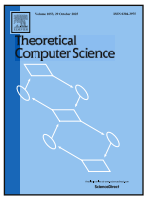
Theoretical Computer Science 1063 (2026) 115645

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs



Elimination of annotation dependencies in validation for Modern JSON Schema

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

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

ARTICLE INFO

Editor: Dr. Vladimiro Vladimiro Sassone

Keywords:
JSON schema
Modern JSON Schema
Schema languages
Schema rewriting
Type theory
Validation

ABSTRACT

JSON Schema is a declarative language that allows one to specify the structure of JSON instances using hierarchical schema objects that combine logical and structural operators.2.2 Early versions of JSON Schema, known collectively as Classical JSON Schema, operated with a straightforward semantics where a schema's meaning was completely determined by which JSON values it could successfully validate. This simple foundation enabled researchers to develop robust theoretical frameworks and practical tools for instance validation and also to determine whether schemas are satisfiable or equivalent to one another. However, Classical JSON Schema had a significant weakness in its inability to effectively express certain kinds of extensions of object schemas.

This limitation prompted a major overhaul in Draft 2019-09, introducing two new features that fundamentally alter how JSON Schema works. The first is *annotation dependency*, where validation now produces more than just a yes/no result. When a schema validates a JSON instance, it also generates an "annotation" that records which fields and items were "evaluated". This annotation then influences the behavior of the new operators "`unevaluatedProperties`" and "`unevaluatedItems`", creating a dependency that did not exist before. The second feature is dynamic references, a separate mechanism that allows for the target of a reference operator to depend on the validation context. These changes were so substantial that all JSON Schema versions from Draft 2019-09 onward are called *Modern JSON Schema*.

This semantic shift invalidated much of the existing theoretical work, and the algorithms that researchers had developed for Classical JSON Schema — particularly those for determining satisfiability and schema inclusion — do not easily adapt to Modern JSON Schema's new behavior. One approach to bridge this gap is "elimination" — converting Modern JSON Schema constructs back into equivalent Classical JSON Schema forms. Previous research successfully developed algorithms for eliminating dynamic references, but annotation dependency remained unsolved.

In this paper we solve this problem, providing three contributions: an *expressibility* result, proving that eliminating annotation-dependent operators is possible; a *succinctness* result, proving that

* Corresponding author.
E-mail addresses: lyes.attouche@dauphine.psl.eu (L. Attouche), mohamed-amine.baazizi@lip6.fr (M.-A. Baazizi), dario.colazzo@dauphine.fr (D. Colazzo), ghelli@di.unipi.it (G. Ghelli), Stefan.Klessinger@uni-passau.de (S. Klessinger), carlo.sartiani@unibas.it;sartiani@gmail.com (C. Sartiani), stefanie.scherzinger@uni-passau.de (S. Scherzinger).

https://doi.org/10.1016/j.tcs.2025.115645
Received 5 January 2025; Received in revised form 14 September 2025; Accepted 6 November 2025
Available online 28 November 2025

eliminating annotation-dependent operators can generally cause schemas to grow exponentially in size, and finally a *practical algorithm* to perform annotation elimination.

Our "practical algorithm" not only matches the asymptotic lower-bound that is provided by the succinctness theorem, but it also presents some specific optimizations that we designed to exploit typical features or real-world schemas. A comprehensive experimental testing, executed on a representative set of 305 schemas retrieved from GitHub, shows that the practical algorithm runs on less than 10 ms on all of them, and in less than 1 ms in the 98 % of the cases, and that, in the 95 % of the cases, it produces schemas in Classical JSON Schema whose size is at most ten times bigger than the source schema written in Modern JSON Schema.

1. Introduction

JSON Schema is a logical language used to define the structure of JSON values, to allow different pieces of software to cooperate. JSON Schema syntax is based on nested *schema* objects. In all versions of JSON Schema until Draft-07 [1] 2.6, which are collectively known as Classical JSON Schema, the semantics of a *schema* was entirely described by the set of JSON values that it validates. This semantics was the basis for a thorough theoretical study and for the development of some semantic tools for JSON Schema, notably tools to decide satisfiability and equivalence of schemas. Unfortunately, Classical JSON Schema suffered a severe limitation in its ability to express extensions of object schemas. After a lengthy discussion, this led to the introduction of two disruptive features in version Draft 2019-09 [2]: annotation dependency and dynamic references.

"Annotation dependency" means that validation of an instance J by a schema S produces a yes/no answer, as in Classical JSON Schema, and also an "annotation", which is a set of *evaluated fields/items*.

This annotation is passed to the annotation-dependent keywords "`unevaluatedProperties`" and "`unevaluatedItems`", whose behavior depends on it. "Dynamic references" refers to the operator "`$dynamicRef`" : k.[1] This operator is executed in a context where the name k may be bound to different schemas, and "`$dynamicRef`" : k chooses, among all these schemas, the first one that has been met; this mechanism allows a recursive schema to be first defined in a basic form, and to be refined later on. Some examples and formal semantics are provided by Attouche et al. [3] and by the official specifications of Draft 2020-12 [4]. This drastic change in semantics is the reason why the versions of JSON Schema from Draft 2019-09 onward are collectively called Modern JSON Schema.

Because of this new semantics, all known formal results about equivalence of schemas or about complexity of some problems must be re-established. In the same way, the algorithms used to decide satisfiability and inclusion for Classical JSON Schema must be redesigned, and, in fact, they are not easy to extend to the new setting. One possible solution is "elimination" — providing an algorithm to translate a schema written in Modern JSON Schema into an equivalent schema in Classical JSON Schema, so that some of the known results and algorithms can be ported to the new language. In this paper, we prove that this is possible.

1.1. JSON Schema, schema factorization, and annotation dependency

The definition of object schemas by extending other schemas is a crucial ability, used in many application fields. Unfortunately, in Classical JSON Schema, this is quite difficult when the resulting object schema is required to be *closed*, where *closed* means that no other properties are allowed apart from those explicitly described.

We use here some examples to introduce JSON Schema, to exemplify the problem, and to show its solution in Modern JSON Schema (the classification of JSON Schema Drafts as Classical vs. Modern has been introduced by Henry Andrews [5]). The following schema defines the shape of the parameters of a protocol in which every exchanged message has "`from`" and "`class`" properties, has either a "`payload`" or an "`errorCode`" property, and has no other property.

```
{ "anyOf": [ { "properties": { "from": { "$ref": ".../address" },
                               "class": { "const": "info" },
                               "payload": { "type": "string" } },
               "required": [ "from", "class", "payload" ],
               "additionalProperties": false },
             { "properties": { "from": { "$ref": ".../address" },
                               "class": { "const": "error"},
                               "errorCode": { "type": "integer" } },
               "required": [ "from", "class", "errorCode" ],
               "additionalProperties": false }
  ]
}
```

[1] More precisely, dynamic references have been introduced through the operator "`$recursiveRef`" : "#" in Draft 2019-09, which has been generalized to "`$dynamicRef`" : k in Draft 2020-12.

The boolean operator "anyOf" : $[S_1, S_2]$ specifies that a JSON value must satisfy either schema S_1 or schema S_2, or both. The keyword "properties" : { "a" : S } specifies that, *if* the "a" property is present, then its value must satisfy S, while "required" forces a property to be present; "additionalProperties" : S specifies that every property that does not match any name introduced by an adjacent "properties" keyword must have a value that satisfies S; when $S =$ false, then no such property can be present, since no value satisfies false; two keywords are adjacent when they are top-level keywords in the same object. In particular, in the above example, the first "additionalProperties" : false forbids any name different from "from", "class", and "payload", while the second forbids any name different from "from", "class" and "errorCode". The "$ref" : u keyword allows any other schema to be referred using a URI u.

We would like to specify these two formats in a different way, by specifying that there is a basic "from" kernel that can be extended in two different ways. This ability is not very relevant in this toy example, but it is crucial in practice. Hence, we may try and rewrite the above schema as follows.

```
{ "properties": { "from": { "$ref": ".../address" } },
  "required": [ "from" ],
  "anyOf": [
    { "properties": { "class": { "const": "info" },
                      "payload": { "type": "string" } },
      "required": [ "class", "payload" ] },
    { "properties": { "class": { "const": "error" },
                      "errorCode": { "type": "integer" } },
      "required": [ "class", "errorCode" ] } ],
  "additionalProperties": false
}
```

This schema enforces the same constraints on "from", "class", "errorCode", and "payload" as the previous schema. However, "additionalProperties" is applied to any property that is not defined by an adjacent "properties" keyword; therefore, in this schema, it would consider "class", "errorCode", and "payload" as "additional" since the "properties" that introduces them is not adjacent to "additionalProperties", hence "additionalProperties": false would be applied, and fail, to any "class", "errorCode", or "payload" property. If we moved "additionalProperties": false inside the branches of the "anyOf", as in the previous version, then "additionalProperties" would be adjacent to the internal "properties" keywords, hence it would consider a "from" property as "additional", and validation of any object with a "from" property would fail.

The problem stems from the fact that the definition of "additionalProperties" depends on the syntactic feature of "being adjacent" to "properties", hence is not robust with respect to schema factorization. To solve this problem, JSON Schema added, with version Draft 2019-09, a new keyword called "unevaluatedProperties", and changed the validation semantics. Now, validation of an instance J against a schema S returns both a boolean result (*valid* or *not valid*) and a set of *evaluated properties* — the properties of the JSON instance under validation that have been evaluated by a "properties" keyword invoked directly or indirectly by the operator. In the rewritten schema above, the first "properties" keyword directly evaluates "from", the "anyOf" keyword indirectly evaluates either "class" and "payload" or "class" and "errorCode", and, if we substitute "additionalProperties" : false with "unevaluatedProperties" : false, this new keyword would regard the properties evaluated by the adjacent "properties" and "anyOf" as "evaluated". Hence, their presence in the object would not cause any problem, while any other property would be regarded as "unevaluated", and in this case the keyword would fail (because of the false parameter), as desired. As a consequence, this new version of the schema, thanks to "unevaluatedProperties", would behave as the original one.

In summary, "unevaluatedProperties" allows the factorization of the definition of "closed" object types, that is, types that admit no other fields apart from those that are explicitly specified, and it relies on the fact that, in Modern JSON Schema, validation returns both a boolean and an *annotation*, specifying which object properties have been evaluated. Modern JSON Schema also defines the operator "unevaluatedItems", which behaves in a similar way for array schemas.

1.2. Motivation of this research

The evaluation model of JSON Schema validation before Draft 2019-09 (Classical JSON Schema) has been studied thoroughly [6, 34], and is quite simple: the behavior of a schema is defined by the set of instances that it validates. With Draft 2019-09 (Modern JSON Schema) the evaluation model changes in two ways:

- the addition of the "unevaluatedProperties" and "unevaluatedItems" keywords requires that validation returns both a boolean (as before) and an "annotation";
- the addition of the "$recursiveRef" operator, generalized by the "$dynamicRef" operator in Draft 2020-12, collectively called *dynamic references*, causes validation to depend on the *dynamic context*, a complex notion that may be defined as "the list of references that have been followed by the operators that caused the invocation of the current operator" [7].

Validation of Classical JSON Schema is known to be P-complete [34]. Attouche et al. [3] proved that the addition of annotation-dependent keywords alone leaves validation inside the P class, while the addition of dynamic references alone makes validation PSPACE-complete, as does the combination of the two. In the same paper, an algorithm has been presented to eliminate

"$dynamicRef", that is, to rewrite all instances of "$dynamicRef" into instances of "$ref" : u keyword, which simply retrieves the schema referred by u with no dependency on the dynamic context. This rewriting requires, in general, an exponential increase in the size of the rewritten schema. In that paper, no result was given on the elimination of "unevaluatedProperties"; however, the fact that validation with "unevaluatedProperties" is still in P left the possibility that "unevaluated*" elimination could be performed with a polynomial size increase.

In this paper, we study how to eliminate "unevaluated*" keywords by rewriting them in terms of annotation-independent keywords. Reasons for doing this, apart from scientific curiosity, are the following:

- Reuse of algorithms and tools: there is a vast ecosystem of tools for Classical JSON Schema; some of them are quite easy to re-engineer for Modern JSON Schema (such as editors or validators), but for tools that decide inclusion, satisfiability, or equivalence, such as those described by Habib et al. [8] and by Attouche et al. [9], no technique is known to extend them to Modern JSON Schema, since they depend on the ability to eliminate conjunction by merging instances of "properties", "patternProperties" and "additionalProperties" that are argument of a logical conjunction into just one instance of "patternProperties" (*and-merge* operation). No algorithm is known to do the same with "unevaluatedProperties"; a translation from Modern JSON Schema to Classical JSON Schema would solve this problem;
- Foundations for the design of native Modern JSON Schema schema analysis algorithms: the known schema-analysis (inclusion, satisfiability, equivalence) algorithms for Classical JSON Schema are based on a set of "normalization" phases, such as *not-elimination* and *and-elimination*, that bring the schema into some specific Disjunctive Normal Form, upon which the final phase of the algorithm is executed. The most natural way to define a native Modern JSON Schema analysis algorithm is, in our opinion, by integrating two phases. The first is the elimination of dynamic references, using the techniques developed by Attouche et al. [3]. The second is the "unevaluated*"-elimination, using the techniques that we develop here, inside the algorithms designed for Classical JSON Schema
- Aid for comprehension: the "unevaluatedProperties" operator is regarded as complex by the same community who designed it, as testified by many public discussions [10]; a translation to Classical JSON Schema simpler operators may be useful to understand the operator and its properties, both in general, and in some specific examples;
- Schema comparison: a translation would make it easier to compare schema versions written in Modern JSON Schema with previous versions written in Classical JSON Schema, and, specifically, to verify whether the two are equivalent or not.

The first point is crucial. While quite technical, it forms the central motivation of this work: all known algorithms for analyzing satisfiability or inclusion in JSON Schema require, at some point, merging a conjunction of many instances of "properties", "patternProperties", and "additionalProperties" into one. The same must be done with the corresponding array operators. However, the fact that "unevaluatedProperties" and "unevaluatedItems" depend on annotations makes it impossible to extend this crucial step to the new operators. No technique, apart from elimination, has been proposed to address this challenge.

It is also worth to notice that there are a few programming languages, e.g., Clojure, Common Lisp, Julia, and Lua, for which, according to what has been reported on https://json-schema.org/tools, there are no validators supporting Modern JSON Schema. While our elimination technique is essential in order to reuse tools that decide schema properties, such as inclusion or equivalence, we do not believe that it should be used to reuse Classical JSON Schema validators to validate Modern JSON Schema. For this specific aim, a direct implementation of the validation rules that we present in Section 3 would be a much simpler approach.

1.3. Contributions

In the following, we use the term *Static Modern JSON Schema* to indicate Modern JSON Schema without dynamic references; we focus on Static Modern JSON Schema since the problem of eliminating dynamic references has already been solved by Attouche et al. [3].

In this paper, we provide the following contributions:

1. Exponential lower bound (succinctness): we show that there exist schemas in Static Modern JSON Schema that contain "unevaluatedProperties" such that any equivalent schema expressed in Classical JSON Schema has an exponentially bigger size, and we show that one occurrence of "unevaluatedProperties" is sufficient for that; we show the same result for the "unevaluatedItems" operator;
2. Elimination algorithm (expressibility): we provide an algorithm to rewrite every occurrence of "unevaluatedItems" and "unevaluatedProperties" in terms of classical keywords; the algorithm is based on a notion of Evaluation Normal Form (ENF) that is studied here for the first time, and which allows us to avoid the exponential blow-up in most practical cases;
3. Test of practical usability: we crawled more than 300 schemas from GitHub that use "unevaluatedProperties" and "unevaluatedItems", and we applied our algorithm on these schemas, and we measured run-time and size expansion; our results confirm that, despite its exponential complexity, the algorithm based on the ENF is fast on real-world schemas, and it produces a small size expansion.

For all the properties that we present in the paper, a full proof is presented in the Appendix.

2. Related work

There exist several schema languages for JSON data other than JSON Schema, e.g., Joi [11] and JSound [12]; Baazizi et al. [13] provided an overview over these languages together with a comparison with JSON Schema. These other languages now have limited diffusion and JSON Schema is affirmed as the de facto standard schema language for JSON data.

Classical JSON Schema has been formally studied in several papers. Pezoa et al. [34] introduced the first formalization of JSON Schema and showed that it cannot be captured by MSO or tree automata because of the "`uniqueItems`" constraints. They focused on validation and proved that the problem is PTIME-complete for Classical JSON Schema; they also proved that satisfiability is EXPTIME-hard by mapping tree automata into JSON Schema (for a more detailed proof see Suárez Barría [14]). Bourhis et al. [6] refined the analysis of Pezoa et al. They mapped JSON Schema onto an equivalent modal logic, called recursive JSL, and proved that satisfiability is EXPTIME-complete for recursive schemas without "`uniqueItems`", and it is in 2EXPTIME for recursive schemas with "`uniqueItems`". Baazizi et al. [15] present a denotational semantics for Classical JSON Schema and discuss negation-completeness for Classical JSON Schema. They show that negation cannot be completely eliminated in JSON Schema because of some minor issues in the way some specific operators are defined. They propose an algebraic version of JSON Schema where negation can be eliminated, and they define an algorithm for not-elimination.

The only existing tool for satisfiability checking has been described by Attouche et al. [9]. This tool, which supports Classical JSON Schema Draft 6 without "`uniqueItems`", works by mapping an input schema into an equivalent algebraic expression, pushing down negations and conjunctions down to the leaves, and transforming the resulting algebraic expression in Disjunctive Normal Form; the tool, finally, checks whether at least one disjunct is satisfiable and, in the positive case, generates a *witness*, that is, a JSON value satisfying the input schema.

The most prominent tool for checking schema inclusion, a problem whose challenges have been studied by Fruth et al. [35], has been described by Habib et al. [8]. This tool has been developed in the context of IBM's open-source AutoML framework LALE [16]. As a consequence, the tool has been highly tailored for this use case, which explains the presence of severe restrictions, such as the lack of support for recursion and for negation on complex schemas. The tool, which supports Classical JSON Schema (Draft 4) without recursion and generalized negation, takes as input two schemas S_1 and S_2, and returns three possible results: (i) *false*, if S_1 is not contained into S_2; (ii) *true*, if S_1 is contained into S_2; or (iii) *unknown*, if the tool was not able to take a decision. The *unknown* result is motivated by the fact the tool exploits a traditional rule-based approach by relying on an incomplete set of rules; therefore, it is possible that no further rule could be applied when comparing two schemas. Despite the lack of support for recursion and generalized negation, the tool has been successfully used to identify 38 real bugs in the LALE framework.

JSON bears some resemblances with XML, and the relationship is quite complex. We refer the reader to Bourhis et al. [6] for a detailed discussion about the differences among these formalisms.

Since JSON Schema is a logical language, it is natural to ask whether it could be mapped onto some well-studied logical languages. First-order languages, such as First Order Logic, Prolog, or Datalog, for example, are not candidates for such a mapping, since these are languages where variables are first-order, that is, they denote one domain value, and sets are defined by collecting sets of first-order variable assignments (often called *valuations*). JSON Schema is a second-order logic, where semantics is defined by a *single* assignment of each variable to a set. A more natural mapping is that of JSON Schema into a modal logic, such as the μ-calculus, since these are logics that share many of the fundamental features of JSON Schema, and this approach has been successfully followed by Bourhis et al. [6] in order to prove the complexity results that we have already described. However, the approaches used to prove satisfiability for these languages [14], cannot be reused for the design of a realistic algorithm.

There exist several tools for data generation starting from a JSON Schema ([17,18], and others). They generate JSON data starting from a schema, but are based on a trial-and-error approach and cannot detect unsatisfiable schemas. Other tools, like the Wisconsin Benchmark Data Generator [19], are of much less interest, as they rely on a single, fixed schema. There are also a few ongoing efforts aiming at using LLMs for generating JSON documents starting from a schema. These approaches usually rely on techniques like *constrained decoding* or *guided generation*, and exhibit significant problems. Geng et al. [20] provide an overview and an experimental evaluation of these approaches.

There exists an active research line that focuses on the definition of query languages for JSON values [6,21]. Differently from the Datalog world, where a unique formalism can be used to describe data and to express queries, the JSON Schema witness generation problem bears a very limited relationship with query answering on JSON data.

Finally, another line of research is schema extraction, where a JSON Schema specification is automatically extracted from one JSON instance, or from a JSON collection. While some approaches focus on discovering schema variants within collection of JSON documents (e.g., Gallinucci et al. [22]), the majority target the extraction of one unified schema. Early works on schema extraction from JSON data ([23,24]) use XML-encoded models to describe the extracted schema, whereas most approaches generate JSON Schema (e.g., Frozza et al. [25], Baazizi et al. [36]). Baazizi et al. [26] use a proprietary format which can be directly mapped to JSON Schema. Notably, Frozza et al. [25] additionally support Binary JSON (BSON), introducing extended data types, which are mapped to standard JSON Schema types in the generated schema description. These approaches have been developed before the introduction of Modern JSON Schema and, consequently, restrict themselves to Classical JSON Schema. In more recent works, Klessinger et al. focus on identifying tagged unions [27,28] and Spoth et al. [29] tackle ambiguity in JSON Schema extraction by heuristically distinguishing between nested collections and entities. Despite targeting complex structures, even these modern approaches only generate schemas in Classical JSON Schema.

All these papers study Classical JSON Schema. To the best of our knowledge, the only formal study about Modern JSON Schema is by Attouche et al. [3], where the authors formalized the new version of the language, and proved that validation becomes

$d \in \mathsf{DNum}, i \in \mathsf{Nat}, n \in \mathsf{Nat}, k \in \mathsf{Str}, b \in \mathsf{Bool}, u \in \mathit{URIs}, p \in \mathit{POSIX\ patterns}, J \in \mathit{JVal}$

$$
\begin{array}{lll}
\mathit{Tp} & ::= & \texttt{"object"} \mid \texttt{"number"} \mid \texttt{"integer"} \mid \texttt{"string"} \mid \texttt{"array"} \mid \texttt{"boolean"} \mid \texttt{"null"} \\
S & ::= & \texttt{true} \mid \texttt{false} \mid \{\ ((\mathit{IK}\ \mathit{Sep}(,))^*\ ,)^?\ ((\mathit{API}\ \mathit{Sep}(,))^*\ ,)^?\ (\mathit{UPUI}\ \mathit{Sep}(,))^*\ \} \\
\mathit{IK} & ::= & \texttt{"minimum"} : d \mid \texttt{"maximum"} : d \mid \texttt{"pattern"} : p \mid \texttt{"const"} : J \mid \texttt{"type"} : \mathit{Tp} \\
 & & \mid \texttt{"anyOf"} : [S_1, \ldots, S_n] \mid \texttt{"allOf"} : [S_1, \ldots, S_n] \mid \texttt{"oneOf"} : [S_1, \ldots, S_n] \\
 & & \mid \texttt{"not"} : S \\
 & & \mid \texttt{"patternProperties"} : \{p_1 : S_1, \ldots, p_n : S_n\} \\
 & & \mid \texttt{"properties"} : \{k_1 : S_1, \ldots, k_n : S_n\} \\
 & & \mid \texttt{"required"} : [k_1, \ldots, k_n] \\
 & & \mid \texttt{"minProperties"} : i \mid \texttt{"maxProperties"} : i \mid \texttt{"propertyNames"} : S \\
 & & \mid \texttt{"prefixItems"} : [S_1, \ldots, S_n] \mid \texttt{"contains"} : S \\
 & & \mid \texttt{"minItems"} : i \mid \texttt{"maxItems"} : i \mid \texttt{"uniqueItems"} : b \\
 & & \mid \texttt{"\$ref"} : u \mid \texttt{"\$defs"} : \{k_1 : S_1, \ldots, k_n : S_n\} \mid \texttt{"\$anchor"} : \mathit{plain\text{-}name} \\
 & & \mid k : J \ \text{(with } k \text{ not previously cited)} \\
\mathit{API} & ::= & \texttt{"additionalProperties"} : S \mid \texttt{"items"} : S \\
\mathit{UPUI} & ::= & \texttt{"unevaluatedProperties"} : S \mid \texttt{"unevaluatedItems"} : S
\end{array}
$$

Fig. 1. Grammar of a core subset of Static Modern JSON Schema, Draft 2020-12.

PSPACE-complete when dynamic references come into play; they also proved that "unevaluatedProperties" and "unevaluatedItems" leave validation in P, and they provided an algorithm to eliminate dynamic references, at the price of an exponential space growth, in the worst case.

3. Syntax and semantics of Modern JSON Schema and Classical JSON Schema

We refer here to Modern JSON Schema as defined in Draft 2020-12 [4]. This language is formalized by Attouche et al. [3]; here we define a subset of the language that is rich enough for our aims.

3.1. Syntax of Modern JSON Schema and Classical JSON Schema

JSON instances J are either base values or nested arrays and objects; the order of object properties (or *fields*) is irrelevant; the names of two different fields in an object must be different, as formalized by the following grammar, where DNum is the set of all decimal numbers (numbers that admit a finite representation in decimal notation).

$s \in \mathsf{Str}, d \in \mathsf{DNum}, n \in \mathsf{Nat}, l_i \in \mathsf{Str}$

$$J ::= \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid d \mid s \mid [J_1, \ldots, J_n] \mid \{l_1 : J_1, \ldots, l_n : J_n\} \qquad i \neq j \Rightarrow l_i \neq l_j$$

In JSON Schema, a schema is expressed in JSON notation. The keywords in a schema appear in any order, but they are evaluated in an order that respects the dependencies among the keywords. Following the approach of Attouche et al. [3], we formalize this behavior by assuming that, before validation, each schema is reordered to respect the grammar in Fig. 1.

The grammar specifies that a schema S is either a boolean schema that matches any JSON instance (`true`) or no instance at all (`false`), or it begins with a possibly empty sequence of *Independent Keywords* (IK), followed by a possibly empty sequence of statically dependent keywords (API, for "additionalProperties" or "items") followed by a possibly empty sequence of annotation-dependent keywords (UPUI, for "unevaluatedProperties" or "unevaluatedItems"). Specifically, the two keywords in API depend on the syntactic content of three keywords in IK ("properties", "patternProperties", "prefixItems"), and the two keywords in UPUI depend on the annotations returned by the keywords in API and in IK. Observe that in JSON Schema jargon, the term *keyword* indicates the entire field, that is, the pair *keywordName: keywordValue*, such as "type" : Tp, not just the "keyword-name"; this can be a bit confusing. In the rest of the paper, we use just K to range on keywords, whenever the distinction $\mathit{IK} - \mathit{API} - \mathit{UPUI}$ is irrelevant.

In the grammar, square brackets "[" and "]", and curly brackets "" and "", are terminal symbols, while we use $(X\ \mathit{Sep}(,))^*$ to indicate a repetition of zero or more instance of the terminal X each followed by the separator ",", and we use $(E)^?$ to indicate optional E; we only use them in the last production of S to indicate the fact that all elements are separated by commas.

Modern JSON Schema introduced the new operators we described and also a minor syntactic reorganization of the annotation-independent array operators, which were called "items" and "additionalItems" in Classical JSON Schema and became "prefixItems" and "items" in Modern JSON Schema. This can be quite confusing; therefore, in this paper, we will always use the Modern JSON Schema syntax for these operators, and we will formalize Classical JSON Schema as "Static Modern JSON Schema without the UPUI production". Translating from this language to an actual dialect of Classical JSON Schema such as Draft-06 just requires a trivial translation of the annotation-independent array operators.

In this grammar, $JVal$ is the set of all JSON values, and *plain-name* denotes any alphanumeric string starting with a letter. A valid schema must also satisfy two more constraints: (1) every URI that is the argument of "`$ref`" must reference a schema, and (2) any two adjacent keywords must have different names, where *adjacent* indicates two fields in the same object.

3.2. Validation judgments

Formalizing the boolean keywords. The behavior of validation is specified by a judgment $\vdash^{S}\ J\ ?\ S \to (r, \kappa)$ that specifies that the validation of an instance J by a schema S returns a boolean r, chosen between $\mathcal{T}$ and $\mathcal{F}$, and an annotation κ, which indicates which of the fields, or items, of J have been evaluated, while the judgment $\vdash^{K}\ J\ ?\ IK \to (r, \kappa)$ specifies the validation behavior of an independent keyword IK. The small letter on top of $\vdash$ is not a variable but a symbol used to differentiate among schema judgments $\vdash^{S}$, keyword judgments $\vdash^{K}$, and list-of-keywords judgments $\vdash^{L}$, introduced later on. We will use $\vdash^{S}\ J\ ?\ S$ to say that S validates J, that is, that $\vdash^{S}\ J\ ?\ S \to (\mathcal{T}, \kappa)$ for some κ. The $\vdash^{S}$ and $\vdash^{K}$ judgments are specified by mutual induction by formal rules such as the following, which specifies that the application of a conjunction keyword "`allOf`" : $[S_1, \ldots, S_n]$ to any instance J yields the conjunction of all the results r_i obtained by applying each S_i schema to J. It also specifies that the set of fields/items evaluated by the "`allOf`" keyword is the union of all the fields/items κ_i that are evaluated by the S_i arguments of the keyword.

$$\frac{\forall i \in \{\!|\, 1 \ldots n \,|\!\}.\ \vdash^{S}\ J\ ?\ S_i \to (r_i, \kappa_i)}{\vdash^{K}\ J\ ?\ \texttt{"allOf"} : [S_1, \ldots, S_n] \to (\bigwedge_{i \in \{\!|\, 1 \ldots n \,|\!\}} r_i, \bigcup_{i \in \{\!|\, 1 \ldots n \,|\!\}} \kappa_i)} \quad (\mathsf{allOf})$$

The disjunction rule is almost identical, but has $\vee$ instead of ς; observe that this rule prevents any short-circuit evaluation: Even if the first argument S_1 was satisfied by J, all other branches must still be evaluated, since one must collect the annotations that they produce.

$$\frac{\forall i \in \{\!|\, 1 \ldots n \,|\!\}.\ \vdash^{S}\ J\ ?\ S_i \to (r_i, \kappa_i)}{\vdash^{K}\ J\ ?\ \texttt{"anyOf"} : [S_1, \ldots, S_n] \to (\bigvee_{i \in \{\!|\, 1 \ldots n \,|\!\}} r_i, \bigcup_{i \in \{\!|\, 1 \ldots n \,|\!\}} \kappa_i)} \quad (\mathsf{anyOf})$$

Finally, rule (oneOf) (see Fig. 2) specifies that, to satisfy "`oneOf`" : $[S_1, \ldots, S_n]$, J must satisfy one, and only one, of $S_1, \ldots, S_n$. Rule (not) (Fig. 2) specifies that to satisfy "`not`" : S, J must not satisfy S.

Terminal keywords. The behavior of keywords that do not refer to any other schemas, which we call *terminal keywords* and which are listed in Table 1, is completely defined by a type and a condition, since they do not return any annotation.

For example, "`required`" : $[k_1, \ldots, k_n]$ is defined by the type $TypeOf(\texttt{"required"}) = \texttt{"object"}$, by the condition $\mathsf{cond}(J, \texttt{"required"} : [k_1, \ldots, k_n]) = \forall i \in 1..n.\ k_i \in \texttt{names}(J)$, and by the two following rules:

$$\frac{TypeOf(J) \neq TypeOf(kw)}{\vdash^{K}\ J\ ?\ (kw : J') \to (\mathcal{T}, \emptyset)} \quad (kw\mathsf{Triv}) \qquad \frac{TypeOf(J) = TypeOf(kw) \quad r = \mathsf{cond}(J, kw : J')}{\vdash^{K}\ J\ ?\ (kw : J') \to (r, \emptyset)} \quad (kw)$$

The generic rules, once instantiated as indicated in Table 1, become as follows. The first rule says that the keyword is satisfied by any instance that is not an object; the second rule says that, when J is an object, then J satisfies "`required`" : $[k_1, \ldots, k_n]$ iff every k_i belongs to $\texttt{names}(J)$, where we use $\texttt{names}(J)$ to indicate the set of the field names of J.

$$\frac{TypeOf(J) \neq \mathsf{object}}{\vdash^{K}\ J\ ?\ \texttt{"required"} : [k_1, \ldots, k_n] \to (\mathcal{T}, \emptyset)} \quad (\mathsf{requiredTriv}) \qquad \frac{TypeOf(J) = \mathsf{object} \quad r = \forall i \in 1..n.\ k_i \in \texttt{names}(J))}{\vdash^{K}\ J\ ?\ \texttt{"required"} : [k_1, \ldots, k_n] \to (r, \emptyset)} \quad (\mathsf{required})$$

Table 1 provides a full specification of all terminal keywords that we consider in this paper. In the first two lines, the notation $TypeOf(\texttt{"type"} : \mathsf{Tp}) = \textit{no type}$ indicates that these keywords do not have the $kw\mathsf{Triv}$ rule, since they are not specific to a single type. In the "`pattern`" line we use the notation $L(p)$ to indicate the set of strings that match p; in the "`min*`" and "`max*`" rules, we use the symbol $|J|$ to indicate the number of fields of an object J and the number of elements of an array J.[2]

[2] Actually, "`propertyNames`" : S_n is not really a "terminal" keyword, since S_n is a schema parameter. As this keyword neither generates nor passes any annotation, it is still fully defined by a type and a condition, hence we can provide its definition in this table.

Table 1
Terminal keywords.

assertion $kw : J'$	$TypeOf(kw)$	$\mathsf{cond}(J, kw : J')$
"const" : J_c	*no type*	$J = J_c$
"type" : Tp	*no type*	$TypeOf(J) = \mathsf{Tp}$
"minimum": d	number	$J \geq \mathrm{d}$
"maximum": d	number	$J \leq \mathrm{d}$
"pattern": p	string	$J \in L(p)$
"propertyNames" : S	object	$\forall k \in \mathtt{names}(J).\ \vdash^{S}\ k\ ?\ S$
"minProperties": i	object	$\lvert J\rvert \geq i$
"maxProperties": i	object	$\lvert J\rvert \leq i$
"required" : $[k_1, \ldots, k_n]$	object	$\forall i.\ k_i \in \mathtt{names}(J)$
"uniqueItems": true	array	$J = [J_1, \ldots, J_n]$ with $n \geq 0$ ς $\forall i, j.\ 1 \leq i \neq j \leq n \Rightarrow J_i \neq J_j$
"uniqueItems": false	array	True
"minItems": i	array	$\lvert J\rvert \geq i$
"maxItems": i	array	$\lvert J\rvert \leq i$

Non-terminal independent structural keywords. Boolean operators collect and transmit the κ_i annotations of their arguments, but annotations are originated by the object and array keywords. We introduce now the rule for the "patternProperties" keyword, starting with an example. Consider an object $J = \{$ "age" : 1, "phone" : null $\}$ and a keyword

$$K = \text{"patternProperties"} : \{\ \text{"a"} : S_a, \text{"g"} : S_g, \text{"x|y"} : S_{xy}\ \}.$$

To compute $\vdash^{K}\ J\ ?\ K$, one first collects in Π all pairs (J-field, K-field) where the J-field name matches the K-field pattern:[3]

$$\Pi = \{\!|\ (\text{"age"} : 1, \text{"a"} : S_a), (\text{"age"} : 1, \text{"g"} : S_g)\ |\!\}.$$

(In this paper we use $\{\!|\ \ldots\ |\!\}$ to indicate sets since we use $\{\ \ldots\ \}$ for JSON objects.) For each matching pair $(k_i : J_i, p_j : S_j)$, we check that J_i is validated by S_j; observe that J can contain fields that are not matched by K (such as "phone": null), and K can contain fields that do not match J (such as "x|y": S_{xy}); such fields are just ignored. The object fields *evaluated* by K are all and only those that appear in Π, as specified in the returned pair $(\ldots, \{\!|\ k_i\ |\!\}^{(k_i : J_i, p_j : S_j) \in \Pi})$.

$$\frac{\begin{array}{c} J = \{\ k_1 : J_1, \ldots, k_n : J_n\ \} \qquad \Pi = \{\!|\ (k_i : J_i, p_j : S_j)\ \mid\ k_i \in L(p_j)\ |\!\} \\ \forall \pi = (k_i : J_i, p_j : S_j) \in \Pi.\ \vdash^{S}\ J_i\ ?\ S_j \rightarrow (r_\pi, \kappa_\pi) \end{array}}{\vdash^{K}\ J\ ?\ \text{"patternProperties"} : \{\ p_1 : S_1, \ldots, p_m : S_m\ \} \rightarrow (\textstyle\bigwedge_{\pi \in \Pi} r_\pi, \{\!|\ k_i\ |\!\}^{(k_i : J_i, p_j : S_j) \in \Pi})}\ \text{(patternProperties)}$$

Observe that the annotations κ_π returned by the judgment in the premise are discarded. The reason is that each rule only returns fields/items that belong to the current instance J, such as the k_i fields that are returned by this rule, while the judgment in the premise does not apply S_j to the current instance J but to the value of a field $k_i : J_i$ of J, so that κ_π contains fields that are not fields of J, but are fields of J_i. This was not the case for the boolean operators, which reapply their arguments S_i to the same instance J. In the JSON Schema jargon, the keywords that analyze the content of the instance are called *structural* keywords, while those that just reapply their schema arguments to the same J are called *in-place* applicators. Structural keywords may originate annotations, but they do not transmit annotations generated by others, while in-place keywords transmit annotations but do not originate them.

The above rule can only be applied to an instance J that is an object; in the other cases, the following "trivial" rule applies. All keywords that refer to one specific type have their own version of this rule: they are trivially *satisfied* by any instance that belongs to any other type.

$$\frac{TypeOf(J) \neq \text{"object"}}{\vdash^{K}\ J\ ?\ \text{"patternProperties"} : J' \rightarrow (\mathcal{T}, \emptyset)}\ \text{(patternPropertiesTriv)}$$

It is worth repeating that this operator does not require the presence of any fields; it only constrains the fields that are present to satisfy the associated schemas.

We give now a rapid overview of the other non-terminal structural rules for the independent keywords (those of the IK production in the grammar); for a complete explanation, we refer to Attouche et al. [3]. The rule (properties) is almost identical to rule (patternProperties). The only difference is that we do not match an instance field $k_i : J_i$ in J with the many, or zero, keyword fields $p_j : S_j$ such that $k_i \in L(p_j)$, but with the one, or zero, keyword field $k_j : S_j$ such that $k_i = k_j$. Rule (prefixItems) specifies that J satisfies "prefixItems" : $[S_1, \ldots, S_m]$ if every element J_i in a position $i \leq m$ satisfies S_i. Similarly to "properties", it does not require the presence of any item — it is always satisfied by an empty array — but it constrains the items that are present to satisfy the corresponding schemas. Rule (contains) specifies that J satisfies "contains" : S if it contains at least one element that satisfies S; the positions of all and only the items of J that satisfy S are returned in κ as "evaluated".

[3] The pattern "a" matches any string that contains "a", hence it matches "age".

Dependent keywords. The keyword judgment $\vdash^{\mathsf{K}} J \ ? \ IK \to (r, \kappa)$ specifies the behavior of a single independent keyword. The behavior of a sequence of keywords $[\![K_1, \ldots, K_n]\!]$, where each K_j may depend on the keywords K_i with $i < j$, is specified by a new judgment $\vdash^{\mathsf{L}} J \ ? \ [\![K_1, \ldots, K_n]\!] \to (r, \kappa)$, which, for the independent keywords, takes the conjunction of the validation results and the union of the evaluated fields; the judgment is inductive; hence we also have a rule for the empty case. Here, $[\![\ldots]\!]$ indicates a list, $[\![ ]\!]$ is an empty list, $\vec{K}$ indicates a list of keywords, and $\vec{K} + K$ is $\vec{K}$ extended with K.

$$\frac{K \in IK \qquad \vdash^{\mathsf{L}} J \ ? \ \vec{K} \to (r_l, \kappa_l) \qquad \vdash^{\mathsf{K}} J \ ? \ K \to (r, \kappa)}{\vdash^{\mathsf{L}} J \ ? \ (\vec{K} + K) \to (r_l \ \varsigma \ r, \kappa_l \cup \kappa)} \ \text{(klist-IK)} \qquad \vdash^{\mathsf{L}} J \ ? \ [\![ ]\!] \to (\mathcal{T}, \emptyset) \ \text{(klist-0)}$$

This keyword-list judgment allows us to specify the behavior of the annotation-dependent keywords, which are "`unevaluatedProperties`" and "`unevaluatedItems`" (production $UPUI$ in the grammar).

The rule for "`unevaluatedProperties`" is the following.

$$\frac{J = \{ k_1 : J_1, \ldots, k_n : J_n \} \qquad \vdash^{\mathsf{L}} J \ ? \ \vec{K} \to (r, \kappa) \qquad \Pi = \{\!| (k_i : J_i) \mid 1 \le i \le n \ \varsigma \ k_i \notin \kappa |\!\} \qquad \forall \pi = (k_i : J_i) \in \Pi. \ \vdash^{\mathsf{S}} J_i \ ? \ S \to (r_\pi, \kappa_\pi)}{\vdash^{\mathsf{K}} J \ ? \ (\vec{K} + \texttt{"unevaluatedProperties"} : S) \to (r \ \varsigma \ \bigwedge_{\pi \in \Pi} r_\pi, \{\!| k_1 \ldots, k_n |\!\})} \ \text{(unevaluatedProperties)}$$

The rule first computes the annotation κ produced by the application of all keywords in $\vec{K}$ to J, and the set Π that contains all fields of J that are not listed in κ. Then, it checks that these unevaluated fields satisfy S. The result combines the boolean r returned by the keywords $\vec{K}$ with the conjunction of the results of the tests performed by the "`unevaluatedProperties`" keyword. The judgment returns, as evaluated fields, the complete list of all object fields ($\to (\ldots, \{\!| k_1 \ldots, k_n |\!\})$), because the combination ($\vec{K}$ + "`unevaluatedProperties`" : S) evaluates all fields of J.

As we explained in the Introduction, the keyword "`unevaluatedProperties`" has been added in Modern JSON Schema to generalize the behavior of "`additionalProperties`". The keyword "`additionalProperties`", which has been retained in Modern JSON Schema, only excludes those fields that are evaluated by an adjacent object keyword "`patternProperties`" or "`properties`" — it ignores what is evaluated by adjacent in-place applicators such as the boolean applicator or the reference applicators. It is thus less powerful than "`unevaluatedProperties`", but it can be defined and implemented in a way that does not depend on annotation passing but relies only on a static analysis of the patterns that are listed in the adjacent object keywords. This is formalized by the rule below, where the set Π does not depend on the annotations κ produced by $\vec{K}$, but on the patterns and keywords that can be statically extracted from the object keywords in $\vec{K}$, as formalized by the function $\texttt{propsOf}(\vec{K})$, defined below the rule.

$$\frac{J = \{ k_1 : J_1, \ldots, k_n : J_n \} \qquad \vdash^{\mathsf{L}} J \ ? \ \vec{K} \to (r, \kappa) \qquad \Pi = \{\!| (k_i : J_i) \mid 1 \le i \le n \ \varsigma \ k_i \notin L(\texttt{propsOf}(\vec{K})) |\!\} \qquad \forall \pi = (k_i : J_i) \in \Pi. \ \vdash^{\mathsf{S}} J_i \ ? \ S \to (r_\pi, \kappa_\pi)}{\vdash^{\mathsf{K}} J \ ? \ (\vec{K} + \texttt{"additionalProperties"} : S) \to (r \ \varsigma \ \bigwedge_{\pi \in \Pi} r_\pi, \{\!| k_1 \ldots, k_n |\!\})} \ \text{(additionalProperties)}$$

$$\begin{array}{lcll}
\texttt{propsOf}(\texttt{"properties"} : \{ k_1 : S_1, \ldots, k_m : S_m \}) & = & \underline{k_1} \cdot \texttt{"|"} \cdot \ldots \cdot \texttt{"|"} \cdot \underline{k_m} & \\
\texttt{propsOf}(\texttt{"patternProperties"} : \{ p_1 : S_1, \ldots, p_m : S_m \}) & = & p_1 \cdot \texttt{"|"} \cdot \ldots \cdot \texttt{"|"} \cdot p_m & \\
\texttt{propsOf}(K) & = & \emptyset & \text{otherwise} \\
\texttt{propsOf}([\![K_1, \ldots, K_n]\!]) & = & \texttt{propsOf}(K_1) \cdot \texttt{"|"} \cdot \ldots \cdot \texttt{"|"} \cdot \texttt{propsOf}(K_n) &
\end{array}$$

$\texttt{propsOf}(\vec{K})$ returns a pattern that matches all, and only, the property names matched by a "`properties`" : O or "`patternProperties`" : O keyword that belongs to $\vec{K}$. In the first line of the definition of `propsOf`, we use $\underline{k_i}$ to indicate a pattern that only matches k_i; for example, $\underline{\texttt{address}}$ is the pattern "`^address$`" (in POSIX notation); $\emptyset$ is a pattern that matches no string.

The rule for the array keyword "`unevaluatedItems`" : S_u (Fig. 2) is similar to that for "`unevaluatedProperties`": Every item that is not evaluated, directly or indirectly, by any adjacent keyword, is evaluated by "`unevaluatedItems`" : S_u, hence it must satisfy S_u; array items are directly evaluated by "`prefixItems`", "`contains`", "`items`", "`unevaluatedItems`", and are indirectly evaluated by the in-place keywords, as happens for object properties.

The keyword "`items`" : S_i was called "`additionalItems`" up to Draft-07[4] and it corresponds to "`additionalProperties`": the items that are evaluated by an adjacent "`prefixItems`" are regarded as "evaluated", but the schema S_i is applied to all other items.

As a very artificial example, consider the two schemas below, both describing arrays. The first one imposes that the first element is a number, the second one is a string, that at least one number is present, and that every item after the string is a number: "`unevaluatedItems`": `false` fails on every item that is not analyzed by any of the previous keywords, including "`anyOf`" and "`contains`". For example, [3, "a", 3] would be validated, and [3, "a", "a"] would fail.

[4] Actually, up to Draft-07, the function of the current "`items`" was distributed between "`items`" and "`additionalItems`", see [30].

The second schema is stricter: it imposes the presence of a number, necessarily in the first position, since no other item would be accepted: "items": false only ignores items that are evaluated by an adjacent "prefixItems". For example, an array [2, "a"] would not be validated by the second schema since the item "a", even if it satisfies the "anyOf" keyword, is analyzed again by "items" : false.

```
{ "type": "array",
  "prefixItems": [ { "type": "number" } ],
  "anyOf": [ { "prefixItems": [ {}, { "type": "string" } ] } ],
  "contains": { "type": "number" },
  "unevaluatedItems": false
}
```

```
{ "type": "array",
  "prefixItems": [ { "type": "number" } ],
  "anyOf": [ { "prefixItems": [ {}, { "type": "string" } ] } ],
  "contains": { "type": "number" },
  "items": false
}
```

This limited expressive power of "items" allows it to be executed without relying on annotations, but only on a statical inspection of the adjacent "prefixItems" keyword, if it exists: in the rule for "items", the function $prefLenOf(\vec{K})$ returns the length of the prefix-array of the only "prefixItems" keyword in $\vec{K}$, or zero in case no such keyword is present in $\vec{K}$. Apart from this, the rule exactly corresponds to that for "additionalProperties".

$$\frac{\begin{array}{c} J = [\,J_1, \ldots, J_n\,] \qquad \vdash^{\mathsf{L}} J \;?\; \vec{K} \to (r, \kappa) \\ \Pi = \{\!|\, (i, J_i) \mid 1 \le i \le n \;\varsigma\; i \notin \{\!|\, 1 \ldots prefLenOf(\vec{K}) \,|\!\} \,|\!\} \\ \forall \pi = (i, J_i) \in \Pi.\ \vdash^{\mathsf{S}} J_i \;?\; S \to (r_\pi, \kappa_\pi) \end{array}}{\vdash^{\mathsf{L}} J \;?\; (\vec{K} + \texttt{"items"} : S) \to (r \;\varsigma\; \bigwedge_{\pi \in \Pi} r_\pi, \{\!|\, 1 \ldots, n \,|\!\})} \quad \text{(items)}$$

Deletion of annotations. A fundamental property of annotations is the fact that, once a schema fails, all of the annotations generated by its keywords are deleted.[5] This behavior is reflected by the rules that define the behavior of an object schema $\{\,\vec{K}\,\}$. Specifically, while rule (objSchema-T) passes the annotation κ generated by $\vec{K}$, rule (objSchema-F) returns the pair $(\mathcal{F}, \emptyset)$: the annotation κ generated by the failing keyword list is forgotten.

$$\frac{\vdash^{\mathsf{L}} J \;?\; [\![\, K_1, \ldots, K_n \,]\!] \to (\mathcal{T}, \kappa)}{\vdash^{\mathsf{S}} J \;?\; \{\, K_1, \ldots, K_n \,\} \to (\mathcal{T}, \kappa)} \text{(objSchema-T)}$$

$$\frac{\vdash^{\mathsf{L}} J \;?\; [\![\, K_1, \ldots, K_n \,]\!] \to (\mathcal{F}, \kappa)}{\vdash^{\mathsf{S}} J \;?\; \{\, K_1, \ldots, K_n \,\} \to (\mathcal{F}, \emptyset)} \text{(objSchema-F)}$$

Deletion of annotations implies that a schema { "not" : S } never generates annotations: if it fails, rule (objSchema-F) suppresses the annotations of { "not" : S }; if it does not fail, this means that S fails, and, in this case, rule (objSchema-F) suppresses the annotations of S. As a consequence, in Modern JSON Schema the equivalence { "not" : { "not" : S } } = S does not hold any longer, since the first schema returns no annotation, while S may generate annotations.

Example 1. Consider the following schema S.

```
{ "not":  { "properties": { "name" : { "type": "string" } },
            "minProperties" : 2
  }
}
```

Consider now the JSON object J = { "name" : "P" }. This object fails the "minProperties" : 2 keyword, hence the schema { "properties" : …, "minProperties" : 2 } fails, hence that schema produces no annotation because of rule objSchema-F — the annotations of "properties" : … are lost. Due to the failure of that inner schema, the entire schema { "not" : { … } } is successful, but it does not have an annotation to transmit.

[5] Differently from failing schemas, failing keywords can produce annotations (see the rule for "patternProperties"); this detail is irrelevant for validation, since a failing keyword forces the surrounding schema to fail hence the annotation is not passed, but it is very important to reduce the number of error messages in some specific situations; this is the only reason why failing keywords generate annotations.

References and definitions. In JSON Schema, the "$ref" : u "reference keyword" allows one to validate an instance using a schema identified by the URI u. Consider, for example, the following reformulation of the example from the Introduction.

```
{ "properties": { "from": { "$ref": "#/$defs/address" } },
  "anyOf": [ { "$ref": "#infoMsgS"}, { "$ref": "#errMsgS" } ],
  "required": [ "from" ],
  "additionalProperties": false,
  "$defs": {
     "infoMsg": { "$anchor": "infoMsgS",
                  "properties": { "class": { "const": "info" },
                                  "payload": { "type": "string" } },
                  "required": [ "class", "payload" ] },
     "errMsg":  { "$anchor": "errMsgS",
                  "properties": { "class": { "const": "error" },
                                  "errorCode": { "type": "integer" } },
                  "required": [ "class", "errorCode" ] },
     "address": { "type": "string" }
  }
}
```

The keyword "$defs" : O, generally used in the top-level schema, is a placeholder used to collect inside O name-schema pairs "nn" : S_{nn}, that are called *definitions*, and we say that S_{nn} is a *named schema*, having "nn" as its name. A named schema with name "nn" can be referred using either the path "#/$defs/nn", or, in case it contains the "$anchor" : aa keyword, also using the shorter form "#aa".

The rule for "$ref" : u is very simple: we retrieve the schema referenced by u using the function $deref(u)$, and we use the retrieved schema to validate J. The URI u has, in general, the shape $absURI \cdot "\#" \cdot f$, where $absURI$ identifies a resource and f identifies a fragment inside the resource. We do not formalize $deref(u)$, because the association between $absURI$ and the resource is implementation dependent. In this paper, we will only use references with shape $\#f$ (empty $absURI$). We use the formalism for local references as introduced by Attouche et al. [3]. For the purposes of this paper, it suffices to stipulate that validation is always performed with respect to an implicit current JSON Schema document S, that $deref("\#/\$defs/nn")$ returns the named schema with name "nn" in the implicit current document S, and that $deref("\#aa")$ returns the only subschema of the implicit current document S that has "$anchor" : aa as a top-level keyword.

$$\frac{S' = deref(u) \qquad \vdash^{S} J \ ? \ S' \rightarrow (r, \kappa)}{\vdash^{K} J \ ? \ "\$ref" : u \rightarrow (r, \kappa)} \qquad (\$ref)$$

The validation behavior of the "$defs" keyword is formalized by rule (other) (see Fig. 2), which specifies that the keyword is ignored during validation; the schemas that "$defs" collects are only used when a "$ref" keyword that refers them is met. The same is true for the "$anchor" keyword: it is not used to validate instances, but only to guide the behavior of the $deref$ function.

Draft 2020-12 also defines the dynamic reference operator "$dynamicRef" : $absURI \cdot "\#" \cdot f$, where the schema that is retrieved depends on the "dynamic context", that is, the sequence of URIs that have been retrieved before dereferencing $absURI \cdot "\#" \cdot f$. We will not discuss it here, since it does not play a role in this paper; a detailed explanation of its behavior can be found in the earlier work by Attouche et al. [3].

A schema is *closed* when all the reference keywords it contains are local and refer to a named schema that is actually present in the current schema — in this paper we will assume that all the schemas with which we work are closed. Of course, the lower bounds that we prove in this paper are not affected by the fact that we only use local references, provided that we consider the size of any remote schema referenced by a schema S as part of the size of S.

References in JSON Schema may be recursive, but (informally) every time we are able to reach a reference from itself, we must traverse at least one structural keyword; this constraint has been called "well-formedness" by Pezoa et al. [34] and by Bourhis et al. [6], "guarded recursion" by Attouche et al. [9], and "no infinite recursive nesting" in the specifications ([4], Section 9.4.1). Guarded recursion can be defined as follows.

Definition 1 (Guarded recursion constraint)**.** In a closed schema S, a subschema S' immediately unguardedly depends on S'' if (a) S' has a top-level boolean keyword that has S'' among its arguments or (b) S' has a "$ref" : u top-level keyword and $S'' = deref(u)$. The guarded-recursion constraint says that the graph defined by this relation is acyclic.

The specifications ([4], Section 9.4.1) require that the repeated application of the validation rules never produce an infinite loop; this is equivalent to Definition 1, because the boolean operators and the reference operators are the only ones that are executed "in-place", that is, which do not shift the attention on a strict subterm of the analyzed instance. This constraint allows us to define

the following measure that we will use in many inductive proofs; the guarded recursion constraint could actually be rephrased as "for any closed schema S, the in-place depth of any subschema of S is well defined".

For technical reasons, rather than giving a direct definition of the in-place depth of "oneOf" : $[S_1, \dots, S_n]$, we define it in terms of the in-place depth of its boolean encoding, as follows.

Definition 2 ($booleanOneOf(S_1, \dots, S_n)$)**.** The schema $booleanOneOf(S_1, \dots, S_n)$ is defined as the following encoding of "oneOf" in terms of the other boolean operators:

$$
\begin{aligned}
&booleanOneOf(S_1, \dots, S_n) = \\
&\ \text{"anyOf"}: \\
&\quad [\quad \{\ \text{"allOf"}: [\ S_1, \{\ \text{"not"}: S_2\ \}, \dots, \{\ \text{"not"}: S_n\ \}\ ]\ \}, \\
&\qquad \dots, \\
&\qquad \{\ \text{"allOf"}: [\ \{\ \text{"not"}: S_1\ \}, \dots, \{\ \text{"not"}: S_{i-1}\ \}, S_i, \{\ \text{"not"}: S_{i+1}\ \}, \dots, \{\ \text{"not"}: S_n\ \}\ ]\ \}, \\
&\qquad \dots, \\
&\qquad \{\ \text{"allOf"}: [\ \{\ \text{"not"}: S_1\ \}, \dots, \{\ \text{"not"}: S_{n-1}\ \}, S_n\ ]\ \} \\
&\quad]
\end{aligned}
$$

Definition 3 (In-place depth)**.** We define the in-place depth ID of any subkeyword or subschema of a schema S as follows (we assume $\max(\emptyset) = 0$):

$$
\begin{array}{lcll}
ID(\{\ K_1, \dots, K_n\ \}) & = & \max(ID(K_1), \dots, ID(K_n)) + 1 & n \geq 0 \\
ID(\texttt{true/false}) & = & 0 & \\
ID(\text{"allOf"/"anyOf"} : [S_1, \dots, S_n]) & = & \max(ID(S_1), \dots, ID(S_n)) + 1 & n \geq 0 \\
ID(\text{"not"} : S) & = & ID(S) + 1 & \\
ID(\text{"oneOf"} : [S_1, \dots, S_n]) & = & ID(booleanOneOf(S_1, \dots, S_n)) + 1 & n \geq 0 \\
ID(\text{"\$ref"} : u) & = & ID(deref(u)) + 1 & \\
ID(K) & = & 0 & \text{otherwise}
\end{array}
$$

In Fig. 2, we report the rules for the non-terminal keywords that we use in the paper; for a formalization of the entire language, we refer to Attouche et al. [3].

The rules for Classical JSON Schema are identical to those for Modern JSON Schema apart from (1) lack of the rules for the "unevaluated*" operators and (2) since validation in Classical JSON Schema does not depend on annotation, the Classical judgment just returns a boolean value r, as in $\vdash^S J\ ?\ S \to r$, rather than a value-annotation pair (r, κ); some examples can be found in Section 3.3.

3.3. Negation-completed Classical JSON Schema

Baazizi et al. [15] proved that negation can be eliminated from Classical JSON Schema, if the language is enriched by keywords: "patternRequired" : { p : S }, "notMultipleOf" : d, "notPattern" : p, "repeatedItems" : b, "containsAfter" : S. These operators are defined by the following table and rules. The rules specify that, if J is an object, then "patternRequired" : { p : S } requires the presence of at least one field whose name matches p and whose value satisfies S. The keyword "containsAfter" : S, if J is an array, requires the presence of at least one item that satisfies S and whose position strictly follows the last position that is described by an adjacent "prefixItems"; "containsAfter" : S requires an item in the same range of positions that are constrained by "items" : S. As happens for all the typed keywords, any instance that is *not* an object (respectively, an array) is validated by "patternRequired" (respectively, by "containsAfter").

assertion $kw : J'$	$TypeOf(kw)$	cond$(J, kw : J')$
"notMultipleOf": d	number	$\forall i \in \text{Int}.\ J \neq (i \times d)$
"notPattern": p	string	$J \notin L(p)$
repeatedItems: true	array	$J = [J_1, \dots, J_n]\ \varsigma\ \exists i, j.\ i \neq j\ \varsigma\ J_i = J_j$
repeatedItems: false	array	True

$$
\frac{J = \{\ k_1 : J_1, \dots, k_n : J_n\ \} \qquad \forall k_i \in L(p).\ \vdash^S J_i\ ?\ S \to r_i \qquad r = (|\{|\ i \mid k_i \in L(p)\ \varsigma\ r_i = \mathcal{T}\ |\}| \geq 1)}{\vdash^S J\ ?\ \text{"patternRequired"} : \{\ p : S\ \} \to r} \quad \text{(patternRequired)}
$$

$$
\frac{J = [\ J_1, \dots, J_n\ ] \qquad \vdash^{\perp} J\ ?\ \vec{K} \to r \qquad \forall i \in \{|\ prefLenOf(\vec{K}) + 1 \dots n\ |\}.\ \vdash^S J_i\ ?\ S \to r_i \qquad r' = (|\{|\ i \mid r_i = \mathcal{T}\ |\}| \geq 1)}{\vdash^{\perp} J\ ?\ (\vec{K} + \text{"containsAfter"} : S) \to r\ \varsigma\ r'} \quad \text{(containsAfter)}
$$

$$\frac{\forall i \in 1..n.\ \ \vdash^{s} J\ ?\ S_i \to (r_i, \kappa_i)}{\vdash^{k} J\ ?\ \texttt{"anyOf"} : [S_1, ..., S_n] \to (\bigvee_{i\in 1..n} r_i, \bigcup_{i\in 1..n} \kappa_i)}\ \text{(anyOf)}$$

$$\frac{\forall i \in 1..n.\ \ \vdash^{s} J\ ?\ S_i \to (r_i, \kappa_i)}{\vdash^{k} J\ ?\ \texttt{"allOf"} : [S_1, ..., S_n] \to (\bigwedge_{i\in 1..n} r_i, \bigcup_{i\in 1..n} \kappa_i)}\ \text{(allOf)}$$

$$\frac{\forall i \in 1..n.\ \ \vdash^{s} J\ ?\ S_i \to \kappa_i \qquad r = (\ |\{\!|\ i \mid r_i = \mathcal{T}\ |\!\}| = 1\ )}{\vdash^{k} J\ ?\ \texttt{"oneOf"} : [S_1, ..., S_n] \to (r, \bigcup_{i\in 1..n} \kappa_i)}\ \text{(oneOf)}$$

$$\frac{\vdash^{s} J\ ?\ S \to (r, \kappa)}{\vdash^{k} J\ ?\ \texttt{"not"} : S \to (\neg r, \kappa)}\ \text{(not)}$$

$$\frac{S' = deref(u) \qquad \vdash^{s} J\ ?\ S' \to (r, \kappa)}{\vdash^{k} J\ ?\ \texttt{"\$ref"} : u \to (r, \kappa)}\ \text{(\$ref)}$$

$$\frac{K \in IK \qquad \vdash^{l} J\ ?\ \vec{K} \to (r_l, \kappa_l) \qquad \vdash^{k} J\ ?\ K \to (r, \kappa)}{\vdash^{l} J\ ?\ (\vec{K} + K) \to (r_l \wedge r, \kappa_l \cup \kappa)}\ \text{(klist-IK)}$$

$$\vdash^{l} J\ ?\ [\,] \to (\mathcal{T}, \emptyset)\ \text{(klist-0)}$$

$$\frac{k \text{ not defined in any other rule}}{\vdash^{k} J\ ?\ k{:}J' \to (\mathcal{T}, \emptyset)}\ \text{(other)}$$

$$\vdash^{s} J\ ?\ \texttt{true} \to (\mathcal{T}, \emptyset)\ \text{(trueSchema)} \qquad \vdash^{s} J\ ?\ \texttt{false} \to (\mathcal{F}, \emptyset)\ \text{(falseSchema)}$$

$$\frac{\vdash^{l} J\ ?\ [K_1, \ldots, K_n] \to (\mathcal{T}, \kappa)}{\vdash^{s} J\ ?\ \{K_1, \ldots, K_n\} \to (\mathcal{T}, \kappa)}\ \text{(objSchema-T)} \qquad \frac{\vdash^{l} J\ ?\ [K_1, \ldots, K_n] \to (\mathcal{F}, \kappa)}{\vdash^{s} J\ ?\ \{K_1, \ldots, K_n\} \to (\mathcal{F}, \emptyset)}\ \text{(objSchema-F)}$$

$$\frac{\begin{array}{c} J = \{k_1{:}J_1, \ldots, k_n{:}J_n\} \\ \Pi = \{\!|\ (k_i{:}J_i, p_j{:}S_j)\ \mid\ k_i \in L(p_j)\ |\!\} \\ \forall \pi = (k_i{:}J_i, p_j{:}S_j) \in \Pi.\ \ \vdash^{s} J_i\ ?\ S_j \to (r_\pi, \kappa_\pi) \\ \vdash^{k} J\ ?\ \texttt{"patternProperties"} : \{p_1{:}S_1, \ldots, p_m{:}S_m\} \\ \to (\bigwedge_{\pi\in\Pi} r_\pi, \{\!|\ k_i\ |\!\}^{(k_i:J_i, p_j:S_j)\in\Pi}) \end{array}}{}\ \text{(patternProperties)}$$

$$\frac{\begin{array}{c} J = \{k'_1{:}J_1, \ldots, k'_n{:}J_n\} \\ \Pi = \{\!|\ (k'_i{:}J_i, k_j{:}S_j)\ \mid\ k'_i = k_j\ |\!\} \\ \forall \pi = (k'_i{:}J_i, k_j{:}S_j) \in \Pi.\ \ \vdash^{s} J_i\ ?\ S_j \to (r_\pi, \kappa_\pi) \\ \vdash^{k} J\ ?\ \texttt{"properties"} : \{k_1{:}S_1, \ldots, k_m{:}S_m\} \\ \to (\bigwedge_{\pi\in\Pi} r_\pi, \{\!|\ k'_i\ |\!\}^{(k'_i:J_i, k_j:S_j)\in\Pi}) \end{array}}{}\ \text{(properties)}$$

$$\frac{\begin{array}{c} J = \{k_1 : J_1, \ldots, k_n : J_n\} \qquad \vdash^{l} J\ ?\ \vec{K} \to (r, \kappa) \\ \Pi = \{\!|\ (k_i{:}J_i)\ \mid\ 1 \le i \le n \wedge\ k_i \notin \kappa\ |\!\} \\ \forall \pi = (k_i{:}J_i) \in \Pi.\ \ \vdash^{s} J_i\ ?\ S \to (r_\pi, \kappa_\pi) \\ \vdash^{k} J\ ?\ (\vec{K} + \texttt{"unevaluatedProperties"} : S) \\ \to (r \wedge \bigwedge_{\pi\in\Pi} r_\pi, \{\!|\ k_1 \ldots, k_n\ |\!\}) \end{array}}{}\ \text{(unevaluatedProperties)}$$

$$\frac{\begin{array}{c} J = \{k_1 : J_1, \ldots, k_n : J_n\} \qquad \vdash^{l} J\ ?\ \vec{K} \to (r, \kappa) \\ \Pi = \{\!|\ (k_i{:}J_i)\ \mid\ 1 \le i \le n \wedge\ k_i \notin L(\texttt{propsOf}(\vec{K}))\ |\!\} \\ \forall \pi = (k_i{:}J_i) \in \Pi.\ \ \vdash^{s} J_i\ ?\ S \to (r_\pi, \kappa_\pi) \\ \vdash^{k} J\ ?\ (\vec{K} + \texttt{"additionalProperties"} : S) \\ \to (r \wedge \bigwedge_{\pi\in\Pi} r_\pi, \{\!|\ k_1 \ldots, k_n\ |\!\}) \end{array}}{}\ \text{(additionalProperties)}$$

$$\frac{\begin{array}{c} J = [\,J_1, ..., J_n\,] \\ \forall i \in 1..n.\ \ \vdash^{s} J_i\ ?\ S \to (r_i, \kappa_i) \qquad \kappa_c = \{\!|\ i \mid r_i = \mathcal{T}\ |\!\} \end{array}}{\vdash^{k} J\ ?\ \texttt{"contains"} : S \to (|\kappa_c| \ge 1, \kappa_c)}\ \text{(cont)}$$

$$\frac{\begin{array}{c} J = [\,J_1, \ldots, J_m\,] \\ \forall i \in 1..\min(n, m).\ \ \vdash^{s} J_i\ ?\ S_i \to (r_i, \kappa_i) \\ \vdash^{k} J\ ?\ \texttt{"prefixItems"} : [\,S_1, \ldots, S_n\,] \\ \to (\bigwedge_{i\in 1..\min(n,m)} r_i, \{\!|\ 1, \ldots, \min(n, m)\ |\!\}) \end{array}}{}\ \text{(prefixItems)}$$

$$\frac{\begin{array}{c} J = [\,J_1, ..., J_n\,] \qquad \vdash^{l} J\ ?\ \vec{K} \to (r, \kappa) \\ \Pi = \{\!|\ (i, J_i)\ \mid\ 1 \le i \le n \wedge\ i \notin \kappa\ |\!\} \\ \forall \pi = (i, J_i) \in \Pi.\ \ \vdash^{s} J_i\ ?\ S \to (r_\pi, \kappa_\pi) \\ \vdash^{l} J\ ?\ (\vec{K} + \texttt{"unevaluatedItems"} : S) \\ \to (r \wedge \bigwedge_{\pi\in\Pi} r_\pi, \{\!|\ 1 \ldots, n\ |\!\}) \end{array}}{}\ \text{(unItems)}$$

$$\frac{\begin{array}{c} J = [\,J_1, ..., J_n\,] \qquad \vdash^{l} J\ ?\ \vec{K} \to (r, \kappa) \\ \Pi = \{\!|\ (i, J_i)\ \mid\ 1 \le i \le n \wedge\ i \notin 1..prefLenOf(\vec{K})\ |\!\} \\ \forall \pi = (i, J_i) \in \Pi.\ \ \vdash^{s} J_i\ ?\ S \to (r_\pi, \kappa_\pi) \\ \vdash^{l} J\ ?\ (\vec{K} + \texttt{"items"} : S) \\ \to (r \wedge \bigwedge_{\pi\in\Pi} r_\pi, \{\!|\ 1 \ldots, n\ |\!\}) \end{array}}{}\ \text{(items)}$$

Fig. 2. Validation rules.

The resulting language is called Negation-Completed Classical JSON Schema, has the same expressive power as Classical JSON Schema, but it enjoys "negation elimination": every schema in Classical JSON Schema can be rewritten into a schema in Negation-Completed Classical JSON Schema whose size is polynomial in that of the original schema, and which is "positive", that is, it does not contain neither negation nor "oneOf". In the paper, we will use Positive Negation-Completed Classical JSON Schema in the proof of the exponential blow-up for "unevaluatedProperties" elimination (Section 5.2), and in the corresponding proof for "unevaluatedItems".

4. Elimination of "unevaluatedProperties": the problem

In this paper, we prove that for any fixed schema, every instance of "unevaluatedProperties" can be transformed into "additionalProperties". However, this process is far from being trivial, mostly for the interaction between "unevaluatedProperties" and boolean operators like "anyOf" and "allOf".

In Classical JSON Schema, all validation keywords K of the IK category of Fig. 1 distribute with respect to "anyOf" and "allOf", that is, when $K \in IK$[6] and when every keyword $K_i^j \in \vec{K}_i$ belongs to IK, we have[7]

$$\{\,\texttt{"anyOf"} : [\{\,\vec{K}_1\,\},\ldots,\{\,\vec{K}_n\,\}],K\,\} \quad\sim\quad \{\,\texttt{"anyOf"} : [\{\,\vec{K}_1,K\,\},\ldots,\{\,\vec{K}_n,K\,\}]\,\}$$
$$\{\,\texttt{"allOf"} : [\{\,\vec{K}_1\,\},\ldots,\{\,\vec{K}_n\,\}],K\,\} \quad\sim\quad \{\,\texttt{"allOf"} : [\{\,\vec{K}_1,K\,\},\ldots,\{\,\vec{K}_n,K\,\}]\,\}$$

where we use $S_1 \sim S_2$ to indicate that schema S_1 is equivalent to schema S_2, meaning that they validate the same instances. 1.6

This property fails for the keywords in API ("additionalProperties" and "items"), since these keywords depend on which other keywords they are syntactically adjacent to, and distributivity changes the keywords to which K is adjacent. The new "unevaluated*" keywords have been defined to eliminate this syntactic dependency, hence one may hope that they satisfy distributivity. If this were the case, we could eliminate "unevaluatedProperties" by distributing it through the logical operators until it reaches the leaves of a schema, where it could be rewritten as "additionalProperties".

Consider, for example, the following schema:

```
{ "anyOf":  [ { "$ref": "#sale" } , { "$ref": "#car" } ],
  "unevaluatedProperties": false,
  "$defs": {
    "sale": { "$anchor": "sale", "properties": { "price": { "type": "integer" }}},
    "car":  { "$anchor": "car",  "properties": { "plate": { "type": "string" }}}
  }
}
```

The above schema describes *sales* with a *price* and *cars* with a *plate*, and considers their disjunction enriched with "unevaluatedProperties" : false. Unlike the example in the Introduction, *sales* and *cars* are not disjoint, and this detail is crucial.

If we distribute "unevaluatedProperties" through the disjunction, we obtain the following schema (we omit the "$defs" keyword).

```
{ "anyOf":  [ { "$ref": "#sale", "unevaluatedProperties": false },
              { "$ref": "#car",  "unevaluatedProperties": false } ]
}
```

This new schema is more restrictive than the original one. Consider the following instance: {"price":100,"plate":"x111"}. It would be accepted by the original schema: it satisfies both "#car" and "#sale"; its first property is evaluated by the first child of "anyOf", the second property by the second child, hence, in the original schema, it has no unevaluated property, and "unevaluatedProperties": false succeeds. In the rewritten schema, the instance property "plate" causes the "unevaluatedProperties": false in the first "anyOf" branch to fail, since it is not evaluated by the keyword "$ref":"#sale", and the instance property "price" causes the second branch to fail, since it is not evaluated by the keyword "$ref":"#car". Hence, validation of the instance {"price":100,"plate":"x111"} fails. Observe that the new schema is more restrictive than the original one only for the instances that satisfy both branches of the original "anyOf"; hence, the two schemas would be equivalent if these branches were incompatible, as happens with the two branches of the example presented in Section 1.1. This fact is very important and will be discussed again and exploited in Section 6.

To distribute "unevaluatedProperties" through this "anyOf", we must consider the possibility that both branches are satisfied, hence we must distinguish three cases: when only {"$ref":"#sale"} is satisfied, hence only "price" is evaluated (if it is present); when only {"$ref":"#car"} is satisfied, hence only "plate" is evaluated; or when both are satisfied, hence both "price" and "plate" are evaluated. The resulting schema is as follows:

```
{ "anyOf":  [ { "$ref": "#sale", "not": {"$ref": "#car"},
                "unevaluatedProperties": false },
              { "$ref": "#car", "not": {"$ref": "#sale"},
```

[6] Apart from K = "$anchor" : k and K = "$defs" : J, that are not validation keywords

[7] In case K has the same name as a keyword in $\vec{K}_i$, instead of $\{\,\vec{K}_i, K\,\}$ we must write $\{\,\texttt{"allOf"} : [\{\,\vec{K}_i\,\},\{\,K\,\}]\,\}$.

```
                "unevaluatedProperties": false },
              { "allOf": [{"$ref": "#sale"}, {"$ref": "#car"}],
                "unevaluatedProperties": false } ]  }
```

Now, we know exactly which properties are evaluated if present in each of the three cases, and "unevaluatedProperties" can be safely rewritten as "additionalProperties" as follows.

```
{ "anyOf":  [ { "$ref": "#sale", "not": {"$ref": "#car"},
                "properties": {"price":{}},
                "additionalProperties": false },
              { "$ref": "#car", "not": {"$ref": "#sale"},
                "properties": {"plate":{}},
                "additionalProperties": false },
              { "allOf": [{"$ref": "#sale"},  {"$ref": "#car"}],
                "properties": {"plate":{}, "price":{}},
                "additionalProperties": false }  ] }
```

The operator "anyOf" is difficult to deal with since it may be satisfied in many different ways — if it has n different operands, it may be satisfied by the satisfaction of any non-empty subset of them, but each different subset may correspond to a different set of evaluated properties, and we will show with Corollary 2 that this problem cannot be avoided.

Although we will see later that "oneOf" and "allOf" are somehow more tractable than "anyOf", neither of them enjoys distributivity. Consider the operator "oneOf", and the following schema.

```
{ "oneOf":  [ { "$ref": "#sale" } , { "$ref": "#car" } ],
  "unevaluatedProperties": false
}
```

If we distribute "unevaluatedProperties" through the operator, we obtain the following schema.

```
{ "oneOf":  [ { "$ref": "#sale", "unevaluatedProperties": false },
              { "$ref": "#car",  "unevaluatedProperties": false } ]
}
```

Again, the two schemas are not equivalent: this time, the rewritten schema is less restrictive. The object { "price" : 100 } would not be accepted by the original "oneOf" schemas: It satisfies both { "$ref" : "#sale" } and { "$ref" : "#car" }, since no field is mandatory in the two schemas.[8] However, this object would be accepted by the rewritten schema, since it satisfies the first branch, but it violates the second one.

Consider now "allOf", and the following schema.

```
{ "allOf":  [ { "$ref": "#sale" } , { "$ref": "#car" } ],
  "unevaluatedProperties": false
}
```

If we distribute "unevaluatedProperties" through the conjunction, we obtain the following schema.

```
{ "allOf":  [ { "$ref": "#sale", "unevaluatedProperties": false },
              { "$ref": "#car",  "unevaluatedProperties": false } ]
}
```

This new schema is more restrictive than the original one: for example, as happens in the "anyOf" case, both of its branches would refuse the instance { "price" : 100, "plate" : "x111" }, which would instead be accepted by the original schema: It is both a sale and a car, and each of its two fields is evaluated by the conjunction. Hence, the "unevaluatedProperties" of the original schema would not create problems.

These examples show that the elimination of "unevaluatedProperties" is far from immediate. In the next section, we give a formal proof of this fact.

[8] Actually, the original schema accepts no instance at all; this example is not meant to be realistic, but only to illustrate failure of distributivity.

Remark 1. Unlike the other boolean operators, rewriting "unevaluatedProperties" in the presence of negation is very easy. Consider the following schema.

```
{ "not": { "$ref": "#sale" } , "unevaluatedProperties": { "type": "string" } }
```

As we have discussed, the successful execution of "not" : S corresponds to a failure of S, and S will not generate any annotation, so "not" : S will not generate any annotation. Hence, "unevaluatedProperties" : S can be rewritten as "additionalProperties" : S, since they are both applied to every field of the object. Hence, the above schema can be rewritten as follows.

```
{ "not": { "$ref": "#sale" } , "additionalProperties": { "type": "string" } }
```

5. Elimination of "unevaluatedProperties" and "unevaluatedItems" requires an exponential blow-up

5.1. Introduction

In the previous section, we have seen that the elimination of "unevaluatedProperties" is not a trivial task. In this section, we prove that there exist families of schemas written in Static Modern JSON Schema for which the elimination can only be achieved through an exponential blow-up of the schema size, by exhibiting a family of schemas S_n such that, for each schema S_n, the smallest corresponding schema written without using "unevaluatedProperties" has a size in $O(2^{|S_n|})$. We then prove that the same property holds for "unevaluatedItems", even if we add the new operators "minContains" and "maxContains" to the target language.

5.2. Exponentially of elimination of "unevaluatedProperties"

Consider the family of schemas where S_n is defined as follows.

```
{ "anyOf": [ { "required": [ "a1" ], "patternProperties": { "a1": true } },
             { "required": [ "a2" ], "patternProperties": { "a2": true } },
               ...
             { "required": [ "an" ], "patternProperties": { "an": true } }],
  "unevaluatedProperties": false
}
```

Recall that a pattern "a1" matches any strings that includes "a1", such as "a1" itself, or "Ba1Ca2K". The following property characterizes the instances that satisfy S_n.

Property 1. *An instance J satisfies S_n if, and only if, it is not an object, or it is an object and:*

1. *J contains at least one property whose name is exactly "a"$\cdot i$, for some $i \leq n$;*
2. *for any property of J, the property name matches one or more patterns "a"$\cdot i$, with $i \leq n$;*
3. *for each property $k : J_k$ of J whose name k matches a set of "a"$\cdot i$ patterns, for at least one of the "a"$\cdot i$ matched by k, there is a property of J whose name is exactly "a"$\cdot i$.*

To understand this characterization, let us observe that the i-th branch of the "anyOf" evaluates a field k iff the branch is successful, i.e. iff J has a field named exactly "a"$\cdot i$, and if k matches "a"$\cdot i$. Hence, condition (1) expresses the fact that "anyOf" needs at least one branch to succeed. Conditions (2) and (3) express the fact that any field k must be evaluated, otherwise the "unevaluatedProperties" keyword will fail, and a field is evaluated by the i-th branch iff conditions (2) and (3) hold for that field. For example, { "a1" : null, "-a1-a3-" : null } satisfies the schema, since "required" : "a1" is satisfied, hence the first branch of "anyOf" evaluates the "-a1-a3-" property; on the contrary, { "a2" : null, "-a1-a3-" : null } does not satisfy the schema, since no successful branch evaluates the property "-a1-a3-".

Before proving the exponentiality of "unevaluatedProperties" elimination for this family of schemas, we introduce a definition and a lemma. Before stating the definition, we recall that we use the term *in-place keywords* for "allOf" : A,"anyOf" : A, "oneOf" : A, "not" : S, and "$ref" : u, and *structural keywords* for any other keyword.

We use *occurrence of a subschema* of S to indicate a path in the syntax tree of S that leads to a schema S', and we say that J satisfies that occurrence when J satisfies that schema S'. We use *occurrence of a subkeyword* of S to indicate a path in the syntax tree of S that leads to a keyword K. We say that J satisfies that occurrence when J satisfies that keyword K, keeping into account the keywords that are adjacent to that occurrence of K when K is either "additionalProperties" : S or "items" : S. For example, an occurrence of "additionalProperties" : false, when adjacent to "patternProperties" : { "a" : true }, is satisfied by all, and only, the objects where every field name matches "a".

Definition 4. For any schema S and for any instance J, we use $\mathcal{S}_{(S,J)}$ to denote all occurrences of subschemas of S that are satisfied by J, $\mathcal{K}_{(S,J)}$ to denote all occurrences of subkeywords of S that are satisfied by J, and $\mathcal{SK}_{(S,J)}$ to denote all occurrences of structural subkeywords of S that are satisfied by J, so that $\mathcal{SK}_{(S,J)} \subseteq \mathcal{K}_{(S,J)}$.

Lemma 1 (Monotonicity)**.** *For any positive schema S in Negation-Completed Classical JSON Schema and for any pair of instances J_1 and J_2, $\mathcal{SK}_{(S,J_1)} \subseteq \mathcal{SK}_{(S,J_2)}$ implies that $\mathcal{K}_{(S,J_1)} \subseteq \mathcal{K}_{(S,J_2)}$ and that $\mathcal{S}_{(S,J_1)} \subseteq \mathcal{S}_{(S,J_2)}$.*

Proof. We want to prove that (1) $\mathcal{SK}_{(S,J_1)} \subseteq \mathcal{SK}_{(S,J_2)}$ and $\vdash^{\mathcal{S}} J_1 \; ? \; S'$ implies $\vdash^{\mathcal{S}} J_2 \; ? \; S'$ and (2) $\mathcal{SK}_{(S,J_1)} \subseteq \mathcal{SK}_{(S,J_2)}$ and $\vdash^{\mathcal{K}} J_1 \; ? \; K$ implies $\vdash^{\mathcal{K}} J_2 \; ? \; K$ for all subschemas S' and subkeywords K of S, by mutual induction on the in-place depth (Definition 3). (1) is immediate for $S = \texttt{true/false}$, and is proved by induction on for $S = \{ K_1, \ldots, K_n \}$. (2) is proved by induction when K is boolean or is "$ref" : u, and follows from $\mathcal{SK}_{(S,J_1)} \subseteq \mathcal{SK}_{(S,J_2)}$ otherwise. □

Corollary 1. *For any positive schema S in Negation-Completed Classical JSON Schema and for any pair of instances J_1 and J_2, if J_1 satisfies S, J_2 does not satisfy S, and $\mathcal{SK}_{(S,J_2)} \supseteq (\mathcal{SK}_{(S,J_1)} \setminus \mathcal{A}^-)$, then $\mathcal{A}^-$ is not empty.*

We now use Corollary 1 to prove the exponentiality result.

Theorem 1. *For any schema S_n of the above family, any equivalent schema S that is expressed using Positive Negation-Completed Classical JSON Schema as defined in Section 3.3, has a dimension that is greater than 2^n.*

Proof.

Consider any n and any schema S, written in Positive Negation-Completed Classical JSON Schema, such that $S \sim S_n$. Consider all the instances of keywords inside S, at any level, that are either "properties" : O, "patternProperties" : O, "additionalProperties" : S', or "propertyNames" : S' — we call them the *prop-keywords*. In our proof, we will only use objects where every property has a null value. For this reason, we define, for each prop-keyword PK, its *accepted language* $L(PK)$ as the set of property names k such that an object $\{ k : \texttt{null} \}$ satisfies PK. By the validation rules that we presented, an object $\{ k_1 : \texttt{null}, \ldots, k_n : \texttt{null} \}$ is accepted a prop-keywords PK if every k_i belongs to the *accepted language* of PK.

For an example, consider the following schema with two prop-keywords:

$$\{ PK_1, PK_2 \} = \{ \texttt{"patternProperties"} : \{ \texttt{"a1"}:\{\}, \texttt{"\^{}a2"}:\{\} \}, \texttt{"additionalProperties"} : \texttt{false} \}$$

Since PK_1 never fails, its accepted language is Σ^*: $L(PK_1) = \Sigma^*$. PK_2 *fails* over $\{ k : \texttt{null} \}$ if, and only if, k is not evaluated by PK_1, that is, iff $k \notin L(\texttt{"a1|\^{}a2"})$, hence the *accepted* language of PK_2 is $L(\texttt{"a1|\^{}a2"})$, and an object $\{ k_1 : \texttt{null}, \ldots, k_n : \texttt{null} \}$ satisfies PK_2 iff every k_i matches "a1|^a2".

For each prop-keyword PK we say that its *pattern signature* is the set of all patterns "a"$\cdot i$ in {|"a1", ..., "an"|} such that $L(\texttt{"a"}\cdot i) \subseteq L(PK)$; for example, if $L(PK) = L(\texttt{"a1|\^{}a2"})$, then its *pattern signature* is {|"a1"|}; the pattern "a2" is not in the signature, since $L(\texttt{"a2"})$ is not completely contained in $L(\texttt{"a1|\^{}a2"})$ (L("a2") contains strings such as "ZZa2", which do not belong to $L(\texttt{"a1|\^{}a2"})$).

Finally, we collect all subkeywords of S with shape "maxProperties" : m or "const" : J_o, and we use BIG to denote a number that is greater than each m parameter and than the length $|J_o|$ of each J_o parameter.

We now show that, for any non-empty subset A of {|"a"$\cdot 1, \ldots,$ "a"$\cdot n$|}, there exists one prop-keyword of S that has exactly A as its pattern signature. For any such $A = \{|\texttt{"a"}\cdot i_1, \ldots, \texttt{"a"}\cdot i_l|\}$, we consider an object J_A that contains the properties {|"a"$\cdot i_1, \ldots,$ "a"$\cdot i_l$|} (with a null value); then, for each prop-keyword PK of S such that there exists "a"$\cdot i_j$ in A that is not in the pattern signature of PK, we add to J_A a null property $PNot(j, PK)$ whose name belongs to $L(\texttt{"a"}\cdot i_j) \setminus L(PK)$; finally, we also add a set of BIG extra null properties $PExtra(1)$,..., $PExtra(BIG)$, where the name of $PExtra(q)$ is obtained by concatenating $q+1$ copies of "a"$\cdot i_1$; we use J_A^+ do denote the result.

By construction, the object J_A^+ satisfies S_n: Consider Property 1; (1): the object J_A^+ contains one property "a"$\cdot i$ (since A is not empty); (2 and 3): these conditions are trivially verified for all properties of J_A, since their names are in A; for every additional property $PNot(j, PK)$ of J_A^+, the property matches by construction the pattern "a"$\cdot i_j$ and "a"$\cdot i_j$ is a property of J_A, hence of J_A^+; finally, every property $PExtra(q)$ matches the pattern "a"$\cdot i_1$, which also belongs to J_A^+.

We consider now the set $\mathcal{SK}_{(S,J_A^+)}$ of all structural keywords inside S (prop-keywords and any other keywords that are not "$ref" or boolean) that are satisfied by J_A^+ to show that some of them are prop-keywords whose pattern signature is exactly A. Observe that the pattern signature of each prop-keyword in $\mathcal{SK}_{(S,J_A^+)}$ is a superset of A: if the pattern signature of a prop-keyword PK' is not a superset of A, then the object J_A^+ contains, by construction, a property $PNot(j, PK')$ that is not in the accepted language of PK', hence such a keyword PK' is not satisfied by J_A^+, hence this PK' cannot be in $\mathcal{SK}_{(S,J_A^+)}$; we use (*) to refer to this fact later on.

We now consider the set $\mathcal{PK}^A$ of those prop-keywords that belong to $\mathcal{SK}_{(S,J_A^+)}$ and whose pattern signature is exactly A, with the aim of proving that $\mathcal{PK}^A$ it is not empty. To this aim, we build an extension J_A^{++} of J_A^+ that satisfies all keywords in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$ but does not satisfy S, as follows.

We consider an enumeration {|"a"$\cdot c_1, \ldots,$ "a"$\cdot c_o$|} of the complement of A, defined as $Co(A) = \{|\texttt{"a"}\cdot 1, \ldots, \texttt{"a"}\cdot n|\} \setminus A$. We build J_A^{++} from J_A^+ by adding a property with name $k^{++} = \texttt{"a"}\cdot c_1 \cdot \ldots \cdot \texttt{"a"}\cdot c_o \cdot \texttt{"\_"}$ (the character "_" at the end of k^{++} is used to handle the case in which $Co(A)$ is a singleton or is empty). The object J_A^{++} violates either condition (2) or condition (3) of Property 1: if $o = 0$, then $k^{++} = \texttt{"\_"}$, which violates condition (2); if $o \geq 1$, then k^{++} matches the patterns of $Co(A)$, but it does not match any of the patterns in A, which are the only properties in J_A^{++} whose name is exactly "a"$\cdot i$ for some i, hence k^{++} violates condition (3), hence it does not satisfy S_n.

We prove now that J_A^{++} satisfies all keywords PK in the set $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$. If PK is a prop-keyword, then, by (*), its pattern signature is a superset of A; since $PK \notin \mathcal{PK}^A$, then its pattern signature is a *strict* superset of A, hence it includes at least one pattern "a" $\cdot\, c_l$ of $Co(A)$; by construction, the new property k^{++} of J_A^{++} matches all patterns in $Co(A)$, hence it matches "a" $\cdot\, c_l$, hence the new property is in $L(PK)$, hence it is accepted by PK; since all other properties of J_A^{++} belong to J_A^+ hence are accepted by PK, then J_A^{++} satisfies the prop-keyword PK. For the structural keywords in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$ that are not prop-keywords, we reason as follows. Since J_A^{++} has more properties than J_A^+, then it satisfies all the "minProperties" : n keywords in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$. Since J_A^{++} contains all properties of J_A^+, then J_A^{++} satisfies all "required" : […] and "patternRequired" : { … } that are in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$. No keyword in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$ is a "maxProperties" : m keyword or a "const" : J_o keyword, since J_A^+ has strictly more than BIG properties. No other structural keyword is able to distinguish one object from another, hence all other keywords in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$ are satisfied by J_A^{++} as well.

Since S is a positive schema, since J_A^+ satisfies S and J_A^{++} does not, since J_A^{++} satisfies all keywords in $\mathcal{SK}_{(S,J_A^+)} \setminus \mathcal{PK}^A$, we can conclude by Corollary 1 that $\mathcal{PK}^A$ is not empty, that is, that there exists at least one occurrence of a subkeyword of S whose pattern signature is exactly A. Hence, we have proved that for every non-empty subset A of {| "a1", … , "an" |} there is one subkeyword of S that has A as its pattern signature. Since {| "a1", … , "an" |} has $2^n - 1$ different non-empty subsets, this implies that S has at least $2^n - 1$ subkeywords.

□

We have proved that S_n needs at least $2^n - 1$ keywords in order to be represented by a positive Negation-Completed Classical JSON Schema schema. The exponentiality result follows for general Classical JSON Schema since it is has been proved by Baazizi et al. [15] that negation can be eliminated from Classical JSON Schema with a polynomial expansion.

Corollary 2. *There exists a family of Static Modern JSON Schema schemas S_n whose only Modern JSON Schema operator is one instance of* "unevaluatedProperties"*, such that, for each family of schemas S'_n expressed in Classical JSON Schema, if each S'_i is equivalent to S_i, then S'_n has a size in $\Omega(2^n)$.*

5.3. Exponentiality of elimination of "unevaluatedItems"

We now present a family S_n^A of array schemas that has the same property of the S_n family of object schemas defined in the previous section. As a technical tool, we also define, for each n, the family $\mathcal{T}_n = \{|\, \text{"T"}\cdot 1, \dots, \text{"T"}\cdot n \,|\}$ of object schemas, with the property that, for any set $\mathcal{T}' = \{|\, \text{"T"}\cdot i_1, \dots, \text{"T"}\cdot i_l \,|\} \subseteq \mathcal{T}_n$, the object { "a"$\cdot i_1$: null, … , "a"$\cdot i_l$: null } satisfies all schemas in $\mathcal{T}'$ and no schema in $\mathcal{T}_n \setminus \mathcal{T}'$.

Definition 5 (S_n^A, "T"$\cdot i$)**.** The schema S_n^A is defined as follows; we will also use "T"$\cdot i$ to denote the schema with the anchor "T"$\cdot i$.

```
{ "anyOf": [
    { "prefixItems": [{"$ref":"#T1"}], "minItems":1, "contains":{"$ref" "#T1"} },
    { "prefixItems": [{"$ref":"#T2"}], "minItems":1, "contains":{"$ref":"#T2"} },
    ...
    { "prefixItems": [{"$ref":"#Tn"}], "minItems":1, "contains":{"$ref":"#Tn"} }
  ],
  "unevaluatedItems": false,
  "$defs": { "T1": { "$anchor": "T1", "required": [ "a1" ] },
             "T2": { "$anchor": "T2", "required": [ "a2" ] },
             ...
             "Tn": { "$anchor": "Tn", "required": [ "an" ] } }
}
```

Each "anyOf" branch requires that the first element of the array satisfies the schema "T" $\cdot\, i$, and, when this condition holds, the branch evaluates all array elements that satisfy "T"$\cdot i$; observe that when "prefixItems" and "minItems" are both satisfied, then "contains" holds as a consequence, but its presence ensures that all other items in the array that satisfy "T" $\cdot\, i$ are also evaluated, so that "unevaluatedItems" : false does not apply to them. In summary, an array satisfies this schema if, and only if:

1. its first element exists and it satisfies at least one of the "T"$\cdot i$ schemas;
2. every other element satisfies at least one of the "T"$\cdot i$ schemas that are satisfied by the first item.

We now prove that the encoding of this family of schemas requires an exponential blow-up.

Remark 2. This family resembles the one that we used for objects, but there are some important differences, especially in the fact that "properties" evaluates all fields whose name matches a given string, independently of their value, while "contains" evaluates the items whose value matches a given schema, independently from their position. The object keywords "properties", "additionalProperties", and "unevaluatedProperties" are all uniformly defined in terms of property names. On the other side, the array keywords "prefixItems", "items" and "unevaluatedItems" are defined in terms of item positions, but "contains"

depends on item content. For this reason, one may expect that "unevaluatedProperties" and "unevaluatedItems" should enjoy different properties with respect to the elimination of "unevaluated*" keywords, but this is not really the case.

Theorem 2. *For any schema $S_n^{\mathcal{A}}$ of the above family, any equivalent schema S that is expressed using Positive Negation-Completed Classical JSON Schema, has a dimension that is bigger than 2^n.*

Proof. Consider any schema S written in Positive Negation-Completed Classical JSON Schema and equivalent to $S_n^{\mathcal{A}}$. Consider the set $\mathcal{T}_n = \{\!|\, \texttt{"T"}\cdot 1, \ldots, \texttt{"T"}\cdot n \,|\!\}$ of the object schemas that appear inside $S_n^{\mathcal{A}}$. For each keyword ik with shape "items" : S' (an *items-keyword*), we say that its *tail signature* TS_{ik} is the set of all schemas "T"$\cdot i$ in $\mathcal{T}_n$ such that S' is satisfied by every element of "T"$\cdot i$.[9] For example, the tail signature of an items-keyword "items" : { "anyOf" : ["T1","T3"] } is $\{\!|$ "T1","T3" $|\!\}$.

We now show that, for any non-empty subset A of $\mathcal{T}_n$, there exists one items-keyword ik somewhere inside S that has A as its tail signature. Assume, towards a contradiction, that there exists a subset A of $\mathcal{T}_n$ such that no items-keyword inside S has A as its tail signature. We fix an integer BIG that is strictly bigger than the maximum m such that "maxItems" : m appears in S, is strictly bigger than the maximum length of the argument J_A of each subkeyword of S that is "prefixItems" : J_A or "const" : J_A, and is strictly bigger than 1. Fixed a non-empty set $A = \{\!|\, \texttt{"T"}\cdot i_1, \ldots, \texttt{"T"}\cdot i_l \,|\!\} \subseteq \mathcal{T}_n$ such that no items-keyword has A as its tail signature, we choose two different objects, J_1 and J_2, whose properties are exactly "a"$\cdot i_1, \ldots,$ "a"$\cdot i_l$, so that both J_1 and J_2 satisfy all the schemas in A and no schema in $\mathcal{T}_n \setminus A$. We build an array J of length at least BIG as follows:

- its first item is J_1;
- its next BIG-1 items, which exist since $BIG > 1$, are equal to J_2, so that this array violates every "uniqueItems" : true keyword inside S and, having at least BIG items, also violates every "maxItems" : m and "const" : J keyword inside S;
- then, for every items-keyword ik with keyword "items" : S_{ik} whose tail signature TS_{ik} does not satisfy $A \subseteq TS_{ik}$, we choose a schema "T"$\cdot i_{ik}$ in $A \setminus TS_{ik}$ and an element J_{ik} of "T"$\cdot i_{ik}$ that does not satisfy S_{ik}, which exists since "T"$\cdot i_{ik}$ is not in the tail signature TS_{ik}, and we add J_{ik} to the array J; this ensures that J does not satisfy "items" : S_{ik}, so that every items-keyword that belongs to $S\mathcal{K}_{(S,J)}$ (i.e., that validates J) has a tail signature TS_{ik} such that $A \subseteq TS_{ik}$; since we assumed that no ik has $TS_{ik} = A$, we conclude that $ik \in S\mathcal{K}_{(S,J)} \Rightarrow TS_{ik} \supset A$.

The array J satisfies $S_n^{\mathcal{A}}$, since its first item satisfies all "T"$\cdot i$ that belongs to A and all its items (J_1, J_2, and all the different J_{ik}'s) satisfy some "T"$\cdot i \in A$, hence all its items are evaluated by a successful branch of $S_n^{\mathcal{A}}$, hence none is passed to "unevaluatedItems" : false. Now, towards a contradiction, we build an array J^+ that satisfies S but does not satisfy $S_n^{\mathcal{A}}$. To this aim, we consider an enumeration $\{\!|\, \texttt{"T"}\cdot j_1, \ldots, \texttt{"T"}\cdot j_p \,|\!\}$ of the complement of A, $Co(A) = \mathcal{T}_n \setminus A$, we let J_a be an object whose field names are exactly $\{\!|\, \texttt{"a"}\cdot j_1, \ldots, \texttt{"a"}\cdot j_p \,|\!\}$, and we observe that J_a satisfies all schemas in $Co(A)$ and no schema of A. We build the array J^+ by adding J_a after the last element of the array J.

J^+ satisfies all keywords in $S\mathcal{K}_{(S,J)}$ because:

- by point (2), no satisfied keyword is either a "maxItems" : m or a "uniqueItems" : true keyword or a "const" : A keyword;
- keywords "contains" : S, "containsAfter" : S, "minItems" : m, and "repeatedItems" : b that were already satisfied by J cannot be invalidated by the addition of one element;
- a keyword "prefixItems" : A satisfied by J cannot be violated by J^+ since J is longer than BIG that is bigger than the argument of any "prefixItems" : A in S, hence the added element is not in the prefix;
- the schema S' of an item-keyword ik = "items" : S' is satisfied by all items of J^+ that belong to J because $ik \in S\mathcal{K}_{(S,J)}$, and S' is also satisfied by J_a because, by (3) above, we have $TS_{ik} \supset A$, hence TS_{ik} contains at least one schema of "T"$\cdot j_p \in Co(A)$, and J_a satisfies that schema by construction, hence J_a satisfies S', hence J^+ satisfies "items" : S';
- any other structural keyword consider any two arrays to be equivalent.

Hence, J satisfies S and J^+ satisfies all keywords of $S\mathcal{K}_{(S,J)}$ hence, since S is written in negation-free JSON Schema, then J^+ satisfies S. However, J^+ does not satisfy $S_n^{\mathcal{A}}$: since the first item of J^+ is J_1, then J^+ satisfies all and only the "anyOf" branches that correspond to the elements of A; since the item J_a satisfies no schema of A, then it is not evaluated by any successful branch of the "anyOf" keyword, hence it causes "unevaluatedItems" to fail; this contradicts the equivalence between S and $S_n^{\mathcal{A}}$. Hence, we have proved that, for any A non-empty subset of $\mathcal{T}_n$, we must have in S at least one items-keyword whose tail signature is A. Since $\mathcal{T}_n$ has $2^n - 1$ different non-empty subsets, this implies that S has at least $2^n - 1$ different keywords. □

Corollary 3. *There exists a family of Static Modern JSON Schema schemas $S_n^{\mathcal{A}}$ whose only Modern JSON Schema operator is one instance of* "unevaluatedItems"*, such that, for each family of schemas S'_n expressed in Classical JSON Schema, if each S'_i is equivalent to S_i, then S'_n has a size in $\Omega(2^n)$.*

5.4. Adding "minContains" and "maxContains"

Draft 2019-09 of JSON Schema introduced the operators "minContains" and "maxContains". When "minContains" : m is adjacent to a "contains" : S keyword, it forces the presence of at least m items that satisfy S; when "maxContains" : q is adjacent to a "contains" : S keyword, it limits the number of items that satisfy S to be less than q, as expressed by the following rule.

[9] We call it *tail* signature since, in case "items" : S' is adjacent to a "prefixItems" : $[\, S_1, \ldots, S_n \,]$ keyword, then "items" : S' only affects the elements in the tail of an instance, that is, those whose position is greater than $n + 1$; differently from the pattern signature of the previous proof, the tail signature of "items" : S' does not depend on the adjacent keywords.

$$\frac{J = [\,J_1, \ldots, J_n\,] \qquad \forall i \in 1..n.\ \vdash^{S}\ J_i\ ?\ S \to (r_i, \kappa_i) \qquad \kappa_c = \{\!|\ i \mid r_i = \mathcal{T}\ |\!\}}{\vdash^{K}\ J\ ?\ \texttt{"contains"} : S, \texttt{"minContains"} : m, \texttt{"maxContains"} : q \to (m \leq |\kappa_c| \leq q, \kappa_c)} \quad \text{(contains)}$$

The addition of these operators to Classical JSON Schema has already been studied: they are considered in the algorithm to verify satisfiability and inclusion presented by Attouche et al. [9], and have been included by Baazizi et al. [15] in Negation-Completed Classical JSON Schema.

Since these operators add a counting ability to JSON Schema, it is natural to ask whether the exponentiality result would still hold if we added "minContains" : m and "maxContains" : q to the target language. We prove here that this is still the case — these operators do not help making the translation more compact.

Modern JSON Schema allows any combination of "contains" : S, "minContains" : m, and "maxContains" : q keywords in the same schema and gives a meaning to each combination. If we allow the ∞ value for the q parameter of "maxContains" : q, the meaning of any combination where some operator is missing can be defined by the following normalization procedure: if "contains" : S is missing, then the two other keywords are deleted; otherwise, a missing "minContains" : m can be inserted as "minContains" : 1, and a missing "maxContains" : q can be inserted as "maxContains" : ∞. In our proof, we assume that every schema is normalized in this way, so that we consider the triple "contains" : S, "minContains" : m, "maxContains" : q to be a single keyword, defined by the above rule.

Theorem 3. *For any schema $S_n^{\mathcal{A}}$ of Definition 5, if S is equivalent to $S_n^{\mathcal{A}}$ and is expressed using Positive Negation-Completed Classical JSON Schema enriched with the* "minContains" *and* "maxContains" *keywords, then S has a dimension that is bigger than 2^n.*

Proof. Consider any schema written in Positive Negation-Completed Classical JSON Schema enriched with the "minContains" and "maxContains" keywords, and equivalent to $S_n^{\mathcal{A}}$. Assume that any "maxContains" : q keyword of S has $q \geq 1$; this can be obtained by rewriting every triple "contains" : S', "minContains" : 0, "maxContains" : 0 as "items" : { "not" : S' } and then performing not-elimination, while triples with "minContains" : n, "maxContains" : 0, and $n \neq 0$ can be rewritten as false. For this schema, we repeat the same construction as in the proof of Property 2, but this time we choose BIG so that it is also strictly bigger than any q such that "maxContains" : q appears in S.

Once we have fixed a non-empty $A = \{\!|\ \texttt{"T"}\cdot i_1, \ldots, \texttt{"T"}\cdot i_l\ |\!\} \subseteq \mathcal{T}_n$ such that no items-keyword has A as its tail signature, we choose, as in the previous proof, two different objects, J_1 and J_2, whose properties are exactly $\texttt{"a"}\cdot i_1, \ldots, \texttt{"a"}\cdot i_l$, so that both J_1 and J_2 satisfy all the schemas in A and no schema in $\mathcal{T}_n \setminus A$. Then, as in the previous proof, for every items-keyword ik with keyword "items" : S_{ik} whose tail signature TS_{ik} does not satisfy $A \subseteq TS_{ik}$, we choose a schema $\texttt{"T"}\cdot i_{ik}$ in $A \setminus TS_{ik}$ and an element J_{ik} of $\texttt{"T"}\cdot i_{ik}$ that does not satisfy S_{ik}, which exists since $\texttt{"T"}\cdot i_{ik}$ is not in TS_{ik}. At this point, we build an array J that contains BIG copies of J_1, followed by BIG copies of J_2, and, for each ik whose tail signature TS_{ik} does not satisfy $A \subseteq TS_{ik}$, we add BIG copies of the item J_{ik} that ensures that $ik \notin \mathcal{SK}_{(S,J)}$. Hence, as in the previous proof, we have that J satisfies $S_n^{\mathcal{A}}$, and that every items-keyword in $\mathcal{SK}_{(S,J)}$ satisfies $TS_{ik} \supset A$.

We now build the array J^+ by adding an element J_a at the end of J as in the previous proof and we prove that J^+ satisfies all keywords in $\mathcal{SK}_{(S,J)}$. For all keywords in $\mathcal{SK}_{(S,J)}$ that are different from "contains" : S_C, "minContains" : m, "maxContains" : q, we reason as in the previous proof.

For every keyword "contains" : S_C, "minContains" : m, "maxContains" : q, since J satisfies the "minContains" : m requirement, then J^+, which contains all items of J plus one more, also satisfies "minContains" : m as well.

For the "maxContains" : q keyword, if $q = \infty$, then it is trivially satisfied. If q is finite, it is strictly smaller then BIG, hence none of the J_1, J_2, and J_{ik} elements of J satisfies S_C, since all of them are repeated at least BIG times and $BIG > q$, hence, if they satisfied S_C, they would violate "maxContains" : q. Hence, the only item in J^+ that may satisfy S_C is J_a but, since $q \geq 1$, a single item is not sufficient to violate "maxContains" : q; hence, J^+ satisfies "maxContains" : q.

As in the previous proof, we have proved that J^+ satisfies every keyword in $\mathcal{SK}_{(S,J)}$ but it does not satisfy $S_n^{\mathcal{A}}$, hence we have a contradiction, hence we have proved that for every A there is a keyword inside S whose tail signature is A, hence we have an exponential number of keywords. As a conclusion, the exponentiality proof holds even if S is allowed to use "minContains" and "maxContains". □

As proved by Baazizi et al. [15], negation can be eliminated from Classical JSON Schema with a polynomial size expansion even in the presence of "minContains" and "maxContains", hence we have the following corollary.

Corollary 4. *There exists a family of Static Modern JSON Schema schemas $S_n^{\mathcal{A}}$ whose only Modern JSON Schema operator is one instance of* "unevaluatedItems"*, such that, for each family of schemas S'_n expressed in Classical JSON Schema, enriched with the* "minContains" *and* "maxContains" *keywords, if each S'_i is equivalent to S_i, then S'_n has a size in $\Omega(2^n)$.*

6. Eliminating unevaluatedProperties using normalization

6.1. Introduction

We have shown in Section 4 that, in general, it is not sound to just push unevaluated $*$ keywords through a logical operator, and in Section 5 we have proved that, in general, the elimination of unevaluated $*$ keywords may require an exponential increase of the schema size.

In this section, we first introduce a baseline technique to eliminate unevaluated $*$ keywords, based on *eXclusive Disjunctive Normal Form* (XDNF), which is simple and general, but not practical. We then use the XDNF intuition to introduce our approach, based on the

notions of *statically characterized schemas*, *cover closure*, and *Evaluation Normal Form* (ENF), which yields schemas that can be much smaller than the baseline.

6.2. *Pushing* `unevaluatedProperties` *through eXclusive DNF*

A schema S is in *DNF* (*Disjunctive Normal Form*) when $S = \{\ \texttt{"anyOf"} : [S_1, \ldots, S_n]\ \}$, and each S_i combines structural keywords K^i_j using conjunctive operators: $S_i = \{\ K^i_1, \ldots, K^i_{m_i}\ \}$ or $S_i = \{\ \texttt{"allOf"} : [\{\ K^i_1\ \}, \ldots, \{\ K^i_{m_i}\ \}]\ \}$.

We have seen in Section 4 that `"unevaluatedProperties"` cannot be pushed through $S = \{\ \texttt{"anyOf"} : [S_i, \ldots, S_n]\ \}$ in general, but this is possible when S is in *eXclusive DNF* (*XDNF*), that is, it is in DNF and all conjunctions S_i are mutually exclusive.

Consider, for example, the following schema that has a core definition `"#/$defs/extensible"` that is extended in different ways; the core definition is "extensible", but, after the extension, the result is closed by adding `"unevaluatedProperties" : false`; this is a common use case of this operator.

```
{ "anyOf": [
    { "$ref": "#/$defs/extensible",
      "properties": { "kind": {"const": "A"}, "address": {"type": "string"} },
      "required": ["kind"] },
    { "$ref": "#/$defs/extensible",
      "properties": { "kind": {"const": "M"}, "model": {"type": "string"} },
      "required": ["kind"] }
  ],
  "unevaluatedProperties": false,
  "$defs": {
       "extensible": { "type": "object",  "properties": { "name": "string"} }
  }
}
```

The first branch is satisfied by objects having a mandatory `"kind"` property with a constant value `"A"` as well as optional properties `"address"` and `"name"` of type string, but it does not impose constraints on other properties; on the other hand, the second branch is satisfied by objects having a mandatory `"kind"` property with a constant value `"M"` as well as optional properties `"model"` and `"name"` of type string, but, as for the previous branch, it does not impose further constraints on other properties.

Since the two branches of `"anyOf"` are mutually exclusive, because of the disjoint values allowed for property `"kind"`, then, for each instance J, when one branch is successful, the other branch fails, hence it does not evaluate any property; hence, `"unevaluatedProperties"` : S_u can be pushed through `"anyOf"`.

Now we observe that we can statically compute the properties $p_1, \ldots, p_n$ that are evaluated by each branch,
and rewrite it as

$$\{K^i_1, \ldots, K^i_{m_i}, \texttt{"allOf"} : [\{\ \texttt{"properties"} : \{\ p_1 : \{\ \}, \ldots, p_n : \{\ \}\ \}, \texttt{"additionalProperties"} : S_u\ \}]\}$$

where the "useless" unary `"allOf"` is necessary since one of the K^i_j is `"properties" : { }`, and we cannot have two different `"properties"` adjacent in the same schema.[10]

Hence, the above schema can be rewritten as follows.

```
{ "anyOf": [
    { "$ref": "#/$defs/extensible",
      "properties": { "kind": {"const": "A"}, "address": {"type": "string"} },
      "required": ["kind"],
      "allOf": [ { "properties": { "name": {}, "kind": {}, "address": {} } ,
                   "additionalProperties": false } ]
    },
    { "$ref": "#/$defs/extensible",
      "properties": { "kind": {"const": "M"}, "model": {"type": "string"} },
      "required": ["kind"],
      "allOf": [ { "properties": { "name": {}, "kind": {}, "model": {} },
                   "additionalProperties": false } ]
    }
```

[10] We could of course merge the two keywords into one, but we use `"allOf"` to keep the approach simpler.

```
  ],
  "$defs":$ { "extensible": { "type": "object",
                              "properties": { "name": {"type": "string"} } }
  }
}
```

This method can be always applied: we can rewrite every schema as an XDNF schema, by first rewriting the schema in DNF, and then by substituting the n arguments of the disjunction "anyOf" : $[S_1, \dots, S_n]$ with $O(2^n)$ mutually exclusive conjunctions: for each non-empty subset S of $\{\!|\, S_1, \dots, S_n \,|\!\}$, we consider the conjunction where all elements of S are required to hold and all those in the complement of S are negated, so that every possible combination of validity/non-validity of the schemas in $\{\!|\, S_1, \dots, S_n \,|\!\}$ that validates "anyOf" : $[S_1, \dots, S_n]$ is satisfied by exactly one of these mutually exclusive conjunctions. Since the result is in XDNF, we can push "unevaluatedProperties" through the "anyOf" inside each branch. Since each branch of an XDNF is a conjunction of structural operators, we can always statically compute which properties are evaluated by that branch, and hence substitute "unevaluatedProperties" with "additionalProperties".

It is easy to prove that the size of this translation is bounded by $O(2^N)$, hence it is asymptotically "optimal", according to the result of Corollary 2. However, the $O(2^N)$ lower bound of Corollary 2 is a worst-case scenario that one hopes to avoid in most practical cases, while this XDNF approach generates an exponential explosion from any disjunction.

A second problem of this approach is the fact that reduction to DNF requires not-elimination, which is not trivial in Modern JSON Schema: while operators such as "allOf" and "anyOf", or "minProperties" and "maxProperties", are each the De Morgan dual of the other; other keywords, such as "patternProperties", do not have a De Morgan dual. This issue can be managed through the techniques introduced by Baazizi et al. [15] for Classical JSON Schema, but the problem in Modern JSON Schema is a bit more complex, since we need to define a negation dual for "unevaluatedProperties" and a negation dual for "not", as discussed in Remark 3

We present here an algorithm that overcomes these two problems: it generalizes the condition of "exclusiveness" to a weaker condition of "cover-closure" that takes advantage from a static characterization of the properties evaluated by each subschema; differently from DNF and XDNF, the cover-closure condition can be reached without any kind of not-elimination, and, in many practical cases, adding a limited number of extra cases to the original disjunction.

Our experiments, described in Section 7, show that this algorithm behaves very well on real-world schemas.

Remark 3. In Classical JSON Schema, "not" is self-dual, which means that { "not" : { "not" : S } } can always be rewritten as S. Unfortunately, this is not the case in Modern JSON Schema, since { "not" : { "not" : S } } yields the same boolean result as S, but returns no annotation; this fact complicates the not-elimination of the "not" operator.

Consider now "unevaluatedProperties" : S; it fails when there exists an unevaluated property that satisfies { "not" : S }, hence, in order to rewrite "not" : { "unevaluatedProperties" : S }, we need an operator "unevaluatedRequired" : S that *requires* the presence of an unevaluated property that satisfies S, in the same way as we had to define "patternRequired" : S in order to eliminate "not" : { "patternProperties" : S }. Moreover, this new "unevaluatedRequired" : S operator would be annotation-dependent, hence we would then need to eliminate it as we do with the other "unevaluated*" operators. All of this is doable, but is quite complex, and expensive; an approach that avoids not-elimination should be preferred.

6.3. Statically characterizing the properties and items that are evaluated by a schema

Our algorithm takes advantage of those situations where one can statically characterize the properties and the items that a schema evaluates, which happens, for example, for all schemas that do not depend on an "anyOf" or "oneOf" operator. We now formalize this notion.

For example, observe that S = { "allOf" : ["prefixItems" : $[S_1, S_2]$,"contains" : S_c] } "evaluates" an item J_i at position i of an array J if, and only if, (1) J satisfies S and (2) either $i \leq 2$ or J_i satisfies S_c. We will express this fact by saying that "the pair $(2, S_c)$ characterizes the item evaluation of S" (Definitions 6.2 and 7.4-5-6). However, if you consider a schema S = { "anyOf" : $[S_1, S_2]$ }, where S_1 is characterized by $(1, S_c^1)$ and S_2 is characterized by $(2, S_c^2)$, you can only have a lower bound and an upper bound of what is evaluated. The lower bound is $(1, \{$ "allOf" : $[S_c^1, S_c^2]$ $\})$: if $\vdash^S J \,?\, S$, whichever is the "anyOf" branch that succeeds, if an element has position 1, or if it satisfies { "allOf" : $[S_c^1, S_c^2]$ }, then the element is evaluated. The upper bound is $(2, \{$ "anyOf" : $[S_c^1, S_c^2]$ $\})$: if an item of a J such that $\vdash^S J \,?\, S$ is evaluated by S, then either its position is ≤ 2 or it satisfies either S_c^1 or S_c^2. We now formalize all of this.

Definition 6 ("k matches $\{\!|\, p_1, \dots, p_n \,|\!\}$", "$J$ satisfies (h, S)").

1. For a set of patterns $P = \{\!|\, p_1, \dots, p_n \,|\!\}$, we define $L(P) = \bigcup_{i \in 1..n} L(p_i)$, and we say that k matches P when $k \in L(P)$.
2. Given an array $[\, J_1, \dots, J_n \,]$ we say that the item J_i at position i satisfies the pair (h, S) with $h \in \mathsf{Nat}$, if either $i \leq h$ or J_i satisfies S.

Definition 7 ("$\{\!|\, p_1, \dots, p_n \,|\!\}/(h, S)$ is an upper bound/is a lower bound/statically characterizes").

1. A set of patterns $\{\!|\, p_1, \dots, p_n \,|\!\}$ *is an upper bound for property evaluation* of a schema S iff, for any object J, if J satisfies S, then every evaluated property matches $\{\!|\, p_1, \dots, p_n \,|\!\}$.

Table 2
Functions $minEP(S)$ and $maxEP(S)$.

Schema	$minEP(S)$	$maxEP(S)$
$\{ K_1, \dots, K_n \}$ $n \geq 2$	$\bigcup_{i\in 1..n} minEP(\{K_i\})$	$\bigcup_{i\in 1..n} maxEP(\{K_i\})$
{ "properties" : $\{ p_1 : S_1, \dots, p_m : S_m \}$ }	$\{\mid \hat{}p_1\$, \dots, \hat{}p_m\$ \mid\}$	$\{\mid \hat{}p_1\$, \dots, \hat{}p_m\$ \mid\}$
{ "patternProperties" : $\{ p_1 : S_1, \dots, p_m : S_m \}$ }	$\{\mid p_1, \dots, p_m \mid\}$	$\{\mid p_1, \dots, p_m \mid\}$
{ "additionalProperties" : S_a }	{| ".*" |}	{| ".*" |}
{ "unevaluatedProperties" : S_u }	{| ".*" |}	{| ".*" |}
{ "$ref" : u }	$minEP(deref(u))$	$maxEP(deref(u))$
{ "anyOf" : $[S_1, \dots, S_n]$ }	$\bigcap_{i\in 1..n} minEP(S_i)$	$\bigcup_{i\in 1..n} maxEP(S_i)$
{ "oneOf" : $[S_1, \dots, S_n]$ }	$\bigcap_{i\in 1..n} minEP(S_i)$	$\bigcup_{i\in 1..n} maxEP(S_i)$
{ "allOf" : $[S_1, \dots, S_n]$ }	$\bigcup_{i\in 1..n} minEP(S_i)$	$\bigcup_{i\in 1..n} maxEP(S_i)$
any other schema	{| |}	{| |}

Table 3
Functions $minEI(S)$ and $maxEI(S)$.

Schema	$minEI(S)$		$maxEI(S)$	
$\{ K_1, \dots, K_n \}$ $n \geq 2$	$(\max_{i\in 1..n}(h_i),$ { "anyOf" : $[S'_1, \dots, S'_n]$ }) $(h_i, S'_i) = minEI(\{K_i\})$	where	$(\max_{i\in 1..n}(h_i),$ { "anyOf" : $[S'_1, \dots, S'_n]$ }) $(h_i, S'_i) = maxEI(\{K_i\})$	where
{ "prefixItems" : $[S_1, \dots, S_n]$ }	$(n, \texttt{false})$		$(n, \texttt{false})$	
{ "contains" : S_a }	$(0, S_a)$		$(0, S_a)$	
{ "items" : s }	$(\infty, \texttt{true})$		$(\infty, \texttt{true})$	
{ "unevaluatedItems" : S_u }	$(\infty, \texttt{true})$		$(\infty, \texttt{true})$	
{ "$ref" : u }	$minEI(deref(u))$		$maxEI(deref(u))$	
{ "anyOf" : $[S_1, \dots, S_n]$ } or { "oneOf" : $[S_1, \dots, S_n]$ }	$(\min_{i\in 1..n}(h_i),$ { "allOf" : $[S'_1, \dots, S'_n]$ }) $(h_i, S'_i) = minEI(S_i)$	where	$(\max_{i\in 1..n}(h_i),$ { "anyOf" : $[S'_1, \dots, S'_n]$ }) $(h_i, S'_i) = maxEI(S_i)$	where
{ "allOf" : $[S_1, \dots, S_n]$ }	$(\max_{i\in 1..n}(h_i),$ { "anyOf" : $[S'_1, \dots, S'_n]$ }) $(h_i, S'_i) = minEI(S_i)$	where	$(\max_{i\in 1..n}(h_i),$ { "anyOf" : $[S'_1, \dots, S'_n]$ }) $(h_i, S'_i) = maxEI(S_i)$	where
any other schema	$(0, \texttt{false})$		$(0, \texttt{false})$	

2. A set of patterns $\{\mid p_1, \dots, p_n \mid\}$ *is a lower bound for property evaluation* of a schema S iff, for any object J, if J satisfies S, then every property of J that matches $\{\mid p_1, \dots, p_n \mid\}$ is evaluated.
3. A set of patterns $\{\mid p_1, \dots, p_n \mid\}$ *characterizes the property evaluation* of a schema S iff it is both an upper bound and a lower bound.
4. A pair (h, S') *is an upper bound for item evaluation* of a schema S iff, for any array J, if J satisfies S, then every item evaluated by S satisfies (h, S').
5. A pair (h, S') *is a lower bound for item evaluation* of a schema S iff, for any array J, if J satisfies S, then every item of J that satisfies (h, S') is evaluated by S.
6. A pair (h, S') *characterizes the item evaluation* of a schema S iff it is both an upper bound and a lower bound.
7. A schema S is *statically characterized* iff there exist $\{\mid p_1, \dots, p_n \mid\}$ that *characterizes its property evaluation* and a pair (h, S') that *characterizes its item evaluation*.

We now define functions $minEP$, $maxEP$, $minEI$, $maxEI$, which compute a lower bound and an upper bound for the properties and the items that are evaluated by a schema. They are defined in Tables 2 and 3.

Property 2. *For any schema S, $minEP(S)$ is a lower bound for its property evaluation, and $maxEP(S)$ is an upper bound.*
For any schema S, $minEI(S)$ is a lower bound for its item evaluation, and $maxEI(S)$ is an upper bound.

Proof. We prove, by induction, what follows. For any schema S, $minEP(S)$ is a lower bound for its property evaluation, and $maxEP(S)$ is an upper bound. For any schema S, $minEI(S)$ is a lower bound for its item evaluation, and $maxEI(S)$ is an upper bound. For any keyword K, $minEP(\{K\})$ is a lower bound for the property evaluation of schema $\{ K \}$, and $maxEP(\{K\})$ is an upper bound for the same schema. For any keyword K, $minEI(\{K\})$ is a lower bound for the item evaluation of schema $\{ K \}$, and $maxEI(\{K\})$ is an upper bound for the same schema.

We start with $minEP$ and $maxEP$.

Cases "properties" and "patternProperties" are trivial. Cases "unevaluatedProperties" is trivial as well - a successful schema { "unevaluatedProperties" } evaluates all properties. Of course, when "unevaluatedProperties" is not the only keyword it does not evaluate all properties but only those that are not evaluated by the other properties, but we deal with this aspect in the inductive proof for case $\{ K_1, \dots, K_n \}$. The same holds for "additionalProperties".

Case "$ref" : u is trivial.

$minEP(\{ K_1, \dots, K_n \}) = \bigcup_{i\in 1..n} minEP(\{K_i\})$: We want to prove that for any object J, if J satisfies $\{ K_1, \dots, K_n \}$, then every property of J that matches $\bigcup_{i\in 1..n} minEP(\{K_i\})$ is evaluated. If a property p matches $\bigcup_{i\in 1..n} minEP(\{K_i\})$, then it matches $minEP(\{K_l\})$ for some l. Since $\{ K_1, \dots, K_n \}$ is satisfied, then we know that K_l is satisfied, hence, by induction, property p is evaluated by the schema $\{ K_l \}$. If K_l is an independent keyword, we conclude that p is evaluated by the keyword K_l, hence is evaluated by the schema. If K_l is

either "unevaluatedProperties" or "additionalProperties", then we know that p is either evaluated by K_l or by a keyword that precedes K_l, hence it is evaluated by the schema.

$maxEP(\{\ K_1, \dots, K_n\ \}) = \bigcup_{i\in 1..n} maxEP(\{K_i\})$: We want to prove that for any object J, if a property p of J is evaluated by $\{\ K_1, \dots, K_n\ \}$, then p matches $\bigcup_{i\in 1..n} maxEP(\{K_i\})$. If p is evaluated by $\{\ K_1, \dots, K_n\ \}$ then p is evaluated by at least one keyword K_l; if K_l is an independent property, then p is also evaluated by $\{\ K_l\ \}$, hence, by induction, p is in $maxEP(\{K_l\})$, hence it is in $\bigcup_{i\in 1..n} maxEP(\{K_i\})$. If K_l is "unevaluatedProperties" : S, then it is possible that p is not evaluated by $\{\ K_l\ \}$ since p may be evaluated by a keyword that precedes K_l, but still p is in $\bigcup_{i\in 1..n} maxEP(\{K_i\})$ since $maxEP(\{K_l\}) =$ ".*", and ".*" matches any property. The same proof holds for "additionalProperties" : S.

$minEP(\text{"oneOf"} : [\ S_1, \dots, S_n\ ]) = \bigcap_{i\in 1..n} minEP(S_i)$: We want to prove that for any object J, if J satisfies $\{(\text{"oneOf"} : [\ S_1, \dots, S_n\ ])\}$, then every property of J that matches $\bigcap_{i\in 1..n} minEP(S_i)$ is evaluated. Assume that J satisfies $\{(\text{"oneOf"} : [\ S_1, \dots, S_n\ ])\}$. Then, J satisfies S_l for some l, hence, by induction, every property of J that matches $minEP(S_l)$ is evaluated. If a property of J matches $\bigcap_{i\in 1..n} minEP(S_i)$, then it matches $minEP(S_l)$, and the thesis follows.

$maxEP(\text{"oneOf"} : [\ S_1, \dots, S_n\ ]) = \bigcup_{i\in 1..n} maxEP(S_i)$: We want to prove that for any object J, if a property p of J is evaluated by "oneOf" : $[\ S_1, \dots, S_n\ ]$, then p matches $\bigcup_{i\in 1..n} maxEP(S_i)$. If a property p of J is evaluated by a successful "oneOf" : $[\ S_1, \dots, S_n\ ]$, then it is evaluated by the only branch of "oneOf" that succeeds; let us call it S_l. By induction, property p matches $maxEP(S_l)$, hence it matches $\bigcup_{i\in 1..n} maxEP(S_i)$.

$minEP(\text{"anyOf"} : [\ S_1, \dots, S_n\ ]) = \bigcap_{i\in 1..n} minEP(S_i)$: We want to prove that for any object J, if J satisfies $\{(\text{"anyOf"} : [\ S_1, \dots, S_n\ ])\}$, then every property of J that matches $\bigcap_{i\in 1..n} minEP(S_i)$ is evaluated. Assume that J satisfies $\{(\text{"anyOf"} : [\ S_1, \dots, S_n\ ])\}$. Then, J satisfies S_l for some l, hence, by induction, every property of J that matches $minEP(S_l)$ is evaluated. If a property of J matches $\bigcap_{i\in 1..n} minEP(S_i)$, then it matches $minEP(S_l)$, and the thesis follows.

$maxEP(\text{"anyOf"} : [\ S_1, \dots, S_n\ ]) = \bigcup_{i\in 1..n} maxEP(S_i)$: We want to prove that for any object J, if a property p of J is evaluated by "anyOf" : $[\ S_1, \dots, S_n\ ]$, then p matches $\bigcup_{i\in 1..n} maxEP(S_i)$. If a property p of J is evaluated by a successful "anyOf" : $[\ S_1, \dots, S_n\ ]$, then it is evaluated by some branches of "anyOf" that succeed; let us consider anyone of them, and call it S_l. By induction, property p matches $maxEP(S_l)$, hence it matches $\bigcup_{i\in 1..n} maxEP(S_i)$.

$minEP(\text{"allOf"} : [\ S_1, \dots, S_n\ ]) = \bigcup_{i\in 1..n} minEP(S_i)$: We want to prove that for any object J, if J satisfies $\{(\text{"allOf"} : [\ S_1, \dots, S_n\ ])\}$, then every property of J that matches $\bigcup_{i\in 1..n} minEP(S_i)$ is evaluated. Assume that J satisfies $\{(\text{"allOf"} : [\ S_1, \dots, S_n\ ])\}$. Then, J satisfies every S_l hence, by induction, every property of J that matches $minEP(S_l)$ is evaluated, for each l, hence every property in $\bigcup_{i\in 1..n} minEP(S_i)$ is evaluated.

$maxEP(\text{"allOf"} : [\ S_1, \dots, S_n\ ]) = \bigcup_{i\in 1..n} maxEP(S_i)$: We want to prove that for any object J, if a property p of J is evaluated by "allOf" : $[\ S_1, \dots, S_n\ ]$, then p matches $\bigcup_{i\in 1..n} maxEP(S_i)$. If a property p of J is evaluated by "allOf" : $[\ S_1, \dots, S_n\ ]$, then it is evaluated by some S_l. By induction, property p matches $maxEP(S_l)$, hence it matches $\bigcup_{i\in 1..n} maxEP(S_i)$.

Any other keyword is either a structural keyword different from the ones listed above or is the "not" keyword. The structural keywords different from the ones listed above do not evaluate any property of the instance under validation. A successful "not" : S keyword is applied to a schema S that does not validate the instance, and a schema that does not validate an instance does not return any validated property. Schemas true and false do not evaluate any properties.

We move to $minEI$ and $maxEI$.

We say that an item J_i of an array $J = [\ J_1, \dots, J_n\ ]$, satisfies the first component of a pair (l, S), if $i \le l$, and that is satisfies the second component if J_i satisfies S. We say that J_i satisfies the pair (l, S) if it satisfies either the first or the second component.

$minEI/maxEI(\{\text{"items"} : S\}) = (\infty, \texttt{true})$: a schema $\{\ \text{"items"} : S\ \}$ evaluates all items of any evaluated array, and every item of an array satisfies the pair $(\infty, \texttt{true})$.

$minEI/maxEI(\{\text{"prefixItems"} : [\ S_1, \dots, S_n\ ]\}) = (n, \texttt{true})$: a schema $\{\ \text{"prefixItems"} : [\ S_1, \dots, S_n\ ]\ \}$ evaluates all and only the items with position $\le n$, which are exactly the items that satisfy $(n, \texttt{false})$.

$minEI/maxEI(\{\text{"contains"} : S\}) = (0, S)$: a schema $\{\ \text{"contains"} : S\ \}$ evaluates all and only the items that satisfy S, which are exactly the items that satisfy $(0, S)$.

All other cases are analogous to those for $minEP/maxEP$, where min on the first component and "allOf" on the second one take the place of $\bigcap$, and max on the first component and "anyOf" on the second one take the place of $\bigcup$. We report a couple of cases as examples.

$minEI(\{\ K_1, \dots, K_n\ \}) = (\max_{i\in 1..n}(h_i), \{\ \text{"anyOf"} : [S'_1, \dots, S'_n]\ \})$ where $(h_i, S'_i) = minEI(S_i)$: We want to prove that for any object J, if J satisfies $\{\ K_1, \dots, K_n\ \}$, then every item of J that satisfies the pair $minEI(\{\ K_1, \dots, K_n\ \})$ is evaluated. If an item J_i satisfies the pair, then either its position is less than $\max_{i\in 1..n}(h_i)$, or it satisfies $\{\ \text{"anyOf"} : [S'_1, \dots, S'_n]\ \}$. In the first case, its position is less then h_l for some l; in the second case, it satisfies S'_l for some l; hence, in both cases, J_i satisfies a pair (h_l, S'_l) for some l. Since $(h_l, S'_l) = minEI(\{K_l\})$, then, by induction, item J_i is evaluated by the schema $\{\ K_l\ \}$. If K_l is an independent keyword, we conclude that J_i is evaluated by the keyword K_l, hence is evaluated by the schema. If K_l is either "items" or "unevaluatedItems", then we know that J_i is either evaluated by K_l or by a keyword that precedes K_l, hence it is evaluated by the schema.

$maxEI(\{\ K_1, \dots, K_n\ \}) = (\max_{i\in 1..n}(h_i), \{\ \text{"anyOf"} : [S'_1, \dots, S'_n]\ \})$ where $(h_i, S'_i) = maxEI(S_i)$: We want to prove that for any object J, if an item J_i of J is evaluated by $\{\ K_1, \dots, K_n\ \}$, then J_i satisfies the pair $maxEI(\{\ K_1, \dots, K_n\ \})$. If J_i is evaluated by $\{\ K_1, \dots, K_n\ \}$ then J_i is evaluated by at least one keyword K_l. If K_l is an independent item, then J_i is also evaluated by $\{\ K_l\ \}$, hence, by induction, J_i satisfies $maxEI(\{K_l\}) = (h_l, S'_l)$, hence, either its position is less then h_l, or it satisfies S'_l. In the first case its position is less then $\max_{i\in 1..n}(h_i)$, in the second case it satisfies $\{\ \text{"anyOf"} : [S'_1, \dots, S'_n]\ \}$, hence it satisfies the pair. If K_l is "unevaluatedProperties" : S, then it is possible that J_i is not evaluated by $\{\ K_l\ \}$ since J_i may be evaluated by a keyword that precedes K_l, but still J_i satisfies

{ "anyOf" : [$S'_1, \ldots, S'_n$] } since $maxEI(\{K_l\}) = (\infty, \texttt{true})$, hence $S'_l = \texttt{true}$ (a second reason why J_i satisfies the pair is that $h_l = \infty$ hence the position of J_i satisfies $\max_{i\in 1..n}(h_i)$). The same reasoning holds when K_l is "items" : S.

$minEI$("oneOf" : [$S_1, \ldots, S_n$]) = ($\min_{i\in 1..n}(h_i)$, { "allOf" : [$S'_1, \ldots, S'_n$] }) where $(h_i, S'_i) = minEI(S_i)$: We want to prove that for any object J, if J satisfies { ("oneOf" : [$S_1, \ldots, S_n$]) }, then every item J_i of J that satisfies the pair $minEI$("oneOf" : [$S_1, \ldots, S_n$]) is evaluated. If an item J_i satisfies the pair, then either its position is less than $\min_{i\in 1..n}(h_i)$, or it satisfies { "allOf" : [$S'_1, \ldots, S'_n$] }. In the first case, its position is less then h_l for every l; in the second case, it satisfies S'_l for every l; hence, in both cases, J_i satisfies every pair (h_l, S'_l), that is, it satisfies $minEI(S_l)$ for every l. Since J satisfies { ("oneOf" : [$S_1, \ldots, S_n$]) }, then, J satisfies S_o for some o, hence, by induction, every item of J that satisfies $minEI(S_o)$ is evaluated, and we have just proved that J_i satisfies every pair $minEI(S_l)$, including $minEI(S_o)$, hence J_i is evaluated.

$maxEI$("oneOf" : [$S_1, \ldots, S_n$]) = ($\max_{i\in 1..n}(h_i)$, { "anyOf" : [$S'_1, \ldots, S'_n$] }) where $(h_i, S'_i) = maxEI(S_i)$: We want to prove that for any object J, if an item J_i of J is evaluated by "oneOf" : [$S_1, \ldots, S_n$], then J_i satisfies the pair $maxEI$("oneOf" : [$S_1, \ldots, S_n$]). If an item J_i of J is evaluated by a successful "oneOf" : [$S_1, \ldots, S_n$], then it is evaluated by the only branch of "oneOf" that succeeds; let us call it S_o. By induction, item J_i satisfies $maxEI(S_o)$, hence it either has a position that is smaller than h_o or it satisfies S'_o. In the first case it satisfies the first component $\max_{i\in 1..n}(h_i)$, in the second case it satisfies the second component { "anyOf" : [$S'_1, \ldots, S'_n$] }. □

When $minEP(S)$ and $maxEP(S)$ denote the same language, then $minEP(S)$ (or, equivalently, $maxEP(S)$) statically characterizes the property evaluation of S, and our algorithm can use this fact to produce an optimized (smaller) encoding (as we will see in Section 6.5). However, deciding whether $minEP(S)$ and $maxEP(S)$ are equivalent is computationally expensive, and therefore different implementations can devote a different effort to this optimization. In order to express this fact, we define a function $exEP_{eq}(S)$, which is parametrized on the function eq that is used to compare two sets of patterns, so that $exEP_{eq}(S) = \{\!|\, p_1, \ldots, p_n \,|\!\}$ when eq is able to prove that $minEP(S)$ and $maxEP(S)$ are both equivalent to some $\{\!|\, p_1, \ldots, p_n \,|\!\}$, and $exEP_{eq}(S) = \uparrow$ otherwise. Parametrizing $exEP_{eq}$ over eq allows us to discuss the effect of different choices for function eq.

Definition 8 ($exEP_{eq}(S)$, $exEI_{eq}(S)$, $exEP_{eq}(S)\downarrow$ /$\uparrow$, $exEI_{eq}(S)\downarrow$ /$\uparrow$)**.** Assume that eq is a binary relation such that

$$\begin{array}{lll} eq(\{\!|\, p_1, \ldots, p_n \,|\!\}, \{\!|\, p'_1, \ldots, p'_m \,|\!\}) & \Rightarrow & L(\{\!|\, p_1, \ldots, p_n \,|\!\}) = L(\{\!|\, p'_1, \ldots, p'_m \,|\!\}) \\ eq((n, S), (m, S')) & \Rightarrow & (n = m) \;\varsigma\; (\forall J.\ (J \text{ satisfies } S) \Leftrightarrow (J \text{ satisfies } S')) \end{array}$$

The functions $exEP_{eq}(S)$ and $exEI_{eq}(S)$ are defined as follows.

$$\begin{array}{rcll} \exists\, eq(minEP(S), maxEP(S)) & \Rightarrow & exEP_{eq}(S) & = \uparrow \\ eq(minEP(S), maxEP(S)) & \Rightarrow & exEP_{eq}(S) & = minEP(S) \\ \exists\, eq(minEI(S), maxEI(S)) & \Rightarrow & exEI_{eq}(S) & = \uparrow \\ eq(minEI(S), maxEI(S)) & \Rightarrow & exEI_{eq}(S) & = minEI(S) \end{array}$$

We use $exEP_{eq}(S)\downarrow$ to indicate that $exEP_{eq}(S) \neq \uparrow$, and similarly for $exEI_{eq}(S)\downarrow$. We use $exEP_{eq}(S)\uparrow$ to indicate that $exEP_{eq}(S) = \uparrow$, and similarly for $exEI_{eq}(S)\uparrow$.

For any eq that enjoys the properties of Definition 8, the following property holds.

Property 3 ($exEP_{eq}(S)$, $exEI_{eq}(S)$, characterize property/item evaluation)**.** *For any schema S, if $exEP_{eq}(S)\downarrow$, then $exEP_{eq}(S)$ characterizes the property evaluation of S. For any schema S, if $exEI_{eq}(S)\downarrow$, then $exEI_{eq}(S)$ characterizes the item evaluation of S.*

From now on, we assume that the function eq is fixed and that it includes at least syntactical equality, so that $eq(P, P)$ and $eq((n, S), (n, S))$ hold.

Example 2. Consider the following schema.

```
{ "anyOf": [
    { "properties": { "a": { "type":"string" } }, "additionalProperties":false },
    { "properties": { "b": { "type":"string" } }, "additionalProperties":false }
  ]
}
```

The first branch S_1 of "anyOf" has $minEP(S_1) = maxEP(S_1) = exEP_{eq}(S_1) = \{\!|$"^a$","."*"$|\!\}$; the second branch S_2 has $minEP(S_1) = maxEP(S_1) = exEP_{eq}(S_1) = \{\!|$"^b$","."*"$|\!\}$. Our algorithm may rewrite both of them as just $\{\!|$"."*"$|\!\}$, but let us assume it does not. Then, we have $minEP$("anyOf" : [S_1, S_2]) = $\{\!|$"^a$","."*"$|\!\} \cap \{\!|$"^b$","."*"$|\!\} = \{\!|$"."*"$|\!\}$, and $maxEP$("anyOf" : [S_1, S_2]) = $\{\!|$"^a$","."*"$|\!\} \cup \{\!|$"^b$","."*"$|\!\} = \{\!|$"^a$","^b$","."*"$|\!\}$.

Now, the value of $exEP_{eq}$("anyOf" : [S_1, S_2]) depends on the function eq. If eq is just set equality, that we indicate as $=$, then we have that $\neg eq(minEP$("anyOf" : [...]), $maxEP$("anyOf" : [...])), hence $exEP_{=}$("anyOf" : [S_1, S_2])$\uparrow$. The function eq that we have implemented, however, regards any two sets that contain "."*" as equivalent, ignoring any other pattern in this case; let us call this function eq^+; in this case, $exEP_{eq^+}$("anyOf" : [S_1, S_2]) = $\{\!|$"."*"$|\!\}$.

We now show that $exEP_{eq}(S)$ is undefined only when S depends on either an "anyOf" or a "oneOf" keyword. We first define the notion of *Å from*.

Definition 9 (K Å from K')**.** For a fixed schema S, we say an occurrence of a subkeyword K is Å from an occurrence a subkeyword K' if there is a path from K to K' that only crosses boolean keywords or references. Formally, K is immediately Å from K' if either (1) K is a boolean keyword "allOf"/"anyOf"/"oneOf" : $[S_1, \dots, S_k]$, and K' is a top-level keyword in one of the S_i arguments of K, or (2) K is a reference keyword "$ref" : u, and K' is a top-level keyword in $deref(u)$. K is Å from K' if there exists a sequence $K_1, \dots, K_n$ of keyword occurrences such that either $n = 1$, or, for each $i \in 1..n-1$, K is immediately Å from K'. An occurrence of a subschema S' of S is Å from K' if one of the top-level keywords of S' is Å from K'.

The following property implies that $exEP_{eq}(S)$ is always well defined unless S is Å from an "anyOf" keyword or a "oneOf" keyword that evaluates different properties in its different branches, and similarly for $exEI_{eq}$.

Property 4 ($exEP_{eq}$, $exEI_{eq}$, and "anyOf"/"oneOf")**.** *$minEP(S) = maxEP(S)$ holds for any schema S which is not Å from any* "anyOf" *or* "oneOf" *keyword K' such that $minEP(\{ K' \}) \neq maxEP(\{ K' \})$. The same property holds if we substitute $minEP/maxEP$ with $minEI/maxEI$.*

Proof. To follow the fine details of the proof, please remember that $minEP$ is always applied to schemas, such as $\{ K \}$, and never directly to keywords, such as K, which forces us to add/remove curly brackets in some points of the proof.

Assume that S is a schema that is not Å from any "anyOf" or "oneOf" keyword K' with $minEP(\{K'\}) \neq maxEP(\{K'\})$. For any line in Table 2 we show that $minEP(S)$ and $minEP(S)$ are equal, by induction on the in-place depth of S.

All lines different from "oneOf", "anyOf", "$ref", "allOf", $\{ K_1, \dots, K_n \}$ are immediate since they have $minEP = maxEP$.

In case $S = \{$ "allOf" : $[K_1, \dots, K_n] \}$, since S is a schema that is not Å from any "anyOf" or "oneOf" keyword K' with $minEP(\{K'\}) \neq maxEP(\{K'\})$, the same property holds for $\{ K_1 \}, \dots, \{ K_2 \}$. Hence all these schemas enjoy $minEP(\{ K_i \}) = maxEP(\{ K_i \})$, hence we have $\bigcup_{i \in 1..n} minEP(\{ K_i \}) = \bigcup_{i \in 1..n} maxEP(\{ K_i \})$, i.e., $minEP(S) = maxEP(S)$. The same proof holds for "allOf". In case $S = \{$ "$ref" : $u \}$, since S is a schema that is not Å from any "anyOf" or "oneOf" keyword K' with $minEP(\{K'\}) \neq maxEP(\{K'\})$, the same property holds for $deref(u)$. By induction on the in-place depth, we have that $minEP(deref(u)) = maxEP(deref(u))$, hence $minEP(S) = maxEP(S)$.

In case $S = \{ K_a \} = \{$ "anyOf" : $[S_1, \dots, S_n] \}$, if we had $minEP(\{ K_a \}) \neq maxEP(\{ K_a \})$ then we would violate the hypothesis that S is not Å from any "anyOf" or "oneOf" keyword K' with $minEP(\{K'\}) \neq maxEP(\{K'\})$, hence we must have $minEP(\{ K_a \}) = maxEP(\{ K_a \})$, i.e., $minEP(S) = maxEP(S)$. The same proof holds for "oneOf".

The same proof presented for $exEP_{eq}$ holds for $exEI_{eq}$. □

6.4. Pair covering and Evaluation Normal Form

We now describe our generalization of the XDNF and our algorithm for the elimination of "unevaluated*". Consider the following version of the schema presented in Section 4.

```
{ "anyOf": [
    { "properties": { "price":{} } },
    { "properties": { "plate":{} } }
  ],
  "unevaluatedProperties": false
}
```

As we have seen, an object with a price and a plate is validated by this schema, since it satisfies both branches, and the price is evaluated by the first branch and the plate by the second one: the two branches cooperate to evaluate all the fields that must be evaluated in order to bypass the "unevaluated*" constraint. Hence, we cannot push "unevaluatedProperties": false through the branches of "anyOf", since, after pushing, the different branches cannot cooperate any more, and hence the object with both fields would not be validated by the new schema. Consider now the following schema (that is equivalent):

```
{ "anyOf": [
    { "properties": { "price":{} } },
    { "properties": { "plate":{} } }
    { "properties": { "price":{}, "plate":{} } },
  ],
  "unevaluatedProperties": false
}
```

It is not in XDNF, but, if we push "unevaluatedProperties": false through the branches of "anyOf", any J that was accepted before is still accepted. Consider the schema after pushing.

```
{ "anyOf": [
   { "properties": { "price":{} }, "unevaluatedProperties": false },
   { "properties": { "plate":{} }, "unevaluatedProperties": false },
   { "properties": { "price":{}, "plate":{} }, "unevaluatedProperties": false }
  ]
}
```

An object with both fields is still accepted by the third branch. This time, it was possible to push "unevaluatedProperties": false through "anyOf" because, for any two branches S_1 and S_2 that can cooperate to evaluate the fields of an instance J, there is a branch S_c that "*covers*" them, i.e., that, if applied to a J that satisfies both S_1 and S_2, then S_c is satisfied, and it evaluates all fields that are evaluated either by S_1 or by S_2. This condition is qualified by "if applied to a J that satisfies both S_1 and S_2", hence it holds trivially when S_1 and S_2 are disjoint, so that, when S_1 and S_2 are disjoint, then any schema (S_1, for example) *covers* the two, but we will see that there are many other situations when two schemas are *covered* by one of the two, or by a third schema. Thus, reaching a disjunctive form where all branches are *covered* by a branch in the disjunction is much easier than reaching an XDNF; this is the basis of our algorithm.

We now formally define the notion of *p-covers* and the corresponding notion of *i-covers*, used to analyze array evaluation.

Definition 10 (S_c *p-covers*(S_1, S_2), S_c *i-covers*(S_1, S_2))**.** We say that S_c *p-covers*(S_1, S_2) iff, for any J that satisfies both S_1 and S_2:

1. J satisfies S_c;
2. every property of J that is evaluated by S_1 is also evaluated by S_c;
3. every property of J that is evaluated by S_2 is also evaluated by S_c.

The definition of S_c *i-covers*(S_1, S_2) is obtained by substituting "property" with "item".

In an XDNF all branches are mutually exclusive, and we generalize that with cover-closure. A disjunction is cover-closed if, for each pair of schemas, it contains a schema that *p-covers* both and a schema that *i-covers* both. For simplicity, from now on we only focus on the *p-covers* relation, but everything that we say can be immediately transferred to the *i-covers* relation.

For any two schemas S_1 and S_2, the schema { "allOf" : $[S_1, S_2]$ } *p-covers* the pair, so we can make a disjunction cover-closed by adding that schema to the list of the disjunction arguments, but we have many interesting special cases when one of the two schemas covers both, so that we have cover-closure at no cost. For example, when two schemas S_1 and S_2 are mutually exclusive, then any schema (such as S_1 itself) trivially *p-covers* both, so that cover-closure generalizes exclusiveness. Moreover, it is easy to check that, if $minEP(S_1) \supseteq maxEP(S_2)$, then S_1 *p-covers* (S_1, S_2), so that, as a consequence:

1. if two schemas S_1 and S_2 are characterized by the same set of patterns, then each of them *p-covers* the pair (S_1, S_2);
2. one schema S_1 that evaluates all properties (since it contains "unevaluatedProperties", for example) *p-covers* any pair (S_1, S_2) formed with another arbitrary schema S_2;
3. given a schema S_2 that evaluates no properties, any arbitrary schema S_1 *p-covers* (S_1, S_2).

Hereafter, given a list of schemas $L = [\![S_1, \ldots, S_n]\!]$, we use "anyOf" : L to indicate the keyword "anyOf" : $[S_1, \ldots, S_n]$, and similarly for "oneOf" and "allOf".

In order to eliminate "unevaluated*" operators from a disjunction, we must (1) push the operators through the branches and (2) in any branch, rewrite the pushed operator using its "annotation independent" counterpart "additionalProperties" or "items". An XDNF allows for (1) because all arguments are pairwise disjoint and (2) because all arguments are conjunctions of structural keywords, which allows us to compute which fields/items are evaluated. We generalize the idea by defining the ENF, which allows (1) because all arguments are *covered* by another one and (2) because all arguments are statically characterized.

This is its formal definition.

Definition 11 (evaluation normal form — ENF)**.** We say that a list of schemas $L = [\![S_1, \ldots, S_n]\!]$ is cover-closed when every pair of schemas (S_i, S_j) of L is *covered* by a schema in L (hence, the empty list and every singleton are cover-closed). We say that a list of schemas $L = [\![S_1, \ldots, S_n]\!]$ is *elementwise statically characterized* when every S_i is statically characterized. We say that a keyword "anyOf" : L or a schema { "anyOf" : L } is cover-closed (or elementwise statically characterized) when the list L is cover-closed (or, respectively, elementwise statically characterized).

We say that a schema is in *evaluation normal form* ENF when:

1. it is a "anyOf" single-keyword schema, that is, it is equal to { "anyOf" : L } for some L;
2. the list L is cover-closed;
3. the list L is elementwise statically characterized.

Every schema { "anyOf" : $[S_1, \ldots, S_n]$ } in XDNF is also in ENF, since every two schemas S_i, S_j that are disjoint are *p-covered* by S_i (and by S_j), and since every product S_i is statically-characterized, by Property 4, since S_i is not Å from any "anyOf" or "oneOf".

A schema { "anyOf" : $[S_1, \ldots, S_n]$ } in ENF is not necessarily in XDNF, nor even in DNF: for example, each of the S_i may contain any nesting of boolean operators and of references, as long as it is statically-characterized, for example because it has shape { "not" : S }; hence, the ENF is much less restrictive than the XDNF.

6.5. Computing the ENF

An arbitrary schema can be transformed in ENF using the algorithm of Definition 13; from now on, we assume that eq is fixed, hence we write $exEP$ rather than $exEP_{eq}$.

The first two lines specify that the function $ENF(S)$ is, essentially, the identity function when S is statically characterized (case $exEP(S)\downarrow$). These first two lines are crucial in keeping the size of $ENF(S)$ similar to the size of S.

By Property 4, for any keyword K that is different from "allOf", "anyOf", "oneOf", or "$ref", we have that $exEP_{eq}(\{\ K\ \})\downarrow$ even with the simplest eq, hence, the only cases where we can have $exEP(S)\uparrow$ are those of multi-keyword schemas, which we immediately reduce to single-keyword schemas (case 3 in the definition), and single-keyword schemas whose keyword is either "allOf", "anyOf", "oneOf", or "$ref", and where $\neg eq(minEP(\{\ K\ \}), maxEP(\{\ K\ \}))$ (cases 4 to 7).

Observe that also these in-place keywords may satisfy $exEP_{eq}(\{\ K\ \})\downarrow$. For example, when K is "oneOf": A or "anyOf": A, we have $exEP_{eq}(\{\ K\ \})\downarrow$ as soon as $eq(minEP(\{\ K\ \}), maxEP(\{\ K\ \}))$. As another example, when K is "allOf": A or "$ref": u, we have $exEP_{eq}(\{\ K\ \})\downarrow$ when they are not Å from any "oneOf": A or "anyOf": A. In all these cases, the trivial rules of case $exEP(S)\downarrow$ are applied (cases 1 and 2).

We now analyze all the lines in the table.

When $S = \{\ \texttt{"allOf"} : [S_1, \ldots, S_m]\ \}$, the algorithm first normalizes all of the subschemas S_i, hence obtaining a conjunction $\{\ \texttt{"allOf"} : [\{\ \texttt{"anyOf"} : L_1\ \}, \ldots, \{\ \texttt{"anyOf"} : L_m\ \}]\ \}$ where every L_i is cover-closed. Then, it uses an auxiliary function And with the following properties (we use $S_1 \sim S_2$ to indicate that the two schemas validate the same instances):

1. $\{\ \texttt{"anyOf"} : And(\llbracket\ L_1, \ldots, L_m\ \rrbracket)\ \} \sim \{\ \texttt{"allOf"} : [\{\ \texttt{"anyOf"} : L_1\ \}, \ldots, \{\ \texttt{"anyOf"} : L_m\ \}]\ \}$;
2. if every L_i is cover-closed, then $And(L_1, \ldots, L_m)$ is cover-closed.

The auxiliary function is defined in Definition 12; it just computes a sort of "Cartesian product" of the lists $L_1, \ldots, L_m$ using the recursive function And, and we will prove that it enjoys the two desired properties. Observe that $And(\mathcal{L})$ transforms a list of lists of schemas ($\mathcal{L} = \llbracket\ L_1, \ldots, L_m\ \rrbracket$) into a flat list of schemas (L).

A multi-keywords schema $\{\ K_1, \ldots, K_n\ \}$ is treated as $\{\ \texttt{"allOf"} : [\{\ K_1\ \}, \ldots, \{\ K_n\ \}]\ \}$.

When S is a single-keyword schema $\{\ \texttt{"anyOf"} : [S_1, \ldots, S_m]\ \}$, the algorithm first normalizes all subschemas, obtaining a disjunction $\{\ \texttt{"anyOf"} : [\{\ \texttt{"anyOf"} : L_1\ \}, \ldots, \{\ \texttt{"anyOf"} : L_m\ \}]\ \}$. Then, it uses an auxiliary function Or with the property that

1. $\{\ \texttt{"anyOf"} : Or(\llbracket\ L_1, \ldots, L_m\ \rrbracket)\ \} \sim \{\ \texttt{"anyOf"} : [\{\ \texttt{"anyOf"} : L_1\ \}, \ldots, \{\ \texttt{"anyOf"} : L_m\ \}]\ \}$;
2. if every L_i is cover-closed, then $Or(L_1, \ldots, L_m)$ is cover-closed.

$Or(\mathcal{L})$ is a recursive function that transforms a list of lists of schemas into a flat list of schemas using the auxiliary function $Close(L', L'')$, which computes a list of schemas that is big enough to ensure that $L' +\!\!+ L'' +\!\!+ Close(L', L'')$ is cover-closed.

$Close(L', L'')$ is defined by a lower bound and an upper bound. The upper bound just collects, for every $S' \in L'$ and $S'' \in L''$, their conjunction $\{\ \texttt{"allOf"} : [S', S'']\ \}$, which ensures that the two are *covered* in the result. The lower bound specifies that an implementation is allowed to omit a conjunction from the result when the pair is *covered* by a schema in $L' +\!\!+ L''$; any implementation is free to decide how aggressively to aim for the lower bound, which yields a smaller ENF but requires a bigger computational effort.[11]

When $S = \{\ \texttt{"oneOf"} : [S_1, \ldots, S_m]\ \}$, the algorithm first rewrites S as a disjunction of m conjunctions $\{\ \texttt{"allOf"} : [\ \{\ \texttt{"not"} : S_1\ \}, \ldots, \{\ \texttt{"not"} : S_{i-1}\ \},\ S_i,\ \{\ \texttt{"not"} : S_{i+1}\ \}, \ldots, \{\ \texttt{"not"} : S_m\ \}\ ]\ \}$, accordingly with the boolean semantics of this operator. Then, it normalizes each conjunction, hence obtaining a disjunction $\texttt{"anyOf"} : [\{\ \texttt{"anyOf"} : L_1\ \}, \ldots, \{\ \texttt{"anyOf"} : L_m\ \}]$. At this point, each L_i is cover-closed and no J may satisfy both a schema in L_i and a schema in L_j with $i \neq j$, hence the list $L_1 +\!\!+ \ldots +\!\!+ L_m$ is cover-closed, hence $\texttt{"anyOf"} : [\{\ \texttt{"anyOf"} : L_1\ \}, \ldots, \{\ \texttt{"anyOf"} : L_m\ \}]$ can be rewritten as $\texttt{"anyOf"} : L_1 +\!\!+ \ldots +\!\!+ L_m$ (see Property 7 for a formal proof).

Finally, when S is a reference, the reference is substituted by its definition, which is recursively normalized. This recursive step can never generate an infinite loop, thanks to the guarded recursion constraint, since normalization only expands boolean operators and references but never crosses the structural operators.[12]

Definition 12 ($And(\mathcal{L}), Close(L', L''), Or(\mathcal{L})$)**.**

$$
\begin{array}{lcl}
And(\llbracket\ \rrbracket) & = & \llbracket\ \texttt{true}\ \rrbracket \\
And(L :: \mathcal{L}) & = & \llbracket\ \{\ \texttt{"allOf"} : [S', S'']\ \}\ \mid\ S' \in L,\ S'' \in And(\mathcal{L})\ \rrbracket \\
Close(L', L'') & \subseteq & \llbracket\ \{\ \texttt{"allOf"} : [S', S'']\ \}\ \mid\ S' \in L',\ S'' \in L''\ \rrbracket \\
Close(L', L'') & \supseteq & \llbracket\ \{\ \texttt{"allOf"} : [S', S'']\ \}\ \mid\ S' \in L',\ S'' \in L'',\ \nexists S \in (L' +\!\!+ L'').\ S\ covers\ (S', S'')\ \rrbracket \\
Or(\llbracket\ \rrbracket) & = & \llbracket\ \rrbracket \\
Or(L :: \mathcal{L}) & = & L +\!\!+ Or(\mathcal{L}) +\!\!+ Close(L, Or(\mathcal{L}))
\end{array}
$$

[11] In our implementation, we just check whether (S', S'') is *covered* by either S' or by S'' because $minEP(S_1) \supseteq maxEP(S_2)$ or $minEP(S_2) \supseteq maxEP(S_1)$ (as discussed in Section 6.4), otherwise we ignore the optimization possibility and insert $\{\ \texttt{"allOf"} : [S', S'']\ \}$ into $Close(L', L'')$.

[12] In the implementation, the definition $deref(u)$ of each reference "$ref" : u is normalized the first time the reference is met, and the result is memorized to be reused when needed, so that $ENF(deref(u))$ is only computed once even if there are many occurrences of "$ref" : u.

Definition 13 ($ENF(S)$)**.**
Function $ENF(S)$ is defined as follows:

If $exEP(S)\downarrow$ **and** $exEI(S)\downarrow$ **(e.g.,** { "not" : S }**, or** { "properties" : S }**, or ...):**				
1	$ENF(S)$	=	S	if $S = \{$ "anyOf" : $L\ \}$
2	$ENF(S)$	=	{ "anyOf" : $[S]$ }	if $S \neq \{$ "anyOf" : $L\ \}$
If $exEP(S)\uparrow$ **or** $exEI(S)\uparrow$ **(**{ $K_1, \ldots, K_m$ }**,** { "anyOf" : L }**,** { "oneOf" : L }**,** { "allOf" : L }**,** { "$ref" : u }**):**				
3	$ENF(\{ K_1, \ldots, K_m \})$ with $m > 1$	=	{ "anyOf" : $And(\llbracket L_1, \ldots, L_m \rrbracket)$ } *where* $ENF(\{ K_i \}) = \{$ "anyOf" : $L_i\ \}$ *for* $i \in 1..m$	
4	$ENF(\{$ "allOf" : $[S_1, \ldots, S_m] \})$	=	{ "anyOf" : $And(\llbracket L_1, \ldots, L_m \rrbracket)$ } *where* $ENF(S_i) = \{$ "anyOf" : $L_i\ \}$ *for* $i \in 1..m$	
5	$ENF(\{$ "anyOf" : $[S_1, \ldots, S_m] \})$	=	{ "anyOf" : $Or(\llbracket L_1, \ldots, L_m \rrbracket)$ } *where* $ENF(S_i) = \{$ "anyOf" : $L_i\ \}$ *for* $i \in 1..m$	
6	$ENF(\{$ "oneOf" : $[S_1, \ldots, S_m] \})$	=	{ "anyOf" : $L_1 +\!\!+ \ldots +\!\!+ L_m$ } *where* $ENF(\{$"allOf" : $[\{$ "not" : $S_1\ \}, \ldots, \{$ "not" : $S_{i-1}\ \}, S_i,$ $\{$ "not" : $S_{i+1}\ \}, \ldots, \{$ "not" : $S_m\ \}]\}) = \{$ "anyOf" : $L_i\ \}$ *for* $i \in 1..m$	
7	$ENF(\{$ "$ref" : $u\ \})$	=	$ENF(deref(u))$	

The function $ENF(S)$ is well-defined; this is essentially a consequence of the guarded recursion constraint.

Property 5. *The function $ENF(S)$ is well-defined.*

Proof Sketch. In cases 3 to 7, $ENF(S)$ is defined in terms of a set of applications of ENF to a set of schemas S_i (just one in case 7), but in all of these cases the in-place depth (Definition 3) of each S_i is smaller than the in-place depth of S; in the case for "oneOf" this happens thanks to the use of $booleanOneOf$ in the definition of in-place depth for "oneOf".

We now prove that And and Or enjoy the properties that we promised in the text.

Property 6 ($And(\mathcal{L}), Or(\mathcal{L})$)**.** *For any list of lists of schemas $\mathcal{L}$ such that every list $L \in \mathcal{L}$ is cover-closed and is elementwise statically characterized, and for any two lists L' and L'' that are cover-closed and elementwise statically characterized:*

1. *$And(\mathcal{L})$ is cover-closed and is elementwise statically characterized;*
2. *$L' +\!\!+ L'' +\!\!+ Close(L', L'')$ is cover-closed and is elementwise statically characterized;*
3. *$Or(\mathcal{L})$ is cover-closed and is elementwise statically characterized;*
4. *"anyOf" : $And(L_1, \ldots, L_n)$ is equivalent to "allOf" : [{ "anyOf" : $L_1\ \}, \ldots, \{$ "anyOf" : $L_n\ \}$];*
5. *"anyOf" : $L' +\!\!+ L'' +\!\!+ Close(L', L'')$ is equivalent to "anyOf" : $L' +\!\!+ L''$;*
6. *"anyOf" : $Or(L_1, \ldots, L_n)$ is equivalent to "anyOf" : [{ "anyOf" : $L_1\ \}, \ldots, \{$ "anyOf" : $L_n\ \}$];*
7. *If each list L_i is cover-closed and is elementwise statically characterized, and, for any $i \neq j$, { "anyOf" : $L_i\ \}$ and { "anyOf" : $L_j\ \}$ are disjoint, then $L_1 +\!\!+ \ldots +\!\!+ L_n$ is cover-closed and is elementwise statically characterized.*

Proof. All lists that are described in points 1, 2, 3, and 7,are built by choosing elements from lists that are elementwise statically characterized and by adding elements with shape { "allOf" : $[S_1, S_2]\ \}$ where S_1 and S_2 are statically characterized; all such elements are hence statically characterized, hence, all these lists are elementwise statically characterized.

(1) $And(\mathcal{L})$ is cover-closed: by induction. Case $And(\llbracket\ \rrbracket)$: holds since every singleton is cover-closed. Case $And(L :: \mathcal{L}) = \llbracket \{$ "allOf" : $[S', S'']\ \} \mid S' \in L,\ S'' \in And(\mathcal{L}) \rrbracket$. Assume S_1 and S_2 belong to $\llbracket \{$ "allOf" : $[S', S'']\ \} \mid S' \in L,\ S'' \in And(\mathcal{L}) \rrbracket$; then, $S_1 = \{$ "allOf" : $[S'_1, S''_1]\ \}$ and $S_2 = \{$ "allOf" : $[S'_2, S''_2]\ \}$ with $S'_1 \in L$, $S''_1 \in L$, $S'_2 \in And(\mathcal{L})$, $S''_2 \in And(\mathcal{L})$. By hypothesis, there exist $S'_3 \in L$ that covers (S'_1, S'_2), and $S''_3 \in And(\mathcal{L})$ that covers (S''_1, S''_2). The schema $S_3 = \{$ "allOf" : $[S'_3, S''_3]\ \}$ belongs to $And(L :: \mathcal{L})$ by construction. If J satisfies both S_1 and S_2, then it satisfies all of S'_1, S''_1, S'_2, S''_2, hence it satisfies both S'_1, S'_2 and S''_1, S''_2, hence it satisfies both S'_3 and S''_3, hence is satisfies S_3. The successful evaluation of S_3 evaluates all properties and items evaluated by S'_3 and by S''_3, hence it evaluates all items evaluated by $S_1 = \{$ "allOf" : $[S'_1, S''_1]\ \}$ and those evaluated by $S_2 = \{$ "allOf" : $[S'_2, S''_2]\ \}$.

(2) $L' +\!\!+ L'' +\!\!+ Close(L', L'')$ is cover-closed: every two elements of L' or of L'' have a cover in L' or in L'', by construction. Any two elements $S' \in L'$ and $S'' \in L''$ are either *covered* by an element of $L' +\!\!+ L''$ or by an element { "allOf" : $[S', S'']\ \}$ in $Close(L', L'')$, by the lower-bound condition of $Close$. Consider now $S_1 \in L'$ and $S_2 \in Close(L', L'')$, where $S_2 = \{$ "allOf" : $[S', S'']\ \}$, with $S' \in L'$ and $S'' \in L''$. S_1 and S' are *covered* by an element S'_1 of L_1; S'_1 and S'', by construction, are either *covered* by an element of $L' +\!\!+ L''$ or by an element { "allOf" : $[S'_1, S'']\ \}$ in $Close(L', L'')$. The same proof holds for $S_1 \in L''$ and $S_2 \in Close(L', L'')$. Finally, consider two elements of $Close(L', L'')$; by the upper-bound condition they can be written as { "allOf" : $[S'_1, S''_1]\ \}$ and { "allOf" : $[S'_2, S''_2]\ \}$. The pair (S'_1, S'_2) is *covered* by S'_3 in L', and the pair (S''_1, S''_2) is *covered* by S''_3 in L''. The pair (S'_3, S''_3) is either *covered* by an element of $L' +\!\!+ L''$ or by an element { "allOf" : $[S'_3, S''_3]\ \}$ in $Close(L', L'')$. This element is satisfied by any J that satisfies { "allOf" : $[S'_1, S''_1]\ \}$ and { "allOf" : $[S'_2, S''_2]\ \}$, and it evaluates all properties and items that are evaluated by these two terms.

(3) $Or(\mathcal{L})$ is cover-closed: by induction. Case $C(\llbracket\ \rrbracket)$: holds since the empty list is cover-closed. Case $Or(L :: \mathcal{L}) = L +\!\!+ Or(\mathcal{L}) +\!\!+ Close(L, Or(\mathcal{L}))$. The list $Or(\mathcal{L})$ is cover-closed by induction, and the thesis follows from the closure of $L' +\!\!+ L'' +\!\!+ Close(L', L'')$ (point 2).

(4) "anyOf" : $And(L_1, \ldots, L_n)$, is equivalent to "allOf" : [{ "anyOf" : L_1 }, ..., { "anyOf" : L_n }]: by induction on n.

Case $n = 0$ is trivial, since both sides are satisfied by any J.

Case $n + 1$: we want to prove that "anyOf" : $And(L_1, \ldots, L_{n+1})$, is equivalent to
"allOf" : [{ "anyOf" : L_1 }, ..., { "anyOf" : L_{n+1} }]:
We use $\mathcal{L}$ to indicate $\llbracket L_2, \ldots, L_{n+1} \rrbracket$, and we have:
$And(L_1, \ldots, L_{n+1}) = \llbracket$ { "allOf" : [S', S''] } $\mid$ $S' \in L$, $S'' \in And(\mathcal{L}) \rrbracket$, hence J satisfies
"anyOf" : $And(L_1, \ldots, L_{n+1})$ iff
$\exists S' \in L,\ S'' \in And(\mathcal{L})$. J satisfies S', J satisfies S', hence
"anyOf" : $And(L_1, \ldots, L_{n+1})$ is equivalent to
"allOf" : [{ "anyOf" : L_1 }, { "anyOf" : $And(\mathcal{L})$ }], equivalent, by induction, to:
"allOf" : [{ "anyOf" : L_1 }, { "allOf" : [{ "anyOf" : L_2 }, ..., { "anyOf" : L_{n+1} }] }], equivalent, by associativity, to:
"allOf" : [{ "anyOf" : L_1 }, { "anyOf" : L_2 }, ..., { "anyOf" : L_{n+1} }], qed.

(5) "anyOf" : $L' +\!\!+ L'' +\!\!+ Close(L', L'')$ is equivalent to "anyOf" : $L' +\!\!+ L''$. Every J that satisfies the right hand side is satisfied by the corresponding element of the left hand side. In the other directions, every J that satisfies an element of L' or L'' satisfies the corresponding element of the right hand side. If J satisfies an element { "allOf" : [S', S''] } of $Close(L', L'')$, then it satisfies both S' in L' and S'' in L'', hence it satisfies the left hand side.

(6) "anyOf" : $Or(L_1, \ldots, L_n)$, is equivalent to "anyOf" : [{ "anyOf" : L_1 }, ..., { "anyOf" : L_n }]: by induction on n.

Case $n = 0$ is trivial, since both sides are equal to "anyOf" : [].

Case $n + 1$: we want to prove that "anyOf" : $Or(L_1, \ldots, L_{n+1})$, is equivalent to
"anyOf" : [{ "anyOf" : L_1 }, ..., { "anyOf" : L_{n+1} }].

We let $\mathcal{L} = \llbracket L_2, \ldots, L_n \rrbracket$
"anyOf" : $Or(L_1, \ldots, L_{n+1})$ is, by definition:
"anyOf" : $(L_1 +\!\!+ Or(\mathcal{L}) +\!\!+ Close(L_1, Or(\mathcal{L})))$; by the Property (5), is equivalent to:
"anyOf" : $(L_1 +\!\!+ Or(\mathcal{L}))$; by associativity:
"anyOf" : $(L_1 +\!\!+$ { "anyOf" : $Or(\mathcal{L})$ }$)$; by induction:
"anyOf" : $(L_1 +\!\!+$ { "anyOf" : [{ "anyOf" : L_2 }, ..., { "anyOf" : L_{n+1} }] }$)$; we conclude by associativity.

(7) If each list L_i is cover-closed and, for any $i \neq j$, { "anyOf" : L_i } and { "anyOf" : L_j } are disjoint, then $L_1 +\!\!+ \ldots +\!\!+ L_n$ is cover-closed.

The proof is immediate: given two schemas S_1 and S_2 from $L_1 +\!\!+ \ldots +\!\!+ L_n$, if they belong to the same list L_i, then they are *covered* by a schema in L_i, since L_i is cover-closed. If they belong to two different lists L_i and L_j than there exists no J that satisfies both S_1 and S_2 since that J would satisfy both { "anyOf" : L_i } and { "anyOf" : L_j }, contradicting the hypothesis that they are disjoint, hence S_1 and S_2 are disjoint hence the pair (S_1, S_2) is trivially *covered* by S_1 (and by S_2 as well). □

Property 7. *For any S, $ENF(S)$ is equivalent to S and is in ENF.*

Proof.

The cases when $exEP(S)\downarrow$ and $exEI(S)\downarrow$ are trivial. All other cases are proved by induction on the in-place depth.

Case { $K_1, \ldots, K_m$ } with $m > 1$:
equivalence: by definition, { $K_1, \ldots, K_m$ } is equivalent to { "allOf" : [{ K_1 }, ..., { K_m }] }; by induction, this is equivalent to { "allOf" : [{ "anyOf" : L_1 }, ..., { "anyOf" : L_m }] }, by Property 6 (4) this is equivalent to { "anyOf" : $And(L_1, \ldots, L_m)$ }.
ENF: by the inductive hypothesis, every L_i is elementwise statically characterized and is cover-closed, hence, by Property 6 (1), $And(L_1, \ldots, L_m)$ is elementwise statically characterized and is cover-closed, hence { "anyOf" : $And(L_1, \ldots, L_m)$ } is in ENF.

Case { "allOf" : [$S_1, \ldots, S_m$] }: essentially the same.

Case { "anyOf" : [$S_1, \ldots, S_m$] }: the same, using Properties 6 (6) and 6 (3).

Case { "oneOf" : [$S_1, \ldots, S_m$] }:
equivalence: by definition, { "oneOf" : [$S_1, \ldots, S_m$] } is equivalent to { "anyOf" : [$S'_1, \ldots, S'_m$] } where $S'_i =$ { "allOf" : [{ "not" : S_1 }, ..., { "not" : S_{i-1} }, S_i, { "not" : S_{i+1} }, ..., { "not" : S_m }] }; by induction, this is equivalent to { "anyOf" : [{ "anyOf" : L_1 }, ..., { "anyOf" : L_m }] }, by associativity, this is equivalent to { "anyOf" : $L_1 +\!\!+ \ldots +\!\!+ L_m$ }.
ENF: by the inductive hypothesis, every L_i is elementwise statically characterized and is cover-closed, hence, by Property 6 (7), $L_1 +\!\!+ \ldots +\!\!+ L_m$ is elementwise statically characterized and is cover-closed, hence { "anyOf" : $L_1 +\!\!+ \ldots +\!\!+ L_m$ } is in ENF.

Case { "$ref" : u }: equivalence: by definition, { "$ref" : u } is equivalent to $deref(u)$, which is equivalent to $ENF(deref(u))$ by induction on the in-place depth. ENF: $ENF(deref(u))$ is in ENF by induction on the in-place depth. □

During the normalization process, we eliminate all duplicates from the arguments of all "allOf" and "anyOf" keywords that we build; this allows us to ensure that the final size of the schema that we get cannot grow more then $O(2^{|S|})$.

Property 8. *$ENF(S)$ generates a schema whose size is in $O(2^{|S|})$.*

Proof. Consider a closed schema S of size N. For every S' that is a subschema of S, we say that S' is a source-node for S, and that { "not" : S' } is a source-node for S, so that the number of different source-nodes for S is $O(N)$. For every n-tuple $S_1, \ldots, S_n$ of source-nodes, we say that { "allOf" : [$S_1, \ldots, S_n$] } is a source-conjunction, and we also consider any source-node S' as a source-conjunction.

We will also assume that, whenever our algorithm computes the conjunction of two source-conjunctions, it flattens them, so that the result is still a source-conjunction.

We prove that:

when S is a source-conjunction, then $ENF(S) = \{\ \texttt{"anyOf"} : L\ \}$, where L is a list of source-conjunctions.

We will then use this fact in order to establish a size bound.

Case (1): we have $S = \{\ \texttt{"anyOf"} : L\ \}$ and $ENF(S) = S$. By hypothesis, S is a source conjunction; since it is neither a conjunction nor a negation, then it is a subschema of the initial schema, hence L is a list of subschemas, hence L is a list of source-conjunctions.

Case (2): here the result is $\{\ \texttt{"anyOf"} : [S]\ \}$, where the only element of the list $[S]$ is S, which is a source-conjunction by hypothesis.

Cases for $exEP(S)\uparrow$ or $exEI(S)\uparrow$ are proved by induction on the in-place depth, and by cases, as follows.

$$\begin{aligned}(3)\ ENF(\{\ K_1, \ldots, K_n\ \}) \quad &= \quad \{\ \texttt{"anyOf"} : And(\llbracket\ L_1, \ldots, L_m\ \rrbracket)\ \} \\ &\quad\ where\ ENF(\{\ K_i\ \}) = \{\ \texttt{"anyOf"} : L_i\ \}\ \ for\ i \in 1..m\end{aligned}$$

By induction, every L_i is a list of source-conjunctions. The result follows since *And* returns a list where every element is a conjunction of elements from the input lists, and the conjunction of many source-conjunctions is still a source-conjunction.

$$\begin{aligned}(4)\ ENF(\{\ \texttt{"allOf"} : [S_1, \ldots, S_m]\ \}) \quad &= \quad \{\ \texttt{"anyOf"} : And(\llbracket\ L_1, \ldots, L_m\ \rrbracket)\ \} \\ &\quad\ where\ ENF(S_i) = \{\ \texttt{"anyOf"} : L_i\ \}\ \ for\ i \in 1..m\end{aligned}$$

Since $\{\ \texttt{"allOf"} : [S_1, \ldots, S_m]\ \}$ is a source-conjunction, then every S_i is a source-node of the original schema, hence every S_i is a source-conjunction, hence, by induction, every L_i is a list of source-conjunctions. The result follows since *And* builds all elements of its result as conjunctions of the elements of the input lists.

$$\begin{aligned}(5)\ ENF(\{\ \texttt{"anyOf"} : [S_1, \ldots, S_m]\ \}) \quad &= \quad \{\ \texttt{"anyOf"} : Or(\llbracket\ L_1, \ldots, L_m\ \rrbracket)\ \} \\ &\quad\ where\ ENF(S_i) = \{\ \texttt{"anyOf"} : L_i\ \}\ \ for\ i \in 1..m\end{aligned}$$

Since $\{\ \texttt{"anyOf"} : [S_1, \ldots, S_m]\ \}$ is a source-conjunction that is neither a "not" nor a "allOf" keyword, then it is a subschema of the original schema, hence every S_i is a subschema of the original schema, hence it is a source-conjunction, hence, by induction, every L_i is a list of source-conjunctions. The result follows since *Or* concatenates elements that are in the input lists with conjunctions of such elements.

$$\begin{aligned}(6)\ ENF(\{\ \texttt{"oneOf"} : [S_1, \ldots, S_m]\ \}) \quad &= \quad \{\ \texttt{"anyOf"} : L_1 +\!\!+ \ldots +\!\!+ L_m\ \} \\ &\quad\ where\ ENF(\{\texttt{"allOf"} : [\{\ \texttt{"not"} : S_1\ \}, \ldots, \{\ \texttt{"not"} : S_{i-1}\ \}, S_i, \\ &\qquad \{\ \texttt{"not"} : S_{i-1}\ \}, \ldots, \{\ \texttt{"not"} : S_m\ \}]\}) = \{\ \texttt{"anyOf"} : L_i\ \} \\ &\quad\ for\ i \in 1..m\end{aligned}$$

As in case (5), every S_i is a subschema of the original schema, hence every

$$\{\texttt{"allOf"} : [\{\ \texttt{"not"} : S_1\ \}, \ldots, \{\ \texttt{"not"} : S_{i-1}\ \}, S_i, \boxminus\{\ \texttt{"not"} : S_{i-1}\ \}, \ldots, \{\ \texttt{"not"} : S_m\ \}]\}$$

is a source-conjunction, hence we can apply induction on the in-place depth and conclude that every L_i is a source-conjunction; list concatenation preserves this property.

$$(7)\ ENF(\{\ \texttt{"\$ref"} : u\ \}) \quad = \quad ENF(deref(u)))$$

By induction on in-place depth, $ENF(deref(u))$ returns a schema with shape $\{\ \texttt{"anyOf"} : L\ \}$ where L is a list of source-conjunctions.

Hence, the algorithm returns a schema $\{\ \texttt{"anyOf"} : [C_1, \ldots, C_n]\ \}$, such that every C_i is a source conjunction $C_i = \{\ \texttt{"allOf"} : [S^i_1, \ldots, S^i_{m_i}]\ \}$. By associativity, commutativity, and idempotence of "allOf", we can systematically eliminate all duplicates from the arguments of the "allOf" of C_i. If we define a named schema for each source-node, every source-conjunction can be represented as a list of N references or negated references, where N is the number of subschemas of the initial schema[13].

When we manipulate a list L of source-conjunctions, in every step of our algorithm we can substitute it with a sublist L' provided that $\texttt{"anyOf"} : L'$ is equivalent to $\texttt{"anyOf"} : L$ (this can be proved by induction). For this reason, whenever a list L of source-conjunctions contains two source-conjunctions $\texttt{"allOf"} : [S_1, \ldots, S_n]$ and $\texttt{"allOf"} : [S'_1, \ldots, S'_n]$ which are pseudo-duplicates, that is, which only differ in the order of their elements, we can eliminate one of the two. Hence, we can ensure that our algorithm only manipulates lists of source-conjunctions such that the source-conjunctions contain no duplicates and the list contains no pseudo-duplicates. Since we only have 2^N different source-conjunctions — if we regard pseudo-duplicates as equal — every list of conjunctions that we manipulate contains less then 2^N source-conjunctions, hence has size smaller than $2^N \times N$. Hence, for every schema S smaller than N, the size of $ENF(S)$ is smaller than $2^N \times N$. □

6.6. *Eliminating* "unevaluatedProperties" *and* "unevaluatedItems"

We can finally use the function $ENF(S)$ to define a function $Elim(S)$ that eliminates the keywords "unevaluatedProperties" and "unevaluatedItems" from every schema S.

[13] just N, and not $2N$, since when both a reference and its negation are present we can rewrite the conjunction as *false*

We say that a schema that contains "unevaluated*" : S_u as a top-level keyword is an uneval-schema; we apply $Elim(S)$, hence $ENF(S)$, to every uneval-schema. In order to keep the size of the result in $O(2^N)$, we do not want to apply ENF to a subschema S', get an exponential blow-up, and then apply ENF again to a schema that contains the result of $ENF(S')$ as a subschema; for this reason, we do not want to have an uneval-schema nested inside another uneval-schema. For this reason, as a preliminary step, we *unnest* the source schema by (1) defining a new definition $u_{S'} : S'$ for each occurrence of an uneval-subschema S' of S that is not a named schema in the "$defs" section and (2) by substituting that occurrence with "$ref" : $u_{S'}$. Hence, after unnesting, every uneval-schema is a named schema in the "$defs" section, which does not contain any other uneval-schema, but only references to such schemas. The unnested schema has size $O(N)$, where N is the size of the schema before unnesting.

We introduce the algorithm through an example, before the formal definition of the involved functions $Elim(S)$, $PushUnProps(S_u, \_)$, $PUPBranch(S_u, \_)$. Consider the following schema.

```
{ "items": { "anyOf":  [ { "$ref": "#sale" } , { "$ref": "#car" } ],
             "unevaluatedProperties": false },
  "$defs": {
    "sale": { "$anchor": "sale", "properties": { "price": { "type": "integer" }}},
    "car":  { "$anchor": "car",  "properties": { "plate": { "type": "string" }}}
  }
}
```

After unnesting, we obtain the following schema, where the only uneval-schema is the named schema "#/$defs/r".

```
{ "items": { "$ref":  "#/$defs/r" },
  "$defs": {
    "sale": { "$anchor": "sale", "properties": { "price": { "type": "integer" }}},
    "car":  { "$anchor": "car",  "properties": { "plate": { "type": "string" }}},
    "r":    { "anyOf":  [ { "$ref": "#sale" } , { "$ref": "#car" } ],
              "unevaluatedProperties": false }
  }
}
```

After the schema has been unnested, we apply $Elim(S)$ to every named schema; if the schema has a shape $\{ K_1, \dots, K_n, \text{"unevaluated*"} : S_u \}$, $Elim(S)$ computes $ENF(\{ K_1, \dots, K_n \})$, and then pushes S_u through all branches of the ENF using $PushUnProps(S_u, ENF(\{ K_1, \dots, K_n \}))$.

For example, in this case, the ENF of the { "anyOf" : A } part of the schema identified by "#/$defs/r" would produce a "anyOf" schema with three arguments:

$$
\begin{aligned}
&Elim(\{\text{"anyOf"} : [\{\text{"\$ref"} : \text{"\#sale"}\}, \{\text{"\$ref"} : \text{"\#car"}\}], \text{"unevaluatedProperties"} : \texttt{false}\}) \\
&= PushUnProps(\texttt{false}, ENF(\{\text{"anyOf"} : [\{\text{"\$ref"} : \text{"\#sale"}\}, \{\text{"\$ref"} : \text{"\#car"}\}]\})) \\
&= PushUnProps(\texttt{false}, \{\text{"anyOf"} : [\ \{\text{"\$ref"} : \text{"\#sale"}\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\text{"\$ref"} : \text{"\#car"}\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\text{"allOf"} : [\{\text{"\$ref"} : \text{"\#sale"}\}, \{\text{"\$ref"} : \text{"\#car"}\}]\} \\
&\qquad\qquad\qquad\qquad\qquad]\})
\end{aligned}
$$

At this point, the function $PushUnProps(S_u, \{\text{"anyOf"} : [S_1, \dots, S_n]\})$ uses $PUPBranch(S_u, S_i)$ to push S_u to each branch S_i of the ENF. $PUPBranch(S_u, S_i)$ returns

$$\{\text{"allOf"} : [S_i, \{\text{"patternProperties"} : \{ p_1 : \{\}, \dots, p_n : \{\} \}, \text{"additionalProperties"} : S_u \}]\}$$

where "patternProperties" : { p_1 : {}, … , p_n : {} } cites all properties of $exEP(S_i)$, so that the keyword "additionalProperties" : S_u applies S_u to the other properties.

$$
\begin{aligned}
&PushUnProps(\texttt{false}, \{\texttt{"anyOf"} : [\quad \{\texttt{"\$ref"} : \texttt{"\#sale"}\},\\
&\qquad\qquad\qquad\qquad\qquad\quad \{\texttt{"\$ref"} : \texttt{"\#car"}\},\\
&\qquad\qquad\qquad\qquad\qquad\quad \{\texttt{"allOf"} : [\{\texttt{"\$ref"} : \texttt{"\#sale"}\}, \{\texttt{"\$ref"} : \texttt{"\#car"}\}]\}\\
&\qquad\qquad\qquad\qquad\quad]\})\\
&= \{\texttt{"anyOf"} : [\quad PUPBranch(\texttt{false}, \{\texttt{"\$ref"} : \texttt{"\#sale"}\}),\\
&\qquad\qquad\quad PUPBranch(\texttt{false}, \{\texttt{"\$ref"} : \texttt{"\#car"}\}),\\
&\qquad\qquad\quad PUPBranch(\texttt{false}, \{\texttt{"allOf"} : [\{\texttt{"\$ref"} : \texttt{"\#sale"}\}, \{\texttt{"\$ref"} : \texttt{"\#car"}\}]\})\\
&\qquad\quad]\}
\end{aligned}
$$

```
= {"anyOf" : [
    {"allOf" :   [{"$ref" : "#sale"},
                  {"patternProperties"{"price" : {} },"additionalProperties" : false }]},
    {"allOf" :   [{"$ref" : "#car"},
                  {"patternProperties"{"plate" : {} },"additionalProperties" : false }]},
    {"allOf" :   [{"allOf" : [{"$ref" : "#sale"}, {"$ref" : "#car"}] },
                  { "patternProperties"{"price" : {},"plate" : {} },
                    "additionalProperties" : false}]}
              ]}
```

We now give the formal definition of $PushUnProps(S_u, S)$ and $PUPBranch(S_u, S)$; the full definition of $Elim(S)$ is only in Definition 17, since we have to discuss arrays before.

Definition 14 ($pProps(\{|\, p_1, \dots, p_n \,|\})$). $pProps(\{|\, p_1, \dots, p_n \,|\})$ stands for "patternProperties" : { p_1 : { }, … , p_n : { } }.

Definition 15 ($PushUnProps(S_u, S)$).

Let $S = \{\texttt{"anyOf"} : [S_1, \dots, S_n]\}$ be a schema where every S_i is statically characterized; $PushUnProps(S_u, S)$ and $PUPBranch(S_u, S_i)$ are defined as follows:

$$
\begin{aligned}
&PUPBranch(S_u, S_i) \quad = \quad \{\texttt{"allOf"} : [S_i, \{\, pProps(exEP(S_i)), \texttt{"additionalProperties"} : S_u \,\}]\}\\
&PushUnProps(S_u, \{\texttt{"anyOf"} : [S_1, \dots, S_n]\})\\
&\qquad\qquad = \quad \{\texttt{"anyOf"} : [\, PUPBranch(S_u, S_1), \dots, PUPBranch(S_u, S_n)\,]\,\}
\end{aligned}
$$

The approach to push and eliminate "unevaluatedItems" is very similar. In this case, we compute, for each branch S_i of the ENF, the two components (h, S_e) of $exEI(S_i)$, which we indicate, respectively, with $exEI_1(S_i)$ and $exEI_2(S_i)$. We use $prefIts(h)$ to indicate a keyword "prefixItems" : [$\texttt{true}^1, \dots, \texttt{true}^h$] that evaluates the first h items of an array. Hence the following schema S^i ensures that all items after $exEI_1(S_i)$ that do not satisfy $exEI_2(S_i)$ satisfy S_u:

$$S^i \;=\; \{\, prefIts(exEI_1(S_i)), \texttt{"items"} : \{\texttt{"anyOf"} : [S_u, exEI_2(S_i)]\}\,\}$$

Hence, we can eliminate "unevaluatedItems" : S from a schema in ENF by adding S^i to each S_i conjunct of the ENF "anyOf" : $[S_1, \dots, S_n]$ as follows, provided that $exEI_1(S_i)$ is finite; when $exEI_1(S_i) = \infty$, or $exEI_2(S_i) = \texttt{true}$, then all items of any array are evaluated, and, hence, $S^i = \texttt{true}$, and $PUIBranch(S_u, S_i) = S_i$: when all items have been evaluated, then "unevaluatedItems" : S_u is trivially satisfied.

Definition 16 ($PushUnItems(S_u, S)$).

Let $S = \{\texttt{"anyOf"} : [S_1, \dots, S_n]\}$ be a schema where every S_i is statically characterized; $PushUnItems(S_u, S)$ and $PUIBranch(S_u, S_i)$ are defined as follows:

$$
\begin{aligned}
&exEI_1(S_i) = \infty \vee exEI_2(S_i) = \texttt{true} :\\
&\quad PUIBranch(S_u, S_i) = \quad S_i\\
&exEI_1(S_i) \neq \infty \;\wr\; exEI_2(S_i) \neq \texttt{true} :\\
&\quad PUIBranch(S_u, S_i)\\
&\qquad\quad = \{\texttt{"allOf"} : [\, S_i, \{\, prefIts(exEI_1(S_i)), \texttt{"items"} : \{\texttt{"anyOf"} : [S_u, exEI_2(S_i)]\}\,\}\,]\,\}\\
&PushUnItems(S_u, \{\texttt{"anyOf"} : [S_1, \dots, S_n]\})\\
&\qquad\quad = \{\texttt{"anyOf"} : [\, PUIBranch(S_u, S_1), \dots, PUIBranch(S_u, S_n)\,]\,\}
\end{aligned}
$$

We are finally ready for the formal definition of $Elim(S)$. The function $Elim(S)$ does not need to recursively descend S since, thanks to unnesting, all the occurrences of "unevaluated*" : S_u are at the top-level of a named schema.

Definition 17 ($Elim(S)$). $Elim(S)$ is defined by the first rule that can be applied to S among the four rules that follow.

$$
\begin{aligned}
&Elim(\{\, K_1, \dots, K_n, \texttt{"unevaluatedProperties"} : S_p, \texttt{"unevaluatedItems"} : S_i\,\})\\
&\quad = \{\texttt{"allOf"}:[\, PushUnProps(S_p, ENF(\{\, K_1, \dots, K_n\,\})), PushUnItems(S_i, ENF(\{\, K_1, \dots, K_n\,\}))\,]\,\}\\
&Elim(\{\, K_1, \dots, K_n, \texttt{"unevaluatedProperties"} : S_p\,\}) = PushUnProps(S_p, ENF(\{\, K_1, \dots, K_n\,\}))\\
&Elim(\{\, K_1, \dots, K_n, \texttt{"unevaluatedItems"} : S_i\,\}) = PushUnItems(S_i, ENF(\{\, K_1, \dots, K_n\,\}))\\
&Elim(S) \quad = \quad S
\end{aligned}
$$

We now prove that $Elim(S)$ is equivalent to S and eliminates all instances of "unevaluated*".

Property 9.

1. *For any S in ENF, J satisfies $PushUnProps(S_u, S)$ iff it satisfies S and all the properties that are not evaluated by S satisfy S_u.*
2. *For any S in ENF, J satisfies $PushUnItems(S_u, S)$ iff it satisfies S and all the items that are not evaluated by S satisfy S_u.*
3. *For any J and S, J satisfies $Elim(S)$ iff J satisfies S.*

Proof. In this proof, we say that a schema S *p-covers* $\mathcal{S}$, where $\mathcal{S}$ is a non-empty set of schemas, iff: (1) every J that satisfies every schema in $\mathcal{S}$ also satisfies S and (2) every property that is evaluated by a schema in $\mathcal{S}$ is also evaluated by S. We prove by induction that if a set of schemas $\mathcal{S}$ is cover-closed, then any subset $\mathcal{S}'$ of $\mathcal{S}$ is *p-covered* by a schema in $\mathcal{S}$. Case $|\mathcal{S}'| = 1$ is trivial. Case $\mathcal{S}' = \{\!|\, S_1, \ldots, S_{n+1} \,|\!\}$: by induction, $\{\!|\, S_1, \ldots, S_n \,|\!\}$ is *p-covered* by an element S_l of $\mathcal{S}$, and, since $\mathcal{S}$ is cover-closed, there is an element S_m of $\mathcal{S}$ that *p-covers* the pair (S_l, S_{n+1}); we prove that S_m *p-covers* $\mathcal{S}'$: every J that satisfies all elements of $\mathcal{S}'$ satisfies both S_l and S_{n+1}, hence it satisfies S_m; if a property is evaluated by an element of $\mathcal{S}'$ then it is evaluated either by S_l or by S_{n+1}, hence it is evaluated by S_m. The same definition and property holds for the notion of *i-covered*. Now we proceed with the proof.

(1) For any S in ENF, J satisfies $PushUnProps(S_u, S)$ iff it satisfies S and all the properties that are not evaluated by S satisfy S_u.

$$
\begin{aligned}
&PushUnProps(S_u, \{\,\texttt{"anyOf"} : [S_1, \ldots, S_n]\,\})\\
&\quad = \{\texttt{"anyOf"} : \quad [\quad \{\,\texttt{"allOf"} : [S_1, \{\, pProps(exEP(S_1)), \texttt{"additionalProperties"} : S_u\,\}]\,\},\\
&\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad \{\,\texttt{"allOf"} : [S_n, \{\, pProps(exEP(S_n)), \texttt{"additionalProperties"} : S_u\,\}]\,\}\\
&\qquad\qquad]\}
\end{aligned}
$$

($\Rightarrow$). If J satisfies $PushUnProps(S_u, S)$, where $S = \{\,\texttt{"anyOf"} : [S_1, \ldots, S_n]\,\}$, then exists l such that it satisfies $\{\,\texttt{"allOf"} : [S_l, \ldots]\,\}$, hence it satisfies S_l, hence it satisfies S. If a property p is not evaluated by S then it is not evaluated by the branch S_l, hence, by Property 3, this property does not belong to $exEP(l)$, hence, since J satisfies $\{\, pProps(exEP(S_l)), \texttt{"additionalProperties"} : S_u\,\}$ we conclude that p satisfies S_u.

($\Leftarrow$). Let us assume that J satisfies S and that all the properties of J that are not evaluated by S satisfy S_u. Let $S = \{\,\texttt{"anyOf"} : [S_1, \ldots, S_n]\,\}$ and let $\mathcal{S}$ be the set of all schemas S_i such that J satisfies S_i. Since S is in ENF, there exists $S_l \in \{\!|\, S_1, \ldots, S_n \,|\!\}$ such that S_l *covers* $\mathcal{S}$. Since J satisfies all schemas in $\mathcal{S}$ and S_l *covers* $\mathcal{S}$, we deduce that J satisfies S_l (that is, $S_l \in \mathcal{S}$). Each property that is evaluated by S is evaluated by some $S_i \in \mathcal{S}$, hence, since S_l *covers* $\mathcal{S}$, it is also evaluated by S_l, hence, by Property 3, each evaluated property matches $exEP(S_l)$. Hence, every property that does not match $exEP(S_l)$ is not evaluated by S hence, by hypothesis, satisfies S_u. Hence, J satisfies $\{\, pProps(exEP(S_l)), \texttt{"additionalProperties"} : S_u\,\}$. Since it also satisfies S_l, then J satisfies the branch $\{\,\texttt{"allOf"} : [S_l, \{\,\ldots\,\}]\,\}$, of $PushUnProps(S_u, S)$, hence it satisfies $PushUnProps(S_u, S)$.

(2) For any S in ENF, J satisfies $PushUnItems(S_u, S)$ iff it satisfies S and all the items that are not evaluated by S satisfy S_u.

$$
\begin{aligned}
&PushUnItems(S_u, \{\,\texttt{"anyOf"} : [S_1, \ldots, S_n]\,\})\\
&= \{\texttt{"anyOf"} : \quad [\quad \{\,\texttt{"allOf"} : [S_1, \{\, prefIts(exEI_1(S_1)), \texttt{"items"} : \{\,\texttt{"anyOf"} : [exEI_2(S_1), S_u]\,\}\,\}]\,\},\\
&\qquad\qquad\quad \ldots\\
&\qquad\qquad\quad \{\,\texttt{"allOf"} : [S_n, \{\, prefIts(exEI_1(S_n)), \texttt{"items"} : \{\,\texttt{"anyOf"} : [exEI_2(S_n), S_u]\,\}\,\}]\,\}\\
&\qquad\quad]\}
\end{aligned}
$$

($\Rightarrow$). If J satisfies $PushUnItems(S_u, S)$, where $S = \{\,\texttt{"anyOf"} : [S_1, \ldots, S_n]\,\}$, then exists l such that J satisfies $\{\,\texttt{"allOf"} : [S_l, \ldots]\,\}$, hence J satisfies S_l, hence J satisfies S. If a item J_i of J is not evaluated by S then it is not evaluated by the branch S_l, hence, by Property 3, this item does not satisfy neither $exEI_1(l)$ nor $exEI_2(l)$, that is, its position is $> exEI_1(l)$ and it does not satisfy $exEI_2(l)$, hence, since J satisfies $\{\, prefIts(exEI_1(S_1)), \texttt{"items"} : \{\,\texttt{"anyOf"} : [exEI_2(S_1), S_u]\,\}\,\}$, we conclude that J_i satisfies S_u.

($\Leftarrow$). Let us assume that J satisfies S and that all the items of J that are not evaluated by S satisfy S_u. Let $S = \{\,\texttt{"anyOf"} : [S_1, \ldots, S_n]\,\}$ and let $\mathcal{S}$ be the set of all schemas S_i such that J satisfies S_i. Since S is in ENF, there exists $S_l \in \{\!|\, S_1, \ldots, S_n \,|\!\}$ such that S_l *covers* $\mathcal{S}$. Since J satisfies all schemas in $\mathcal{S}$ and S_l *covers* $\mathcal{S}$, we deduce that J satisfies S_l. Each item that is evaluated by S is evaluated by some $S_i \in \mathcal{S}$, hence, since S_l *covers* $\mathcal{S}$, it is also evaluated by S_l, hence, by Property 3, this item satisfies $exEI(S_l)$. Hence, every item that does not satisfy $exEI(S_l)$ is not evaluated by S hence, by hypothesis, this item satisfies S_u; we use (*) to refer to this fact. We show that J satisfies $\{\, prefIts(exEI_1(S_l)), \texttt{"items"} : \{\,\texttt{"anyOf"} : [exEI_2(S_l), S_u]\,\}\,\}$: if an item satisfies $exEI_1(S_l)$, then it is not examined by "items"; it an item satisfies $exEI_2(S_l)$ then it satisfies $\{\,\texttt{"anyOf"} : [exEI_2(S_l), \ldots]\,\}$, hence it passes the "items" test, and if an item does not satisfy neither $exEI_1(S_l)$ nor $exEI_2(S_l)$, then it does not satisfy $exEI(S_l)$, hence it satisfies S_u (by *), else it satisfies $\{\,\texttt{"anyOf"} : [\ldots, S_u]\,\}$, hence also this item passes the "items" test, hence J satisfies $\{\, prefIts(exEI_1(S_1)), \texttt{"items"} : \{\,\texttt{"anyOf"} : [exEI_2(S_1), S_u]\,\}\,\}$. Since J also satisfies S_l, then J satisfies the branch $\{\,\texttt{"allOf"} : [S_l, \{\,\ldots\,\}]\,\}$ of $PushUnItems(S_u, S)$, hence it satisfies $PushUnItems(S_u, S)$.

(3) For any J and S, J satisfies $Elim(S)$ iff J satisfies S.

$Elim(S)$ is defined by the first rule that can be applied to S among the four rules that follows.

$$
\begin{aligned}
&Elim(\{\, K_1, \ldots, K_n, \texttt{"unevaluatedProperties"} : S_p, K_n, \texttt{"unevaluatedItems"} : S_i\,\})\\
&= \{\,\texttt{"allOf"} : [PushUnProps(S_p, ENF(\{\, K_1, \ldots, K_n\,\})), PushUnItems(S_i, ENF(\{\, K_1, \ldots, K_n\,\}))]\,\}\\
&Elim(\{\, K_1, \ldots, K_n, \texttt{"unevaluatedProperties"} : S_p\,\}) = PushUnProps(S_p, ENF(\{\, K_1, \ldots, K_n\,\}))\\
&Elim(\{\, K_1, \ldots, K_n, \texttt{"unevaluatedItems"} : S_i\,\}) = PushUnItems(S_i, ENF(\{\, K_1, \ldots, K_n\,\}))\\
&Elim(S) \quad = \quad S
\end{aligned}
$$

The first three cases are immediate consequences of cases (1)+ Property 7 and (2)+Property 7. The last case is trivial. □

Property 10. *A closed schema S that has been unnested and where every named schema S has been substituted with $Elim(S)$, does not contain any residual instance of* "unevaluated*".

Proof. $Elim(S)$ is defined by the first rule that can be applied to S among the four rules that follows.

$$
\begin{aligned}
&Elim(\{\, K_1, \ldots, K_n, \texttt{"unevaluatedProperties"} : S_p, K_n, \texttt{"unevaluatedItems"} : S_i \,\}) \\
&= \{\, \texttt{"allOf"} : [PushUnProps(S_p, ENF(\{\, K_1, \ldots, K_n \,\})), PushUnItems(S_i, ENF(\{\, K_1, \ldots, K_n \,\}))] \,\} \\
&Elim(\{\, K_1, \ldots, K_n, \texttt{"unevaluatedProperties"} : S_p \,\}) = PushUnProps(S_p, ENF(\{\, K_1, \ldots, K_n \,\})) \\
&Elim(\{\, K_1, \ldots, K_n, \texttt{"unevaluatedItems"} : S_i \,\}) = PushUnItems(S_i, ENF(\{\, K_1, \ldots, K_n \,\})) \\
&Elim(S) \quad = \quad S
\end{aligned}
$$

After all schemas are unnested, neither the keywords K_i nor the schemas S and S_u contain any instance of "unevaluated*". We then prove by induction in the in-place depth that the application of ENF to a schema $\{\, K_1, \ldots, K_n \,\}$ that contains no instance of "unevaluated*" returns a schema with the same property, and using the fact that the list manipulation operators, $And\mathcal{L}$, $And\mathcal{L}$, and $L' +\!\!+ L''$, do not insert in the result any keyword that was not present in the parameters, with the only exception of "allOf". Finally, we observe that, when S_u and S_{ENF} contain no instance of "unevaluated*", then the same property holds for $PushUnProps(S_u, S_{ENF})$ and for $PushUnItems(S_u, S_{ENF})$. □

7. Experimental evaluation

7.1. Implementation and execution environment

Our research prototype implements our unevaluated ∗ elimination algorithm and is written in Scala 2.13. The dispatcher and evaluation scripts are written in Bash and Python 3.10. The experiments were executed in a Docker container running Ubuntu 22.0.4. We configured a maximum heap space of 10 GB for the Java Virtual Machine (JVM). Our execution platform is a server with two 20-core Intel Xeon Gold 6242R 3,1 GHz processors and 192 GB of RAM.

7.2. Research hypotheses

We test the following hypotheses using our research prototype:

- H0 / Correctness of the implementation: Schemas before/after unevaluated ∗ elimination are equivalent.
- H1 / Runtime: Given real-world schemas, the implementation has acceptable runtime, despite the exponential worst-case lower bound of the unevaluated ∗ elimination algorithm.
- H2 / Size blow-up: Given real-world schemas, the blow-up in size is still reasonable, despite the exponential worst-case lower bound of the unevaluated ∗ elimination algorithm.

H0 is not really a research hypothesis, it is rather an hypothesis about the quality of our code. We describe its verification in this section for two reasons. First of all, the reliability of the results regarding H1 and H2 depends on the reliability of the code; therefore, we find it appropriate to describe here how the code has been tested. Second, because, in order to test for H0, we developed a corpus of schemas and witnesses, and these artifacts may be reused by other research groups.

7.3. Software tools

In our experiments, we use several external tools: JSON Schema validators, a JSON Schema data generator, and a tool for equivalence checking of JSON Schema. We provide a brief overview of these tools here. In Section 7.5, we explain their role in our experiments.

Validators. JSON Schema validators assess the validity of a JSON instance against a JSON Schema. A plethora of JSON Schema validators is available which unfortunately do not agree on some specific tests. To solve this issue, we employ Bowtie [31], a framework that offers a uniform interface to different validators, to systematically compare the results of 22 validators that are integrated into the tool and that support Modern JSON Schema. We discovered that some of them never failed on our test cases, while others would fail on different cases. Hence, we decided to run each test with the 22 validators and adopt the majority result. In practice, the result is the same as what we would get by choosing one of the robust tools, but the approach protects us against overconfidence on a single tool.

Data generator. JSON data generators synthesize instances from a structural description, such as JSON Schema. We use JSON Generator [17] (version 0.4.7), a tool which employs a heuristic approach to randomly create instances that conform to the provided JSON Schema. These instances are validated, and the program terminates once a valid instance is found. While JSON Generator aims to generate valid instances, the integrated validator may occasionally produce false positives, by creating invalid instances. We leverage this behavior to generate both valid and invalid instances, assigning a ground truth (valid/invalid with respect to the schema) to each using the Bowtie framework, as described above.

Table 4
Schema collections with number and size of schemas, and with the number of valid and invalid instances which were either generated using the data generator (*Random*), or which were written by hand (*Manual*).

Collection	#Schemas	Average Size (KB)	Maximum Size (KB)	#Total Instances	**#Valid** Instances	**#Invalid** Instances	Generation Method
GitHub	305	63.21	429.90	1347	515	832	*Random*
Test Suite	65	0.34	1.58	182	99	83	*Manual*
andwritten	60	2.27	8.10	387	138	249	*Manual*

Equivalence checking. To check the equivalence between the manually eliminated schemas and the eliminated schemas produced by our implementation, as we will describe in Section 7.5, we rely on the JSON Schema witness generation tool described by Attouche et al. [9]. While it can be quite slow, this tool was shown to be very accurate, and thus serves as a reliable indicator of the correctness of our implementation.

7.4. Schema collections and instances

We analyze our approach using a diverse set of schemas and instances, consisting of real-world examples and of carefully handcrafted tests. Table 4 describes these collections.

GitHub. We systematically crawled schemas from GitHub using the GitHub code search API. Specifically, we downloaded all JSON files that contain the properties "`unevaluatedProperties`" or "`unevaluatedItems`". We removed duplicates as well as any ill-formed JSON Schemas. We further removed 34 schemas that use features that are not yet supported by our implementation: 24 schemas referencing nested definitions, 6 schemas using "`dependentSchemas`", 3 schema using "`$id`" and "`$anchor`" keywords, 1 schema expressing dynamic references through "`$dynamicRef`" and "`$dynamicAnchor`". Notably, these are not inherent limitations of our approach, but simply deliberate restrictions of our research prototype. A manual inspection of the 305 remaining schemas revealed that around 20 % are toy examples or were designed to test validators (such as the JSON Schema Test Suite [32]), while the remaining schemas seem to be used in practical, real-world applications.

We generated valid and invalid instances using the *JSON Generator* [17] data generator, as explained in Section 7.3. Therefore, we consider most of the schemas to constitute a realistic snapshot of real-world schemas that developers use, while all instances are artificial.[14]

Test suite. The JSON Schema Test Suite [32] is a community-curated collection of schemas with valid and invalid instances designed to benchmark validators. The schemas and instances are handcrafted to broadly cover the JSON Schema language and are very small, following a unit-test style. The collection contains 70 schemas with "`unevaluated*`" keywords. As with the GitHub collection, we removed 5 schemas because of features that are not supported by our prototype: 3 schemas expressing dynamic references through "`$dynamicRef`" and "`$dynamicAnchor`", and 2 schemas using "`dependentSchemas`".

Handwritten. We carefully hand-crafted 60 schemas that complement the GitHub and test suite collections, as will be explained in Section 7.5. For each handwritten schema, we created several valid and invalid instances, along with an equivalent schema in which we manually eliminated "`unevaluatedProperties`" and "`unevaluatedItems`", as a ground truth for equivalence checking.

We made the Handwritten collection available on GitHub [33].

7.5. How we test correctness of the implementation

We developed three different approaches to spot correctness problems: (1) random witness testing, (2) manual witness testing, and (3) handwritten translation testing.

(1) In *random witness testing*, given an input Modern JSON Schema schema S_i from the GitHub collection, we use an external tool to generate positive witnesses, which are instances J_y such that S_i validates J_y, and also negative witnesses, which are instances J_n such that S_i does not validate J_n. If $T(S_i)$ is the Classical JSON Schema result of our `unevaluated` $*$ elimination tool, we use external validators to check that $T(S_i)$ validates every positive witness J_y and also does not validate any J_n negative witness; if this does not hold, our tool has an error.

Of course, this “pointwise” approach is not exhaustive: the fact that $T(S_i)$ is equivalent to S_i on a limited set of instances does not ensure that the two are equivalent, but every approach based on a finite set of tests has this problem. A more serious limitation is that many, or even most, of the generated “random” witnesses may not really depend on how S_i deals with unevaluated properties; for example, when the positive witness J_y is just the empty record “{ }”, then its validation does not depend on the presence of an "`unevaluatedProperties`" : O keyword in S_i.

[14] Unfortunately, licensing restrictions prevent us from making this collection publicly available.

Table 5
Correctness results of random/manual witness testing for each collection.

Collection	#Schemas	#Instances	Method	Success	Errors
GitHub	305	1347	*Random*	100 %	0 %
Test Suite	65	182	*Manual*	100 %	0 %
Handwritten	60	387	*Manual*	100 %	0 %

Table 6
Correctness results of handwritten translation testing.

Collection	#Schemas	#Success	#Logical errors	#Timeout	**#Unsupported**
Handwritten	60	30	0	29	1

(2) In *manual witness testing*, the schema and the witness have been designed purposely to test correctness. To this aim, we exploited the standard JSON Schema Test Suite, which contains some examples of schemas that contain "`unevaluated*`" keywords with positive and negative witnesses, and, more importantly, we designed and wrote our own Handwritten collection. This is a collection of Modern JSON Schema schemas that use the "`unevaluated*`" keywords in a variety of ways, in combination with a variety of other keywords. For each schema, we prepared a set of positive and negative examples whose validation, or non-validation, depends on the "`unevaluated*`" keyword and on the way that annotations are passed by the other operators; this is similar in spirit to the JSON Schema Test Suite, but is much more focused on this specific test case, and hence is much more thorough. For each such schema S_h and for each positive and negative witness, we then used an external validator to check that the witness is actually positive (or negative), and then to check whether the `unevaluated` $*$-eliminated schema $T(S_h)$ is equivalent to S_h on these specific test cases.

This approach is much more likely to identify translation problems than the previous approach, as witnesses are designed to depend on the "`unevaluated*`" keyword and its precise placement. However, design and production of this kind of test case requires significant effort.

(3) In *handwritten translation testing*, we prepared a set of Modern JSON Schema schemas that use the "`unevaluated*`" keywords in a variety of ways and, for each schema S_h, we prepared a manual Classical JSON Schema translation S_m that is supposedly equivalent to S_h. Then, for each schema S_h, we used an external Classical JSON Schema equivalence checker to verify that the algorithmic translation $T(S_h)$ was equivalent to the handwritten translation S_m.

Before carrying out the actual experiment, we verified the correctness of our translation using approach (2), which is quite reliable. However, we cannot guarantee that the manual translation is correct, which could lead to false positives or false negatives. False positives are not a concern, as they are inherent to finite testing. False negatives have been removed, as follows: when the external tool discovers that $T(S_h)$ is not equivalent to S_m, it also returns a non-equivalence witness J, that is, an instance that is validated by $T(S_h)$ but not by S_m, or vice versa; e.g., we may have $\vdash^S J ? T(S_h)$ and not $\vdash^S J ? S_m$. At this point, we use an external validator to check the validity of J for S_h. If S_h and $T(S_h)$ are equivalent on J, e.g. $\vdash^S J ? T(S_h)$ and $\vdash^S J ? S_h$, this means that S_m was not a correct translation of S_h, and we can correct it; If S_h and $T(S_h)$ are not equivalent on J, in our example $\vdash^S J ? T(S_h)$ and not $\vdash^S J ? S_h$, then we have a real, verified logical error of the translation tool.

This third approach is much more effective than the previous two, as it is not based on “pointwise” tests on some specific witnesses, but on a full equivalence analysis between $T(S_h)$ and S_m. However, it has its own drawbacks. First, its preparation is even more costly than that of the other approaches, as the manual translation of a complex schema requires considerable effort. Secondly, equivalence checking for Classical JSON Schema schemas containing complex combinations of boolean operators is computationally expensive. As we will see later, the only external tool available that supports the required set of JSON Schema features often failed to complete the equivalence test within the allotted time.

To reduce manual effort, we use the same handwritten schemas S_h with their positive and negative witnesses in two experiments. In *handwritten translation testing* (3), we use the witnesses in the preliminary phase where we verify the equivalence between S_h and S_m, while they play no role in the $S_m \sim T(S_h)$ equivalence test. In *manual witness testing* (2), we use the witnesses for the “pointwise" comparison between S_h and $T(S_h)$, while the manually translated schemas S_m play no role.

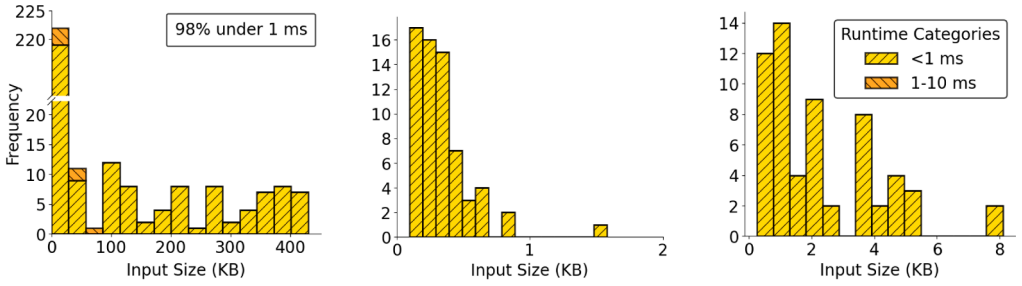
An external equivalence tool that can directly check the equivalence between S_i and $T(S_i)$ would give us a much more robust and easier way to check the correctness of our implementation. However, as we specified in the Introduction, in this moment, no algorithm has yet been described to check equivalence between two schemas that use "`unevaluated*`" keywords, apart from the one we provide in this paper: we use our algorithm to eliminate "`unevaluated*`" from the schemas, and then the algorithm described by Attouche et al. [9] to check the equivalence of the results.

7.6. Experimental results

Correctness of the implementation. For hypothesis H0, we used three collections. The JSON Schema Test Suite offers complete language coverage, the GitHub collection covers realistic use cases, and the Handwritten collection covers complex interactions between different operators.

Table 7
Runtime results (average of 100 runs), showing median, 95th percentile, and average runtime of each collection.

Collection	#Schemas	Avg. Size (KB)	Median Runtime	95 %-tile Runtime	Avg. Runtime
GitHub	305	63.21	0.13 ms	0.68 ms	0.26 ms
Test Suite	65	0.34	0.14 ms	0.33 ms	0.17 ms
Handwritten	60	2.27	0.14 ms	0.39 ms	0.18 ms



(a) GitHub (b) Test Suite (c) Handwritten

Fig. 3. Histogram over the schema sizes for the schema collections. Yellow bars for `unevaluated` $*$ elimination within less than 1 ms, orange bars for runtimes of 1 to 10 ms. Runtimes averaged over 100 runs.

Table 5 reports the results of random and manual *witness testing* (Section 7.5), and it shows that almost 2000 validity tests, on three different collections built in three very different ways, did not spot any problems with our implementation of the algorithm described here.

Table 6 shows the result of *handwritten translation testing* (Section 7.5). In this case, for each file S_h in the Handwritten test, we manually prepared an equivalent file S_m written in Classical JSON Schema. It is worth specifying that our manual translation was not just an application of our algorithm; for example, for all schemas that feature the "`oneOf`" operator, we used a completely different technique, not described in the paper for space reasons, that is much easier to compute by hand. As described in Section 7.5, we tested the actual equivalence between S_m and S_h by the same pointwise testing that we documented in Table 5. Further, we used the tool described in Section 7.3 to verify the equivalence between the result of our algorithm on S_h and the manual translation S_m.

Table 6 shows the results. *Success* states the number of eliminated schemas that are proved equivalent to the manually translated schema, while *Logical Error* states the number of eliminated schemas that are not equivalent. Equivalence checking was performed with a timeout of three hours per pair of schemas; pairs exceeding this timeout are counted in *Timeout*, while *Unsupported* indicates runtime errors in the equivalence testing tool. Unfortunately, equivalence testing is an exponential problem and some of the Handwritten schemas exhibit complex combinations of boolean and structural operators that are quite challenging. As a result, the external tool could complete the equivalence analysis for only half of our schemas; all completed analyzes confirmed the expected perfect equivalence between the algorithm result and manual elimination.

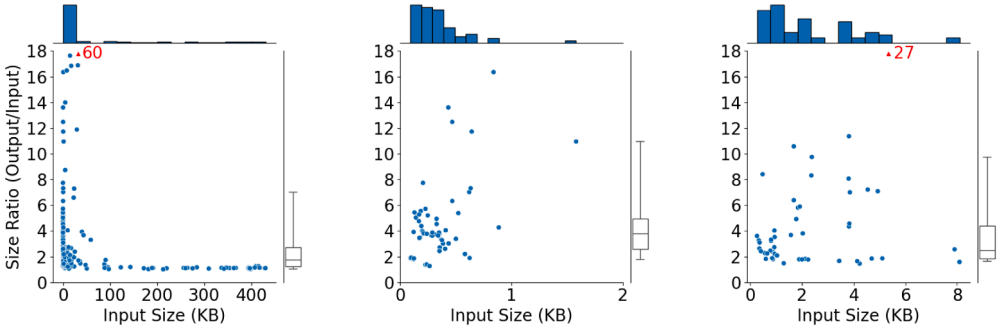
Runtime. We next verify hypothesis H1 by measuring the runtime of `unevaluated` $*$ elimination on our collections. Table 7 shows the runtimes averaged over 100 runs for each collection, specifically the median, 95th percentile, and average runtime. The measurements exclude the initial schema parsing. As runtime measurements under 1 millisecond are subject to measurement errors, these statistics should only be viewed as rough indicators.

Fig. 3 shows the distribution of schemas with respect to input size. Runtimes are visualized as stacked bars, distinguishing schemas processed in under 1 ms (yellow) and those taking between 1 and 10 ms (orange). In the GitHub collection, 98 % of schemas finish in under 1 millisecond and all remaining schemas take between 1 and 10 milliseconds. In the other collections, elimination for each schema is always performed in less than 1 millisecond. Interestingly, there is little correlation between input size and runtime, which may be explained by the fact that we do not include the parsing time in these runtime measurements. If we also consider parsing time, runtimes for all schemas larger than 50 kilobytes in the GitHub collection exceed 1 millisecond, yet no schema exceeds a runtime of 10 milliseconds, and 70 % remain below 1 millisecond.

These results confirm hypothesis H1: Our implementation demonstrates perfectly acceptable runtime performance on real-world schemas, processing over 98 % of schemas in under 1 millisecond each and never exceeding 10 milliseconds.

Size blow-up. We test hypothesis H2 by comparing the size of the eliminated schemas to the size of the original schemas. Fig. 4 shows the size ratio (output size divided by input size) of schemas processed by our implementation compared to the input size of the schemas. 87 % of schemas across all collections have a size ratio below 5, while less than 5 % of schemas exceed a size ratio of 10.

While the Test Suite and Handwritten collections contain only small schemas of up to 8 kilobytes, the GitHub collection also contains a number of larger schemas, up to 430 kilobytes, with 75 % of schemas in the GitHub collection having an input size below



(a) GitHub (b) Test Suite (c) Handwritten

Fig. 4. Investigating size blow-up: Schema input size vs. size ratio (output size/input size) of schemas for the schema collections. Outliers and their size ratio (rounded) are shown in red. Histograms along the horizontal axis show the distribution of schemas by input size. Boxplots along the vertical axis show distribution of size ratios, with whiskers reaching to the 5th/95th percentile. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

50 kilobytes. Noticeably, almost all larger schemas (more than 50 kilobytes) in the GitHub collection have a size ratio close to 1. This can be explained by two factors: First, we measure the blow-up relative to the size of the input schema, and thus the same absolute size blow-up caused by `unevaluated` $*$ elimination is more noticeable in small schemas than in large ones. Second, all 59 schemas in the GitHub collection that are larger than 120 kilobytes originate from the same GitHub repository, exhibiting considerable similarities in their use of "`unevaluated*`" keywords, which may explain why the size ratio shows very little variation for these schemas.

There are two noticeable outliers, one in the GitHub collection, with a size ratio of 60 (rounded) and one in the Handwritten collection, with a size ratio of 27 (rounded). The high size ratio of these outliers is caused by an unusually extensive use of logical operators.

These results support hypothesis H2, with the vast majority of analyzed real-world schemas exhibiting a size ratio below 5. Although some schemas exceed a size ratio of 10 and one outlier even reaches a size ratio of 60, we observe these higher values only for schemas with an input size below 50 KB, meaning that the absolute output size is still reasonable: on average, schemas with a size ratio above 10 have an output size of around 200 KB.

7.7. Discussion

We evaluated the correctness of our research prototype on more than 400 real-world and artificial schemas. Using almost 2000 instances for random and manual witness testing, along with a set of manually translated schemas for handwritten translation testing, we found no correctness issues with our implementation. Runtime analysis showed that our algorithm processes most schemas in under 1 millisecond and never exceeds 10 milliseconds, suggesting that it is well suited for practical use. Despite the exponential worst-case lower bound, the observed size blow-up remained manageable for most real-world schemas, with only one noticeable outlier.

These results are coherent with our research hypotheses and show the applicability of our algorithm for real-world schemas.

8. Conclusions

"`unevaluatedProperties`" and "`unevaluatedItems`" have been added to Modern JSON Schema to solve important and practical problems, such as, for example, the bad interaction between "`additionalProperties`" and factorization. These novel operators differ from existing operators in their sensitivity to annotations that now guide the validation process. While very useful, these operators are not compatible with the algorithms that have been previously defined to decide inclusion, equivalence and satisfiability of JSON Schema. In this paper, we have shown that these operators add no expressive power to the language, i.e., they can be rewritten in terms of other operators. This rewriting comes at the price of an exponential blowup of schema size, and we proved that this blowup cannot be avoided.

We have then defined a rewriting algorithm, based on the static characterization of the evaluated properties and items, on the notion of cover closure, and on the Evaluation Normal Form (ENF). The ENF approach is designed in order to keep the size of the output, and hence the computation time, under strict control on real-world schemas. We have designed and executed a set of experiments to validate the algorithm and to analyze its performance on real-world schemas, and these experiments confirm our hypothesis.

Modern JSON Schema adds both the "`unevaluated*`" keywords and dynamic references to Classical JSON Schema; Attouche et al. [3] proved that the elimination of dynamic references requires an exponential explosion, and here we prove that the same holds for the "`unevaluated*`" keywords. It is difficult to guess whether it would be possible to eliminate both classes of operators with a single exponential blow-up or whether there exist schemas where a double exponential blow-up is unavoidable — we leave this as an open problem.

We are in contact with the community in charge of maintaining the standard, have communicated our results, and have received very valuable feedback; we thank them here for their support, questions, and suggestions.

CRediT authorship contribution statement

Lyes Attouche: Software; **Mohamed-Amine Baazizi:** Writing – review & editing, Software; **Dario Colazzo:** Writing – review & editing, Writing – original draft, Conceptualization; **Giorgio Ghelli:** Writing – review & editing, Writing – original draft, Conceptualization; **Stefan Klessinger:** Writing – review & editing, Software; **Carlo Sartiani:** Writing – review & editing, Writing – original draft, Project administration; **Stefanie Scherzinger:** Writing – review & editing, Software.

Data availability

We have shared the link to our data in the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partly supported by the European Union under the scheme HORIZON-INFRA-2021-DEV-02-01 — Preparatory phase of new ESFRI research infrastructure projects, Grant Agreement n.101079043, “SoBigData RI PPP: SoBigData RI Preparatory Phase Project”. We also acknowledge the support of the PRIN Project “BioConceptum” (2022AEEKXS), under the NRRP MUR program funded by the NextGenerationEU.

References

[1] A. Wright, H. Andrews, G. Luff, JSON schema validation: a vocabulary for structural validation of JSON - draft-handrews-json-schema-validation-01, 2018, Retrieved 10 September 2025., https://json-schema.org/draft-07/draft-handrews-json-schema-validation-01.
[2] A. Wright, H. Andrews, B. Hutton, JSON schema validation: a vocabulary for structural validation of JSON - draft-handrews-json-schema-validation-02, 2019, Retrieved 10 September 2025., https://tools.ietf.org/html/draft-handrews-json-schema-validation-02.
[3] L. Attouche, M.A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, Validation of Modern JSON Schema: Formalization and complexity, Proc. ACM Program. Lang. 8 (POPL) (2024) 1451–1481. https://doi.org/10.1145/3632891
[4] A. Wright, H. Andrews, B. Hutton, G. Dennis, JSON schema: a media type for describing JSON documents - draft-bhutton-json-schema-01, 2022, Retrieved 10 September 2025., https://json-schema.org/draft/2020-12/json-schema-core.html.
[5] H. Andrews, Modern JSON schema, 2023, Available online at https://modern-json-schema.com/.
[6] P. Bourhis, J.L. Reutter, F. Suárez, D. Vrgoc, JSON: data model, query languages and schema specification, in: E. Sallinger, J.V.d. Bussche, F. Geerts (Eds.), Proc. PODS, ACM, 2017, pp. 123–135. https://doi.org/10.1145/3034786.3056120
[7] A. Wright, H. Andrews, B. Hutton, JSON schema validation: a vocabulary for structural validation of JSON - draft-bhutton-json-schema-validation-00, 2020, Retrieved 10 September 2025., https://tools.ietf.org/html/draft-bhutton-json-schema-validation-00.
[8] A. Habib, A. Shinnar, M. Hirzel, M. Pradel, Finding data compatibility bugs with JSON subschema checking, in: Proc. ISSTA, 2021, pp. 620–632. https://doi.org/10.1145/3460319.3464796
[9] L. Attouche, M.A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, Witness generation for JSON Schema, Proc. VLDB Endow. 15 (13) (2022) 4002–4014. https://www.vldb.org/pvldb/vol15/p4002-sartiani.pdf.
[10] Aznan, How does unevaluatedProperties interact with nested properties?, 2023, Available online at https://stackoverflow.com/questions/76465431/how-does-unevaluatedproperties-interact-with-nested-properties, retrieved 10 September 2025.
[11] Object schema description language and validator for JavaScript objects, 2025 Available on GitHub at https://github.com/hapijs/joi. retrieved 10 September 2025.
[12] JSound schema definition language, 2025 Available on GitHub at http://www.jsoniq.org/docs/JSound/html-single/index.html. retrieved 10 September 2025.
[13] M.A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, Schemas and types for JSON data: from theory to practice, in: Proc. SIGMOD Conference, ACM, 2019, pp. 2060–2063.
[14] F.S. Barría, Formal Specification, Expressiveness, and Complexity Analysis for JSON Schema, Master’s thesis, Pontificia Universidad Católica de Chile, Santiago, Chile, 2016. https://repositorio.uc.cl/handle/11534/16908.
[15] M.A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, Negation-closure for JSON Schema, Theor. Comput. Sci. 955 (2023) 113823. https://doi.org/10.1016/J.TCS.2023.113823
[16] G. Baudart, M. Hirzel, K. Kate, P. Ram, A. Shinnar, Lale: consistent automated machine learning, (2020). https://arxiv.org/abs/2007.01977.
[17] J. Blackler, JSON generator, 2022, Available on GitHub at https://github.com/jimblackler/jsongenerator. retrieved 10 September 2025.
[18] hypothesis-jsonschema, 2024, Available on GitHub at https://github.com/python-jsonschema/hypothesis-jsonschema. retrieved 10 September 2025.
[19] S. Jahangiri, Wisconsin benchmark data generator: to JSON and beyond, in: Proc. SIGMOD, ACM, 2021, pp. 2887–2889. https://doi.org/10.1145/3448016.3450577
[20] S. Geng, H. Cooper, M. Moskal, S. Jenkins, J. Berman, N. Ranchin, R. West, E. Horvitz, H. Nori, Generating structured outputs from language models: benchmark and studies, CoRR:abs/2501.10868 (2025). https://doi.org/10.48550/ARXIV.2501.10868
[21] J. Hidders, J. Paredaens, J.V.d. Bussche, J-Logic: logical foundations for JSON querying, in: E. Sallinger, J.V.d. Bussche, F. Geerts (Eds.), Proc. PODS 2017, Chicago, IL, USA, May 14–19, 2017, ACM, 2017, pp. 137–149. https://doi.org/10.1145/3034786.3056106
[22] E. Gallinucci, M. Golfarelli, S. Rizzi, Schema profiling of document-oriented databases, Inf. Syst. 75 (2018) 13–25. https://doi.org/10.1016/J.IS.2018.02.007
[23] J.L.C. Izquierdo, J. Cabot, Discovering implicit schemas in JSON data, in: F. Daniel, P. Dolog, Q. Li (Eds.), Proc. ICWE, 7977 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 68–83. https://doi.org/10.1007/978-3-642-39200-9_8
[24] D.S. Ruiz, S.F. Morales, J.G. Molina, Inferring versioned schemas from NoSQL databases and its applications, in: P. Johannesson, M. Lee, S.W. Liddle, A.L. Opdahl, O.P. López (Eds.), Proc. ER, 9381 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 467–480. https://doi.org/10.1007/978-3-319-25264-3_35
[25] A.A. Frozza, R. dos Santos Mello, F. de Souza da Costa, An approach for schema extraction of JSON and extended JSON document collections, in: Proc. IRI, IEEE, 2018, pp. 356–363. https://doi.org/10.1109/IRI.2018.00060

[26] M.A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, Parametric schema inference for massive JSON datasets, VLDB J. 28 (4) (2019) 497–521.
[27] S. Klessinger, M. Klettke, U. Störl, S. Scherzinger, Extracting JSON Schemas with tagged unions, in: C. Cappiello, S. Geisler, M. Vidal (Eds.), Proc. DEco@VLDB, 3306 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 27–40. https://ceur-ws.org/Vol-3306/paper4.pdf.
[28] S. Klessinger, Capturing data-inherent dependencies in JSON Schema extraction, in: S. Das, I. Pandis, K.S. Candan, S. Amer-Yahia (Eds.), SIGMOD Companion, ACM, 2023, pp. 295–297. https://doi.org/10.1145/3555041.3589396
[29] W. Spoth, O. Kennedy, Y. Lu, B.C. Hammerschmidt, Z.H. Liu, Reducing ambiguity in Json Schema discovery, in: Proc. SIGMOD Conference, ACM, 2021, pp. 1732–1744.
[30] F. Galiegue, K. Zyp, JSON Schema: interactive and non interactive validation - draft-fge-json-schema-validation-00, 2013, Retrieved 10 September 2025, https://tools.ietf.org/html/draft-fge-json-schema-validation-00.
[31] J. Berman, Bowtie JSON schema meta validator, 2023, Available on GitHub at https://github.com/bowtie-json-schema/bowtie. version 0.67.0. retrieved 10 September 2025.
[32] J. Berman, JSON-schema-test-suite (draft2020-12), 2023, Available on Github at https://github.com/json-schema-org/JSON-Schema-Test-Suite/tree/main/tests/draft2020-12. retrieved 10 September 2025.
[33] L. Attouche, M.-A. Baazizi, D. Colazzo, G. Ghelli, S. Klessinger, C. Sartiani, S. Scherzinger, Uneval Elimination: Handwritten Collection, 2024, Available on GitHub at https://github.com/sdbs-uni-p/uneval-elimination-handwritten-collection. retrieved 10 September 2025.
[34] F. Pezoa, J.L. Reutter, F. Suárez, M. Ugarte, D. Vrgoc, Foundations of JSON Schema, Proc. WWW, 2016, pp. 263–273. https://doi.org/10.1145/2872427.2883029
[35] M. Fruth, M.A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger Challenges in checking JSON Schema containment over evolving real-world schemas, Proc. ER 2020 Workshops, 2020, pp. 220–230. https://doi.org/10.1007/978-3-030-65847-2_20
[36] M. Klettke, U. Störl, S. Scherzinger, Schema extraction and structural outlier detection for JSON-based NoSQL data stores, Proc. BTW P-24 (2015) 425–444.