Validation of Modern JSON Schema: Formalization and Complexity

Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger

Validation of Modern JSON Schema: Formalization and Complexity

LYES ATTOUCHE, Université Paris-Dauphine – PSL, France
MOHAMED-AMINE BAAZIZI, Sorbonne Université, LIP6 UMR 7606, France
DARIO COLAZZO, Université Paris-Dauphine – PSL, France
GIORGIO GHELLI, Dip. Informatica, Università di Pisa, Italy
CARLO SARTIANI, DIMIE, Università della Basilicata, Italy
STEFANIE SCHERZINGER, Universität Passau, Germany

JSON Schema is the standard language used to describe the structure of JSON documents. The standard is constantly evolving, and Draft 2019-09 added two new features, *dynamic references* and *annotation-dependent validation*. This addition changes the evaluation model, so that the versions after Draft 2019-09 are now called *Modern JSON Schema*, while *Classical JSON Schema* indicates those that precede it.

These new features solve important practical problems, but they are not easy to understand, and the modification of the evaluation model renders the theory that has been developed for Classical JSON Schema not applicable to the new versions. In this paper we face these problems. We give the first formal description of Modern JSON Schema, we then use this formal apparatus in order to study the complexity of validation for Modern JSON Schema, and we present a surprising result: with the addition of dynamic references, the problem of JSON Schema validation moves from PTIME-complete to PSPACE-hard. We also show that the problem is PSPACE-complete, by providing a polynomial space algorithm. We show that the problem is in PTIME in the special case when there is a fixed bound on the number of distinct dynamic references in each schema, by defining a deterministic polynomial time algorithm. We study the influence of schema size and of instance size, showing that the problem is PSPACE-complete with respect to the schema size, but is in PTIME when the schema is fixed and only the instance size is allowed to vary. Finally, we run experiments that show that there are schemas where the asymptotic complexity difference between dynamic and static references is extremely visible with small schemas already.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: JSON Schema, complexity of validation

1 INTRODUCTION

1.1 Evolution of JSON Schema

JSON is a data language whose values are nested objects and arrays, and JSON Schema [25] is the de-facto standard schema language for JSON. It is based on the combination of structural operators, describing base values, objects, and arrays, through logical operators such as disjunction, conjunction, negation, and recursive references.

JSON Schema passed through many versions, the most important being Draft-04 [18], which was the first to be widely adopted, Draft-06 [31], which introduced extensions without changing the validation model, Draft 2019-09 [29], and Draft 2020-12 [30].

The evaluation model of Draft-04 and Draft-06 is quite easy to understand and to formalize, and it has been studied in many papers, such as [8, 15, 16, 26]. However, Draft 2019-09 introduces

Authors' addresses: Lyes Attouche, Université Paris-Dauphine – PSL, Paris, France, lyes.attouche@dauphine.fr; Mohamed-Amine Baazizi, Sorbonne Université, LIP6 UMR 7606, Paris, France, baazizi@ia.lip6.fr; Dario Colazzo, Université Paris-Dauphine – PSL, Paris, France, dario.colazzo@dauphine.fr; Giorgio Ghelli, Dip. Informatica, Università di Pisa, Pisa, Italy, ghelli@di.unipi.it; Carlo Sartiani, DIMIE, Università della Basilicata, Potenza, Italy, carlo.sartiani@unibas.it; Stefanie Scherzinger, Universität Passau, Passau, Germany, stefanie.scherzinger@uni-passau.de.

2023. 2475-1421/2023/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

two important novelties to the evaluation model: annotation-dependent validation, and dynamic references; according to the terminology introduced by Henry Handrews in [7], because of these modifications to the evaluation model, Draft 2019-09 is the first Draft that defines *Modern JSON Schema*, while the previous Drafts define variations of *Classical JSON Schema*. These novelties are motivated by application needs, but none of them is faithfully represented by the abstract models that had been developed and studied for Classical JSON Schema. Further, both novelties seem in need of formal clarification, as documented by many online discussions, such as [24] and [20]. Draft 2020-12 [30] is the version of Modern JSON Schema that we study in this paper.

1.2 An example of Modern JSON Schema

A JSON Schema schema is a formal specification of a validation process that can be applied to a JSON value called "the instance". A schema, such as the one in Figure 1, can contain nested schemas. A schema is either `true`, `false`, or it is an object whose fields, such as `"type" : "array"`, are called *keywords*, and the term "keyword" is also used to indicate just the name of the field. Two keywords with the same parent object are said to be *adjacent*.

```
1 {   "$schema": "https://json-schema.org/draft/2020-12/schema",
2     "$id": "https://example.com/simple-tree",
3     "$anchor": "tree",
4     "type": "object",
5     "properties": {
6         "data": true,
7         "children": {
8             "type": "array",
9             "items": { "$ref": "https://example.com/simple-tree#tree" }
10        }
11    },
12    "examples": [
13       { "data": 3, "children": [ { "data": null, "children": [] },
14                                  { "data": " ", "children": []  } ] },
15       { "children": [ { "data": null }, { "children": [ { } ] } ] },
16       { "daat": 3, "hcilreden": true }
17    ]
18 }
```

Fig. 1. A schema representing trees.

Looking at Figure 1, the `"$schema"` keyword specifies that this schema is based on Draft 2020-12.

Subschemas of a JSON schema can be identified with a URI with structure *baseURI*·`"#"`·*fragmentId* (we use $s_1 \cdot s_2$ for string concatenation); the `"$id"` keyword assigns a base URI to its parent schema, and the `"$anchor"` keyword gives a fragment identifier to its parent schema, so that, in this case, the fragment named `"tree"` is the entire schema. Hence, this schema can be referred to as either `"https://example.com/simple-tree"` or as `"https://example.com/simple-tree#tree"`.

The `"type"` keyword specifies that this schema only validates objects, while it fails on the other JSON types, that is, arrays and base values. The `"properties"` keyword specifies that, if the instance under validation contains a `"data"` property, then its value has no constraint, and, if it contains a `"children"` property, then its value must satisfy the nested subschema of lines 7-10: it must be an array whose elements (if any) must all satisfy `"$ref" : "https://example.com/simple-tree#tree"`.

`"$ref"` is a reference operator that invokes a local or remote subschema, which, in this case, is the entire current schema.[1]

[1]By the rules of URI reference resolution [13], the URI `"https://example.com/simple-tree#tree"` could be substituted by the local reference `"$ref" : "#tree"`, since `"https://example.com/simple-tree"` is the base URI of this schema.

This schema is satisfied by all the instances that are listed in the "examples" array.[2] The second example shows that no field is mandatory — fields can be made mandatory by using the keyword "required". The third example shows that fields that are different from "data" and "children" are just ignored — we will discuss later how one can forbid such extra fields. To sum up, the only JSON instances where this schema fails are those that associate "children" to a value that is not an array, such as {"data" : 3, "children" : "aaa"}.

Dynamic references have been added as an extension mechanism, allowing one to first define a base form of a data structure, and then to refine it, very much in the spirit of "self" refinement in object-oriented languages. To this aim, the basic data structure is named using "$dynamicAnchor" and is referred using the "$dynamicRef" keyword, as in Figure 2, lines 3 and 9. This combination indicates that "$dynamicRef" : "https://example.com/simple-tree#tree" is *dynamic*, which means that, when this schema is accessed through a different context, this dynamic reference may be redefined so that it refers to a fragment that is still named "tree", but which is defined in a schema that is not "https://example.com/simple-tree". We underline the absolute URI part of the dynamic reference to remind the reader of the fact that this URI is only used if it is not redefined in the "dynamic context", as we will explain later.

```
1 {   "$schema": "https://json-schema.org/draft/2020-12/schema",
2     "$id": "https://example.com/simple-tree",
3     "$dynamicAnchor": "tree",
4     "type": "object",
5     "properties": {
6         "data": true,
7         "children": {
8             "type": "array",
9             "items": { "$dynamicRef": "https://example.com/simple-tree#tree" }
10        }
11    }
12 }
```

Fig. 2. A schema representing extensible trees.

The contextual redefinition mechanism is illustrated by Figure 3.

The schema "https://example.com/strict-tree" redefines the dynamic anchor "tree" (line 4), so that it now indicates a conjunction between "$ref" : "https://example.com/simple-tree#tree" and the keyword "unevaluatedProperties" : false, which forbids the presence of any property that does not match those listed in "https://example.com/simple-tree#tree". If one applies this schema, it will invoke "$ref" : "https://example.com/simple-tree#tree" (Figure 3-line 5), which will execute the schema of Figure 2 in a "dynamic scope" where "https://example.com/strict-tree" has redefined the meaning of "$dynamicRef" : "https://example.com/simple-tree#tree", because "https://example.com/strict-tree" has been met before "https://example.com/simple-tree" (this rule will be formalized in Section 3.7 and discussed in Section 4).[3]

Hence, the "outermost" (or "first") schema that contains "$dynamicAnchor" : "fragmentName" is the one that fixes the meaning of that anchor for any other schema S' that will invoke "$dynamicRef" : "absURI"·"#"·"fragmentName" later, independently from the absolute URI "absURI" used by S'.

This semantics is quite surprising, and we believe it needs a formal definition. We are going to provide that definition, and this was actually the original motivation of this work.

[2]JSON Schema validation will just ignore all non-validation keywords such as "examples".

[3]In Figure 3 we use a non-standard keyword "non-examples" to indicate instances that are validated by "$ref" : "https://example.com/simple-tree#tree" but not by the present schema — again, JSON Schema allows any non-standard keyword, and just ignores it.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/strict-tree",
    "$dynamicAnchor": "tree",
    "$ref": "https://example.com/simple-tree#tree",
    "unevaluatedProperties": false,
    "non-examples" : [
       { "dat": 3 },
       { "data": 3, "childrn": [ ] },
       { "children": [ { "data": null }, { "childrn": [ ] } ] }
    ]
}
```

Fig. 3. A schema that refines trees.

1.3 Annotation dependency

In Modern JSON Schema, keywords produce *annotations*, and the validation result of a keyword may depend on the annotations produced by adjacent keywords. These annotations carry a lot of information, but the information that is relevant for validation is which children of the current instance (that is, which properties, if it is an object, or which items, if it is an array) have already been *evaluated*. This information is then used by the operators `"unevaluatedProperties"` and `"unevaluatedItems"`, since they are only applied to children that have *not* been *evaluated*, where the exact meaning of this term will be discussed later.

For example, in the schema of Figure 3, the assertion `"unevaluatedProperties" : false` depends on the annotations returned by the adjacent keyword `"$ref" : "https://example.com/simple-tree#tree"` (Figure 1). In this case, `"$ref"` *evaluates* all and only the fields whose name is either `"data"` or `"children"`, hence `"unevaluatedProperties" : false` is applied to any other field, hence it fails if, and only if, fields with a different name exist.

The order in which the keywords appear in the schema is irrelevant for this mechanism; as formalized later, the result is the same as if the `"unevaluatedProperties"` keyword were always evaluated last among the keywords inside a same schema.

The definition of *evaluated* in the specifications of Draft 2020-12 ([30]) presents many ambiguities, as testified by online discussions such as [24] and [20]. These ambiguities do not affect the final result of evaluation, but only the error messages that are generated; these aspects are discussed in Appendix G. We believe that an important contribution of this work is that it provides a precise and succinct language where this kind of ambiguities can be discussed and settled.

1.4 Our Contribution

We provide the following contributions.

i) We provide a formalization of Modern JSON Schema through a set of rules that take into account both annotation-dependent validation and dynamic references (Sections 2 and 3). We implemented a corresponding validator for Modern JSON Schema written in Scala, and used it to verify their correctness using the JSON Schema standard test suite (Section 9).

ii) We analyze the complexity of validating a JSON instance against a schema and show that, when dynamic references come into play, the validation problem becomes PSPACE-hard (Section 5); validation was known to be PTIME-complete for Classical JSON Schema.

iii) We prove that the bound is strict, hence the problem is PSPACE-complete, by providing a polynomial space algorithm for validation (Section 6).

iv) We show that annotation-dependent validation alone does not change the PTIME complexity of JSON Schema validation, by providing an explicit algorithm for Modern JSON Schema

that runs in polynomial time on schemas that do not contain dynamic references (Section 7). The algorithm actually runs in polynomial time on any family of schemas where the number of dynamic references is bounded by a constant.

v) We study the *data complexity* of validation and prove that, when fixing a schema S, validation remains polynomial even in the presence of dynamic references. To this aim, we provide a technique to substitute dynamic references with static references, at the price of an exponential increase of the schema size (Section 8).

vi) We run experiments that show that there are families of schemas where the distinction between dynamic and static references is clearly visible in the experimental result; the experiment also shows that many established validators generally exhibit an exponential behaviour also on the PTIME fragment of JSON Schema (Section 9).

2 FORMALIZING JSON SCHEMA SYNTAX

2.1 Introduction

For reasons of space, we only formalize here a crucial subset of Modern JSON Schema, while the rest is in Appendix A. There are two aspects of JSON Schema, URI management and keyword reordering, that we prefer to address separately from the other aspects, in order to simplify the presentation, through a "normalization process" that we define in Section 2.2; we then formalize the syntax of normalized JSON Schema in Section 2.3.

2.2 URI resolution and resource flattening

The keywords `"$id"`, `"$ref"`, and `"$dynamicRef"` accept any URI reference as value, they apply the *resolution* process defined in [13], and they then interpret the resulting resolved URI according to JSON Schema rules. Since resolution is already specified in [13], we will not formalize it here, and we will assume that, in every schema that is interpreted through our rules, the values of these three keywords have been already resolved, and that the result of that resolution has the shape $absURI \cdot \texttt{"\#"} \cdot fragmentId$, where *fragmentId* may be empty for `"$ref"`, and `"$dynamicRef"`, and must be empty for `"$id"`.

A $\texttt{"\$id"} : absURI$ keyword at the top-level of a schema object that is nested inside a JSON Schema document indicates that that object is a separate resource, identified by *absURI*, that is embedded inside that document but is otherwise independent. When the $\texttt{"\$id"} : absURI$ keyword is found in an object that is not interpreted as a schema, for example inside an unknown keyword or a non validation keyword such as `"default"`, then it has no special meaning. The set of positions that are interpreted as schemas is defined by the grammar presented in Section 2.3.

Embedded resources are an important feature, since they allow the distribution of different resources with just one file, but they present some problems when they are nested inside arbitrary keywords, and when a reference crosses the boundaries between resources, as does `"$ref" : "#/properties/foo/items"` in Figure 4 (see also Section 9.2.1 of the specifications [30]).

To avoid this kind of problems, we assume that every JSON Schema document is *resource-flattened* before validation. Resource flattening consists in moving every embedded resource identified as *absURI* into the value of a field named *absURI* of a `"$defs"` keyword at the top level of the document, and replacing the resource with an equivalent schema $\{\texttt{"\$ref"} : absURI \cdot \texttt{"\#"}\}$ that invokes that resource; `"$defs"` is a placeholder keyword that is not evaluated, but which provides a place to collect schemas that can be invoked using `"$ref"` or `"$dynamicRef"`. During this phase, we also replace any reference that crosses resource boundaries with a *canonical* reference that directly refers the target (as suggested in Section 8.2 of the specifications [30]) (Figure 4).

```
{   "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/resource1",
    "type": "object",
    "$ref": "#/properties/foo/items",
    "properties": {
        "foo": {
           "$schema": "https://json-schema.org/draft/2019-09/schema",
           "$id": "https://example.com/embedded",
            "items": {
                 "$id": "https://example.com/embeddedInside",
                 "type": "number"
            }
        }
    },
    "default":  { "$id": "https://example.com/irrelevantId", "type": "number"  }
}

{   "$schema": ..., "$id": ..., "type": ...,
    "$ref": "https://example.com/flattened#",
    "properties": {
        "foo": { "$ref": "https://example.com/embedded" }
    },
    "default":  { "$id": "https://example.com/irrelevantId", "type": "number  },
    "$defs": {
        "https://example.com/embedded" :  {
            "$schema": "https://json-schema.org/draft/2019-09/schema",
            "$id": "https://example.com/embedded",
            "items": { "$ref": "https://example.com/embeddedInside"}
        },
        "https://json-schema.org/draft/2020-12/schema" : {
                 "$schema": "https://json-schema.org/draft/2019-09/schema",
                 "$id": "https://example.com/embeddedInside",
                 "type": "number"
        }
    }
}
```

Fig. 4. A schema with embedded resources and its flattened version.

Closed schemas. The input of a validation problem includes a schema S and all schemas that are recursively reachable from S by following the URIs used in the "$ref" and "$dynamicRef" operators. For complexity evaluation we will only consider *closed* schemas, that is, schemas that include all the different resources that can be recursively reached from the top-level schema (see Figure 6 for an example). There is no loss of generality, since external schemas can be embedded in a top-level one by copying them in the "$defs" section, using the "$id" operator to preserve their base URI.

2.3 JSON Schema Normalized Grammar

JSON Schema syntax is a subset of JSON syntax. We present in Figure 5 the grammar for a subset of the keywords, which is rich enough to present our results — the full grammar is in Appendix A. In this grammar, the meta-symbols are $(X)^*$, which is Kleene star of X, and $(X)^?$, which is an optional X. Non-terminals are italic words, and everything else — including { [, :] } — are terminal symbols.

Our grammar imposes a specific order among keywords, because the result of some JSON Schema keywords depends on that of some adjacent keywords. Specifically, the two keywords in *FstDep* depend on some keywords in *IndKey* (such as "properties" and "patternProperties"), and the two keywords in *SndDep* depend on the keywords in *FstDep*, and on many keywords in *IndKey*, such as "properties", "patternProperties","anyOf", "allOf", "$ref", and others.

JSON Schema allows the keywords to appear in any order, and evaluates them in an order that respects the dependencies among keywords. We formalize this behaviour by assuming that, before validation, each schema is reordered to respect the grammar in Figure 5. The grammar specifies that

a schema S is either a boolean schema, or it begins with a possibly empty sequence of independent keywords or triples, followed by a possibly empty sequence of first-level dependent keywords, followed by a possibly empty sequence of second-level dependent keywords.

$q \in \mathsf{Num}, i \in \mathsf{Int}, k \in \mathsf{Str}, \mathit{absURI} \in \mathsf{Str}, f \in \mathsf{Str}, \mathit{format} \in \mathsf{Str}, p \in \mathsf{Str}, J \in \mathcal{J}\mathit{Val}$

$$
\begin{array}{lcl}
\mathit{Tp} & ::= & \texttt{"object"} \mid \texttt{"number"} \mid \texttt{"integer"} \mid \texttt{"string"} \mid \texttt{"array"} \mid \texttt{"boolean"} \mid \texttt{"null"} \\
S & ::= & \texttt{true} \mid \texttt{false} \mid \{\ \mathit{IndKey}\ (,\mathit{IndKey})^*\ (,\mathit{FstDep})^*\ (,\mathit{SndDep})^*\ \} \\
 & & \mid \{\ \mathit{FstDep}\ (,\mathit{FstDep})^*\ (,\mathit{SndDep})^*\ \} \mid \{\ \mathit{SndDep}\ (,\mathit{SndDep})^*\ \} \mid \{\ \} \\
\mathit{IndKey} & ::= & \texttt{"minimum"} : q \mid \texttt{"maximum"} : q \mid \texttt{"pattern"} : p \mid \texttt{"required"} : [\ k_1, \ldots, k_n\ ] \\
 & & \mid \texttt{"type"} : \mathit{Tp} \mid \texttt{"\$id"} : \mathit{absURI} \mid \texttt{"\$defs"} : \{\ k_1 : S_1, \ldots, k_n : S_n\ \} \\
 & & \mid \texttt{"\$ref"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f \mid \texttt{"\$dynamicRef"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f \\
 & & \mid \texttt{"\$anchor"} : \mathit{plain-name} \mid \texttt{"\$dynamicAnchor"} : \mathit{plain-name} \\
 & & \mid \texttt{"anyOf"} : [\ S_1, \ldots, S_n\ ] \mid \texttt{"allOf"} : [\ S_1, \ldots, S_n\ ] \mid \texttt{"oneOf"} : [\ S_1, \ldots, S_n\ ] \\
 & & \mid \texttt{"not"} : S \mid \texttt{"patternProperties"} : \{\ p_1 : S_1, \ldots, p_m : S_m\ \} \\
 & & \mid \texttt{"properties"} : \{\ k_1 : S_1, \ldots, k_m : S_m\ \} \\
 & & \mid k : J \ \text{(with } k \text{ not cited as keyword in any other production)} \\
\mathit{FstDep} & ::= & \texttt{"additionalProperties"} : S \mid \texttt{"items"} : S \\
\mathit{SndDep} & ::= & \texttt{"unevaluatedProperties"} : S \mid \texttt{"unevaluatedItems"} : S
\end{array}
$$

Fig. 5. Minimal grammar of normalized JSON Schema Draft 2020-12.

This grammar specifies the predefined keywords, the type of the associated value (here $\mathcal{J}\mathit{Val}$ is the set of all JSON values), and their order. We do not formalize here further restrictions on patterns p, absolute URIs absURI, and fragment identifiers f. A valid schema must also satisfy two more constraints: (1) every URI that is the argument of `"$ref"` or `"$dynamicRef"` must reference a schema, and (2) any two adjacent keywords must have different names.

3 JSON SCHEMA VALIDATION

3.1 JSON data model

JSON values are either base values, or nested arrays and objects; the order of object fields is irrelevant.

$s \in \mathsf{Str}, d \in \mathsf{Num}, n \in \mathsf{Int}, n \geq 0$

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

In this paper we reserve the notation $\{\ldots\}$ to JSON objects, hence we use $\{\!|\ a_1, \ldots, a_n\ |\!\}$ and $\{\!|\ a_i\ |\!\}^{i \in N}$ to indicate a set. When the order of the elements is relevant, we use the list notation $[\![\ a_1, \ldots, a_n\ ]\!]$; we also use $\vec{a}$ to indicate a list.

3.2 Introduction to the proof system

We are going to define a judgment that describes the result and the annotations that are returned when a keyword $K = k : P$ is applied to an instance J in a context C, where C provides the information needed to interpret dynamic references. Hence, we read the following judgment

$$C \vdash^{\mathsf{K}} J\ ?\ K \rightarrow (r, \kappa)$$

as: the application of the keyword K to the instance J, in the context C, returns the boolean r and the annotations κ. The annotations as defined in [30] are a complex data structure, but we only represent here the small subset that is relevant for validation, that is, the set of *evaluated* children,

of the instance J. The *evaluated* children of an object are represented by their names, and the *evaluated* children of an array by their position, so that:

$$\begin{array}{llll} C \vdash^{\mathsf{K}} J \,?\, K \to (r,\kappa) & \wedge & J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} & \Rightarrow \quad \kappa \subseteq \{\!|\, k_1, \ldots, k_n \,|\!\} \\ C \vdash^{\mathsf{K}} J \,?\, K \to (r,\kappa) & \wedge & J = [\, J_1, \ldots, J_n \,] & \Rightarrow \quad \kappa \subseteq \{\!|\, 1, \ldots, n \,|\!\} \\ C \vdash^{\mathsf{K}} J \,?\, K \to (r,\kappa) & \wedge & J \text{ a base value} & \Rightarrow \quad \kappa = \emptyset \end{array}$$

We define a similar *schema judgment* $C \vdash^{\mathsf{S}} J \,?\, S \to (r,\kappa)$ in order to describe the result of applying a schema S to an instance J, and we define a *list evaluation* judgment $C \vdash^{\mathsf{L}} J \,?\, [\![K_1, \ldots, K_n]\!] \to (r,\kappa)$ in order to apply a list of keywords to J, passing the annotations produced by a sublist $[\![K_1, \ldots, K_i]\!]$ to the following keyword K_{i+1}. Observe that the letters `K`, `S`, and `L` that appear on top of $\vdash$ are not metavariables but just symbols used to differentiate the three judgments.

In the next sections we define the rules for keywords and for schemas. Keywords are called *assertions* when they assert properties of the analyzed instance, so that `"$id"` is not an assertion, while `"type"` is. Assertions are called *applicators* when they have schema parameters, such as S_1 and S_2 in `"anyOf"` : $[S_1, S_2]$, that they apply either to the instance, in which case they are *in-place applicators* (e.g, `"anyOf"` : $[S_1, S_2]$), or to elements or items of the instance, in which case they are *object applicators* or *array applicators* (e.g., `"properties"` : $\{\, k_1 : S_1, k_2 : S_2 \,\}$).

In the following sections we present the rules for "terminal" assertions, Boolean in-place applicators, and for object and array applicators. We also illustrate the rules for sequential evaluation and for the schema judgments, and finally the rules for static and dynamic references.

3.3 Terminal assertions

Terminal assertions are those that do not contain any subschema to reapply. The great majority of them are conditional on a type T: they are trivially satisfied when the instance J does not belong to T, and they otherwise verify a specific condition on J. Hence, these keyword are defined by a couple of rules, as exemplified here for the keyword `"minimum"` : q. The first rules always returns T (true) when J is not a number, while the second one, applied to numbers, returns the same boolean $r \in \{\!|\, T, F \,|\!\}$ as checking whether $J \geq q$. The set of *evaluated* children is $\emptyset$.

$$\frac{\mathit{TypeOf}(J) \neq \mathsf{number}}{C \vdash^{\mathsf{K}} J \,?\, \texttt{"minimum"} : q \to (T, \emptyset)}\ (\mathsf{minimumTriv}) \qquad \frac{\mathit{TypeOf}(J) = \mathsf{number} \qquad r = (J \geq q)}{C \vdash^{\mathsf{K}} J \,?\, \texttt{"minimum"} : q \to (r, \emptyset)}\ (\mathsf{minimum})$$

Typed terminal assertions are completely defined by a type and a condition; a complete list of these keywords, with the associated type and condition, can be found in Appendix A.2.

We also have four *type-uniform* terminal assertions, that do not single out any specific type for a special treatment. They are `"enum"`, `"const"`, `"type"` : $[\mathsf{Tp}_1, \ldots, \mathsf{Tp}_n]$, and `"type"` : Tp. We only define here the rule for `"type"` : Tp, where $\mathit{TypeOf}(J)$ extracts the type of the instance J.

$$\frac{r = (\ \mathit{TypeOf}(J) = \mathsf{Tp}\ )}{C \vdash^{\mathsf{K}} J \,?\, \texttt{"type"} : \mathsf{Tp} \to (r, \emptyset)}\ (\mathsf{type})$$

Hence, the rule for a type-uniform terminal assertion is completely defined by a boolean condition, as reported in Table 1.

3.4 Boolean applicators

JSON Schema boolean applicators apply a list of schemas to the instance, obtain a list of intermediate boolean results, and combine the intermediate results using a boolean operator. For the annotations, all assertions always return a union of the annotations produced by their subschemas, even when

assertion: kw:**J'**	**condition:** cond(**J**, kw:**J'**)
"enum" : $[J_1, \ldots, J_n]$	$J \in \{\!\{ J_1, \ldots, J_n \}\!\}$
"const" : J_c	$J = J_c$
"type" : Tp	$\mathit{TypeOf}(J) = \mathsf{Tp}$
"type" : $[\mathsf{Tp}_1, \ldots, \mathsf{Tp}_n]$	$\mathit{TypeOf}(J) \in \{\!\{ \mathsf{Tp}_1, \ldots, \mathsf{Tp}_n \}\!\}$

Table 1. Boolean conditions for type-uniform terminal assertions.

the assertion fails. This should be contrasted with the behaviour of schemas, where a failing schema never returns any annotation (Section 3.6).[4]

Hence, this is the rule for the disjunctive applicator "anyOf"; it combines the intermediate results using the $\vee$ operator, and a child of J is *evaluated* if, and only if, it has been *evaluated* by any subschema S_i.

$$\frac{\forall i \in \{\!\{ 1 \ldots n \}\!\}.\ C \vdash^{\mathsf{S}} J\ ?\ S_i \rightarrow (r_i, \kappa_i) \qquad r = \vee(\{\!\{ r_i \}\!\}^{i \in \{\!\{ 1 \ldots n \}\!\}})}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"anyOf"} : [S_1, ..., S_n] \rightarrow (r, \bigcup_{i \in \{\!\{ 1 \ldots n \}\!\}} \kappa_i)} \quad \text{(anyOf)}$$

The rules for "allOf" and for "not" are analogous: "allOf" is successful if all premises are successful and negation is successful if its premise fails.

$$\frac{\forall i \in \{\!\{ 1 \ldots n \}\!\}.\ C \vdash^{\mathsf{S}} J\ ?\ S_i \rightarrow (r_i, \kappa_i) \qquad r = \wedge(\{\!\{ r_i \}\!\}^{i \in \{\!\{ 1 \ldots n \}\!\}})}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"allOf"} : [S_1, ..., S_n] \rightarrow (r, \bigcup_{i \in \{\!\{ 1 \ldots n \}\!\}} \kappa_i)} \quad \text{(allOf)}$$

$$\frac{C \vdash^{\mathsf{S}} J\ ?\ S \rightarrow (r, \kappa)}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"not"} : S \rightarrow (\neg r, \kappa)} \quad \text{(not)}$$

3.5 Independent object and array applicators (independent structural applicators)

Independent structural applicators are those that reapply a subschema to some children of the instance (*structural*) and whose behavior does not depend on adjacent keywords (*independent*).

We start with the rules for the "patternProperties" : $\{ p_1 : S_1, \ldots, p_m : S_m \}$ applicator that asserts that, if J is an object, then every property of J whose name matches a pattern p_j has a value that satisfies S_j. This rule constraints all instance fields whose name matches any pattern p_j in the applicator, but it does not force any of the p_j's to be matched by any property name, nor any property name to match any p_j; if there is no match, the keyword is satisfied.

We first have the trivial rule for the case when J is not an object: non-object instances trivially satisfy the operator.

$$\frac{\mathit{TypeOf}(J) \neq \text{object}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"patternProperties"} : \{ p_1 : S_1, \ldots, p_m : S_m \} \rightarrow (T, \emptyset)} \quad \text{(patternPropertiesTriv)}$$

In the non-trivial case, where $J = \{ k'_1 : J_1, \ldots, k'_n : J_n \}$, we first collect the set $\{\!\{ (i_1, j_1), \ldots, (i_l, j_l) \}\!\}$ of all pairs (i, j) such that $k'_i \in L(p_j)$, where $L(p_j)$ is the language of the pattern p_j. For each such pair (i_q, j_q), we collect the boolean r_q that specifies whether $C \vdash^{\mathsf{S}} J_{i_q}\ ?\ S_{j_q}$ holds or not, and the entire keyword is successful over J if the conjunction $r = \wedge(\{\!\{ r_q \}\!\}^{q \in \{\!\{ 1 \ldots l \}\!\}})$ is T. According to the standard convention, the empty conjunction $\wedge(\{\!\{ \}\!\})$ evaluates to T, hence this rule does not force any matching.

[4]Other interpretations of the specifications of Draft 2020-12 are possible; see Appendix G for a discussion.

The *evaluated* properties are all the properties k_{i_q} for which a corresponding pattern p_{i_q} exists, independently of the result r_q of the corresponding validation, and independently of the overall result r of the keyword. Observe that the sets κ_q of the children that are *evaluated* in the subproofs are discarded; this happens because elements of κ_q are children of a child J_{i_q} of J; we collect information about the evaluation of the children of J, and are not interested in children of children.

$$\frac{\begin{array}{c} J = \{\, k'_1 : J_1, \ldots, k'_n : J_n \,\} \qquad \{\!|\, (i_1, j_1), \ldots, (i_l, j_l) \,|\!\} = \{\!|\, (i, j) \mid k'_i \in L(p_j) \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J_{i_q}\ ?\ S_{j_q} \to (r_q, \kappa_q) \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"patternProperties"} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\} \to (r, \{\!|\, k'_{i_1}, \ldots, k'_{i_l} \,|\!\})} \text{(patternProperties)}$$

The rule for $\texttt{"properties"} : \{\, k_1 : S_1, \ldots, k_m : S_m \,\}$ is essentially the same, with equality $k'_i = k_j$ taking the place of matching $k'_i \in L(p_j)$; also in this case, no name match is required, but, if a match happens, then the corresponding child of J must satisfy the subschema with the same name.

$$\frac{\begin{array}{c} J = \{\, k'_1 : J_1, \ldots, k'_n : J_n \,\} \qquad \{\!|\, (i_1, j_1), \ldots, (i_l, j_l) \,|\!\} = \{\!|\, (i, j) \mid k'_i = k_j \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J_{i_q}\ ?\ S_{j_q} \to (r_q, \kappa_q) \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"properties"} : \{\, k_1 : S_1, \ldots, k_m : S_m \,\} \to (r, \{\!|\, k'_{i_1}, \ldots, k'_{i_l} \,|\!\})} \text{(properties)}$$

Of course, we also have the trivial rule (propertiesTriv), analogous to rule (patternPropertiesTriv): when J is not an object, "properties" is trivially satisfied. The rules for the other independent object applicators, and for the independent array applicators, can be found in Appendix A.

The independent keywords presented in this section (and in the previous one) produce (respectively, collect and transmit) annotations that influence the behavior of the dependent keywords, which are "additionalProperties", "items", "unevaluatedProperties", and "unevaluatedItems". These dependencies are formalized in the next sections.

3.6 The semantics of schemas: sequential evaluation of keywords

3.6.1 Schemas and sequential evaluation. We have defined the semantics of the independent keywords. We now introduce the rules for schemas and for the sequential executions of keywords.

The rules for the `true` and `false` schemas are trivial.

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

The rule for an object schema $\{\, \vec{K} \,\}$ is based on the keyword-list judgment $C \vdash^{\mathsf{L}} J\ ?\ \vec{K} \xrightarrow{\vec{r}} \kappa$, which applies the keywords in the ordered list $\vec{K}$, passing the annotations from left to right.

Rule (schema-false) specifies that, as dictated by [30], when schema validation fails, no annotation is passed, hence no instance child is regarded as *evaluated*. This is a crucial difference with keywords, since the $C \vdash^{\mathsf{K}} J\ ?\ K \to (r, \kappa)$ judgment may return non empty annotations even when $r = F$. [5]

$$\frac{C \vdash^{\mathsf{L}} J\ ?\ [\!|\, K_1, \ldots, K_n \,|\!] \to (T, \kappa)}{C \vdash^{\mathsf{S}} J\ ?\ \{\, K_1, \ldots, K_n \,\} \to (T, \kappa)}\ \text{(schema-true)} \qquad \frac{C \vdash^{\mathsf{L}} J\ ?\ [\!|\, K_1, \ldots, K_n \,|\!] \to (F, \kappa)}{C \vdash^{\mathsf{S}} J\ ?\ \{\, K_1, \ldots, K_n \,\} \to (F, \emptyset)}\ \text{(schema-false)}$$

We now describe the rules for the sequential evaluation judgment $C \vdash^{\mathsf{L}} J\ ?\ \vec{K} \to (r, \kappa)$. The rules are specified for each list $\vec{K} + K$ by induction on $\vec{K}$ and by cases on K.

We start with the crucial rule, that for $\vec{K} + \texttt{"unevaluatedProperties"} : S$. To evaluate $\vec{K} + \texttt{"unevaluatedProperties"} : S$ we first evaluate $\vec{K}$, which yields a set of *evaluated* children κ, we

[5] See the discussion in Appendix G.

then evaluate S on the other children, and we combine the results by conjunction. We return every property as *evaluated*.

$$\frac{\begin{array}{c} J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} \qquad C \vdash^{\mathsf{L}} J \,?\, \vec{K} \to (r, \kappa) \\ \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, i \mid 1 \le i \le n \wedge\ k_i \notin \kappa \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J_{i_q} \,?\, S \to (r_q, \kappa_q) \qquad r' = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{L}} J \,?\, (\vec{K} + \texttt{"unevaluatedProperties"} : S) \to (r \wedge r', \{\!|\, k_1 \ldots, k_n \,|\!\})} \text{(unevaluatedProperties)}$$

The rule for $\vec{K} + \texttt{"additionalProperties"} : S$ is identical, apart from the fact that we only eliminate the properties that have been *evaluated* by *adjacent* keywords. The specifications [30] indicate that this information should be passed as annotation, but that a static analysis is acceptable if it gives the same result. We formalize this second approach since it is slightly simpler. We define a function $\texttt{propsOf}(\vec{K})$ that extracts all the patterns and all the names that appear in any `"properties"` and `"patternProperties"` keywords that appear in $\vec{K}$ and combines them into a pattern; a property is directly evaluated by a keyword in $\vec{K}$ if, and only if, it belongs to $L(\texttt{propsOf}(\vec{K}))$. The notation $\underline{k_i}$ used in the first line indicates a pattern whose language is $\{\!|\, k_i \,|\!\}$; $\emptyset$ in the third line is a pattern such that $L(\emptyset) = \emptyset$.

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

The keyword `"additionalProperties"` was already present in Classical JSON Schema, and it does not really depends on the annotations passed by the previous keywords, but only on information that can be statically extracted from `"properties"` and `"patternProperties"`; critically, it is not influenced by what is *evaluated* by an adjacent `"$ref"`, as happens to `"unevaluatedProperties"` in the example of Figure 3. Modern JSON Schema introduced `"unevaluatedProperties"` in order to overcome this limitation.

$$\frac{\begin{array}{c} J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} \qquad C \vdash^{\mathsf{L}} J \,?\, \vec{K} \to (r, \kappa) \\ \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, i \mid 1 \le i \le n \wedge\ k_i \notin L(\texttt{propsOf}(\vec{K})) \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J_{i_q} \,?\, S \to (r_q, \kappa_q) \qquad r' = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{L}} J \,?\, (\vec{K} + \texttt{"additionalProperties"} : S) \to (r \wedge r', \{\!|\, k_1 \ldots, k_n \,|\!\})} \text{(additionalProperties)}$$

The rules for `"unevaluatedItems"` : S and `"items"` : S are similar, and they can be found in Appendix A.

Having exhausted the rules for dependent keywords, we have a catch-all rule for all other keywords, that says that, when K is an independent keyword, we combine the results of $C \vdash^{\mathsf{L}} J \,?\, \vec{K} \to (r_l, \kappa_l)$ and $C \vdash^{\mathsf{K}} J \,?\, K \to (r, \kappa)$, but no information is passed between the two judgments.

$$\frac{K \in \mathit{IndKey} \qquad C \vdash^{\mathsf{L}} J \,?\, \vec{K} \to (r_l, \kappa_l) \qquad C \vdash^{\mathsf{K}} J \,?\, K \to (r, \kappa)}{C \vdash^{\mathsf{L}} J \,?\, (\vec{K} + K) \to (r_l \wedge r, \kappa_l \cup \kappa)} \text{(klist-(N+1))}$$

Since the above rule is inductive, we need a terminal rule for the case of an empty list of keywords.

$$C \vdash^{\mathsf{L}} J \,?\, [\,] \to (T, \emptyset) \qquad \text{(klist-0)}$$

3.7 Static and dynamic references

The "$ref" and "$dynamicRef" in-place applicators allow a URI-identified subschema to be applied to the current instance, but the two applicators interpret the URI in a very different way.

As specified in Section 2.2, we assume that the parameter of any "$ref" and "$dynamicRef" is a resolved URI with structure $absUri \cdot \texttt{"\#"} \cdot fragmentId$, where the fragment identifier may be empty; $\texttt{"\$ref"} : absURI \cdot \texttt{"\#"} \cdot fragmentId$ retrieves the resource S identified by $absUri$, which may be the current schema or a different one, retrieves the subschema S' of S identified by $fragmentId$, and applies S' to the current instance J.

The application of $absUri \cdot \texttt{"\#"} \cdot fragmentId$ changes the dynamic scope; this fact is recorded by extending the context C in the premise with $absUri$. The context C is the ordered list of the absolute URIs of the "dynamic scopes" that have been met by the current branch of the validation proof, and is used to interpret dynamic references — in the rule, $L + e$ denotes the operation of adding an element e at the end of a list L.

$$\frac{S' = \texttt{get}(\texttt{load}(absURI), f) \qquad C + absURI \;\vdash^{\texttt{S}}\; J \;?\; S' \rightarrow (r, \kappa)}{C \vdash^{\texttt{K}}\; J \;?\; \texttt{"\$ref"} : absURI \cdot \texttt{"\#"} \cdot f \rightarrow (r, \kappa)} \qquad (\$\text{ref})$$

Here, $\texttt{load}(absURI)$ returns the schema S identified by $absURI$, an operation that we cannot formalize since the standards leave it undefined [13, 30]. $\texttt{get}(S, f)$ returns the subschema identified by f inside S; the fragment f may either be empty, hence identifying the entire S, or a plain-name, which is matched by a corresponding "$anchor" operator inside S,[6] or a JSON Pointer, that begins with "/" and is interpreted by navigation — the get function is formally defined in Appendix B.

For simplicity, we assume that the schema has already been analyzed to ensure the following properties; it would not be difficult to formalize these conditions in the rules:

(1) the load function will not fail;
(2) the get function will not fail;
(3) every "$id" operator assigns to its schema a URI that is different from the URI of any other resource recursively reachable from its schema.

The applicator "$dynamicRef" is very different, and is defined as follows (see [30] Section 8.2.3.2):

> If the initially resolved starting point URI includes a fragment that was created by the "$dynamicAnchor" keyword, the initial URI MUST be replaced by the URI (including the fragment) for the outermost schema resource in the dynamic scope (Section 7.1) that defines an identically named fragment with "$dynamicAnchor". Otherwise, its behavior is identical to "$ref", and no runtime resolution is needed.

This sentence is not easy to decode, but it means that, given an assertion $\texttt{"\$dynamicRef"} : absURI \cdot \texttt{"\#"} \cdot f$, one first verifies whether the resource referenced by the "starting point URI" $absURI$ contains a dynamic anchor $\texttt{"\$dynamicAnchor"} : f'$ with $f' = f$. In this is the case, $\texttt{"\$dynamicRef"} : absURI \cdot \texttt{"\#"} \cdot f$ will be interpreted according to the dynamic interpretation specified in the second part of the sentence, otherwise it will be interpreted as if it were a static reference "$ref"; this verification is formalized by the premises $\texttt{dget}(\texttt{load}(absURI), f) \neq \bot$ and $\texttt{dget}(\texttt{load}(absURI), f) = \bot$ of the two rules that we present below for "$dynamicRef". The function $\texttt{dget}(S, f)$ looks inside S for a subschema that contains $\texttt{"\$dynamicAnchor"} : f$, but it returns $\bot$ if there is no such subschema.[7] After this check is passed, the dynamic interpretation focuses on the fragment f, and it looks for the

[6] Actually, it can also be matched by a "$dynamicAnchor" operator, which, in this case, is interpreted as exactly as "$anchor".
[7] Observe that $\texttt{dget}(\texttt{load}(absURI), f) \neq \bot$ is a static check that may be performed once for all when the schema is loaded. This check is called "the bookending requirement", and it may be dropped in future Drafts (see Remove $dynamicRef bookending requirement), but our results would not be affected by this decisions.

first (the "outermost") resource in C^+ that contains a subschema identified by "$dynamicAnchor" : f, where C^+ is the dynamic context C extended with the initial URI *absURI*.

We formalize this specification using two functions: $\texttt{dget}(S, f)$ and $\texttt{fstURI}(C, f)$. The function $\texttt{dget}(S, f)$ returns the subschema S' that is identified in S by a plain-name f that has been defined by "$dynamicAnchor" : "f", and returns $\bot$ when no such subschema is found in S, and its definition is given in Appendix B. The function $\texttt{fstURI}(L, f)$ returns the first URI in the list L that defines f, that is, such that $\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot$:

$$\begin{array}{lcll} \texttt{fstURI}([\![\ ]\!], f) & = & \bot & \\ \texttt{fstURI}(\mathit{absURI} + L, f) & = & \mathit{absURI} & \text{if } \texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot \\ \texttt{fstURI}(\mathit{absURI} + L, f) & = & \texttt{fstURI}(L, f) & \text{if } \texttt{dget}(\texttt{load}(\mathit{absURI}), f) = \bot \end{array}$$

We can finally formalize the dynamic reference rule. It first checks that the initial URI refers to a dynamic anchor, but, after this check, the result of $\texttt{load}(\mathit{absURI})$ is forgotten. Instead, we look for the first URI *fURI* in $C + \mathit{absURI}$ where the dynamic anchor f is defined, and we extract the corresponding subschema S' by executing $\texttt{dget}(\texttt{load}(\mathit{fURI}), f)$.

$$\frac{\begin{array}{cc} \texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot & \mathit{fURI} = \texttt{fstURI}(C + \mathit{absURI}, f) \\ S' = \texttt{dget}(\texttt{load}(\mathit{fURI}), f) & C + \mathit{fURI} \vdash^{\mathsf{S}} J\ ?\ S' \to (r, \kappa) \end{array}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"\$dynamicRef"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f \to (r, \kappa)} \quad (\$\text{dynamicRef})$$

Remark 1. Observe that $\texttt{fstURI}(C + \mathit{absURI}, f)$ searches *fstURI* into a list that contains the dynamic context extended with the URI *absURI*. We have the impression that the specifications, that we copied above, would rather require $\mathit{fURI} = \texttt{fstURI}(C, f)$, but we contacted the specification authors, and we checked some online verifiers that are widely adopted, and there seems to be a general agreement that $\mathit{fURI} = \texttt{fstURI}(C + \mathit{absURI}, f)$ is the correct formula (see Appendix C for a concrete example). We do not really understand where the specifications would mandate to search into *absURI* as well, and the current version of the official test cases (as for 26^{th} Jan. 2023) does not help us resolving the ambiguity, but this is not essential, since none of our theoretical results is affected by this choice.

On the other side, we believe that these difficulties in the interpretation of the natural language specifications are a further indication of the relevance of a formal specification.

The second rule for "$dynamicRef" applies when the initially resolved starting point URI does not include a fragment that was created by the "$dynamicAnchor" keyword, hence "$dynamicRef" behaves as "$ref".

$$\frac{\begin{array}{c} \texttt{dget}(\texttt{load}(\mathit{absURI}), f) = \bot \\ S' = \texttt{get}(\texttt{load}(\mathit{absURI}), f) \qquad C + \mathit{absURI} \vdash^{\mathsf{S}} J\ ?\ S' \to (r, \kappa) \end{array}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"\$dynamicRef"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f \to (r, \kappa)} \quad (\$\text{dynamicRefAsRef})$$

3.8 Compressing the context by *saturation*

The only rule that depends on the context C is rule ($dynamicRef) that uses $\texttt{fstURI}(C + \mathit{absURI}, f)$ to retrieve the first URI in $C + \mathit{absURI}$ that identifies a schema where f identifies a dynamic fragment. For this operation, when URI is already present in C, its addition at the end of C is irrelevant; for each URI we could just retain its first occurrence in C. Let us define $C{+}?\mathit{URI}$, that we read as C *saturated with* URI, as $C{+}?\mathit{URI} = C$ when $\mathit{URI} \in C$ and $C{+}?\mathit{URI} = C + \mathit{URI}$ when $\mathit{URI} \notin C$. By the observation above, we can substitute $C + \mathit{URI}$ with $C{+}?\mathit{URI}$ in the premises of rules ($ref), ($dynamicRef), and ($dynamicRefAsRef), obtaining, for example the following rule.

$$\frac{\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot \qquad \mathit{fURI} = \texttt{fstURI}(C{+}?\mathit{absURI}, f) \qquad S' = \texttt{dget}(\texttt{load}(\mathit{fURI}), f) \qquad C{+}?\mathit{fURI} \vdash^{S} J\,?\,S' \longrightarrow (r, \kappa)}{C \vdash^{K} J\,?\,\texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f \longrightarrow (r, \kappa)} \quad (\$\text{dynamicRef}_c)$$

This observation will be crucial in our complexity evaluations, hence, from now on, we adopt saturation in the reference rules, and we assume that contexts are URI lists with no repetition.

4 DYNAMIC REFERENCES AS A PARAMETRIZATION MECHANISM

Dynamic references look like a data parametrization mechanism, similar to the X variable in a classical $\mathit{List} < X >$ data structure, but with a peculiar instantiation mechanism. While parametric data structures such as $\mathit{List} < X >$ are usually instantiated by parameter-passing, as in $\mathit{newList} < \mathit{Integer} > ()$ (Java syntax), dynamic references are instantiated by giving a first definition, as we do in Figure 2 for the `"#tree"` dynamic anchor, and by refining it later on, as we do in Figure 3, where we recall the original definition and add an extra condition `"unevaluatedProperties" : false`. Here, "later on" describes the typical order in which one would design this piece of code; crucially, the order is inverted at run-time, when we would invoke `"https://example.com/strict-tree#tree"` first, and this order would cause the definition in `"https://example.com/strict-tree"` to overwrite the definition in `"https://example.com/tree"`. Hence, dynamic references are not "instantiated" by parameter passing, but rather "overwritten", in a way that is driven by invocation order. For this reason, they resemble *self* parametrization in object-oriented languages, where the structure is first defined in a superclass, is later refined in subclasses, and the programmer can decide which version to use, which will depend on their entry-point, that is, on which class they choose to instantiate. A further parallelism with object-oriented inheritance is the emphasis on refinement: in many standard examples, the new definition of a dynamic reference invokes the previous definition and adds more constraints, as we do in Figure 3; this looks very similar to inheritance, where the subclass refines the superclass.

This peculiar instantiation mechanism is related to the intended application of dynamic references. Dynamic references have been introduced (as `"recursiveRef"`) in Draft 2019-09 as a refinement mechanism for recursive data structures, as in our example. The driving example was the metaschema of JSON Schema, that is, a schema specifying what is a well formed JSON Schema schema. If you consider the grammar in Figure 5, they needed a mechanism to describe, in JSON Schema, that grammar, together with different extensions. That mechanism had to allow different extensions to that grammar to be specified, each one adding its own cases, and all of them recursively interpreting S according to a combination of all different extensions [5] (core metaschema). Hence, the mechanism has not been designed to support "parametric data types", but rather "refinable recursive data types", which explains the reliance on "rebinding" rather than "parameter passing".

5 PSPACE HARDNESS: USING DYNAMIC REFERENCES TO ENCODE A QBF SENTENCE

Dynamic references add a seemingly minor twist to the validation rules, but this twist has a dramatic effect on validation complexity. We prove here that dynamic references make validation PSPACE-hard, by reducing quantified Boolean formulas (QBF) validity, a well-known PSPACE-complete problem [27, 28], to JSON Schema validation. In detail, we encode an arbitrary closed QBF formula ψ as a schema S_ψ whose size is linear in ψ and with the property that, given any JSON instance J, the assertion $[\![\, b \,]\!] \vdash^{S} J\,?\,S_\psi$, where b is the base URI of S_ψ, holds if, and only if, S_ψ is valid. Observe that the actual value of J is irrelevant: the schema S_ψ is either satisfied by any instance, or by none at all. In our encoding we use a Boolean subset of JSON Schema, which only includes `"$ref"`, `"$anchor"`, `"$dynamicRef"`, `"$dynamicAnchor"`, `"anyOf"`, `"allOf"`, `true`, and `false`, where `true`, `"$ref"`,

and "$anchor" are only used to improve readability: every "anyOf" : [true] could just be removed, while "$ref" and "$anchor" could be substituted by "$dynamicRef" and "$dynamicAnchor".

We start with an example. Consider the following QBF formula: $\forall x1.\ \exists x2.\ (x1 \wedge \neg x2) \vee (\neg x1 \wedge x2)$; Figure 6 shows how it can be encoded as a JSON Schema schema.

```
1 {
2     "$schema": "https://json-schema.org/draft/2020-12/schema",
3     "$id": "urn:psi",
4     "$ref": "urn:start#forall.x1",
5     "$defs": {
6         "urn:start": {
7              "$id": "urn:start",
8              "$defs": {
9                  "forall.x1" : {
10                    "$anchor": "forall.x1",
11                    "allOf": [ { "$ref": "urn:setvar1#afterq1" },
12                               { "$ref": "urn:start#afterq1" }]
13                 },
14                 "exists.x2": {
15                    "$anchor": "exists.x2",
16                    "anyOf": [ { "$ref": "urn:setvar2#afterq2" },
17                               { "$ref": "urn:start#afterq2" }]
18                 },
19                 "afterq1": { "$anchor": "afterq1", "$ref": "urn:start#exists.x2" },
20                 "afterq2": { "$anchor": "afterq2", "$ref": "urn:close#phi" }
21             }
22         },
23         "urn:setvar1": {
24             "$id": "urn:setvar1",
25             "$defs": {
26                 "afterq1": { "$anchor": "afterq1",       "$ref": "urn:start#exists.x2" },
27                 "x1":      { "$dynamicAnchor": "x1",     "anyOf": [true] },
28                 "not.x1":  { "$dynamicAnchor": "not.x1", "anyOf": [false] }
29             }
30         },
31         "urn:setvar2": {
32             "$id": "urn:setvar2",
33             "$defs": {
34                 "afterq2": { "$anchor": "afterq2",       "$ref": "urn:close#phi" },
35                 "x2":      { "$dynamicAnchor": "x2",     "anyOf": [true] },
36                 "not.x2":  { "$dynamicAnchor": "not.x2", "anyOf": [false] }
37             }
38         },
39         "urn:close": {
40             "$id": "urn:close",
41             "$defs": {
42                 "x1":     { "$dynamicAnchor": "x1",     "anyOf": [false] },
43                 "not.x1": { "$dynamicAnchor": "not.x1", "anyOf": [true] },
44                 "x2":     { "$dynamicAnchor": "x2",     "anyOf": [false] },
45                 "not.x2": { "$dynamicAnchor": "not.x2", "anyOf": [true] },
46                 "phi": {
47                    "$anchor": "phi",
48                    "anyOf": [
49                        { "allOf": [ { "$dynamicRef": "urn:close#x1" },
50                                     { "$dynamicRef": "urn:close#not.x2" }]
51                        },
52                        { "allOf": [ { "$dynamicRef": "urn:close#not.x1" },
53                                     { "$dynamicRef": "urn:close#x2" }]
54                        }
55 ]}}}}}
```

Fig. 6. Encoding $\forall x1.\ \exists x2.\ (x1 \wedge \neg x2) \vee (\neg x1 \wedge x2)$ as a JSON Schema.

For each variable xi we define a resource "urn:setvar"$\cdot i$ (lines 23-30 and 31-38 of Figure 6), each defining two dynamic schemas, one with plain-name "x"$\cdot i$ and value true, and the other

one with plain-name "not.x"·i and value false[8] (lines 27-28, 35-36). The body of the formula to evaluate is defined in the "phi" property inside a resource "urn:close" (line 46-55). All variables in "urn:close#phi" are encoded as dynamic references, so that their value depends on the resources that are in-scope at the moment of the formula evaluation. If the context of the evaluation of the formula "urn:close"#"phi"includes the resource "urn:setvar"·i, then this resource is in the context before the resource "urn:close", hence "$dynamicRef" : "urn:close#x"·i resolves to the true subschema defined there. If the context of the evaluation does not include the resource "urn:setvar" · i, then "urn:close#x" · i resolves to the schema defined for xi inside the "urn:close" resource, whose value is false.

Now we describe how we encode the quantifiers. The first quantifier is encoded inside the "urn:start" resource. If the quantifier is ∀, as in this case, then we apply "allOf" to two references (lines 9-13), one that checks whether the rest of the formula holds when $x1$ is true, by invoking "urn:setvar1#afterq1", which sets $x1$ to true by bringing "urn:setvar1" in scope, and the second one that evaluates the rest of the encoding in a context that does not contain "urn:setvar1", so that $x1$ will be set to false by the definition that is provided by "urn:close".

The existential quantifier is encoded exactly in the same way (lines 14-18), with the only difference that we use "anyOf", so that "urn:start#exists.x2" holds if the rest of the formula holds for at least one boolean value of $x2$. This technique allows one to encode any QBF formula. The size of the "urn:start" part of that schema is linear in the number of variables. Each "urn:setvar"·i resource has a constant size, and their number is linear in the number of variables. Finally, the "urn:close" resource contains a part whose size is linear in the number of variables and a second part whose size is linear with the rest of the formula. Hence, the size of the entire schema is linear in the size of the encoded sentence.

We now formalize this encoding. We substitute the names "#forall.x"·i and "#exists.x"·i, used in the example, with "#quant.x"·i, in order to simplify the description and the proofs.

Definition 1 (S_ψ). Consider a generic closed QBF formula:

$$\psi = Q_1 x_1 \dots Q_n x_n.\ \phi$$

where $Q_i \in \{\!| \forall, \exists |\!\}$ and ϕ is generated by:

$$\phi ::= x_i \mid \neg x_i \mid \phi \vee \phi \mid \phi \wedge \phi.$$

The schema S_ψ contains $n+3$ resources: "urn:psi" (the root), "urn:start", "urn:setvar"·1, …, "urn:setvar"· n, and "urn:close", and its root contains the only assertion

"$ref" : "urn:start#quant.x1"

The "urn:start" resource contains, for each x_i, two subschemas, named "urn:start#quant.x"·i and "urn:start#afterq"·i. Every named subschema has a "$anchor" or a "$dynamicAnchor" keyword that assigns it a name, and an assertion that we call its "body". The body of "urn:start#quant.x" · i is

$$boolOp : [\ \{\text{"\$ref" : "urn:setvar"}\cdot i\cdot\text{"\#afterq"}\cdot i\}\ ,\ \{\text{"\$ref" : "urn:start\#afterq"}\cdot i\}\ ]$$

where $boolOp$ = "allOf" when $Q_i = \forall$, and $boolOp$ = "anyOf" when $Q_i = \exists$. The body of "urn:start#afterq"·i is "$ref" : "urn:start#quant.x"·(i+1) when $i < n$ and is "$ref" : "urn:close#phi" when $i = n$.

Each "urn:setvar"·i resource contains 3 subschemas: "afterq"·i, "x"·i, and "not.x"·i. The body of "afterq"·i is equal to the body of "urn:start#afterq"·i; the body of "x"·i is "anyOf" : [true], and the body of "not.x"·i is "anyOf" : [false].

[8] More precisely, it is "anyOf" : [false], since we cannot add an anchor to a schema that is just false; "anyOf" : [true] in the body of "x"·i is clearly redundant, and is there only for readability.

Finally, the "urn:close" resource contains, for each variable x_i, two subschemas, named "urn:close#x"$\cdot i$ and "urn:close#not.x"$\cdot i$, and a schema "phi". The body each "x"$\cdot i$ is "anyOf" : [false], and the body of each "not.x"$\cdot i$ is "anyOf" : [true]. For a formula $Q_1 x_1 \dots Q_n x_n.\ \phi$, the body of "phi" is S_ϕ, which is recursively defined as follows:

$$\begin{array}{lcl} S_{x_i} & = & \{\text{"\$dynamicRef" : "urn:close\#x"}\cdot i\} \\ S_{\neg x_i} & = & \{\text{"\$dynamicRef" : "urn:close\#not.x"}\cdot i\} \\ S_{\phi_1 \vee \phi_2} & = & \text{"anyOf" : [} S_{\phi_1}, S_{\phi_2} \text{]} \\ S_{\phi_1 \wedge \phi_2} & = & \text{"allOf" : [} S_{\phi_1}, S_{\phi_2} \text{]}. \end{array}$$

The following theorem, whose proof is in Appendix D, states the correctness of the translation.

Theorem 2. Given a QBF closed formula $\psi = Q_1 x_1 \dots Q_n x_n.\ \phi$ and the corresponding schema S_ψ with base URI b, ψ is valid if, and only if, for every J, $[\![\, b \,]\!] \vdash^{S}\ J\ ?\ S_\psi \to T$.

Since the encoding has linear size, PSPACE-hardness is an immediate corollary; as already discussed, true, "$ref", and "$anchor" are not really necessary to our encoding.

Corollary 3 (PSPACE-hardness). *Validation in any fragment of Modern JSON Schema that includes* "$dynamicRef", "$dynamicAnchor", "anyOf", "allOf", *and* false, *is PSPACE-hard.*

6 VALIDATION IS IN PSPACE

After we established that validation of dynamic references is PSPACE-hard, we show here that the bound is tight, by exhibiting a validation algorithm that runs in polynomial space. To this aim, we consider the algorithm that applies the typing rules through recursive calls, using a list of already-met subproblems in order to cut infinite loops.

The algorithm is specified in Figure 1. For each schema, it evaluates its keywords, passing the current value of the boolean result and of the *evaluated* children from one keyword to the next one. Independent keywords (such as "anyOf" and "patternProperties") execute their own rule and update the current result and the current *evaluated* items using conjunction and union, as dictated by rule (klist-(n+1)), while each dependent keyword, (such as "unevaluatedProperties"), updates these two values as specified by its own rule.

In Algorithm 1 we exemplify one in-place independent applicator ("anyOf"), one in-place applicator that updates the context ("$dynamicRef"), one structural independent applicator ("patternProperties"), and one dependent applicator ("unevaluatedProperties").

Function SchemaValidate (*Context, Instance, Schema, StopList*) applies *Schema* in the context *Context*, that is a list of absolute URIs without repetitions, to *Instance*, and uses *StopList* in order to avoid infinite recursion. The *Context* list is extended by evaluation of dynamic and static references using the function Saturate (*Context, URI*) (line 31), which adds *URI* to *Context* only if it is not already there. The *StopList* records the (Context, Instance, Schema) triples that have been met in the current call stack. It stops the algorithm when the same triple is met twice in the same evaluation branch, which prevents infinite loops, since any infinite branch must find the same triple infinitely many times, because every instance and schema that is met is a subterm of the input, and only finitely many different contexts can be generated.

We prove now that this algorithm runs in polynomial space. To this aim, the key observation is the fact that we have a polynomial bound of the length of the call stack. The call stack is a sequence of alternating tuples SchemaValidate (Context, Instance, Schema, StopList) - KeywordValidate (…) - k(...) - SchemaValidate (Context', Instance', Schema', StopList'), where $k(\dots)$ is the keyword-specific function invoked by KeywordValidate. We focus on the sequence of SchemaValidate (Context, Instance, Schema, StopList) tuples, ignoring the intermediate calls. This sequence can

Algorithm 1: Validation

```
1  SchemaValidate(Context,Instance,Schema,StopList)
2      if (Schema == True)) then return (True,EmptySet) ;
3      if (Schema == False)) then return (False,EmptySet) ;
       /* "Input" represents a triple                                                    */
4      Input := (Context,Instance,Schema);
5      if (Present (Input,StopList)) then raise ("error: infinite loop") ;
       /* Result and Eval are initialized, and then updated by each call to KeywordValidate   */
6      Result := True; Eval := EmptySet;
7      for Keyword in Keywords (Schema) do
8          (Result,Eval) := KeywordValidate (Context, Instance, Keyword, Result, Eval, StopList+Input) ;
9      if Result == True then return (Result,Eval);
10     else return (Result,EmptySet) ;
11
12 KeywordValidate(Context,Instance, Keyword, PrevRes, PrevEval, StopList)
13     switch Keyword do
14         case "anyOf": List do
15             return (AnyOf (Context,Instance,List,PrevRes,PrevEval,StopList));
16         case "dynamicRef": absURI "#" fragmentId do
17             return (DynamicRef (Context,Instance,absURI,fragment,PrevRes,PrevEval,StopList));
18         ...
19
20 AnyOf(Context, Instance, List, PrevRes, PrevEval, StopList)
21     Result := True; Eval := EmptySet;
22     for Schema in List do
23         (SchemaRes,SchemaEval) = SchemaValidate (Context, Instance, Keyword, StopList) ;
24         Result := Or (Result,SchemaRes); Eval := Union (Eval,SchemaEval);
25     return (And (PrevResult,Result), Union (PrevEval,Eval));
26
27 DynamicRef(Context, Instance, AbsURI, fragment, PrevRes, PrevEval, StopList)
28     if (dget(load(AbsURI),fragment) = bottom)) then return (StaticRef (...));
29     for URI in Context+AbsURI do if (dget(load(URI),fragment) != bottom) then { fstURI := URI; break; }
30     fstSchema ::= get(load(fstURI),fragment);
31     (SchemaRes,SchemaEval) = SchemaValidate(Saturate (Context,fstURI),Instance,fstSchema,StopList);
32     return (And (PrevResult,SchemaRes), Union (PrevEval,SchemaEval));
33
34 PatternProperties(Context, Instance, SchemaMap, PrevRes, PrevEval, StopList)
35     if (Instance is not Object) then return (True,EmptySet) ;
36     Result := True; Eval := EmptySet;
37     for (name,J) in Instance do
38         for (patt,Schema) in SchemaMap do
39             if (name matches patt) then
40                 (SchemaRes,Ignore) = SchemaValidate (Context, J, Schema, StopList) ;
41                 Result := And (Result,SchemaRes); Eval := Union (Singleton (name),Eval);
42     return (And (PrevResult,Result), Union (PrevEval,Eval));
43
44 UnevaluatedProperties(Context, Instance, Schema, PrevRes, PrevEval, StopList)
45     if (Instance is not Object) then return (True,EmptySet) ;
46     Result := True;
47     for (a,J) in Instance do
48         if (a not in PrevEval) then
49             (SchemaRes,Ignore) = SchemaValidate (Context, J, Schema, StopList) ;
50             Result := And (Result,SchemaRes);
51     return (And (PrevResult,Result), NamesOf (Instance));
```

be divided in at most n subsequences, if n is the input size, the first one with a context that only contains one URI, the second one with contexts with two URIs, and the last one having a number of URIs that is bound by the input size, since no URI is repeated twice in a context. In each subsequence all the (Instance, Schema) pairs are different, since the stoplist test would otherwise raise a failure. Since every instance in a call stack tuple is a subinstance of the initial one, and every schema is a subschema of the initial one, we have at most n^2 elements in each subsequence, hence the entire call stack never exceeds n^3. We finally observe that every single function invocation can be executed in polynomial space plus the space used by the functions that it invokes, directly and indirectly; the result follows, since we have seen that these functions are never more than $3 * n^3$ at the same time. This is the basic idea behind the following Theorem, whose full proof can be found in Appendix D. Since our algorithm runs in polynomial space, the problem of validation for Modern JSON Schema is PSPACE-complete.

Theorem 4. For any closed schema S and instance J whose total size is less than n, Algorithm 1 applied to J and S requires an amount of space that is polynomial in n.

7 POLYNOMIAL TIME VALIDATION FOR STATIC REFERENCES

While dynamic references make validation PSPACE-hard, annotation-dependent validation alone does not change the PTIME complexity of Classical JSON Schema validation. We prove this fact here by defining an optimized variant of Algorithm 1 that runs in polynomial time in situations where there is a fixed bound on the maximum number of dynamic references, hence, a fortiori, for schemas where no dynamic reference is present.

Our optimized algorithm exploits a memoization technique. To this aim, it returns, from each evaluation of S over J in a context C, not only the boolean result and the *evaluated* children, but also a *DFragSet*, that specifies which dynamic fragments have been resolved during that evaluation. For each evaluated judgment $C \vDash^{S} J ? S \rightarrow (r, \kappa)$, we write the tuple $(C, J, S, r, \kappa, \mathit{DFragSet})$ in an updatable store. When, during the same validation, we evaluate again J and S in an arbitrary context C', we retrieve any previous evaluation with the same pair (J, S), and we verify whether the new C' is equivalent to the context C used for that evaluation, with respect to the set of fragments that have been actually evaluated, reported in the *DFragSet*; here *equivalent* means that, for each fragment f in *DFragSet*, $\texttt{fstURI}(C, f)$ and $\texttt{fstURI}(C', f)$ coincide. If the two contexts are equivalent, then we do not recompute the result, but we just return the previous $(r, \kappa, \mathit{DFragSet})$ triple. It is easy to prove that, when the number of different dynamic references is bounded, this equivalence relation has a number of equivalence classes that is polynomial in the size of S, hence that memoization limits the number of recursive call below a polynomial bound.

For simplicity, in our algorithm we keep the UpdatableStore and the StopList separated; it would not be difficult to merge them in a single data structure that can be used for the purposes of both. We show here how `SchemaValidate` changes from Algorithm 1. In Appendix E we also report how `KeywordValidate` is modified.

This optimized algorithm returns the same result as the base algorithm, and runs in polynomial time if the number of different dynamic fragments is limited by a fixed bound. The proofs can be found in Appendix D.

Theorem 5. Algorithm 2 applied to $(C, J, S, \emptyset, \emptyset)$ returns (r, κ, d), for some d, if, and only if, $C \vDash^{S} J ? S \rightarrow (r, \kappa)$.

Theorem 6. Consider a family of closed schemas S and judgments J such that $(|S| + |J|) \leq n$, and let D be the set of different fragments f that appear in the argument of a `"$dynamicRef"` : $\mathit{initURI} \cdot \texttt{"\#"} \cdot f$ in S. Then, Algorithm 2 runs on S and J in time $O(n^{k+|D|})$ for some constant k.

Algorithm 2: Polynomial Time Validation

```
   /* The UpdatableStore maps each evaluated Instance-Schema pair to the list, maybe empty, of
      all contexts where it has been evaluated, each context paired to the associated result  */
1  SchemaValidateAndStore(Context,Instance,Schema,StopList,UpdatableStore)
2      if (Schema == True) then return (True,EmptySet,EmptySet) ;
3      if (Schema == False) then return (False,EmptySet,EmptySet) ;
4      Input := (Context,Instance,Schema);
5      if (Present (Input,StopList)) then  raise ("error: infinite loop") ;
6      PreviousResultsForSameSchemaAndInstance := UpdatableStore.get(Instance,Schema);
7      for (OldContext,OldResult,OldEval,OldDFragSet) in PreviousResultsForSameSchemaAndInstance do
8          if (Equivalent (Context,OldContext,OldDFragSet)) then /* If the current Context is equivalent
               to the OldContext we reuse the old output                                          */
9          return (OldOutput);
10     Output := (True,EmptySet,EmptySet);
11     for Keyword in Keywords (Schema) do
12         Output := KeywordValidate (Context, Instance, Keyword, Output, StopList+Input,UpdatableStore) ;
13     (Result,Eval,DFragSet) := Output;
14     UpdatableStore.addToList((Instance,Schema),(Context,Result,Eval,DFragSet));
15     if Result == True then
16         return (Result, Eval, DFragSet);
17     else
18         return (Result, EmptySet, DFragSet);
19
20 Equivalent (Context,OldContext,DFragSet)
21     Result := True;
22     for f in DFragSet do  Result := And (Result, (FirstURI (Context,f) ==FirstURI (OldContext,f)));
23     return (Result);
```

Corollary 7. *Validation is in PTIME for every family of schemas where the maximum number of different fragments that are argument of* `"$dynamicRef"` *is bounded by a constant.*

8 PTIME DATA COMPLEXITY THROUGH ELIMINATION OF DYNAMIC REFERENCES

As we have seen, dynamic references change the computational and algebraic properties of JSON Schema. We define here a process to eliminate dynamic references, by substituting them with static references; this allows us to reuse results and algorithms that have been defined for Classical JSON Schema. Specifically, we will prove here that dynamic references can be eliminated from a schema, and substituted with static references, at the price of a potentially exponential increase in the size of the schema. This entails that the data complexity of the problem is polynomial (Corollary 9).

A dynamic reference `"$dynamicRef"` : $initURI \cdot$`"#"`$\cdot f$ is resolved, during validation, to a URI reference `fstURI`$(C+?initURI, f)\cdot$`"#"`$\cdot f$ that depends on the context C of the validation (Section 3.7), so that the same schema S behaves in different ways when applied in different contexts. This context-dependency extends to static references: a static reference `"$ref"` : $absURI \cdot$`"#"`$\cdot f$ is always resolved to the same subschema, however, when this subschema invokes some dynamic reference, directly or through a chain of static references, then the validation behaviour of this subschema depends on the context, as happens with `"$ref" : "urn:close#phi"` in our example, which is a static reference, but the behaviour of the schema it refers to depends on the context.

To obtain the same effect without dynamic references, we observe that, once the context C is fixed, then every dynamic reference has a fixed behaviour, and it can be encoded using a static reference `"$ref"` : `fstURI`$(C+?initURI, f)\cdot$`"#"`$\cdot f$. Every dynamic reference can be eliminated if we iterate this

process by defining, for each subschema S' and for each context C, a context-injected version $\texttt{CI}(C, S')$, which describes how S' behaves when the context is C. The context-injected $\texttt{CI}(C, S')$ is obtained by (1) substituting in S' every dynamic reference $\texttt{"\$dynamicRef"} : \mathit{initURI}\cdot\texttt{"\#"}\cdot f$ with a static reference to the context-injected version of the schema identified by $\texttt{fstURI}(C{+}?\mathit{initURI}, f)\cdot\texttt{"\#"}\cdot f$, and (2) substituting every static reference $\texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ with a static reference to the context-injected version of the schema identified by $\mathit{absURI}\cdot\texttt{"\#"}\cdot f$. Step (2) is crucial, since a static reference may recursively invoke a dynamic one, hence the context must be propagated through the static references.

Before giving a formal definition of the process that we outlined, we start with an example. Consider the context $C = [\![\,\texttt{"urn:psi"}, \texttt{"urn:start"}, \texttt{"urn:setvar1"}\,]\!]$ and a reference $\texttt{"\$ref"} : \texttt{"urn:close\#phi"}$, which refers to the following schema S' that contains four dynamic references.

```
1 "urn:close#phi": {
2   "$anchor": "phi",
3   "anyOf": [
4       { "allOf": [ { "$dynamicRef": "urn:close#x1" },
5                    { "$dynamicRef": "urn:close#not.x2" } ] },
6       { "allOf": [ { "$dynamicRef": "urn:close#not.x1" },
7                    { "$dynamicRef": "urn:close#x2" } ] }
8   ]
9 }
```

The corresponding context-injected schema $\texttt{CI}(C, S')$ is the following one. When a schema is identified by $\mathit{absURI}\cdot\texttt{"\#"}\cdot f$, we identify its context-injected version $\texttt{CI}(C, S')$ using $\mathit{absURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$, where $\underline{C}$ is an invertible encoding of C into a plain-name.[9]

```
1 "urn:close#urn:psi_urn:start_urn:setvar1_phi": {
2    "$anchor": "urn:psi_urn:start_urn:setvar1_phi",
3    "anyOf": [
4        { "allOf": [ { "$ref": "urn:setvar1#urn:psi_urn:start_urn:setvar1_urn:close_x1" },
5                     { "$ref": "urn:close#urn:psi_urn:start_urn:setvar1_urn:close_not.x2" } ] },
6        { "allOf": [ { "$ref": "urn:setvar1#urn:psi_urn:start_urn:setvar1_urn:close_not.x1" },
7                     { "$ref": "urn:close#urn:psi_urn:start_urn:setvar1_urn:close_x2" } ] }
8    ]
9 },
```

Let us use C^+ to abbreviate $C{+}?\texttt{"urn:close"}$. Observe that, in $\texttt{CI}(C, S')$, all dynamic references have been substituted with static references, which depend on the context C: for example, `"x1"` in line 4 has a URI that starts with `"urn:setvar1"`, which is $\texttt{fstURI}(C^+, \texttt{"x1"})$, while `"x2"` in line 7 has a URI that starts with `"urn:close"`, which is $\texttt{fstURI}(C^+, \texttt{"x2"})$. The fragment name of a static reference, such as `"urn:psi_urn:start_urn:setvar1_phi"`, encodes both the injected context (`"urn:psi_..._urn:setvar1_"`) and the original fragment name (`"_phi"`).

Observe that the injected context of the `"$ref"`'s in lines 4-7 extends `"urn:psi_..._urn:setvar1_"` with `"urn:close_"`. This corresponds to what happens in the rule ($dynamicRef):

$$\frac{\begin{array}{ll}\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot & \mathit{fURI} = \texttt{fstURI}(C{+}?\mathit{absURI}, f) \\ S' = \texttt{dget}(\texttt{load}(\mathit{fURI}), f) & C{+}?\mathit{fURI} \;\vdash^{\mathsf{S}}\; J\;?\;S' \to (r, \kappa)\end{array}}{C \vdash^{\mathsf{K}}\; J\;?\;\texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f \to (r, \kappa)} \quad (\$\text{dynamicRef})$$

The rule dictates that the schema S' must be analyzed in the extended context C^+, not in the original context. Hence, in $\texttt{CI}(C, S')$, each dynamic reference to $\texttt{"urn:close\#"}\cdot f_i'$ has been substituted with a static reference to the context-injected schema $\texttt{CI}(C^+, \texttt{fstURI}(C^+, f_i')\cdot\texttt{"\#"}\cdot f_i')$, whose URI-reference is $\texttt{fstURI}(C^+, f_i')\cdot\texttt{"\#"}\cdot\underline{C^+}\cdot f_i'$. The complete unfolding is in Appendix F.

[9] In the example, we encode a sequence of absolute URIs such as $[\![\,\texttt{"urn:psi"}, \texttt{"urn:start"}, \texttt{"urn:setvar1"}\,]\!]$ as `"urn:psi_urn:start_urn:setvar1_"`, that is, we escape any underscore inside the URIs (not exemplified here), and we terminate each URI with an underscore.

We can now give a formal definition of the translation process. For simplicity, we assume that all fragment identifiers are plain-names defined using `"$anchor"` or `"$dynamicAnchor"`, without loss of generality, since JSON Pointers can be easily translated using the anchor mechanism.

Given a judgment S_0 with base URI b, we first define a "local" translation function `CI` that maps every pair (C, S), where C is a list of URIs from S_0 without repetitions and S is a subschema of S_0, into a schema without dynamic references, and that maps every pair (C, K) to a keyword without dynamic references. This function maps references as specified below, and acts as an homomorphism on all the other operators, as exemplified here with `"anyOf"`.

$$\begin{array}{lcl}
\texttt{CI}(C, \texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f) & = & \texttt{"\$ref"} : \texttt{fstURI}(C{+?}\mathit{absURI}, f)\cdot\texttt{"\#"}\cdot\underline{C}\cdot f \\
\texttt{CI}(C, \texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f) & = & \texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f \\
\texttt{CI}(\texttt{"anyOf"} : [\, S_1, \ldots, S_n \,]) & = & \texttt{"\$ref"} : \texttt{"anyOf"} : [\, \texttt{CI}(C, S_1), \ldots, \texttt{CI}(C, S_n) \,] \\
\ldots &&
\end{array}$$

Consider now a schema S_0 and the set $\mathcal{C}$ of all possible contexts, that is, of all lists with no repetitions of absolute URIs of resources inside S_0; a *fragment* of S_0 is any subschema that defines a static or a dynamic anchor (e.g., the subschema identified by `"urn:close#phi"` is a fragment). The static translation of S_0, $\texttt{Static}(S_0)$, is obtained by substituting, in S_0, each fragment S_f identified by $\mathit{absURI}\cdot\texttt{"\#"}\cdot f$ with many fragments $\underline{C}\cdot f$, one for any context $C \in \mathcal{C}$, where the schema identified by each $\mathit{absURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$ is $\texttt{CI}(C, S_f)$, as exemplified in Appendix F.[10] If we have n_U absolute URIs in S_0, we have $\Sigma_{i \in \{\!|\, 0 \ldots n_U \,|\!\}}(i!)$ lists of URIs without repetitions, hence, if we have n_f fragments, the possible (C, S_f) pairs are $(\Sigma_{i \in \{\!|\, 0 \ldots n_U \,|\!\}}(i!)) \times n_f$, which is included between $n_U! \times n_f$ and $(n_U + 1)! \times n_f$. This last formula gives an exponential upper bound, or, more precisely, an upper bound in $O(2^{n \cdot \log n})$. In practice, the number of reachable schemas will be much smaller, but our example shows that it can actually be exponential.

We can now prove that this process preserves schema behaviour; the proof is in Appendix D.

Theorem 8 (Encoding correctness). Let S be a closed schema with base URI b. Then:

$$[\![\, b \,]\!] \vdash^{S} \; J \,?\, \texttt{Static}(S) \to (r, \kappa) \quad \Leftrightarrow \quad [\![\, b \,]\!] \vdash^{S} \; J \,?\, S \to (r, \kappa)$$

As an immediate consequence of this construction, and of the polynomiality of validation in JSON Schema without dynamic references, we have the fact that validation in Modern JSON Schema has an $O(2^{n \cdot \log n})$ upper bound when the schema size is considered, but is polynomial when the schema is fixed and it is only the instance that varies. This is an important result, reminiscent of data complexity in query evaluation, but the parallelism is not precise: while queries are smaller than data in most application fields, there are many situations where JSON Schema documents are generally bigger than the checked instances, for example when complex schemas are used in order to validate function parameters, that may be quite small.

Corollary 9 (Instance complexity). *When S is fixed, the validation problem $[\![\, b \,]\!] \vdash^{S} J \,?\, S \to (r, \kappa)$ is in PTIME.*

9 EXPERIMENTS

We implemented Algorithm 2 for the entire JSON Schema language. This algorithm applies the rules in Appendix A. Our Scala implementation can be accessed (anonymously) as an online tool from

[10] We say that a translated fragment $\mathit{absURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$ refers another translated fragment if the schema of the first contains a reference to the second; when we translate a schema S_0, we do not actually need to generate every possible fragment $\mathit{absURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$, but we only need to translate those that are recursively reachable from the root fragment $b\cdot\texttt{"\#"}\cdot\underline{[\![\, b \,]\!]}\cdot S_0$; this is exemplified in the translation of our example, in Appendix F, where we only translate the context-fragment pairs that are reachable.

https://validationproofs.oa.r.appspot.com/.[11] We further make the schemas used in our experiments available (see below).

9.1 Correctness of formalization

We applied our algorithm to the official JSON Schema test suite [12].[12] We pass all tests apart from those that contain references to external files (concerning 24 schemas out of 345, at the time of writing). These references must be resolved at runtime, a feature that we do not yet support. This is merely a technical limitation that we plan to overcome within the upcoming weeks.

Discussion. This experiment proves that the rules that we presented, and which are faithfully reflected by our algorithm, are correct and complete with respect to the standard test suite.

9.2 Complexity

We have proved that validation in Modern JSON Schema is PSPACE-complete in the presence of dynamic references, while it is in PTIME when dynamic references are not present. In the upcoming experiment, we test (1) whether there exist families of schemas where this difference is reflected by considerable validation times, and (2) whether they already manifest with small schemas.

Schemas. We designed three families of schemas that are universally satisfiable, and we validate the JSON instance *null* against them.

The family of schemas from `"dyn1.js"` up to `"dyn100.js"` generalizes our running example; the file `"dyn"`$\cdot i \cdot$`".js"` contains the encoding of

$$\forall x_1.\ \exists x_2.\ \ldots \forall x_{2i}.\ \exists x_{2i+1}.\ ((x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)) \wedge \ldots \wedge ((x_{2i} \wedge x_{2i+1}) \vee (\neg x_{2i} \wedge \neg x_{2i+1}))$$

as defined in Section 5. We then have a corresponding family of Draft-04 schemas `"stat1.js"` up to `"stat100.js"`, where dynamic references are substituted with static references, and boolean schemas `true` and `false` are substituted with { } and { `"not"` : { } }.

The third family of schemas, from `"dyn.bounded1.js"` up to `"dyn.bounded100.js"`, encodes

$$\forall x_1.\ \exists x_2.\ \ldots \forall x_{2i}.\ \exists x_{2i+1}.\ ((x_1 \wedge x_{2i+1}) \vee (\neg x_1 \wedge \neg x_{2i+1})).$$

These "dynamic-bounded" schemas, hence, contain dynamic references that respect the condition of Corollary 7, which allows an optimized algorithm to run in polynomial time.

We provide the schemas online: https://anonymous.4open.science/r/modern-jsonschema-908C/.

Validators. For running third-party validators, we employ the meta-validator Bowtie [11], which invokes validators encapsulated in Docker containers. We tested all 14 different open-source validators currently provided by Bowtie, written in 10 different programming languages, as detailed in Table 2, Appendix H. We also prepared a container for the validator from [26], to include it in our experiments.

Execution environment. Our execution environment is a 40-core Debian server with 384MB of RAM. Each core runs with with 3.1Gz and CPU frequency set to performance mode. We are running Docker version 20.10.12, Bowtie version 0.67.0, and Scala version 2.12.

All runtimes with Bowtie were measured with GNU time, averaged over three runs, and include the overhead of invoking Bowtie and Docker. We overrode the default timeout setting in Bowtie. Our Scala implementation was run outside of Bowtie ("solo"). Here, we measure at the level of

[11] As an AppEngine application, the online version of the tool comes with strict limitations regarding main memory and timeouts. It is not suitable for benchmarking, but allows one to interact with the system and to inspect the proofs.
[12] We only focus on *main* schemas and do not consider the *optional* ones.

microseconds, from within the Scala code. Out of 5 such runs for each experiment, we discard the smallest and the largest runtime (to ignore caching effects), and average over the remaining three.

Lines in our plots terminate when either the validator produces a logical validation error, a runtime exception (most commonly, a stack overflow), or when Bowtie reported that the validator did not respond.

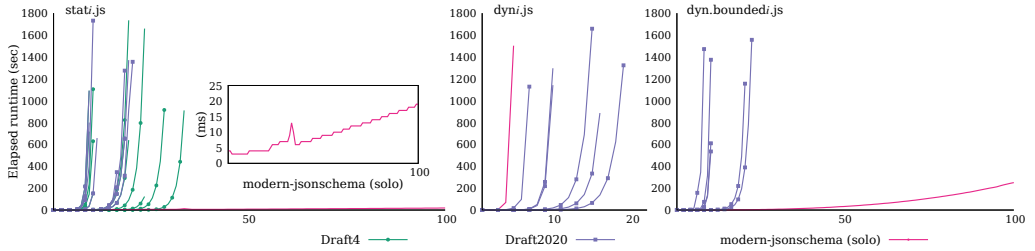

Fig. 7. Runtimes of validators on the three schema families. Distinguishing validators that support Draft 4, Draft 2020, as well as our algorithm "modern-jsonschema". The inlet shows a closeup of modern-jsonschema.

Results. The results in Figure 7 are perfectly coherent with the theoretical results in the paper. Results on the "dyn" and "stat" schemas show that, on this specific example, the difference in the asymptotic complexity of the static and the dynamic version is extremely visible: if we focus on our validator (red line), we see that validation with dynamic references can become impractical even with reasonably-sized files (e.g., schema "stat5.js" counts fewer than 250 lines when pretty-printed), while the runtime remains very modest when dynamic references are substituted with static references. The runtime on the "bounded" family reflects Corollary 7, showing the effectiveness of the proposed optimization on this specific example.

The results on the other implementations strongly suggest that many validators have chosen to implement an algorithm that is exponential even when no dynamic reference is present. This is not surprising for the validators designed for Draft 2020-12, since we have been the first to describe an algorithm (Algorithm 2) that runs in polynomial time over the static fragment of Draft 2020-12. It is a bit more surprising for the algorithm published as additional material for [26], since it implements Draft-04, which belongs to Classical JSON Schema, whose validation problem is in PTIME, as proved for the first time in that same paper.

Discussion. This experiment shows there are families of schemas where the PSPACE-hardness of the problem is visible, and that the algorithm we describe in Section 7 is extremely effective when the dynamic references are substituted with static references, or limited in number. In this paper we focus on worst-case asymptotic complexity, and we do not present claims about real-world relevance of our algorithm, which is an important issue, but is not in the scope of this paper.

10 RELATED WORK

To the best of our knowledge, modern JSON Schema has not been formalized before, nor validation in the presence of dynamic references has been studied.

Overviews over schema languages for JSON can be found in [9, 10, 15, 26]. In [26] Pezoa et al. proposed the first formalization of Classical JSON Schema Draft-04 and studied the complexity of validation. They proved that JSON Schema Draft-04 expressive power goes beyond MSO and tree automata, and showed that validation is PTIME-complete. They also described and experimentally analyzed a Python validator that exhibits good performance and scalability. Their formalization

of semantics and validation, however, cannot be extended to modern JSON Schema due to the presence of dynamic references.

In [15] Bourhis et al. refined the analysis of Pezoa et al. They mapped Classical JSON Schema onto an equivalent modal logic, called recursive JSL, and studied the complexity of validation and satisfiability. In particular, they proved that validation for recursive JSL and Classical JSON Schema is PTIME-complete and that it can be solved in $O(|J|^2|S|)$ time; then they showed that satisfiability for Classical JSON Schema is EXPTIME-complete for schemas without `uniqueItems` and is in 2EXPTIME otherwise. Again, their approach does not seem very easy to extend to modern JSON Schema, as it relies on modal logic and a very special kind of alternating tree automata.

While we are not aware of any other formal study about JSON Schema validation, dozens of validators have been designed and implemented in the past (please, see [6] for a rather complete list of about 50 implementations). Only some of them (about 11), like ajv [3] and Hyperjump [4], support modern JSON Schema and dynamic references. These validators, usually compile schemas in an efficient internal representation, that is later used for validation purposes. ajv, for instance, uses modern code generation techniques and compiles a schema into a specialized validator, designed to support advanced v8 optimization.

Validation has been widely studied in the context of XML data (see [22, 23], for instance). However, schema languages for XML are based on regular expressions, while JSON Schema exploits record types, recursion, and full boolean logics, and this makes it very difficult to import techniques from one field to the other.

Schema languages such as JSON Schema and type systems for functional languages are clearly related, and a lot of work has been invested in the analysis of the computational complexity of type checking and type inference for programming languages and for module systems (we will only cite [19], as an example). We are well aware of this research field, but we do not think that it is related to this specific work, since in that case the focus is on the analysis of *code* while JSON Schema validation analyses *instances of data structures*.

11 CONCLUSIONS AND OPEN PROBLEMS

Modern JSON Schema introduced annotation-dependent validation and dynamic references, which are widely regarded as complex additions, and which changed the evaluation model, hence invalidating the theory developed for Classical JSON Schema.

Here we provide the first published formalization for Modern JSON Schema. This formalization provides a language to unambiguously describe and discuss the standard, and a tool to understand its subtleties.

We use our formalization to study the complexity of validation of Modern JSON Schema. We proved that the problem is PSPACE-complete, and that a very small fragment of the language is already PSPACE-hard. We proved that this increase in asymptotic complexity is caused by dynamic references, while annotation-dependent validation without dynamic references can be decided in PTIME, and we have defined and implemented and experimented an explicit algorithm to this aim.

We defined a technique to eliminate dynamic references, at the price of a potential exponential increase in the schema size, and we used it to prove that data-complexity of validation is in PTIME.

We leave many open problems, such as the definition of a new notion of schema equivalence and inclusion that is compatible with annotation-dependent validation, the study of its properties, and the study of the computational complexity of the problems of satisfiability, validity, inclusion, and example generation.

REFERENCES

[1] 2022. jschon.dev. https://jschon.dev Online tool. Retrieved 14 October 2022..

[2] 2022. json-everything validator. https://json-everything.net/json-schema/ Online tool. Retrieved 14 October 2022..

[3] 2023. Ajv JSON Schema validator. https://ajv.js.org Retrieved 10 January 2023..

[4] 2023. Hyperjump JSON Schema Validator. https://json-schema.hyperjump.io/ Online tool. Retrieved 10 January 2023..

[5] 2023. JSON Schema meta-schema. https://json-schema.org/draft/2020-12/schema Retrieved 28 February 2023..

[6] 2023. JSON Schema validators. https://json-schema.org/implementations.html#validators Retrieved 10 January 2023..

[7] Henry Andrews. 2023. Modern JSON Schema. Available online at https://modern-json-schema.com/.

[8] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness Generation for JSON Schema. *Proc. VLDB Endow.* 15, 13 (2022), 4002–4014. https://www.vldb.org/pvldb/vol15/p4002-sartiani.pdf

[9] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas And Types For JSON Data. In *Proc. EDBT*. 437–439.

[10] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. In *Proc. SIGMOD Conference*. 2060–2063.

[11] Julian Bergman. 2023. Bowtie JSON Schema Meta Validator. https://github.com/bowtie-json-schema/bowtie Online tool. Version 0.67.0..

[12] Julian Bergman. 2023. JSON-Schema-Test-Suite (draft2020-12). https://github.com/json-schema-org/JSON-Schema-Test-Suite/tree/main/tests/draft2020-12

[13] T. Berners-Lee, R. Fielding, and L. Masinter. January 2005. *Uniform Resource Identifier (URI): Generic Syntax*. Technical Report. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc3986

[14] Jim Blackler. 2022. JSON Generator. Available at https://github.com/jimblackler/jsongenerator. Retrieved 19 September 2022..

[15] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proc. PODS*. 123–135. https://doi.org/10.1145/3034786.3056120

[16] Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoc. 2020. JSON: Data model and query languages. *Inf. Syst.* 89 (2020), 101478. https://doi.org/10.1016/j.is.2019.101478

[17] P. Bryan, K. Zyp, and M. Nottingham. Aprile 2013. *JavaScript Object Notation (JSON) Pointer*. Technical Report. Internet Engineering Task Force. https://www.rfc-editor.org/info/rfc6901

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

[19] Fritz Henglein and Harry G. Mairson. 1994. The Complexity of Type Inference for Higher-Order Typed lambda Calculi. *J. Funct. Program.* 4, 4 (1994), 435–477. https://doi.org/10.1017/S0956796800001143

[20] Mark Jacobson. 2021. The meaning of "additionalProperties" has changed. Available online at https://github.com/orgs/json-schema-org/discussions/57.

[21] Klaus-Jörn Lange and Peter Rossmanith. 1992. The Emptiness Problem for Intersections of Regular Languages. In *Mathematical Foundations of Computer Science 1992, 17th International Symposium, MFCS'92, Prague, Czechoslovakia, August 24-28, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 629)*, Ivan M. Havel and Václav Koubek (Eds.). Springer, 346–354. https://doi.org/10.1007/3-540-55808-X_33

[22] Wim Martens, Frank Neven, and Thomas Schwentick. 2009. Complexity of Decision Problems for XML Schemas and Chain Regular Expressions. *SIAM J. Comput.* 39, 4 (2009), 1486–1530. https://doi.org/10.1137/080743457

[23] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. 2006. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.* 31, 3 (2006), 770–813. https://doi.org/10.1145/1166074.1166076

[24] Oliver Neal. 2022. Ambiguous behaviour of "additionalProperties" when invalid. Available online at https://github.com/json-schema-org/json-schema-spec/issues/1172.

[25] JSON Schema Org. 2022. JSON Schema. Available at https://json-schema.org.

[26] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON Schema. In *Proc. WWW*. 263–273.

[27] Larry J. Stockmeyer. 1976. The Polynomial-Time Hierarchy. *Theor. Comput. Sci.* 3, 1 (1976), 1–22. https://doi.org/10.1016/0304-3975(76)90061-X

[28] Larry J. Stockmeyer and Albert R. Meyer. 1973. Word Problems Requiring Exponential Time: Preliminary Report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong (Eds.). ACM, 1–9. https://doi.org/10.1145/800125.804029

[29] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02 Retrieved 19 September 2022..

[30] A. Wright, H. Andrews, B. Hutton, and G. Dennis. 2022. *JSON Schema: A Media Type for Describing JSON Documents - draft-bhutton-json-schema-01.* Technical Report. Internet Engineering Task Force. https://json-schema.org/draft/2020-12/json-schema-core.html Retrieved 15 October 2022..

[31] A. Wright, G. Luff, and H. Andrews. 2017. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-wright-json-schema-validation-01.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-wright-json-schema-validation-01 Retrieved 19 September 2022..

A COMPLETE LIST OF RULES

A.1 Complete grammar

$q \in \mathsf{Num}, i \in \mathsf{Int}, k \in \mathsf{Str}, \mathit{absURI} \in \mathsf{Str}, f \in \mathsf{Str}, \mathit{format} \in \mathsf{Str}, p \in \mathsf{Str}, J \in \mathcal{J}\mathit{Val}$

$$
\begin{array}{lcl}
\mathit{Tp} & ::= & \texttt{"object"} \mid \texttt{"number"} \mid \texttt{"integer"} \mid \texttt{"string"} \mid \texttt{"array"} \mid \texttt{"boolean"} \mid \texttt{"null"} \\
S & ::= & \{\ \mathit{IndKeyOrT}\ (,\mathit{IndKeyOrT})^*\ (,\mathit{FstDep})^*\ (,\mathit{SndDep})^*\ \} \\
 & & \mid \{\ \mathit{FstDep}\ (,\mathit{FstDep})^*\ (,\mathit{SndDep})^*\ \} \mid \{\ \mathit{SndDep}\ (,\mathit{SndDep})^*\ \} \mid \{\ \} \\
 & & \mid \texttt{true} \mid \texttt{false} \\
\mathit{IndKeyOrT} & ::= & \mathit{IndKey} \mid \mathit{ITETriple} \mid \mathit{ContainsTriple} \mid \mathit{Other} \\
\mathit{IndKey} & ::= & \texttt{"exclusiveMinimum"} : q \mid \texttt{"exclusiveMaximum"} : q \\
 & & \mid \texttt{"minimum"} : q \mid \texttt{"maximum"} : q \mid \texttt{"multipleOf"} : q \\
 & & \mid \texttt{"pattern"} : p \mid \texttt{"minLength"} : i \mid \texttt{"maxLength"} : i \\
 & & \mid \texttt{"minProperties"} : i \mid \texttt{"maxProperties"} : i \\
 & & \mid \texttt{"required"} : [\ k_1, \ldots, k_n\ ] \\
 & & \mid \texttt{"uniqueItems"} : \texttt{true} \mid \texttt{"uniqueItems"} : \texttt{false} \\
 & & \mid \texttt{"minItems"} : i \mid \texttt{"maxItems"} : i \mid \texttt{"format"} : \mathit{format} \\
 & & \mid \texttt{"dependentRequired"} : \{\ k_1 : [\ k_1^1, \ldots, k_{o_1}^1\ ] \ldots, k_n : [\ k_1^n, \ldots, k_{o_n}^n\ ]\ \} \\
 & & \mid \texttt{"enum"} : [J_1, \ldots, J_n] \mid \texttt{"const"} : J_c \\
 & & \mid \texttt{"type"} : \mathit{Tp} \mid \texttt{"type"} : [\mathit{Tp}_1, \ldots, \mathit{Tp}_n] \\
 & & \mid \texttt{"\$id"} : \mathit{absURI} \mid \texttt{"\$ref"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f \mid \texttt{"\$dynamicRef"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f \\
 & & \mid \texttt{"\$defs"} : \{\ k_1 : S_1, \ldots, k_n : S_n\ \} \mid \\
 & & \mid \texttt{"\$anchor"} : \mathit{plain-name} \mid \texttt{"\$dynamicAnchor"} : \mathit{plain-name} \\
 & & \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{"prefixItems"} : [\ S_1, \ldots, S_n\ ] \mid \texttt{"contains"} : S \\
 & & \mid \texttt{"patternProperties"} : \{\ p_1 : S_1, \ldots, p_m : S_m\ \} \\
 & & \mid \texttt{"properties"} : \{\ k_1 : S_1, \ldots, k_m : S_m\ \} \\
 & & \mid \texttt{"propertyNames"} : S \mid \texttt{"dependentSchemas"} : \{\ k_1 : S_1, \ldots, k_n : S_n\ \} \\
\mathit{ITETriple} & ::= & (\texttt{"if"} : S,)^?\ \texttt{"then"} : S,\ \texttt{"else"} : S \\
\mathit{ContainsTriple} & ::= & (\texttt{"contains"} : S,)^?\ \texttt{"minContains"} : i\ (,\texttt{"maxContains"} : i)^? \\
\mathit{Other} & ::= & \texttt{"\$schema"} : k \mid \texttt{"\$vocabulary"} : k \mid \texttt{"\$comment"} : k \\
 & & \mid \texttt{"title"} : k \mid \texttt{"description"} : k \\
 & & \mid \texttt{"deprecated"} : b \mid \texttt{"readOnly"} : b \mid \texttt{"writeOnly"} : b \\
 & & \mid \texttt{"default"} : J \mid \texttt{"examples"} : [J_1, \ldots, J_n] \\
 & & \mid k : J \ \text{(with } k \text{ not cited as keyword in any other production)} \\
\mathit{FstDep} & ::= & \texttt{"additionalProperties"} : S \mid \texttt{"items"} : S \\
\mathit{SndDep} & ::= & \texttt{"unevaluatedProperties"} : S \mid \texttt{"unevaluatedItems"} : S
\end{array}
$$

Fig. 8. Complete grammar of normalized JSON Schema Draft 2020-12.

The complete grammar groups the keywords "if"-"then"-"else", and specifies that the presence of any keyword among "if"-"then"-"else" implies the presence of "then" and "else", which is enforced by adding a trivial "then" : {}, or "else" : {}, when one or both are missing; this presentation reduces the number of rules needed to formalize "if"-"then"-"else". In the same way, the grammar groups "contains"-"minContains"-"maxContains" and imposes the presence of "minContains" when any of

the three is present, which is enforced by adding the default `"minContains" : 1` when `"minContains"` is missing.

A.2 Terminal keywords

The types and the conditions of the terminal keywords are specified in the following table. There, the length operator $|J|$ counts the number of characters of a string, the number of fields of an object, and the number of elements of an array. names(J) extracts the names of an object. When p is a pattern or a format, we use $L(p)$ to indicate the corresponding set of strings.

When *TypeOf*(*kw*) is *no type* then the assertion does not have the (*kw*Triv) rule and does not have the condition $\mathit{TypeOf}(J) = \mathit{TypeOf}(kw)$ in the (*kw*) rule.

assertion *kw*:J'	*TypeOf*(*kw*)	cond(J,*kw*:J')
`"enum"` : $[J_1, \ldots, J_n]$	*no type*	$J \in \{\!\|\, J_1, \ldots, J_n \,\|\!\}$
`"const"` : J_c	*no type*	$J = J_c$
`"type"` : Tp	*no type*	$\mathit{TypeOf}(J) = \mathsf{Tp}$
`"type"` : $[\mathsf{Tp}_1, \ldots, \mathsf{Tp}_n]$	*no type*	$\mathit{TypeOf}(J) \in \{\!\|\, \mathsf{Tp}_1, \ldots, \mathsf{Tp}_n \,\|\!\}$
`"exclusiveMinimum"`: q	number	$J > q$
`"exclusiveMaximum"`: q	number	$J < q$
`"minimum"`: q	number	$J \geq q$
`"maximum"`: q	number	$J \leq q$
`"multipleOf"`: q	number	$\exists i \in \mathsf{Int}.\ J = i \times q$
`"pattern"`: p	string	$J \in L(p)$
`"minLength"`: i	string	$\|J\| \geq i$
`"maxLength"`: i	string	$\|J\| \leq i$
`"minProperties"`: i	object	$\|J\| \geq i$
`"maxProperties"`: i	object	$\|J\| \leq i$
`"required"` : $[\, k_1, \ldots, k_n \,]$	object	$\forall i.\ k_i \in \mathsf{names}(J)$
`"uniqueItems"`: `true`	array	$J = [J_1, \ldots, J_n]$ with $n \geq 0$ $\wedge\ \forall i, j.\ 1 \leq i \neq j \leq n \Rightarrow J_i \neq J_j$
`"uniqueItems"`: `false`	array	T
`"minItems"`: i	array	$\|J\| \geq i$
`"maxItems"`: i	array	$\|J\| \leq i$
`"format"`: *format*	string	$J \in L(\mathit{format})$
`"dependentRequired"` : $\{\, k_1 : [\, k_1^1, \ldots, k_{o_1}^1 \,]$ $\ldots, k_n : [\, k_1^n, \ldots, k_{o_n}^n \,]\ \}$	object	$\forall i \in \{\!\|\, 1 \ldots n \,\|\!\}.$ $k_i \in \mathsf{names}(J)$ $\Rightarrow \{\!\|\, k_1^i, \ldots, k_{o_i}^i \,\|\!\} \subseteq \mathsf{names}(J)$

A.3 The boolean applicators

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

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

$$\frac{\forall i \in \{\!\|\, 1 \ldots n \,\|\!\}.\ C \vdash^{\mathsf{S}} J\ ?\ S_i \longrightarrow (r_i, \kappa_i) \qquad r = (\ |\{\!\|\, i \mid r_i = T \,\|\!\}| = 1\ )}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"oneOf"} : [S_1, ..., S_n] \longrightarrow (r, \bigcup_{i \in \{\!\|\, 1 \ldots n \,\|\!\}} \kappa_i)} \quad \text{(oneOf)}$$

$$\frac{C \vdash^{\mathsf{S}} J\ ?\ S \longrightarrow (r, \kappa)}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"not"} : S \longrightarrow (\neg r, \kappa)} \quad \text{(not)}$$

A.4 References

$$\frac{S' = \mathtt{get}(\mathtt{load}(absURI), f) \qquad C{+}?absURI \ \vdash^{\mathsf{S}} \ J \ ? \ S' \rightarrow (r, \kappa)}{C \vdash^{\mathsf{K}} \ J \ ? \ \texttt{"\$ref"} : absURI \cdot \texttt{"\#"} \cdot f \rightarrow (r, \kappa)} \quad (\$\mathrm{ref})$$

$$\frac{\begin{array}{c}\mathtt{dget}(\mathtt{load}(absURI), f) \neq \bot \qquad fURI = \mathtt{fstURI}(C{+}?absURI, f) \\ S' = \mathtt{dget}(\mathtt{load}(fURI), f) \qquad C{+}?fURI \ \vdash^{\mathsf{S}} \ J \ ? \ S' \rightarrow (r, \kappa)\end{array}}{C \vdash^{\mathsf{K}} \ J \ ? \ \texttt{"\$dynamicRef"} : absURI \cdot \texttt{"\#"} \cdot f \rightarrow (r, \kappa)} \quad (\$\mathrm{dynamicRef})$$

$$\frac{\begin{array}{c}\mathtt{dget}(\mathtt{load}(absURI), f) = \bot \\ S' = \mathtt{get}(\mathtt{load}(absURI), f) \qquad C{+}?absURI \ \vdash^{\mathsf{S}} \ J \ ? \ S' \rightarrow (r, \kappa)\end{array}}{C \vdash^{\mathsf{K}} \ J \ ? \ \texttt{"\$dynamicRef"} : absURI \cdot \texttt{"\#"} \cdot f \rightarrow (r, \kappa)} \quad (\$\mathrm{dynamicRefAsRef})$$

A.5 If-then-else

$$\frac{C \vdash^{\mathsf{S}} \ J \ ? \ S_i \rightarrow (T, \kappa_i) \qquad C \vdash^{\mathsf{S}} \ J \ ? \ S_t \rightarrow (r, \kappa_t)}{C \vdash^{\mathsf{L}} \ J \ ? \ (\texttt{"if"} : S_i + \texttt{"then"} : S_t \ + \texttt{"else"} : S_e \ ) \rightarrow (r, \kappa_i \cup \kappa_t)} \quad (\text{if-true})$$

$$\frac{C \vdash^{\mathsf{S}} \ J \ ? \ S_i \rightarrow (F, \kappa_i) \qquad C \vdash^{\mathsf{S}} \ J \ ? \ S_e \rightarrow (r, \kappa_e)}{C \vdash^{\mathsf{L}} \ J \ ? \ (\texttt{"if"} : S_i + \texttt{"then"} : S_t \ + \texttt{"else"} : S_e \ ) \rightarrow (r, \kappa_i \cup \kappa_e)} \quad (\text{if-false})$$

$$C \vdash^{\mathsf{L}} \ J \ ? \ (\texttt{"then"} : S_t \ + \texttt{"else"} : S_e \ ) \rightarrow (T, \emptyset) \quad (\text{missing-if})$$

A.6 Object and array applicators: independent applicators

Independent applicators are defined by a set of two rules: (kwTriv) and (kw); the (kwTriv) rules are specified by the following table; the (kw) rules are specified after the table.

assertion $kw : J'$	*TypeOf* (kw)
$\texttt{"patternProperties"} : \{ p_1 : S_1, \ldots, p_m : S_m \}$	object
$\texttt{"properties"} : \{ k_1 : S_1, \ldots, k_m : S_m \}$	object
$\texttt{"propertyNames"} : S$	object
$\texttt{"prefixItems"} : S$	array
$\texttt{"contains"} : S$	array
$\texttt{"dependentSchemas"} : \{ k_1 : S_1, \ldots, k_n : S_n \}$	object

$$\frac{\begin{array}{c}J = \{ k'_1 : J_1, \ldots, k'_n : J_n \} \qquad \{\!|\, (i_1, j_1), \ldots, (i_l, j_l) \,|\!\} = \{\!|\, (i, j) \mid k'_i \in L(p_j) \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}. \, C \vdash^{\mathsf{S}} \ J_{i_q} \ ? \ S_{j_q} \rightarrow (r_q, \kappa_q) \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}})\end{array}}{C \vdash^{\mathsf{K}} \ J \ ? \ \texttt{"patternProperties"} : \{ p_1 : S_1, \ldots, p_m : S_m \} \rightarrow (r, \{\!|\, k'_{i_1}, \ldots, k'_{i_l} \,|\!\})} \quad (\text{patternProperties})$$

$$\frac{\begin{array}{c}J = \{ k'_1 : J_1, \ldots, k'_n : J_n \} \qquad \{\!|\, (i_1, j_1), \ldots, (i_l, j_l) \,|\!\} = \{\!|\, (i, j) \mid k'_i = k_j \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}. \, C \vdash^{\mathsf{S}} \ J_{i_q} \ ? \ S_{j_q} \rightarrow (r_q, \kappa_q) \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}})\end{array}}{C \vdash^{\mathsf{K}} \ J \ ? \ \texttt{"properties"} : \{ k_1 : S_1, \ldots, k_m : S_m \} \rightarrow (r, \{\!|\, k'_{i_1}, \ldots, k'_{i_l} \,|\!\})} \quad (\text{properties})$$

$$\frac{J = \{ k_1 : J_1, \ldots, k_n : J_n \} \qquad \forall i \in \{\!|\, 1 \ldots n \,|\!\}. \, C \vdash^{\mathsf{S}} \ k_i \ ? \ S \rightarrow (r_i, \kappa_i) \qquad r = \wedge(\{\!|\, r(\kappa_i) \,|\!\}^{i \in \{\!|\, 1 \ldots n \,|\!\}})}{C \vdash^{\mathsf{K}} \ J \ ? \ \texttt{"propertyNames"} : S \rightarrow (r, \emptyset)} \quad (\text{propertyNames})$$

$$\frac{J = [\,J_1, \ldots, J_m\,] \qquad \forall i \in \{\!|\, 1 \ldots \min(n,m) \,|\!\}.\ C \vdash^{\mathsf{S}} J_i \,?\, S_i \to (r_i, \kappa_i) \qquad r = \wedge(\{\!|\, r_i \,|\!\}^{i \in \{\!|\, 1 \ldots \min(n,m) \,|\!\}})}{C \vdash^{\mathsf{K}} J \,?\, \texttt{"prefixItems"} : [\,S_1, \ldots, S_n\,] \to (r, \{\!|\, 1, \ldots, \min(n,m) \,|\!\})} \text{(prefixItems)}$$

$$\frac{\begin{array}{c} J = [\,J_1, ..., J_n\,] \\ \forall i \in \{\!|\, 1 \ldots n \,|\!\}.\ C \vdash^{\mathsf{S}} J_i \,?\, S \to (r_i, \kappa_i) \qquad \kappa_c = \{\!|\, i \mid r_i = T \,|\!\} \qquad r_c = (i \le |\kappa_c| \le j) \end{array}}{C \vdash^{\mathsf{K}} J \,?\, (\texttt{"contains"} : S + \texttt{"minContains"} : i + \texttt{"maxContains"} : j) \to (r_c, \kappa_c)} \text{(contains-max)}$$

$$\frac{\begin{array}{c} \text{rule (contains-max) does not apply and} \qquad J = [\,J_1, ..., J_n\,] \\ \forall i \in \{\!|\, 1 \ldots n \,|\!\}.\ C \vdash^{\mathsf{S}} J_i \,?\, S \to (r_i, \kappa_i) \qquad \kappa_c = \{\!|\, i \mid r_i = T \,|\!\} \qquad r_c = (i \le |\kappa_c|) \end{array}}{C \vdash^{\mathsf{K}} J \,?\, (\texttt{"contains"} : S + \texttt{"minContains"} : i) \to (r_c, \kappa_c)} \text{(contains-no-max)}$$

$$\frac{\text{rules (contains-max) and (contains-no-max) do not apply}}{C \vdash^{\mathsf{K}} J \,?\, (\texttt{"minContains"} : i\ (+\texttt{"maxContains"} i)^{?}) \to (T, \emptyset)} \text{(missing-contains)}$$

$$\frac{\begin{array}{c} J = \{\, k'_1 : J_1, \ldots,\ k'_m : J_m \,\} \qquad \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, i \mid i \in \{\!|\, 1 \ldots n \,|\!\},\ k_i \in \{\!|\, k'_1, \ldots,\ k'_m \,|\!\} \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J \,?\, S_{i_q} \to \kappa_q \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{K}} J \,?\, \texttt{"dependentSchemas"} : \{\, k_1 : S_1, \ldots, k_n : S_n \,\} \to (r, \bigcup_{q \in \{\!|\, 1 \ldots l \,|\!\}} \kappa_q)} \text{(dependentSchemas)}$$

A.7 Dependent keywords

Dependent keywords are defined by a set of two rules: (kwTriv) and (kw); the (kwTriv) rule is specified by the following table.

assertion $kw : J'$	*TypeOf* (kw)
"unevaluatedProperties" : S	object
"additionalProperties" : S	object
"unevaluatedItems" : S	array
"items" : S	array

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

$$\frac{\begin{array}{c} J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} \qquad C \vdash^{\mathsf{L}} J \,?\, \vec{K} \to (r, \kappa) \\ \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, i \mid 1 \le i \le n \wedge\ k_i \notin L(\texttt{propsOf}(\vec{K})) \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J_{i_q} \,?\, S \to (r_q, \kappa_q) \qquad r' = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{L}} J \,?\, (\vec{K} + \texttt{"additionalProperties"} : S) \to (r \wedge r', \{\!|\, k_1 \ldots, k_n \,|\!\})} \text{(additionalProperties)}$$

$$\frac{\begin{array}{c} J = \{\, k_1 : J_1, \ldots, k_n : J_n \,\} \qquad C \vdash^{\mathsf{L}} J \,?\, \vec{K} \to (r, \kappa) \\ \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, i \mid 1 \le i \le n \wedge\ k_i \notin \kappa \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}} J_{i_q} \,?\, S \to (r_q, \kappa_q) \qquad r' = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{L}} J \,?\, (\vec{K} + \texttt{"unevaluatedProperties"} : S) \to (r \wedge r', \{\!|\, k_1 \ldots, k_n \,|\!\})} \text{(unevaluatedProperties)}$$

$$\texttt{maxPrefixOf}(\vec{K}) = \quad m \quad \text{if } (\texttt{"prefixItems"} : [\, S_1, \ldots, S_m \,]) \in \vec{K} \text{ for some } S_1, \ldots, S_n$$
$$\texttt{maxPrefixOf}(\vec{K}) = \quad 0 \quad \text{if } (\texttt{"prefixItems"} : [\, S_1, \ldots, S_m \,]) \notin \vec{K} \text{ for any } S_1, \ldots, S_n$$

$$\frac{J = [\, J_1, ..., J_n \,] \qquad C \vdash^{\texttt{L}} J \,?\, \vec{K} \to (r, \kappa) \qquad \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, 1 \ldots n \,|\!\} \setminus \{\!|\, 1 \ldots \mathit{maxPrefixOf}(\vec{K}) \,|\!\} \qquad \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\texttt{S}} J_{i_q} \,?\, S \to \kappa_q \qquad r' = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}})}{C \vdash^{\texttt{L}} J \,?\, (\vec{K} + \texttt{"items"} : S) \to (r \wedge r', \{\!|\, 1 \ldots, n \,|\!\})} \text{(items)}$$

$$\frac{J = [\, J_1, ..., J_n \,] \qquad C \vdash^{\texttt{L}} J \,?\, \vec{K} \to (r, \kappa) \qquad \{\!|\, i_1, \ldots, i_l \,|\!\} = \{\!|\, 1 \ldots n \,|\!\} \setminus \kappa \qquad \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\texttt{S}} J_{i_q} \,?\, S \to \kappa_q \qquad r' = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}})}{C \vdash^{\texttt{L}} J \,?\, (\vec{K} + \texttt{"unevaluatedItems"} : S) \to (r \wedge r', \{\!|\, 1 \ldots, n \,|\!\})} \text{(unevaluatedItems)}$$

$$\frac{k : J' \in \mathit{IndKeyOrT} \qquad C \vdash^{\texttt{L}} J \,?\, \vec{K} \to (r_l, \kappa_l) \qquad C \vdash^{\texttt{K}} J \,?\, K \to (r, \kappa)}{C \vdash^{\texttt{L}} J \,?\, (\vec{K} + K) \to (r_l \wedge r, \kappa_l \cup \kappa)} \text{(klist-(N+1))}$$

$$C \vdash^{\texttt{L}} J \,?\, [\!|\,|\!] \to (T, [\!|\,|\!]) \text{(klist-0)}$$

A.8 The trivial rule

$$\frac{kw \in \mathit{Other}}{C \vdash^{\texttt{K}} J \,?\, kw : J' \to (T, \emptyset)} \text{(KW-trivial)}$$

A.9 Schema rules

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

$$\frac{C \vdash^{\texttt{L}} J \,?\, [\!|\, K_1, \ldots, K_n \,|\!] \to (T, \kappa)}{C \vdash^{\texttt{S}} J \,?\, \{\, K_1, \ldots, K_n \,\} \to (T, \kappa)} \text{(schema-true)} \qquad \frac{C \vdash^{\texttt{L}} J \,?\, [\!|\, K_1, \ldots, K_n \,|\!] \to (F, \kappa)}{C \vdash^{\texttt{S}} J \,?\, \{\, K_1, \ldots, K_n \,\} \to (F, \emptyset)} \text{(schema-false)}$$

B FUNCTIONS $\texttt{get}(S, f)$ AND $\texttt{dget}(S, f)$

$\texttt{dget}(S, f)$ searches inside S for a subschema $\{\, \texttt{"\$dynamicAnchor"} : f, K_1, \ldots, K_n \,\}$ and returns it. However, it only searches inside known keywords that contain a schema as a parameter, and the search is stopped by the presence of an internal "$id" keyword, because "$id" indicates that the subschema is a separate resource, with a different URI.

We formalize this behaviour by defining two functions:

(1) $\texttt{dgets}(S, f)$, that searches $\texttt{"\$dynamicAnchor"} : f$ into S, and, if not found, invokes $\texttt{dgetk}(k : J, f)$ on each keyword $k : J$ to search inside the schema;
(2) $\texttt{dgetk}(k : J, f)$ that invokes $\texttt{dgets}(S, f)$ to search f inside the parts of J that are known to contain a subschema.

The function $\texttt{dget}(S, f)$ is defined as $\texttt{dgets}(\texttt{StripId}(S), f)$, where $\texttt{StripId}(S)$ removes any outermost "$id" keyword from S. This is necessary since dgets interrupts its search when it meets a "$id" keyword, but the presence of an "$id" in the outermost schema should not interrupt the search:

$$\begin{array}{lcll} \texttt{StripId}(\{\, \texttt{"\$id"} : \mathit{URI}, \vec{K} \,\}) & = & \{\, \vec{K} \,\} & \\ \texttt{StripId}(S) & = & S & \text{otherwise} \end{array}$$

The functions $\texttt{dgets}(S, f)$ and $\texttt{dgetk}(S, f)$ are defined as follows, where *max* returns the maximum element of a set that contains either schemas or $\bot$, according to the trivial order defined by $\bot \leq S$: that is, *max* select the only element in the set that is different from $\bot$, if it exists and is unique. If we assume that no plain-name is used by two different anchors in the same schema, then, in line 4, there exists at most one value of i such that $\texttt{dget}(K_i, f) \neq \bot$, hence the maximum is well defined, and the same holds for lines 6 and 7.

0	$\texttt{dget}(S, f)$	=	$\texttt{dgets}(\texttt{StripId}(S), f)$	
1	$\texttt{dgets}(\texttt{true/false}, f)$	=	$\bot$	
2	$\texttt{dgets}(\{\, \vec{K} \,\}, f)$	=	$\bot$	if $\texttt{"\$id"} : \mathit{URI} \in \vec{K}$
3	$\texttt{dgets}(\{\, \vec{K} \,\}, f)$	=	$\{\, \vec{K} \,\}$	if $\texttt{"\$dynamicAnchor"} : f \in \vec{K}$ and 2 does not apply
4	$\texttt{dgets}(\{\, K_1, \ldots, K_n \,\}, f)$	=	$max_{i \in \{\!\mid 1 \ldots n \mid\!\}} \texttt{dgetk}(K_i, f)$	if 2, 3 do not apply
5	$\texttt{dgetk}(kw : S, f)$	=	$\texttt{dgets}(S, f)$	$kw \in \mathit{kwSimPar}$
6	$\texttt{dgetk}(kw : [\, S_1, \ldots, S_n \,], f)$	=	$max_{i \in \{\!\mid 1 \ldots n \mid\!\}} \texttt{dgets}(S_i, f)$	$kw \in \mathit{kwArrPar}$
7	$\texttt{dgetk}(kw : \{\, k_1 : S_1, \ldots, k_n : S_n \,\}, f)$	=	$max_{i \in \{\!\mid 1 \ldots n \mid\!\}} \texttt{dgets}(S_i, f)$	$kw \in \mathit{kwObjPar}$
8	$\texttt{dgetk}(kw : J, f)$	=	$\bot$	$kw \in \mathit{kwOther}$

where

$$
\begin{aligned}
\mathit{kwSimPar} &= \{\!\mid \texttt{"not"}, \texttt{"contains"}, \texttt{"propertyNames"}, \texttt{"items"}, \texttt{"additionalProperties"}, \\
&\qquad \texttt{"unevaluatedProperties"}, \texttt{"unevaluatedItems"} \mid\!\} \\
\mathit{kwObjPar} &= \{\!\mid \texttt{"\$defs"}, \texttt{"patternProperties"}, \texttt{"properties"}, \texttt{"dependentSchemas"} \mid\!\} \\
\mathit{kwArrPar} &= \{\!\mid \texttt{"anyOf"}, \texttt{"allOf"}, \texttt{"oneOf"}, \texttt{"prefixItems"} \mid\!\} \\
\mathit{kwOther} &= \mathrm{Str} \setminus \mathit{kwSimPar} \setminus \mathit{kwObjPar} \setminus \mathit{kwArrPar}
\end{aligned}
$$

The four lines of `dgetk` definition specify that the search for a `"$dynamicAnchor"` keyword is performed only inside keywords that are known and whose parameter contains a schema object in a known position. For example, the search is not performed inside a user-defined keyword or inside `"const"` or `"default"`.

When f is a plain-name, then the function `get` is identical to `dget`, but it extends case 3, since it matches both `"$anchor"` and `"$dynamicAnchor"`:

0	$\texttt{get}(S, f)$	=	$\texttt{gets}(\texttt{StripId}(S), f)$	
1	$\texttt{gets}(\texttt{true/false}, f)$	=	$\bot$	
2	$\texttt{gets}(\{\, \vec{K} \,\}, f)$	=	$\bot$	if $\texttt{"\$id"} : \mathit{URI} \in \vec{K}$
3	$\texttt{gets}(\{\, \vec{K} \,\}, f)$	=	$\{\, \vec{K} \,\}$	if $\texttt{"\$dynamicAnchor"} : f \in \vec{K}$ or $\texttt{"\$anchor"} : f \in \vec{K}$ and 2 does not apply
4	$\texttt{gets}(\{\, K_1, \ldots, K_n \,\}, f)$	=	$max_{i \in \{\!\mid 1 \ldots n \mid\!\}} \texttt{getk}(K_i, f)$	if 2, 3 do not apply
5	$\texttt{getk}(kw : S, f)$	=	$\texttt{gets}(S, f)$	$kw \in \mathit{kwSimPar}$
...				

When f is a JSON Pointer, it should be interpreted by $\texttt{get}(S, F)$ according to JSON Pointer specifications of RFC6901 [17]. However, Draft 2020-12 specs [30] (Section 9.2.1) specify that the behaviour is undefined when the pointer crosses resource boundaries .

C WHERE TO SEARCH FOR DYNAMIC ANCHORS

In Remark 1 we discuss the difference between looking for a dynamic reference in C or in $C+?absURI$. We report here a concrete test.

When the schema of Figure 9 is applied to the JSON instance { "children" : { } }, the keyword "$dynamicRef" : "https://example.com/tree#node" is applied in a dynamic context that only contains "https://example.com/outer", which denotes a resources that does not contain any "$dynamicAnchor" : "node" keyword. The only such keyword in the schema is at line 7, and is belongs to the embedded resource "https://example.com/tree". This specific example relies on the fact that an embedded resource, as identified by the "$id" keyword, is not part of the scope of the embedding resource, the one identified by "https://example.com/outer", but it could be rephrased by putting the embedded resource in a separate file. Hence, if "$dynamicRef" : "https://example.com/tree#node" were resolved using the dynamic context only, if would raise a run-time failure. But our rule dictates that it is resolved in $C+?absURI$, that is in

"https://example.com/outer" + "https://example.com/tree"

hence raising a validation error since the value of "children" is not an array. In Figure 9 we see how four different validators behaved at the date of 10^{th} Jan 2023: the first two exhibit an internal bug, and the other two use the approach that we formalized; none of them reports a resolution error due to the fact that "$dynamicAnchor" : "node" is not present in the dynamic scope. Hence, they validate the choice to look for the fragment id f inside $C+?absURI$, and not inside C only.

Schema:

```
1 { "$schema": "https://json-schema.org/draft/2020-12/schema","$id":
2   "https://example.com/outer",
3   "$dynamicRef": "https://example.com/tree#node",
4   "$defs": {
5       "https://example.com/tree": {
6         "$id": "https://example.com/tree",
7         "$dynamicAnchor": "node",
8         "type": "object",
9         "properties": {
10            "data": true,
11            "children": {
12               "type": "array",
13               "items": { "$dynamicRef": "#node" }
14            }
15          }
16        }
17    }
18 }
```

Instance:

```
  { "children" : {}}
```

Output:

```
1 https://json-schema.hyperjump.io/:        TypeError: s[t] is undefined
2 https://tryjsonschematypes.appspot.com:   An error occurred - java.lang.NullPointerException
3 https://json-everything.net/json-schema/ "valid": false
4                                            "evaluationPath": "/$dynamicRef/properties/children"
5 https://jschon.dev/:                  "valid": false
6                                         "keywordLocation": "/$dynamicRef/properties/children/type"
```

Fig. 9. Different behaviour of several JSON Schema validators.

D PROOFS

From Section 5:

Theorem 2. Given a QBF closed formula $\psi = Q_1 x_1 \dots Q_n x_n.\ \phi$ and the corresponding schema S_ψ with base URI b, ψ is valid if, and only if, for every J, $[\![\, b \,]\!] \vdash^S J \,?\, S_\psi \to T$.

Proof. Consider a QBF formula $\psi = Q_1 x_1 \dots Q_n x_n.\ \phi$ and the corresponding schema S_ψ. We say that a context C is well-formed for S_ψ if (1) its elements are URIs of resources of S_ψ and (2) in case it contains `"urn:close"`, then `"urn:close"` is the last element of C. We associate every well-formed context C to an assignment A_C as follows:

$$A_C(x_i) = T \;\Leftrightarrow\; \texttt{"urn:setvar"}\cdot i \in C$$
$$A_C(x_i) = F \;\Leftrightarrow\; \texttt{"urn:setvar"}\cdot i \notin C$$

Given a QBF formula ψ' that may contain open variables and an assignment A that is defined for every variable of ψ', we use $Valid(\psi', A)$ to indicate the fact that ψ' is valid when every open variable is substituted with its value in A.

The index of a context C, $Ind(C)$, is the maximal i such that $\texttt{"urn:setvar"}\cdot i \in C$, and is 0 if no $\texttt{"urn:setvar"}\cdot i$ belongs to C.

We prove, by induction, the following properties, for every context C well-formed for S_ψ, where we define $Tail_0(\psi) = \phi$ and $Tail_{i+1}(\psi) = Q_{n-i} x_{n-i} Tail_i(\psi)$, so that $Tail_i(\psi) = Q_{n-i+1} x_{n-i+1} \dots Q_n x_n.\ \phi$:

(1a) $\texttt{"urn:close"} \in C \;\Rightarrow\; C \vdash^S J \,?\, \{\ \texttt{"\$dynamicRef"} : \texttt{"x"}\cdot i\ \} \to T \Leftrightarrow A_C(x_i) = T$

(1b) $\texttt{"urn:close"} \in C \;\Rightarrow\; C \vdash^S J \,?\, \{\ \texttt{"\$dynamicRef"} : \texttt{"not.x"}\cdot i\ \} \to T \Leftrightarrow A_C(x_i) = F$

(2) $C \vdash^S J \,?\, \{\ \texttt{"\$ref"} : \texttt{"urn:close\#phi"}\ \} \to T \;\Leftrightarrow\; Valid(\phi, A_C)$

(3) $Ind(C) \le n - i \Rightarrow$
$C \vdash^S J \,?\, \{\ \texttt{"\$ref"} : \texttt{"urn:start\#quant.x"}\cdot(n - i + 1)\ \} \to T \;\Leftrightarrow\; Valid(Tail_i.\ \phi, A_C)$

The theorem follows from case (3), in the case $i = n$, with C equal to the base context $[\![\, \texttt{"urn:psi"} \,]\!]$.

Property (1a): it holds since $\texttt{"x"}\cdot i$ is defined as `"anyOf" : [true]` in $\texttt{"urn:setvar"}\cdot i$ and as `"anyOf" : [false]` in `"urn:close"`. In a well-formed context that contains `"urn:close"`, when $\texttt{"urn:setvar"}\cdot i$ is present, then it precedes `"urn:close"`, hence the value of $\{\ \texttt{"\$dynamicRef"} : \texttt{"x"}\cdot i\ \}$ is determined by $\texttt{"urn:setvar"}\cdot i$, hence is `"anyOf" : [true]`. If $\texttt{"urn:setvar"}\cdot i$ is absent, then the value of $\{\ \texttt{"\$dynamicRef"} : \texttt{"x"}\cdot i\ \}$ is defined by `"urn:close"`, hence is `"anyOf" : [false]`. (1b) is analogous.

Property (2): by induction on ϕ, using property (1) for the x_i and $\neg x_i$, and induction for $S_1 \wedge S_2$ and $S_1 \vee S_2$.

Property (3): by induction on i. Case $i = 1$: we want to prove that

$$Ind(C) \le n - 1 \;\Rightarrow\; C \vdash^S J \,?\, \{\ \texttt{"\$ref"} : \texttt{"urn:start\#quant.x"}\cdot(n)\ \} \to T \;\Leftrightarrow\; Valid(Tail_1.\ \phi, A_C)$$

By definition, $Tail_1.\ \phi = Q_n x_n.\phi$; we assume that $Q_n = \forall$; the case for $Q_n = \exists$ is analogous.

From $Q_n = \forall$ we get:

$$Valid(Tail_1.\ \phi, A_C) \Leftrightarrow (Valid(\phi, A_C[x_n \leftarrow T]) \wedge Valid(\phi, A_C[x_n \leftarrow F]))$$

Since $Q_n = \forall$, then the body of $\texttt{"urn:start\#quant.x"}\cdot n$ is

$$\texttt{"allOf"} : [\ \{\texttt{"\$ref"} : \texttt{"urn:setvar"}\cdot n\cdot\texttt{"\#afterq"}\cdot n\}\ ,\ \{\texttt{"\$ref"} : \texttt{"urn:start\#afterq"}\cdot n\}\ ]$$

The body of $\texttt{"urn:setvar"}\cdot n\cdot\texttt{"\#afterq"}\cdot n$ is `"$ref" : "urn:close#phi"`, and the body of $\texttt{"urn:start\#afterq"}\cdot n$ is equal. Hence, the assertion $\texttt{"\$ref"} : \texttt{"urn:setvar"}\cdot n\cdot\texttt{"\#afterq"}\cdot n$ evaluates `"$ref" : "urn:close#phi"` in a context C^T which is C extended with $\texttt{"\$ref"} : \texttt{"urn:setvar"}\cdot n$; by definition, A_{C^T} is equal to $A_C[x_n \leftarrow T]$. Hence, by property (2),

$$C^T \vdash^S J \,?\, \{\ \texttt{"\$ref"} : \texttt{"urn:close\#phi"}\ \} \to T \;\Leftrightarrow\; Valid(\phi, A_{C^T}) \;\Leftrightarrow\; Valid(\phi, A_{C[x_n \leftarrow T]})$$

The assertion $\texttt{"\$ref"} : \texttt{"urn:start\#afterq"} \cdot n$ evaluates $\texttt{"\$ref"} : \texttt{"urn:close\#phi"}$ in the same context C, hence A_C sets x_n to F, since $Ind(C) \leq n-1$. Hence, by property (2),

$$C \vDash^S J\ ?\ \{\ \texttt{"\$ref"} : \texttt{"urn:close\#phi"}\ \} \rightarrow T \quad \Leftrightarrow \quad Valid(\phi, A_C) \quad \Leftrightarrow \quad Valid(\phi, A_{C[x_n \leftarrow F]})$$

We can then conclude that

$$\texttt{"allOf"} : [\ \{\texttt{"\$ref"} : \texttt{"urn:setvar"} \cdot n \cdot \texttt{"\#afterq"} \cdot n\}\ ,\ \{\texttt{"\$ref"} : \texttt{"urn:start\#afterq"} \cdot n\}\ ]$$

evaluates to T if, and only if $Valid(\phi, A_C[x_n \leftarrow T]) \wedge Valid(\phi, A_C[x_n \leftarrow F])$, that is, $Valid(Tail_1.\ \phi, A_C)$.

Case $2 \leq i \leq n$: we want to prove that

$$Ind(C) \leq n-i \implies C \vDash^S J\ ?\ \{\ \texttt{"\$ref"} : \texttt{"urn:start\#quant.x"} \cdot (n-i+1)\ \} \rightarrow T \quad \Leftrightarrow \quad Valid(Tail_i.\ \phi, A_C)$$

By definition, $Tail_i.\ \phi = Q_{n-i+1} x_{n-i+1}.Tail_{i-1}.\ \phi$; we assume that $Q_{n-i+1} = \forall$; the case for $Q_{n-i+1} = \exists$ is analogous.

From $Q_{n-i+1} = \forall$ we get:

$$Valid(Tail_i.\ \phi, A_C) \Leftrightarrow (Valid(Tail_{i-1}.\ \phi, A_C[x_{n-i+1} \leftarrow T]) \wedge Valid(Tail_{i-1}.\ , A_C[x_{n-i+1} \leftarrow F]))$$

Since $Q_{n-i+1} = \forall$, then the body of $\texttt{"urn:start\#quant.x"} \cdot (n-i+1)$ is

$$\texttt{"allOf"} : [\ \{\texttt{"\$ref"} : \texttt{"urn:setvar"} \cdot (n-i+1) \cdot \texttt{"\#afterq"} \cdot (n-i+1)\}\ ,\ \{\texttt{"\$ref"} : \texttt{"urn:start\#afterq"} \cdot (n-i+1)\}\ ]$$

The body of $\texttt{"urn:setvar"} \cdot (n-i+1) \cdot \texttt{"\#afterq"} \cdot (n-i+1)$ is $\texttt{"\$ref"} : \texttt{"urn:start\#quant.x"} \cdot (n-i+2)$, and the body of $\texttt{"urn:start\#afterq"} \cdot (n-i+1)$ is equal. Hence, the assertion $\texttt{"\$ref"} : \texttt{"urn:setvar"} \cdot (n-i+1) \cdot \texttt{"\#afterq"} \cdot (n-i+1)$ evaluates $\texttt{"urn:start\#quant.x"} \cdot (n-i+2)$ in a context C^T which is C extended with $\texttt{"\$ref"} : \texttt{"urn:setvar"} \cdot (n-i+1)$; by definition, A_{C^T} is equal to $A_C[x_{n-i+1} \leftarrow T]$. Hence, by property (2),

$$C^T \vDash^S J\ ?\ \{\ \texttt{"\$ref"} : \texttt{"urn:close\#phi"}\ \} \rightarrow T \quad \Leftrightarrow \quad Valid(\phi, A_{C[x_{n-i+1} \leftarrow T]})$$

The assertion $\texttt{"\$ref"} : \texttt{"urn:start\#afterq"} \cdot (n-i+1)$ evaluates $\texttt{"urn:start\#quant.x"} \cdot (n-i+2)$ in the same context C with $Ind(C) \leq n-i$, hence A_C sets x_{n-i+1} to F, by definition of A_C.

We rewrite $(n-i+2)$ as $(n-(i-1)+1)$ and we observe that $Ind(C) \leq n-i$ implies that $Ind(C + \texttt{"urn:setvar"} \cdot (n-i+1)) \leq n-(i-1)$, so that we can apply induction on i, which provides the following inductive hypothesis:

$$C + \texttt{"urn:setvar"} \cdot (n-i+1) \vDash^S J\ ?\ \{\ \texttt{"\$ref"} : \texttt{"urn:start\#quant.x"} \cdot (n-i+2)\ \} \rightarrow T \\ \Leftrightarrow \quad Valid(Tail_{i-1}.\ \phi, A_C[x_{n-i+1} \leftarrow T])$$

and:

$$C \vDash^S J\ ?\ \{\ \texttt{"\$ref"} : \texttt{"urn:start\#quant.x"} \cdot (n-i+2)\ \} \rightarrow T \quad \Leftrightarrow \quad Valid(Tail_{i-1}.\ \phi, A_C[x_{n-i+1} \leftarrow F])$$

We can then conclude that

$$\texttt{"allOf"} : [\ \{\texttt{"\$ref"} : \texttt{"urn:setvar"} \cdot (n-i+1) \cdot \texttt{"\#afterq"} \cdot (n-i+1)\}\ ,\ \{\texttt{"\$ref"} : \texttt{"urn:start\#afterq"} \cdot (n-i+1)\}\ ]$$

evaluates to T if, and only if

$$Valid(Tail_{i-1}.\ \phi, A_C[x_{n-i+1} \leftarrow T]) \wedge Valid(Tail_{i-1}.\ , A_C[x_{n-i+1} \leftarrow F])$$

which concludes the proof. □

From Section 6:

Theorem 4. For any closed schema S and instance J whose total size is less than n, Algorithm 1 applied to J and S requires an amount of space that is polynomial in n.

Proof. We first prove that the call stack has a polynomial depth, thanks to the following properties, where we use n for the size of the input instance, and m for the size of the input schema.

(1) The depth of the call stack is always less than $|StopList| \times 3$.
(2) Each StopList generated during validation can be divided in M distinct sublists $sl_1, \ldots, sl_M$, such that all elements in the same sublist have the same context, and the context of the sublist sl_{i+1}, with $i \geq 1$, contains one more URI than the context of the sublist sl_i.
(3) The size of each sublist sl_i is at most $n \times m$.

Property (1) holds since *SchemaValidate(...,StopList)* invokes *KeywordValidate* with a stoplist with length $|StopList| + 1$, and *KeywordValidate* invokes a rule-specific function that may invoke *SchemaValidate* with the same stoplist, so that the StopList grows by 1 every time the call stack grows by 3, and vice versa.

Property (2) holds by induction: the function *SchemaValidate* passes to *KeywordValidate* the same context it receives, and the function *KeywordValidate* passes to *SchemaValidate* either the same context it receives or a context that contains one more URI, which happens when the keyword analyzed is either `"$ref"` or `"$dynamicRef"` and the target URI was not yet in the context.

Property (3) holds since no two elements of the stoplist can be equal, since a failure is raised when the input is already in the StopList parameter. All elements in a sublist have the same context, hence they must differ either in the instance or in the schema. The instance must be a sub-instance of the input instance, hence we have at most n choices. The schema must must be a sub-schema of the input schema, hence we have at most m choices. Hence, the size of each sublist sl_i is at most $n \times m$.

The combination of these three properties implies that the StopList parameter contains at most $(n \times m) \times (m + 1)$ elements, since it can be decomposed in at most $m + 1$ sublists, which implies that this parameter has a polynomial size, and that the call stack, by property (1), has a polynomial depth.

We now provide the rest of the proof, which is quite straightforward.

We observe that `SchemaValidate` *(Context,Instance,Schema,StopList)* scans all keywords inside *Schema* in sequence, and it only needs enough space to keep the pair *(Result,Eval)* between one call and the other one, and the size of this pair is in $O(n)$, since the list *Eval* of *evaluated* properties or items cannot be bigger than the instance. Since we reuse the same space for all keywords, we only need to prove that each single keyword can be recursively evaluated in polynomial space, including the space needed for the call stack and for the parameters.

Every keyword has its own specific algorithm, some of which are exemplified in Figure 1.

We first analyze `"patternProperties"`. All the four parameters can be stored in polynomial space; *Context* since it is a list of URIs that belong to the schema and because it contains no repetition, and the other parameters have already be discussed. Each matching pair can be analyzed in polynomial space, apart from the recursive call. For the recursive call, the call stack has a polynomial length, and every called function employs polynomial space.

We need to repeat the same analysis for all rules, but none of them is more complex than `"patternProperties"`. For example, the dependent keywords such as `"unevaluatedProperties"` have one extra parameter that contains the already *evaluated* properties and items, but it only takes polynomial space, as already discussed.

This completes the proof of the fact that the algorithm runs in polynomial space, hence the problem of validation for JSON Schema 2020-12 is PSPACE-complete. □

Theorem 5. Algorithm 2 applied to $(C, J, S, \emptyset, \emptyset)$ returns (r, κ, d), for some d, if, and only if, $C \vdash^{S} J \,?\, S \longrightarrow (r, \kappa)$.

Proof. In the following proof, we use the following metavariables conventions:

- *sl* (stoplist) denotes a list of triples (C, J, S)

- d (DFragSet) is a list of fragments f
- O (Output) is a triple (r, κ, d)
- us (UpdatableStore) is a list of quadruples (C, J, S, O)

This means that when we say "for any $sl\ldots$" we actually mean "for any list sl of tuples $(C, J, S)\ldots$"

We first define an enriched version of the typing rules, that defines a judgment $C \vdash^{\mathsf{Sd}} J\ ?\ S \to (r, \kappa, d)$ that returns in d the fragment names of the dynamic references that are resolved. The most important rule is ($\$\mathtt{dynamicRef}_d$), that adds the resolved f to the list d.

$$\frac{\begin{array}{c}\mathtt{dget}(\mathtt{load}(absURI), f) \neq \bot \qquad fURI = \mathtt{fstURI}(C{+}?absURI, f) \\ S' = \mathtt{dget}(\mathtt{load}(fURI), f) \qquad C{+}?fURI \ \vdash^{\mathsf{S}}\ J\ ?\ S' \to (r, \kappa, d)\end{array}}{C \vdash^{\mathsf{K}}\ J\ ?\ \mathtt{"\$dynamicRef"} : absURI \cdot \mathtt{"\#"} \cdot f \to (r, \kappa, d \cup \{\!|\, f \,|\!\})} \ (\$\mathsf{dynamicRef}_d)$$

All the applicators pass the evaluated dynamic references to their result, and a schema passes all the evaluated dynamic references even when it fails. We present here the failing schema rule and the rule for ($\mathsf{patternProperties}_d$), ($\mathsf{schema\text{-}false}_d$), and ($\mathsf{klist}\text{-}(n+1)_d$), the other rules are analogous.

$$\frac{\begin{array}{c}J = \{\, k'_1 : J_1, \ldots, k'_n : J_n \,\} \qquad \{\!|\, (i_1, j_1), \ldots, (i_l, j_l) \,|\!\} = \{\!|\, (i, j) \mid k'_i \in L(p_j) \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{\mathsf{S}}\ J_{i_q}\ ?\ S_{j_q} \to (r_q, \kappa_q, d_q) \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}})\end{array}}{C \vdash^{\mathsf{Kd}}\ J\ ?\ \mathtt{"patternProperties"} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\} \to (r, \{\!|\, k'_{i_1}, \ldots, k'_{i_l} \,|\!\}, \bigcup_{q \in \{\!|\, 1 \ldots l \,|\!\}} d_q)} \ (\mathsf{patternProperties}_d)$$

$$\frac{C \vdash^{\mathsf{Ld}}\ J\ ?\ [\![\, K_1, \ldots, K_n \,]\!] \to (F, \kappa, d)}{C \vdash^{\mathsf{Sd}}\ J\ ?\ \{\, K_1, \ldots, K_n \,\} \to (F, \emptyset, d)} \ (\mathsf{schema\text{-}false}_d)$$

$$\frac{k : J' \in \mathit{IndKeyOrT} \qquad C \vdash^{\mathsf{Ld}}\ J\ ?\ \vec{K} \to (r_l, \kappa_l, d_l) \qquad C \vdash^{\mathsf{Kd}}\ J\ ?\ K \to (r, \kappa, d)}{C \vdash^{\mathsf{Ld}}\ J\ ?\ (\vec{K} + K) \to (r_l \wedge r, \kappa_l \cup \kappa, d_l \cup d)} \ (\mathsf{klist}\text{-}(n+1)_d)$$

These new judgments just return some extra information with respect to the basic judgment, hence the following facts are immediate.

$$\begin{array}{rcl} C \vdash^{\mathsf{K}}\ J\ ?\ K \to (r, \kappa) & \Leftrightarrow & \exists d.\ C \vdash^{\mathsf{Kd}}\ J\ ?\ K \to (r, \kappa, d) \\ C \vdash^{\mathsf{S}}\ J\ ?\ S \to (r, \kappa) & \Leftrightarrow & \exists d.\ C \vdash^{\mathsf{Sd}}\ J\ ?\ S \to (r, \kappa, d) \\ C \vdash^{\mathsf{L}}\ J\ ?\ \vec{K} \to (r, \kappa) & \Leftrightarrow & \exists d.\ C \vdash^{\mathsf{Ld}}\ J\ ?\ \vec{K} \to (r, \kappa, d) \end{array}$$

We define an equivalence relation $C \sim_d C'$ defined as:

$$C \sim_d C' \ \Leftrightarrow\ \forall f \in d.\ \mathtt{fstURI}(C, f) = \mathtt{fstURI}(C', f)$$

We prove the following property:

$$\begin{array}{rcl} C \sim_d C' \ \wedge\ C \vdash^{\mathsf{Kd}}\ J\ ?\ K \to (r, \kappa, d') \wedge d' \subseteq d & \Rightarrow & C' \vdash^{\mathsf{Kd}}\ J\ ?\ K \to (r, \kappa, d') \\ C \sim_d C' \ \wedge\ C \vdash^{\mathsf{Ld}}\ J\ ?\ \vec{K} \to (r, \kappa, d') \wedge d' \subseteq d & \Rightarrow & C' \vdash^{\mathsf{Ld}}\ J\ ?\ \vec{K} \to (r, \kappa, d') \\ C \sim_d C' \ \wedge\ C \vdash^{\mathsf{Sd}}\ J\ ?\ S \to (r, \kappa, d') \wedge d' \subseteq d & \Rightarrow & C' \vdash^{\mathsf{Sd}}\ J\ ?\ S \to (r, \kappa, d') \end{array}$$

We prove it by induction on the rules. The proof is immediate for the terminal rules. For the applicators, consider rule ($\mathsf{patternProperties}_d$).

$$\frac{\begin{array}{c} J = \{\, k'_1 : J_1, \ldots, k'_n : J_n \,\} \qquad \{\!|\, (i_1, j_1), \ldots, (i_l, j_l) \,|\!\} = \{\!|\, (i,j) \mid k'_i \in L(p_j) \,|\!\} \\ \forall q \in \{\!|\, 1 \ldots l \,|\!\}.\ C \vdash^{S} J_{i_q}\ ?\ S_{j_q} \to (r_q, \kappa_q, d_q) \qquad r = \wedge(\{\!|\, r_q \,|\!\}^{q \in \{\!|\, 1 \ldots l \,|\!\}}) \end{array}}{C \vdash^{\mathsf{Kd}} J\ ?\ \texttt{"patternProperties"} : \{\, p_1 : S_1, \ldots, p_m : S_m \,\} \to (r, \{\!|\, k'_{i_1}, \ldots, k'_{i_l} \,|\!\}, \bigcup_{q \in \{\!|\, 1 \ldots l \,|\!\}} d_q)} \ (\mathsf{patternProperties}_d)$$

From the hypothesis that $\bigcup_{q \in \{\!|\, 1 \ldots l \,|\!\}} d_q \subseteq d$ we immediately deduce that for $q \in \{\!|\, 1 \ldots l \,|\!\}$ we have $d_q \subseteq d$, hence we can apply the inductive hypothesis to each $C \vdash^{S} J_{i_q}\ ?\ S_{j_q} \to (r_q, \kappa_q, d_q)$, and conclude. All the rules that do not modify the context are proved in the same way. We are left with the rules that modify the context, which are ($\mathsf{\$ref}_d$) and ($\mathsf{\$dynamicRef}_d$). This is rule ($\mathsf{\$ref}_d$).

$$\frac{S' = \mathtt{get}(\mathtt{load}(absURI), f) \qquad C{+}?absURI \vdash^{\mathsf{Sd}} J\ ?\ S' \to (r, \kappa, d)}{C \vdash^{\mathsf{Kd}} J\ ?\ \texttt{"\$ref"} : absURI \cdot \texttt{"\#"} \cdot f \to (r, \kappa, d)} \ (\mathsf{\$ref}_d)$$

We prove now that $C \sim_d C'$ implies $C{+}?absURI \sim_d C'{+}?absURI$. Consider any $f \in d$; if $\mathtt{fstURI}(C, f)$ are both different from $\bot$ we have:

$$\mathtt{fstURI}(C{+}?absURI, f) = \mathtt{fstURI}(C, f) = \mathtt{fstURI}(C', f) = \mathtt{fstURI}(C'{+}?absURI, f)$$

if $\mathtt{fstURI}(C, f)$ are both equal to $\bot$ we have:

$$\mathtt{fstURI}(C{+}?absURI, f) = absURI = \mathtt{fstURI}(C'{+}?absURI, f).$$

Hence $C{+}?absURI \sim_d C'{+}?absURI$, and we can apply induction to the ($\mathsf{\$ref}$) rule.

Consider finally rule ($\mathsf{\$dynamicRef}_d$), applied to C and to C'.

$$\frac{\begin{array}{c} \mathtt{dget}(\mathtt{load}(absURI), f) \neq \bot \qquad fURI = \mathtt{fstURI}(C{+}?absURI, f) \\ S_0 = \mathtt{dget}(\mathtt{load}(fURI), f) \qquad C{+}?fURI \vdash^{S} J\ ?\ S_0 \to (r, \kappa, d_0) \end{array}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"\$dynamicRef"} : absURI \cdot \texttt{"\#"} \cdot f \to (r, \kappa, d_0 \cup \{\!|\, f \,|\!\})} \ (\mathsf{\$dynamicRef}_d)$$

$$\frac{\begin{array}{c} \mathtt{dget}(\mathtt{load}(absURI), f) \neq \bot \qquad fURI' = \mathtt{fstURI}(C'{+}?absURI, f) \\ S'_0 = \mathtt{dget}(\mathtt{load}(fURI'), f) \qquad C'{+}?fURI' \vdash^{S} J\ ?\ S'_0 \to (r', \kappa', d'_0) \end{array}}{C' \vdash^{\mathsf{K}} J\ ?\ \texttt{"\$dynamicRef"} : absURI \cdot \texttt{"\#"} \cdot f \to (r', \kappa', d'_0 \cup \{\!|\, f \,|\!\})} \ (\mathsf{\$dynamicRef}_d)$$

By hypothesis, $(d_0 \cup \{\!|\, f \,|\!\}) \subseteq d$, hence $f \in d$. By reasoning as for rule ($\mathsf{\$ref}$), we deduce that $\mathtt{fstURI}(C{+}?absURI, f) = \mathtt{fstURI}(C'{+}?absURI, f)$, hence $fURI = fURI'$, hence $S'_0 = S_0$. By reasoning as for rule ($\mathsf{\$ref}$), $absURI = absURI'$ implies that $C{+}?fURI \sim_d C'{+}?fURI'$. Hence, since $d_0 \subseteq (d_0 \cup \{\!|\, f \,|\!\}) \subseteq d$, we can apply the induction hypothesis, and deduce that $(r', \kappa', d'_0) = (r, \kappa, d_0)$, from which we deduce that

$$(r', \kappa', d'_0 \cup \{\!|\, f \,|\!\}) = (r, \kappa, d_0 \cup \{\!|\, f \,|\!\})$$

Hence, we have proved that the following deduction rule is sound, that is, that its application does not allow any new judgment to be deduced:

$$\frac{C \vdash^{\mathsf{Sd}} J\ ?\ S \to (r, \kappa, d) \qquad C \sim_d C'}{C' \vdash^{\mathsf{Sd}} J\ ?\ S \to (r, \kappa, d)} \ (\mathsf{reuse}_d)$$

Our algorithm applies the rules of $C \vdash^{\mathsf{Sd}} J\ ?\ S$ plus (reuse_d) that is sound, hence it can only prove correct statements. On the other direction, completeness of the algorithm follows from the fact that it is correct and that it always terminates. □

We can now prove that Algorithm 2 runs in polynomial time when we have a constant bound on the number of dynamic references.

Theorem 6. Consider a family of closed schemas S and judgments J such that $(|S| + |J|) \leq n$, and let D be the set of different fragments f that appear in the argument of a `"$dynamicRef"` : $initURI \cdot$`"#"`$\cdot f$ in S. Then, Algorithm 2 runs on S and J in time $O(n^{k+|D|})$ for some constant k.

Proof. Consider S, J, n, and D, as in the statement of the Theorem. Consider the call tree of a run of Algorithm 2 applied to the tuple $(C, J, S, \emptyset, \emptyset)$, with S. The top node of this tree is a call to `SchemaValidateAndStore`, from now on abbreviates as `SVAS`, applied to $(C, J, S, \emptyset, \emptyset)$, which in turn invokes from 0 to n times `KeywordValidate`$(C, J, K_i, \ldots)$, each applying one specific rule, each rule invoking a certain number of times `SVAS`(C', J', S', sl', up').

We first observe that, for any invocation `SVAS`(C', J', S', sl', up') in the call tree rooted in `SVAS`$(C, J, S, \emptyset, \emptyset)$, with S closed, J' is a subtree of J, S' is a subschema of S, and C' is a list of URIs of resources in S with no repetitions. We also observe that the call tree is finite, since any infinite call tree would have an infinite branch, every infinite branch would contain two different invocations of `SVAS` with the same triple (C, J, S), and this possibility is prevented by the test executed on sl.

We observe that every two nodes labeled with `SVAS`(C', J', S', sl', up') and `SVAS`$(C'', J'', S'', sl'', up'')$ in that call tree either differ in J', or they differ in S', or they enjoy the property that $C \nsim_D C'$, since the check performed on us prevents a second call to `SVAS` with the same J and S when $C \sim_{d'} C'$ where d' is the *DFragSet* returned for (C, J, S), hence, for any two calls with the same (J, S), the corresponding C and C' enjoy $C \nsim_{d'} C'$ for some $d' \subseteq D$, hence they enjoy $C \nsim_D C'$. Each equivalence class of the relation $C \sim_D C'$ is characterized by a different function that maps each $f \in D$ to a URI, or to $\perp$; if we have n different URIs in S, the equivalence relation has at most $(n+1)^{|D|}$ distinct classes, since we have at most $n+1$ possible different choices for each fragment identifier. We have at most n subschemas of S and subterms of J, and $(n+1)^{|D|}$ different equivalence classes for C, hence the call tree for `SVAS`$(C, J, S, \emptyset, \emptyset)$ has at most $(n+1)^{|D|} \times n \times n$ nodes labeled with `SVAS`. The time needed to invoke the at most n instances of `SVAS` that are immediately invoked by a call to `SVAS` is polynomial — the most complicated case is (`patternProperties`) where each of the n label in J must be matched against each of the n regular expressions of the patterns of K, hence `SVAS`$(C, J, S, \emptyset, \emptyset)$ runs in time $O(n^{k+|D|})$ for some constant k. □

From Section 8:

Theorem 8 (Encoding correctness). Let S be a closed schema with base URI b. Then:

$$[\![\, b \,]\!] \vdash^{S} \; J \; ? \; \texttt{Static}(S) \longrightarrow (r, \kappa) \quad \Leftrightarrow \quad [\![\, b \,]\!] \vdash^{S} \; J \; ? \; S \longrightarrow (r, \kappa)$$

Proof. For any schema S_0 with base URI b, we want to prove that

$$[\![\, b \,]\!] \vdash^{S} \; J_0 \; ? \; \texttt{Static}(S_0) \longrightarrow (r_0, \kappa_0) \quad \Leftrightarrow \quad [\![\, b \,]\!] \vdash^{S} \; J_0 \; ? \; S_0 \longrightarrow (r_0, \kappa_0)$$

Our validation rules are deterministic, hence for each triple C, J, S or C, J, K we can define its "standard proof", that is just the only proof obtained by the application of the rules.

We then consider the standard proof for

$$[\![\, b \,]\!] \vdash^{S} \; J_0 \; ? \; S_0 \longrightarrow (r_0, \kappa_0)$$

and we prove that, for any judgment $C \vdash^{S} J \; ? \; S \longrightarrow (r, \kappa)$ in that proof, we have that:

$$C \vdash^{S} \; J \; ? \; S \longrightarrow (r, \kappa) \;\Leftrightarrow\; C \vdash^{S} \; J \; ? \; \texttt{CI}(C, S) \longrightarrow (r, \kappa)$$

and similarly that for any judgment $C \vdash^{K} J \; ? \; K \longrightarrow (r, \kappa)$ in that proof, we have that:

$$C \vdash^{K} \; J \; ? \; K \longrightarrow (r, \kappa) \;\Leftrightarrow\; C \vdash^{K} \; J \; ? \; \texttt{CI}(C, K) \longrightarrow (r, \kappa).$$

We prove that by mutual induction on the size of the proof. The base cases are trivial, since $\texttt{CI}(C, K)$ is equal to K when K is not a reference operator and does not contain any subschema. All the applicators different from references, such as "anyOf", are immediate by induction.

The only interesting cases are those for "$ref" and "$dynamicRef".

$$\begin{aligned}\texttt{CI}(C, \texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f) &= \texttt{"\$ref"} : \texttt{fstURI}(C{+?}\mathit{absURI}, f)\cdot\texttt{"\#"}\cdot\underline{C}\cdot f\\ \texttt{CI}(C, \texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f) &= \texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f\end{aligned}$$

Let us consider the ("$dynamicRef") case. Let this be the root of the proof of $C \vdash^{\mathsf{K}} J\ ?\ K \to (r, \kappa)$:

$$\frac{\begin{array}{cc}\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot & \mathit{fURI} = \texttt{fstURI}(C{+?}\mathit{absURI}, f)\\ S' = \texttt{dget}(\texttt{load}(\mathit{fURI}), f) & C{+?}\mathit{fURI} \vdash^{\mathsf{S}} J\ ?\ S' \to (r, \kappa)\end{array}}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f \to (r, \kappa)}\ (\$\text{dynamicRef})$$

By definition,

$$\texttt{CI}(C, \texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f) = \texttt{"\$ref"} : \texttt{fstURI}(C{+?}\mathit{absURI}, f)\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$$

That is

$$\texttt{CI}(C, \texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f) = \texttt{"\$ref"} : \mathit{fURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$$

Hence, the proof for $C \vdash^{\mathsf{K}} J\ ?\ \texttt{CI}(C, K) \to (r, \kappa)$ is the following one.

$$\frac{S'_1 = \texttt{get}(\texttt{load}(\mathit{fURI}), \underline{C}\cdot f) \qquad C{+?}\mathit{fURI} \vdash^{\mathsf{S}} J\ ?\ S'_1 \to (r_1, \kappa_1)}{C \vdash^{\mathsf{K}} J\ ?\ \texttt{"\$ref"} : \mathit{fURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f \to (r_1, \kappa_1)}\ (\$\text{ref})$$

By definition of $\texttt{Static}(S_0)$, if S' is the schema identified by $\mathit{fURI}\cdot\texttt{"\#"}\cdot f$, then $\mathit{fURI}\cdot\texttt{"\#"}\cdot\underline{C}\cdot f$ refers to $\texttt{CI}(C, S')$, hence $S'_1 = \texttt{CI}(C, S')$, and we conclude by induction.

The case for ($ref) is similar but simpler.

□

E ALGORITHM 2, PART 2

From Section 7.

Algorithm 3: Polynomial Time Validation - part 2

```
   /* Apply a keyword and return a new output based on PrevOutput                    */
 1 KeywordValidate(Context,Instance, Keyword, PrevOutput, StopList, Up)
 2     switch Keyword do
 3         case "anyOf": List do
 4             return (AnyOf (Context,Instance,List,PrevOutput,StopList,Up));
 5         case "dynamicRef": absURI "#" fragmentId do
 6             return (DynamicRef (Context,Instance,absURI,fragmentId,PrevOutput,StopList,Up));
 7         ...
 8
 9 AnyOf(Context, Instance, List, PrevOutput, StopList, Up)
10     (PrevRes, PrevEval, PrevDFragSet) := PrevOutput;
11     Result := True; Eval := EmptySet; DFragSet := EmptySet;
12     for Schema in List do
13         SchemaOutput := SchemaValidateAndStore (Context, Instance, Keyword, StopList, Up) ;
14         (SchemaRes,SchemaEval,SchemaDFragSet) := SchemaOutput ;
15         Result := Or (Result,SchemaRes);
16         Eval := Union (Eval,SchemaEval);
17         DFragSet := Union (DFragSet,SchemaDFragSet);
18     return (And (PrevResult,Result), Union (PrevEval,Eval), Union (PrevDFragSet,DFragSet));
19
20 DynamicRef(Context, Instance, AbsURI, fragmentId, PrevOutput, StopList, Up)
21     (PrevRes, PrevEval, PrevDFragSet) := PrevOutput;
22     if (dget(load(AbsURI),fragmentId) = bottom)) then StaticRef (...);
23     fstURI := FirstURI (Context+?AbsURI,fragmentId) ;
24     fstSchema := get(load(fstURI),fragmentId);
25     SchOutput := SchemaValidateAndStore(Saturate (Context,fstURI),Instance,fstSchema,StopList,Up);
26     (SchRes,SchEval,SchDT) := SchOutput;
27     return (And (PrevRes,SchRes), Union (PrevEval,SchEval), Union (SchDT, Singleton (fragmentId)));
28
29 FirstURI (context,fragmentId)
30     for URI in context do
31         if (dget(load(URI),fragmentId) != bottom) then { return (URI); }
32     return (Bottom);
```

F UNFOLDING OF THE EXAMPLE

We report here the complete unfolding of the running example, using the procedure defined in Section 8. It is the following document, where contexts are represented inside anchors using the abbreviations "urn:psi" ⟶ "p", "urn:start" ⟶ "s", "urn:setvar1" ⟶ "sv1", "urn:setvar2" ⟶ "sv2", and "urn:close" ⟶ "c". This is the first part.

```
{   "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "urn:psi",
    "$ref": "urn:start#p_s_forall.x1",
    "$defs": {
        "urn:start": {
             "$id": "urn:start",
             "$defs": {
                 "p_s_forall.x1" : {
                    "$anchor": "p_s_forall.x1",
                    "allOf": [ { "$ref": "urn:setvar1#p_s_afterq1" },
                               { "$ref": "urn:start#p_s_afterq1" }  ]  },
                 "p_s_exists.x2": {
                    "$anchor": "p_s_exists.x2",
                    "anyOf": [ { "$ref": "urn:setvar2#p_s_afterq2" },
                               { "$ref": "urn:start#p_s_afterq2" } ] },
                 "p_s_sv1_exists.x2": {
                    "$anchor": "p_s_sv1_exists.x2",
                    "anyOf": [ { "$ref": "urn:setvar2#p_s_sv1_afterq2" },
                               { "$ref": "urn:start#p_s_sv1_afterq2" }  ]
                 },
                 "p_s_afterq1": { "$anchor": "p_s_afterq1", "$ref": "urn:start#p_s_exists.x2" },
                 "p_s_afterq2": { "$anchor": "p_s_afterq2", "$ref": "urn:close#p_s_phi" },
                 "p_s_sv1_afterq2": { "$anchor": "p_s_sv1_afterq2", "$ref": "urn:close#
                      p_s_sv1_phi" }
             }
        },
        "urn:setvar1": {
             "$id": "urn:setvar1",
             "$defs": {
                 "p_s_afterq1": { "$anchor": "p_s_afterq1", "$ref": "urn:start#p_s_sv1_exists.x2"
                      },
                 "p_s_sv1_sv2_c_x1":     { "$anchor": "p_s_sv1_sv2_c_x1",     "anyOf": [true] },
                 "p_s_sv1_c_x1":         { "$anchor": "p_s_sv1_c_x1",         "anyOf": [true] },
                 "p_s_sv1_sv2_c_not.x1": { "$anchor": "p_s_sv1_sv2_c_not.x1", "anyOf": [false] },
                 "p_s_sv1_c_not.x1":     { "$anchor": "p_s_sv1_c_not.x1",     "anyOf": [false] }
             }
        },
        "urn:setvar2": {
             "$id": "urn:setvar2",
             "$defs": {
                 "p_s_sv1_afterq2": { "$anchor": "p_s_sv1_afterq2",
                                      "$ref": "urn:close#p_s_sv1_sv2_phi" },
                 "p_s_afterq2":     { "$anchor": "p_s_afterq2",
                                      "$ref": "urn:close#p_s_sv2_phi" },
                 "p_s_sv1_sv2_c_x2":     { "$anchor": "p_s_sv1_sv2_c_x2",     "anyOf": [true] },
                 "p_s_sv2_c_x2":         { "$anchor": "p_s_sv2_c_x2",         "anyOf": [true] },
                 "p_s_sv1_sv2_c_not.x2": { "$anchor": "p_s_sv1_sv2_c_not.x2", "anyOf": [false] },
                 "p_s_sv2_c_not.x2":     { "$anchor": "p_s_sv2_c_not.x2",     "anyOf": [false] }
             }
        },
        "urn:close": {
             "$id": "urn:close",
             "$defs": {
                 "p_s_sv2_c_x1": { "$anchor": "p_s_sv2_c_x1", "anyOf": [false] },
                 "p_s_c_x1":     { "$anchor": "p_s_c_x1",     "anyOf": [false] },
                 "p_s_sv2_c_not.x1": { "$anchor": "p_s_sv2_c_not.x1", "anyOf": [true] },
                 "p_s_c_not.x1": { "$anchor": "p_s_c_not.x1", "anyOf": [true] },
                 "p_s_sv1_c_x2": { "$anchor": "p_s_sv1_c_x2", "anyOf": [false] },
                 "p_s_c_x2":     { "$anchor": "p_s_c_x2",     "anyOf": [false] },
                 "p_s_sv1_c_not.x2": { "$anchor": "p_s_sv1_c_not.x2", "anyOf": [true] },
                 "p_s_c_not.x2": { "$anchor": "p_s_c_not.x2", "anyOf": [true] },
                 "p_s_sv1_sv2_phi": {...
```

Here is the last part; since "urn:close#phi" can be reached from four different contexts, we need four different definitions, as follows. In the four cases, the way the dynamic variables "x1" and "x2" are resolved varies depending on the context C, encoded in the anchor $\underline{C} \cdot f$.

```
"urn:close#urn:psi_urn:start_urn:setvar1_urn:setvar2_phi": {
   "$anchor": "urn:psi_urn:start_urn:setvar1_urn:setvar2_phi",
   "anyOf": [
      {"allOf": [ {"$ref": "urn:setvar1#..._urn:setvar1_urn:setvar2_urn:close_x1"},
                  {"$ref": "urn:setvar2#..._urn:setvar1_urn:setvar2_urn:close_not.x2"} ] },
      {"allOf": [ {"$ref": "urn:setvar1#..._urn:setvar1_urn:setvar2_urn:close_not.x1"},
                  {"$ref": "urn:setvar2#u..._urn:setvar1_urn:setvar2_urn:close_x2"} ] }
   ]
},
"urn:close#urn:psi_urn:start_urn:setvar1_phi": {
    "$anchor": "urn:psi_urn:start_urn:setvar1_phi",
    "anyOf": [
        {"allOf": [ {"$ref": "urn:setvar1#urn:psi_urn:start_urn:setvar1_urn:close_x1"},
                    {"$ref": "urn:close#urn:psi_urn:start_urn:setvar1_urn:close_not.x2"} ] },
        {"allOf": [ {"$ref": "urn:setvar1#urn:psi_urn:start_urn:setvar1_urn:close_not.x1"},
                    {"$ref": "urn:close#urn:psi_urn:start_urn:setvar1_urn:close_x2"} ] }
    ]
},
"urn:close#urn:psi_urn:start_urn:setvar2_phi": {
    "$anchor": "urn:psi_urn:start_urn:setvar1_urn:setvar2_phi",
    "anyOf": [
        {"allOf": [ {"$ref": "urn:close#urn:psi_urn:start_urn:setvar2_urn:close_x1"},
                    {"$ref": "urn:setvar2#urn:psi_urn:start_urn:setvar2_urn:close_not.x2"} ] },
        {"allOf": [ {"$ref": "urn:close#urn:psi_urn:start_urn:setvar2_urn:close_not.x1"},
                    {"$ref": "urn:setvar2#urn:psi_urn:start_urn:setvar2_urn:close_x2"} ] }
    ]
},
"urn:close#urn:psi_urn:start_phi": {
    "$anchor": "urn:psi_urn:start_urn:setvar1_phi",
    "anyOf": [
        {"allOf": [ {"$ref": "urn:close#urn:psi_urn:start_urn:close_x1"},
                    {"$ref": "urn:close#urn:psi_urn:start_urn:close_not.x2"} ] },
        {"allOf": [ {"$ref": "urn:close#urn:psi_urn:start_urn:close_not.x1"},
                    {"$ref": "urn:close#urn:psi_urn:start_urn:close_x2"} ] }
    ]
}
}}}}
```

G WHAT DOES *EVALUATED* MEAN?

The keyword "unevaluatedProperties" is applied to the instance properties that have not been *evaluated* by adjacent keywords, as discussed in Section 1.3, but, unfortunately, the definition in [30] of what counts as *evaluated* presents some ambiguities.

The *successful* application of a "properties" keyword *evaluates* all instance properties whose name appears in the value of the "properties" keyword; in Figure 1, these are the properties named "data" or "children". Moreover, a property is *evaluated* by the successful application of a keyword that invokes a schema that *evaluates* that property, as happens in Figure 1 with the keyword "$ref" : "https://example.com/simple-tree#tree".

If, however, the "properties" keyword fails, then the specifications of Draft 2020-12 ([30]) are ambiguous. They state without any ambiguity that validation will fail, but they give contradictory indications about which instance fields are *evaluated*, which is a problem, since it affects the error messages and the annotations returned by the validation tool. Consider the following example.

Example 1. The following schema expresses the fact that a property "a", if present, must have type "integer", and that no other property, neither equal to "b" nor different from "a" and "b", may appear.

```
{
 "$id": "https://frml.edu/onlya_no_b.json",
 "type": "object",
 "properties":  {
       "a": {"type": "integer"} ,
       "b": false
 },
 "unevaluatedProperties" : false
}
```

When applied to the instance { "a" : 0, "b" : 0, "c" : 0 }, this schema fails, because of properties "b" and "c". While this is clear, it is not clear which properties count as *evaluated* by the "properties" keyword, hence, which error messages should be produced by "unevaluatedProperties". This is testified by the discussion in [24], by that in [20], and by several more that are cited therein.

We tested the above schemas with the json-everything validator [2], Jim Blackler's JSON tools [14], the jschon.dev validator [1], and Hyperjump [4], in the online-versions on 10/30/2022. These are established well-known validators that support Draft 2019-09 and Draft 2020-12. By analyzing their output, reported respectively in Figures 10, 11, 12, and 13, (Appendix I), we observe that the first two validators apply "unevaluatedItems" to "c" only, hence they regard every matching field as *evaluated* even when "properties" fails; we call it the failure-tolerating interpretation: every property that matches is *evaluated*, even if the keyword fails, and even if the subschema applied to that property fails, as happens here to "b". However, jschon.dev applies "unevaluatedProperties" to "a", "b", and "c", hence it considers no property as *evaluated*; we call it the success-only interpretation: when the keyword fails no property is regarded as *evaluated*, neither "a", whose value satisfies the "type" : "integer" assertion, neither "b". Finally, Hyperjump does not apply "properties" to any object property, because it adopts an interpretation where the failure of "properties" produces a special *null* annotation that completely prevents the execution of "unevaluatedProperties"; we call it the null-annotation interpretation.

In this paper we formalize the failure-tolerating interpretation, since it seems the one that is currently more accepted by the community. Moreover, we provide a formal framework where these different interpretations can be formalized and discussed.

It is important to clarify that these ambiguities do not affect validation, since they only arise with failing schemas, but they are still very relevant, because they affect the error messages and, more importantly, the mental model of the specification reader. Whoever reads the specification

document builds a mental model about the internal logics of JSON Schema, and the way they define a schema, or the way they implement a validator, are guided by this mental model. If the mental model is unclear, or ambiguous, their work becomes more complicated then it should be.

H LIST OF DOWNLOADED VALIDATORS

List of the validators from Section 9.

Table 2. Validators used in our experiments, first listing active open-source projects, then two academic implementations. Stating programming language and supported drafts. (×) means that draft is officially supported, but logical or runtime errors occurred in our experiments.

Validator	Language	Draft4	Draft 2020	Origin/Version
clojure-json-schema	clojure	×		Bowtie, release tag b07e5
cpp-valijson	C++	×		Bowtie, release tag b07e5
go-jsonschema	go	×	×	Bowtie, release tag b07e5
js-ajv	javascript	×	(×)	Bowtie, release tag b07e5
js-hyperjump	javascript	(×)	×	Bowtie, release tag b07e5
lua-jsonschema	lua	×		Bowtie, release tag b07e5
ruby-json_schemer	ruby	×		Bowtie, release tag b07e5
rust-boon	python	×	(×)	Bowtie, release tag 992c2
rust-jsonschema	rust	×	×	Bowtie, release tag b07e5
python-fastjsonschema	python	×		Bowtie, release tag b07e5
python-jschon	python		×	Bowtie, release tag b07e5
python-jsonschema	python	×	×	Bowtie, release tag b07e5
ts-vscode-json-languageservice	typescript	×	(×)	Bowtie, release tag b07e5
dotnet-jsonschema-net	dotnet		×	Bowtie, release tag b07e5
foundations-jsonsonschema	python	×		Code from [21], integrated into Bowtie
modern-jsonschema	Scala		×	Algorithm 2, integrated into Bowtie

I OUTPUT EXAMPLES

Instance:

```
{ "a": 0, "b": 0, "c": 0 }
```

Schema:

```
{
  "$id": "https://formalize.edu/onlya_no_b.json",
  "type": "object",
  "properties":  {
        "a": {"type": "integer"} ,
        "b": false
  },
  "unevaluatedProperties" : false
}
```

Output:

```
{
  "valid": false,
  "keywordLocation": "",
  "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#",
  "instanceLocation": "",
  "errors": [
    {
      "valid": true,
      "keywordLocation": "/properties/a",
      "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#/properties/a",
      "instanceLocation": "/a"
    },
    {
      "valid": false,
      "keywordLocation": "/properties/b",
      "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#/properties/b",
      "instanceLocation": "/b",
      "error": "All values fail against the false schema"
    },
    {
      "valid": false,
      "keywordLocation": "/unevaluatedProperties",
      "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#/unevaluatedProperties",
      "instanceLocation": "/c",
      "error": "All values fail against the false schema"
    }
  ]
}
```

Fig. 10. Annotations returned by json-everything, example 1.

Instance:

```
{ "a": 0, "b": 0, "c": 0 }
```

Schema:

```
{
  "$id": "https://formalize.edu/onlya_no_b.json",
  "type": "object",
  "properties":  {
        "a": {"type": "integer"} ,
        "b": false
  },
  "unevaluatedProperties" : false
}
```

Output:

```
{
    "valid": false,
    "keywordLocation": "https://formalize.edu/onlya_no_b.json",
    "absoluteKeywordLocation": "",
    "instanceLocation": "",
    "errors": [
        {
            "valid": false,
            "error": "False",
            "keywordLocation": "https://formalize.edu/onlya_no_b.json#/properties/b",
            "absoluteKeywordLocation": "#/properties/b",
            "instanceLocation": "#/b"
        },
        {
            "valid": false,
            "error": "False",
            "keywordLocation": "https://formalize.edu/onlya_no_b.json#/unevaluatedProperties",
            "absoluteKeywordLocation": "#/unevaluatedProperties",
            "instanceLocation": "#/c"
        }
    ]
}
```

Fig. 11. Annotations returned by Jim Blackler's JSON tools, example 1.

Instance:

```
{ "a": 0, "b": 0, "c": 0 }
```

Schema:

```
{
  "$id": "https://formalize.edu/onlya_no_b.json",
  "type": "object",
  "properties":  {
        "a": {"type": "integer"} ,
        "b": false
  },
  "unevaluatedProperties" : false
}
```

Output:

```
{
    "valid": false,
    "instanceLocation": "",
    "keywordLocation": "",
    "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#",
    "errors": [
        {
            "instanceLocation": "/b",
            "keywordLocation": "/properties/b",
            "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#/properties/b",
            "error": "The instance is disallowed by a boolean false schema"
        },
        {
            "instanceLocation": "",
            "keywordLocation": "/unevaluatedProperties",
            "absoluteKeywordLocation": "https://formalize.edu/onlya_no_b.json#/
                 unevaluatedProperties",
            "error": [
                "a",
                "b",
                "c"
            ]
        }
    ]
}
```

Fig. 12. Annotations returned by jschon.dev, example 1.

Instance:

```
{ "a": 0, "b": 0, "c": 0 }
```

Schema:

```
{
  "$id": "https://formalize.edu/onlya_no_b.json",
  "type": "object",
  "properties":  {
        "a": {"type": "integer"} ,
        "b": false
  },
  "unevaluatedProperties" : false
}
```

Output:

```
# fails schema constraint https://formalize.edu/onlya_no_b.json#/properties
#/b fails schema constraint https://formalize.edu/onlya_no_b.json#/properties/b
```

Fig. 13. Annotations returned by Hyperjump, example 1.