Validation of Modern JSON Schema: Formalization and Complexity

LYES ATTOUCHE, Université Paris-Dauphine - PSL, France
MOHAMED-AMINE BAAZIZI, Sorbonne University, France
DARIO COLAZZO, Université Paris-Dauphine - PSL, France
GIORGIO GHELLI, University of Pisa, Italy
CARLO SARTIANI, University of Basilicata, Italy
STEFANIE SCHERZINGER, University of Passau, Germany

JSON Schema is the de-facto standard schema language for JSON data. The language went through many minor revisions, but the most recent versions of the language, starting from Draft 2019-09, added two novel features, *dynamic references* and *annotation-dependent validation*, that change the evaluation model. *Modern JSON Schema* is the name used to indicate all versions from Draft 2019-09, which are characterized by these new features, while *Classical JSON Schema* is used to indicate the previous versions.
These new "modern" features make the schema language quite difficult to understand and have generated many discussions about the correct interpretation of their official specifications; for this reason, we undertook the task of their formalization. During this process, we also analyzed the complexity of data validation in Modern JSON Schema, with the idea of confirming the polynomial complexity of Classical JSON Schema validation, and we were surprised to discover a completely different truth: data validation, which is expected to be an extremely efficient process, acquires, with Modern JSON Schema features, a PSPACE complexity.

In this paper, we give the first formal description of Modern JSON Schema, which we have discussed with the community of JSON Schema tool developers, and which we consider a central contribution of this work. We then prove that its data validation problem is PSPACE-complete. We prove that the origin of the problem lies in the Draft 2020-12 version of dynamic references, and not in annotation-dependent validation. We study the schema and data complexities, showing that the problem is PSPACE-complete with respect to the schema size even with a fixed instance but is in P when the schema is fixed and only the instance size is allowed to vary. Finally, we run experiments that show that there are families of schemas where the difference in asymptotic complexity between dynamic and static references is extremely visible, even with small schemas.

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

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

ACM Reference Format:
Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of Modern JSON Schema: Formalization and Complexity. *Proc. ACM Program. Lang.* 8, POPL, Article 49 (January 2024), 31 pages. https://doi.org/10.1145/3632891

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

ACM 2475-1421/2024/1-ART49
https://doi.org/10.1145/3632891

1 INTRODUCTION

JSON Schema [Org 2022] is the de-facto standard schema language for JSON data. It is based on the combination of structural operators, which describe 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 [Galiegue and Zyp 2013], Draft-06 [Wright et al. 2017], which introduced extensions without changing the validation model, Draft 2019-09 [Wright et al. 2019], and Draft 2020-12 [Wright et al. 2022].

The evaluation model of Draft-04 and Draft-06 is quite easy to understand and to formalize, and it has been studied in [Pezoa et al. 2016], [Bourhis et al. 2017, 2020], [Attouche et al. 2022], yielding many interesting complexity results. However, Draft 2019-09 introduces two important novelties to the evaluation model: annotation-dependent validation, and dynamic recursive references, which have been generalized as general-purpose dynamic references in Draft 2020-12. According to the terminology introduced by Henry Andrews in [Andrews 2023], 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 need formal clarification and specification, as documented by many online discussions, such as [Neal 2022] and [Jacobson 2021]. These discussions involve main actors behind the JSON Schema design and show that the informal JSON Schema specification leads to several, different, yet reasonable interpretations of the semantics of the new operators in Modern JSON Schema.

Our Contribution. Draft 2020-12 [Wright et al. 2022] is the version of Modern JSON Schema that we study in this paper. We provide the following contributions.[1]

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 3 and 4); to the best of our knowledge this is the first formal specification and study of Modern JSON Schema. In addition, we implemented, in Scala, a Modern JSON Schema data validator by a direct translation of our formal system and used it to verify the correctness of our formal system with respect to the JSON Schema standard test suite (Section 10).

ii) We analyze the complexity of validating a JSON instance against a schema and show that, surprisingly enough, when the general-purpose dynamic references of Draft 2020-12 come into play, the validation problem becomes PSPACE-hard (Section 5); validation was known to be P-complete for Classical JSON Schema [Bourhis et al. 2017; Pezoa et al. 2016]. We also prove that the bound is strict and, hence, the problem is PSPACE-complete (Section 6).

iii) We show that annotation-dependent validation alone, on the contrary, does not change the P complexity of validation, by providing an explicit algorithm for Modern JSON Schema that runs in polynomial time on any family of schemas where the number of dynamic references is bounded by a constant (Section 7). We also show that the validation for Draft 2019-09 was still in P, because of the restrictions that it imposed on dynamic references. 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.

iv) We provide a technique for substituting dynamic references with static references, at the price of an exponential increase of the schema size (Section 8).

v) We run experiments on a rich set of validators that show that there are families of schemas where the distinction between dynamic and static references is clearly visible in the experimental

[1] Unless otherwise noticed, all proofs are in the full version [Attouche et al. 2023c].

results; the experiments also show that many established validators exhibit an exponential behavior also on the P fragment of JSON Schema (Section 10).

This paper is about *Formalization* and *Complexity*. We believe that the *formalization* that we provide answers a need that has been clearly expressed in many public discussions, and we will use our connections with the JSON Schema community in order to disseminate our results.

Our *complexity* results, on the other hand, belong, for the moment, to the class of "foundational" results, that is, theoretical results that shed light on a phenomenon whose practical relevance *may* manifest in the future. Concretely, this is the first time that we encounter a construct of a data-validation language whose complexity is not in P (unless P=PSPACE). As a consequence, from now on, we know that the addition of a dynamic re-binding mechanism to a data validation language may have an unexpected impact on its computational complexity.

2 MODERN JSON SCHEMA THROUGH EXAMPLES

2.1 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*. Two keywords with the same parent object, such as `"$id" : "https:.../schema"` and `"type" : "object"` in Figure 1, are said to be *adjacent*, so that keywords are *adjacent* when they are siblings in the parse tree of the schema.

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

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 through a URI with structure $baseURI \cdot$ `"#"` $\cdot$ $fragmentId$ (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 assigns a fragment identifier to its parent schema, so that, in this case, the fragment named `"tree"` by `"$anchor"` is the entire schema. Hence, this schema can be referred to as either `"http://mjs.ex/st"` or as `"http://mjs.ex/st#tree"`.

The `"type"` keyword specifies that this schema only validates objects, while it fails on arrays and base values. The `"properties"` keyword specifies that, if the instance under validation contains a `"data"` property, then its value has no constraint (`"data" : true`), and, if it contains a `"children"` property, then its value must satisfy the nested subschema of line 7: it must be an array whose elements (if any) must all satisfy `"$ref" : "http://mjs.ex/st#tree"`. `"$ref"` is a reference operator that invokes a local or remote subschema, which, in this case, is the entire current schema.[2]

[2]By the standard rules of URI reference resolution [Berners-Lee et al. 2005], the URI `"http://mjs.ex/st#tree"` could be substituted by the local reference `"$ref" : "#tree"`, since `"http://mjs.ex/st"` is the base URI of this schema.

This schema is satisfied by all the instances that are listed in the "examples" array.[3] 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. 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"}.

Introducing dynamic references. Dynamic references have been added in Modern JSON Schema 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 7. This combination indicates that "$dynamicRef" : "http://mjs.ex/simple-tree#tree" is *dynamic*, which means that, when this schema is accessed through a different context, as exemplified later, this dynamic reference may refer to a fragment that is still named "tree", but which is defined in a schema that is not "http://mjs.ex/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": "http://mjs.ex/simple-tree",
3   "$dynamicAnchor": "tree",
4   "type": "object",
5   "properties": {
6       "data": true,
7       "children": {  "type": "array", "items": { "$dynamicRef": "http://mjs.ex/simple-tree#tree" }}}
8 }
```

Fig. 2. A schema representing extensible trees.

The contextual redefinition mechanism is illustrated in the schema shown in Figure 3. The schema "http://mjs.ex/strict-tree" redefines the dynamic anchor "tree" (line 3), so that it now indicates a conjunction between "$ref" : "http://mjs.ex/simple-tree#tree" (line 4) and the keyword "unevaluatedProperties" : false (line 5), which forbids the presence of any property whose name does not match those listed in "http://mjs.ex/simple-tree#tree". If one applies this schema, it will invoke "$ref" : "http://mjs.ex/simple-tree#tree" (Figure 3 - line 4), which will execute the schema of Figure 2 in a "dynamic scope" where "http://mjs.ex/strict-tree" has redefined the meaning of "$dynamicRef" : "http://mjs.ex/simple-tree#tree". In detail, in Draft 2020-12, the "outermost" (or "first") schema that contains "$dynamicAnchor" : *fragmentName* is the one that fixes the meaning of that anchor for any other schema S' that will invoke "$dynamicRef" : *absURI*·"#"·*fragmentName* later, independently from the absolute URI *absURI* used by S', hence, in this case, the meaning of "http://mjs.ex/simple-tree#tree" in Figure 2 is fixed to be "http://mjs.ex/strict-tree#tree" because the schema of Figure 3 is invoked before the one in Figure 2. Observe that the meaning of the static reference "$ref" : "http://mjs.ex/simple-tree#tree" to the dynamic anchor "#tree" (Figure 3 line 4) is fixed and does not depend on the dynamic context.

As another example, in Figure 4, we have a simplified version of the metaschema of JSON Schema.

The metaschema "https://json-schema.org/draft/2020-12/meta/applicator" has a dynamic anchor (i.e., name) "meta", and specifies that the value of a "properties" keyword is an object J_p, where every field of J_p has an arbitrary name (like "data" or "children" in Figure 2 line 6), and the value of every field of J_p (such as true and { "type" : "array", ... } in Figure 2 line 7) is itself a schema; the

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

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

Fig. 3. A schema that refines trees.

```
{  "$id": "https://json-schema.org/draft/2020-12/meta/applicator",
   "$dynamicAnchor": "meta",
   "properties": {
        "properties": {
            "type": "object",
            "patternProperties": {
                  ".*": { "$dynamicRef": "https://json-schema.org/draft/2020-12/meta/applicator#meta" }
            }
        }
    }
}

{   "$id": "https://json-schema.org/draft/2020-12/meta/uneval",
    "$dynamicAnchor": "meta",
    "properties": {
        "unevaluatedProperties": { "$dynamicRef": "https://json-schema.org/draft/2020-12/meta/uneval#meta" }
    }
}

{   "$id": "https://json-schema.org/draft/2020-12/meta/core",
    "$dynamicAnchor": "meta",
    "type" : [ "boolean", "object" ],
    "properties": {
        "$id": { "type": "string"  },
        "$dynamicAnchor": { "type": "string"  },
        "$title": { "type": "string"  }
    }
}

{   "$id": "https://json-schema.org/draft/2020-12/schema",
    "$dynamicAnchor": "meta",
    "allOf": [ {"$ref": "https://json-schema.org/draft/2020-12/meta/applicator"},
               {"$ref": "https://json-schema.org/draft/2020-12/meta/uneval"},
               {"$ref": "https://json-schema.org/draft/2020-12/meta/core"} ]
}

{   "$id": "https://json-schema.org/draft/2020-12/mini-schema",
    "$dynamicAnchor": "meta",
    "allOf": [ {"$ref": "https://json-schema.org/draft/2020-12/meta/applicator"},
               {"$ref": "https://json-schema.org/draft/2020-12/meta/core"} ]
}
```

Fig. 4. JSON Schema metaschema

use of "$dynamicRef" : "https://json-schema.org/draft/2020-12/meta/applicator#meta" allows the programmer to later redefine what constitutes a valid schema.

In the same way, the metaschema "https://json-schema.org/draft/2020-12/meta/uneval" has the same dynamic anchor (name) "meta", and specifies that the value of a "unevaluatedProperties" keyword is itself a schema (we will describe "unevaluatedProperties" later on).

The metaschema "https://json-schema.org/draft/2020-12/meta/core" just specifies that every schema is either a boolean or an object, and that the value of keywords "$id", "$dynamicAnchor", and "title", is a string.

The metaschema "https://json-schema.org/draft/2020-12/schema" combines the three fragments above, and redefines the dynamic anchor "meta", so that, when an instance is validated using "https://json-schema.org/draft/2020-12/schema", any dynamic reference "...#meta" found inside any of the three fragments actually refers to "https://json-schema.org/draft/2020-12/schema", that is, to their conjunction.

Some validators do not support the "unevaluatedProperties" keyword; these validators would then use the "https://json-schema.org/draft/2020-12/mini-schema" metaschema. It combines only two of the fragments above, and redefines the dynamic anchor "meta", so that, when an instance is validated using ".../mini-schema", any"$dynamicRef": "...#meta" found inside any of the two fragments refers to ".../mini-schema".

To sum up, the dynamic nature of "...#meta" allows one to define a "customized" metaschema by choosing the specific fragments to combine. This example is very important, since it provided the initial motivation for the introduction of dynamic references: allowing the definition of different versions of the metaschema by choosing which fragments to combine.

The semantics of dynamic references is not easy to understand, and we believe that it needs a formal definition. We are going to provide that definition.

2.2 Annotation Dependency

In Modern JSON Schema, the application of a keyword to an instance produces *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*.

For example, the assertion "unevaluatedProperties" : false in the schema of Figure 3 depends on the annotations returned by the adjacent keyword "$ref" : "http://mjs.ex/simple-tree#tree" (which refers to the schema in Figure 2). In this case, "$ref" *evaluates* all and only the fields whose name is either "data" or "children". Hence, "unevaluatedProperties" : false is applied to any other field, and 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 was always evaluated last among the keywords inside its schema (i.e., its *adjacent* keywords).

The definition of *evaluated* in the specifications of Draft 2020-12 ([Wright et al. 2022]) presents many ambiguities, as testified by online discussions such as [Neal 2022] and [Jacobson 2021]. These ambiguities do not affect the final result of the evaluation, but only the error messages that are generated; these aspects are further discussed in the full version, and we will formalize here the interpretation that is more widely accepted. We believe that an important contribution of this work is that it provides a precise and succinct language in which these ambiguities can be discussed and settled.

3 FORMALIZING JSON SCHEMA SYNTAX

A key contribution of this work is a formalization of the entire Modern JSON Schema language, but, for reasons of space, we only report here a crucial subset that illustrates the approach and is sufficient to carry out the complexity analysis; the remaining part is in the full version.

In this section, we formalize the syntax, which does not present any technical difficulty; the validation behavior is defined in the next section.

Schemas are structured as *resources*, which are collected into *documents* and refer to each other using URIs. We deal with these aspects through a "normalization process" (Section 3.1), where we eliminate the issues related to URI resolution and to the collection of many resources in just one document; we then formalize the syntax of normalized JSON Schema in Section 3.2.

3.1 URI Resolution, Resource Flattening, Schema Closing

The keywords "$id", "$ref", and "$dynamicRef" accept any URI reference as value, they apply the *resolution* process defined in [Berners-Lee et al. 2005], and then interpret the resulting resolved URI according to the JSON Schema rules. Since resolution is already specified in [Berners-Lee et al. 2005], we will not formalize it here, and we will assume that, in every schema that is interpreted through our rules, the values of these three keywords have already been resolved, and that the result of that resolution has the shape *absURI*·"#"·*fragmentId* for "$ref", and "$dynamicRef", where *fragmentId* may be empty and has the shape *absURI*, with no fragment, for "$id".

A "$id" : *absURI* keyword at the top-level of a schema object S_{id} that is nested inside a JSON Schema document S indicates that the schema object S_{id} is a separate resource, identified by *absURI*, that is embedded inside the document S but is otherwise independent. Embedded resources are an important feature, since they allow the distribution of different resources with just one file, but present some problems when they are nested inside arbitrary keywords, and when a reference crosses the boundaries between resources, as does "$ref" : "http://mjs.ex/top#/properties/foo/items" in Figure 5, line 3 (see also Section 9.2.1 of the specifications [Wright et al. 2022]) (a fragment identifier following "#" may be either an anchor, as in the previous examples, or a JSON path, as in this case).

```
1 { "$schema": "https://json-schema.org/draft/2020-12/schema",
2   "$id": "http://mjs.ex/top",
3   "$ref": "http://mjs.ex/top#/properties/foo/items",
4   "properties": {
5       "foo": { "$schema": "https://json-schema.org/draft/2019-09/schema",
6                "$id": "http://mjs.ex/nestedFoo",
7                "items": { "type": "object"}}},
8   "unevaluatedProperties": { "$id": "http://mjs.ex/nestedUnevaluatedProps", "type": "string" }}
```

```
1 { "$schema": "https://json-schema.org/draft/2020-12/schema",
2   "$id": "http://mjs.ex/top",
3   "$ref": "http://mjs.ex/nestedFoo#/items",
4   "properties": { "foo": { "$ref": "http://mjs.ex/nestedFoo#" }},
5   "unevaluatedProperties": { "$ref": "http://mjs.ex/nestedUnevaluatedProps#" },
6   "$defs": { "http://mjs.ex/nestedFoo" :  {
7                  "$schema": "https://json-schema.org/draft/2019-09/schema",
8                  "$id": "http://mjs.ex/nestedFoo",
9                  "items": { "type": "object"}}},
10            "http://mjs.ex/nestedUnevaluatedProps" : {
11                "$id": "http://mjs.ex/nestedUnevaluatedProps",
12                "type": "string"}}
```

```
1 Valid instances:          { "foo": [ { "aa" : 3 }, {} ] }
2                           { "bar": "any string" }
3                           { "foo": [ { "bb" : "a" } ], "other" : "z" }
4 Non valid instances:      { "foo": [ 3 ] }
5                           { "any": 3 }
```

Fig. 5. A schema with embedded resources, its flattened version, and some examples.

To avoid this kind of problem, we assume that every JSON Schema document is *resource-flattened* before validation. Resource flattening consists in moving every embedded resource identified by

"$id" : *absURI* into the value of a field named *absURI* of a "$defs" keyword at the top level of the document, and replacing the moved resource with an equivalent schema {"$ref" : *absURI*·"#"} that invokes that resource;[4] "$defs" is a placeholder keyword that is not evaluated but is a place to collect schemas that can be invoked using "$ref" or "$dynamicRef". [5] During this phase, we also replace any reference that crosses resource boundaries with an equivalent reference whose URI is the base URI of the target, as suggested in Section 8.2 of the specifications [Wright et al. 2022] (e.g., line 3 in the first schema of Figure 5, which crosses the boundary of the internal resource "http://mjs.ex/nestedFoo", is substituted with the equivalent reference of line 3 in the second schema); the two steps are exemplified in Figure 5.

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. 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.

3.2 JSON Schema Normalized Grammar

JSON Schema syntax is a subset of JSON syntax. We present in Figure 6 the grammar for a subset of the keywords, which is rich enough to present our results. 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.

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 behavior by assuming that, before validation, each schema is reordered to respect the grammar in Figure 6. The grammar specifies that a schema S is either a boolean schema, that matches any value (true) or no value at all (false), or it begins with a possibly empty sequence of Independent Keywords or triples (IK), followed by a possibly empty sequence of First-Level Dependent keywords (FLD), followed by a possibly empty sequence of Second-Level Dependent keywords (SLD). Specifically, the two keywords in *FLD*, "additionalProperties" and "items", depend on some keywords in *IK* (such as "properties" and "patternProperties"), and the two keywords in *SLD* depend on the keywords in *FLD*, and on many keywords in *IK*, such as "properties", "patternProperties","anyOf", "allOf", "$ref", and others.

This grammar specifies the predefined keywords, the type of the associated value (here $\mathcal{J}Val$ is the set of all JSON values, and *plain-name* denotes any alphanumeric string starting with a letter), 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.

4 JSON SCHEMA VALIDATION

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

$$s \in \mathsf{Str}, d \in \mathsf{Num}, n \in \mathsf{Int}, n \geq 0, l_i \in \mathsf{Str}$$
$$J ::= \quad \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid d \mid s \mid [\, J_1, \ldots, J_n \,] \mid \{\, l_1 : J_1, \ldots, l_n : J_n \,\} \quad i \neq j \Rightarrow l_i \neq l_j$$

[4] The empty fragment identifier after *absURI*·"#" refers to the root of the resource *absURI*.
[5] The "$defs" keyword is the "Modern" version of the "definitions" keyword of Classical JSON Schema, and the act of collecting all embedded resources in the "$defs" section is described as "bundling" in Draft 2020-12, Section 9.3.

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

$$
\begin{array}{lcl}
\mathit{Tp} & ::= & \texttt{"object"} \mid \texttt{"number"} \mid \texttt{"integer"} \mid \texttt{"string"} \mid \texttt{"array"} \mid \texttt{"boolean"} \mid \texttt{"null"} \\
S & ::= & \texttt{true} \mid \texttt{false} \mid \{\ \mathit{IK}\ (,\mathit{IK})^*\ (,\mathit{FLD})^*\ (,\mathit{SLD})^*\ \} \\
 & & \mid \{\ \mathit{FLD}\ (,\mathit{FLD})^*\ (,\mathit{SLD})^*\ \} \mid \{\ \mathit{SLD}\ (,\mathit{SLD})^*\ \} \mid \{\ \} \\
\mathit{IK} & ::= & \texttt{"minimum"} : q \mid \texttt{"maximum"} : q \mid \texttt{"pattern"} : p \mid \texttt{"required"} : [\ k_1, \ldots, k_n\ ] \\
 & & \mid \texttt{"type"} : \mathit{Tp} \mid \texttt{"type"} : [\mathit{Tp}_1, \ldots, \mathit{Tp}_n] \mid \texttt{"\$defs"} : \{\ k_1 : S_1, \ldots, k_n : S_n\ \} \\
 & & \mid \texttt{"\$id"} : \mathit{absURI} \mid \texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f \mid \texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f \\
 & & \mid \texttt{"\$anchor"} : \mathit{plain\text{-}name} \mid \texttt{"\$dynamicAnchor"} : \mathit{plain\text{-}name} \\
 & & \mid \texttt{"anyOf"} : [\ S_1, \ldots, S_n\ ] \mid \texttt{"allOf"} : [\ S_1, \ldots, S_n\ ] \mid \texttt{"not"} : S \\
 & & \mid \texttt{"patternProperties"} : \{\ p_1 : S_1, \ldots, p_m : S_m\ \} \\
 & & \mid \texttt{"properties"} : \{\ k_1 : S_1, \ldots, k_m : S_m\ \} \\
 & & \mid k : J \ \text{(with } k \text{ not previously cited)} \\
\mathit{FLD} & ::= & \texttt{"additionalProperties"} : S \mid \texttt{"items"} : S \\
\mathit{SLD} & ::= & \texttt{"unevaluatedProperties"} : S \mid \texttt{"unevaluatedItems"} : S
\end{array}
$$

Fig. 6. Minimal grammar of normalized JSON Schema Draft 2020-12.

In this paper, we reserve the notation $\{\ldots\}$ to JSON objects, hence we use $\{\!|\, a_1, \ldots, a_n \,|\!\}$ and $\{\!|\, a_i \,|\!\}^{i \in I}$ 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.

Given a pattern p, we will use $L(p)$ to denote the language generated by p, i.e., the set of all strings that match that pattern.

4.1 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 judgment $C \vdash^{\mathsf{K}} J : K \to (r, \kappa)$ 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 [Wright et al. 2022], 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:

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

Hence, the set of annotations κ can contain member names (strings) or array positions (integers).

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 $\texttt{"\$id"} : \mathit{absURI}$ is not an assertion, while $\texttt{"type"} : \mathit{Tp}$ is. Assertions are called *applicators* when they have schema parameters, such as S_1 and S_2 in $\texttt{"anyOf"} : [S_1, S_2]$, 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, 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, for static and dynamic references. Rules are shown in Figure 7.

4.2 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 keywords are defined by a couple of rules, as exemplified here for the keyword "minimum" : q. Rule (minimumTriv) (Figure 7) always returns $\mathcal{T}$ (true) when J is not a number, while rule (minimum), applied to numbers, returns the same boolean $r \in \{\!|\, \mathcal{T}, \mathcal{F} \,|\!\}$ as checking whether $J \geq q$. The set of *evaluated* children is $\emptyset$: in our model, these two rules do not generate an annotation.

These typed terminal assertions are completely defined by a type and a condition; a complete list of these keywords, with the associated type and condition, is in the full version.

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

In Figure 7 we show the rule for "type" : Tp, where $TypeOf(J)$ extracts the type of the instance J.

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

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	$TypeOf(J) = \mathsf{Tp}$
"type" : $[\mathsf{Tp}_1, \ldots, \mathsf{Tp}_n]$	$TypeOf(J) \in \{\!|\, \mathsf{Tp}_1, \ldots, \mathsf{Tp}_n \,|\!\}$

4.3 Boolean Applicators

JSON Schema boolean applicators, such as "anyOf" : $[S_1, ..., S_n]$, 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 the assertion fails; this should be contrasted with the behavior of schemas, where a failing schema never returns any annotation (Section 4.5).[6]

The rule for the disjunctive applicator "anyOf" 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.

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.

4.4 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" 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 constrains all instance fields whose name matches any pattern p_j in the applicator, but it does not

[6] Other interpretations of the specifications of Draft 2020-12 are possible; see the full version for a discussion.

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

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

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

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

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

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

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

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

$$C \vdash^{\mathsf{S}} J : \mathtt{true} \to (\mathcal{T}, \emptyset) \text{(true)} \qquad C \vdash^{\mathsf{S}} J : \mathtt{false} \to (\mathcal{F}, \emptyset) \text{(false)}$$

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

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

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

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

$$C \vdash^{\mathsf{L}} J : [\!|\, |\!] \to (\mathcal{T}, \emptyset) \text{(klist-0)}$$

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

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

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

Fig. 7. Validation rules.

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. Patterns p_j may have a non-empty intersection of their languages, so that a single instance field may match two or more patterns.

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

In the non-trivial case (rule (patternProperties)), 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 $\mathcal{T}$ (*true*). According to the standard convention, the empty conjunction $\wedge(\{\!|\ |\!\})$ evaluates to $\mathcal{T}$, hence this rule does not force any matching.

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 we are not interested in children of children.

The rule (properties) for `"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 $k'_i \in L(p_j)$, hence l is the number of pairs (i, j) such that $k'_i = k_j$; likewise, no name match is required, but in case of a match, the corresponding child of J must satisfy the subschema with the same name.

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 and array applicators can be found in the full version.

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"`. All these dependencies are formalized in the following sections.

4.5 The Semantics of Schemas: Sequential Evaluation of Keywords

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

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

The rules (schema-true) and (schema-false) for an object schema $\{ \vec{K} \}$ are 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-true) just reuses the result of the keyword-list judgment, but (schema-false) specifies that, as dictated by [Wright et al. 2022], when schema validation fails, all annotations are removed, and hence no instance child is regarded as *evaluated*. This is a crucial difference with keyword-lists, since the $C \vDash^{\mathsf{K}} J : K \to (r, \kappa)$ judgment may return non-empty annotations even when $r = \mathcal{F}$.[7]

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 $|\vec{K}| + |K|$ and by cases on K.

We start with the crucial rule (unevaluatedProperties), for $\vec{K} +$ `"unevaluatedProperties"` : S, which forces all unevaluated properties to conform to S. To evaluate $\vec{K} +$ `"unevaluatedProperties"` : S we

[7] See the discussion in the full version.

first evaluate $\vec{K}$, which yields a set of *evaluated* children κ, we then evaluate S on the other children, and we combine the results by conjunction. We return every property of the instance as *evaluated*.

The rule for $\vec{K}$ + `"additionalProperties"` : S (additionalProperties) is identical, apart from the fact that we only eliminate the properties that have been *evaluated* by *adjacent* keywords. The specifications [Wright et al. 2022] 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 $L(\emptyset) = \emptyset$.

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

The keyword `"additionalProperties"` was already present in Classical JSON Schema, and, as shown by our formalization, it does not need to access the annotations passed by the previous keywords, but can be implemented on the basis of 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 the new keyword `"unevaluatedProperties"` in order to overcome this limitation.

The rules for `"unevaluatedItems"` : S and `"items"` : S are similar and can be found in the full version.

Having exhausted the rules for the dependent keywords, we have a catch-all rule (klist-(n+1)) 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. Rule (klist-0) is just the base case for induction.

4.6 Static and Dynamic References

Annotation-dependent validation and dynamic references are the two additions that characterize Modern JSON Schema. Dynamic references are those that had the greatest need for formalization.

The reference operators `"$ref"` : *absURI* · `"#"` · *fragmentId* and `"$dynamicRef"` : *absURI* · `"#"` · *fragmentId* are in-place applicators that allow a URI-identified subschema to be applied to the current instance, but the two applicators interpret the URI in a very different way.

`"$ref"` : *absURI*·`"#"`·*fragmentId* retrieves the resource S identified by *absURI*, which may be the current schema or a different one, retrieves the subschema S' of S identified by *fragmentId*, and applies S' to the current instance J (rule (`"$ref"`)). The `"$dynamicRef"` keyword, instead, interprets the reference in a way that depends on the *dynamic scope*, which is, informally, the ordered list of all resources that have been visited in the current branch of the proof tree, which we represent in the rules by listing their URIs in the context C.

Specifically, as shown in rule (`"$ref"`), the evaluation of `"$ref"` : *absURI*·`"#"`·*fragmentId* changes the dynamic scope, by extending the context C in the premise with *absURI* — in the rule, $L + e$ denotes the operation of adding an element e at the end of a list L.

In rule ($ref), `load`(*absURI*) returns the schema S identified by *absURI*, an operation that we cannot formalize since the standards leave it undefined [Berners-Lee et al. 2005; Wright et al. 2022];

we can regard it as an access to an immutable store that associated URIs to schemas. $\texttt{get}(S, f)$ 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,[8] or a JSON Pointer, that begins with "/" and is interpreted by navigation. The get function is formally defined in the full version.

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 $\texttt{load}(\mathit{absURI})$ invocation will not fail, that is, *absURI* is a valid URI;
(2) the $\texttt{get}(\texttt{load}(\mathit{absURI}), f)$ invocations will not fail, that is, $\texttt{load}(\mathit{absURI})$ contains a subschema identified by f (which is either a JSON Pointer or an anchor name);
(3) the $\texttt{dget}(\texttt{load}(\mathit{fURI}), f)$ invocation will not fail, that is, $\texttt{load}(\mathit{fURI})$ contains a subschema identified by the dynamic anchor f;
(4) every "$id" operator assigns to its schema a URI that is different from the URI of any other resource recursively reachable from its schema;
(5) there exist no two "$anchor" and "$dynamicAnchor" keywords that assign the same name to two different schemas inside one specific resource.

If any of these conditions does not hold, the validator should raise a failure.

The behavior of "$dynamicRef" is very different from that of "$ref", and is defined as follows (see [Wright et al. 2022] 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 $\texttt{"\$dynamicRef"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f$, one first verifies whether the resource referenced by the "starting point URI" *absURI* contains a dynamic anchor $\texttt{"\$dynamicAnchor"} : f'$ with $f' = f$. If this is the case, $\texttt{"\$dynamicRef"} : \mathit{absURI} \cdot \texttt{"\#"} \cdot f$ 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 $\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot$ and $\texttt{dget}(\texttt{load}(\mathit{absURI}), f) = \bot$ of the two rules that we present above 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.[9] After this check is passed, the dynamic interpretation focuses on the fragment f, and it looks for the first (the "outermost") resource in C^+ that contains a subschema identified by $\texttt{"\$dynamicAnchor"} : f$, 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 $\texttt{"\$dynamicAnchor"} : \texttt{"f"}$, and returns $\bot$ when no such subschema is found in S, and its definition is given in the full version. The function $\texttt{fstURI}(L, f)$ returns the first URI in the list L that defines f, that is, such that $\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot$.

We can finally formalize the dynamic reference rule ($dynamicRef). 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,

[8] Actually, it can also be matched by a "$dynamicAnchor" operator, which, in this case, is interpreted as exactly as "$anchor".
[9] Observe that $\texttt{dget}(\texttt{load}(\mathit{absURI}), f) \neq \bot$ 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), yet this decision would not affect our results.

we look for the first URI *fURI* in $C + absURI$ where the dynamic anchor f is defined, and extract the corresponding subschema S' by executing $\texttt{dget}(\texttt{load}(fURI), f)$.

Example 1. The dynamic references mechanism allows one to define multiple successive refinements of a same schema. One may first refine the trees of Figure 2 to the strict trees of Figure 3, and then to the integer strict trees of Figure 8 below.

```
{ "$id": "http://mjs.ex/strict-int-tree",
  "$dynamicAnchor": "tree",
  "$ref": "http://mjs.ex/strict-tree#tree",
  "properties": { "data" : { "type" : "integer"  } }
}
```

Fig. 8. Refining a refined type.

In an empty context, a reference `"$ref" : "http://mjs.ex/strict-int-tree#tree"` would invoke this schema, which would then invoke the strict-tree of Figure 3, which would invoke the simple-tree of Figure 2. At this point we find the only dynamic reference, the following line in Figure 2, which says that the children of a tree are trees:

```
    "children": {  "type": "array", "items": { "$dynamicRef": "http://mjs.ex/simple-tree#tree" }}}}
```

Here, the dynamic context C is:

[| `"http://mjs.ex/strict-int-tree"`, `"http://mjs.ex/strict-tree"`, `"http://mjs.ex/simple-tree"` |]

The dynamic reference is resolved as `"http://mjs.ex/strict-int-tree#tree"`, since `"http://mjs.ex/strict-int-tree"` is the first resource in C that defines the dynamic anchor `"tree"`.

Remark 1. Observe that $\texttt{fstURI}(C + absURI, f)$ searches *fstURI* into a list that contains the dynamic context extended with the URI *absURI*. We have the impression that the specifications (as copied above) would rather require $fURI = \texttt{fstURI}(C, f)$, but we contacted the authors and checked some online verifiers that are widely adopted. There seems to be a general agreement that $fURI = \texttt{fstURI}(C + absURI, f)$ is the correct formula (see the full version for a concrete example).

This is a typical example of the problems generated by natural language specifications, where different readers interpret the same document in different ways, and one needs to discover the current consensus by social interaction and experiments. Formal specifications would be extremely useful to address this kind of problem.

The second rule for `"$dynamicRef"` (rule ($dynamicRefAsRef)) applies when the initially resolved starting point URI does not include a fragment that was created by the `"$dynamicAnchor"` keyword ($\texttt{dget}(\texttt{load}(absURI), f) = \bot$), in which case `"$dynamicRef"` behaves as `"$ref"`.

4.7 Compressing the Context by *Saturation*

The only rule that depends on the context C is rule ($dynamicRef) that uses $\texttt{fstURI}(C + absURI, f)$ to retrieve, in $C + absURI$, the first URI that identifies a schema that contains f as a dynamic anchor. When URI is already present in C, its addition at the end of C does not affect the result of fstURI, hence, for each URI, we could just retain its first occurrence in C. Let us define $C{+}?URI$, that we read as C *saturated with* URI, as $C{+}?URI = C$ when $URI \in C$ and $C{+}?URI = C + URI$ when $URI \notin C$. By the observation above, we can substitute $C + URI$ with $C{+}?URI$ in the premises of rules ($ref), ($dynamicRef), and ($dynamicRefAsRef), obtaining, for example the following rule.

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

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

4.8 Ruling Out Infinite Proof Trees

A *proof tree* is a tree whose nodes are judgments and such that, for every node N of the tree, there is a deduction rule that allows N to be deduced from its children. A judgment N is *proved* when there is a *finite* proof tree whose root is N.

The *naive application algorithm*, given a triple C, J, S, builds the proof tree rooted in $C \vdash^{S} J : S \to (r, \kappa)$ by finding the deduction rule whose conclusion matches C, J, S, and by recurring on all the judgments in its premises.

Consider now a schema that reapplies itself to the current instance, such as:

$$S_{loop} \;=\; \{\,\texttt{"\$id"} : \texttt{"here"}, \texttt{"allOf"} : [\,\{\,\texttt{"\$ref"} : \texttt{"here"}\,\}\,]\,\}.$$

The naive algorithm would produce an infinite loop when applied to a triple (C, J, S_{loop}), for any C and J, which reflects the fact that any proof tree whose root is $C \vdash^{S} J : S_{loop} \to (r, \kappa)$ is infinite.

The JSON Schema specifications forbid any schema which may generate infinite proof trees. Pezoa et al. [Pezoa et al. 2016] formalized this constraint for Classical JSON Schema as follows (we use the terminology of [Attouche et al. 2022]).

Definition 1 (Unguardedly references in Classical JSON Schema; well-formed schema). Given a closed Classical JSON Schema schema S, a subschema S_i of S "unguardedly references" a subschema S_j of S if the following three conditions hold:

(1) $\texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ is a keyword of a subschema S'_i of S_i (that is, $\texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ is one of the fields of the object S'_i);
(2) every keyword (if any) in the path from S_i to S'_i is a boolean applicator (S'_i is *unguarded*);
(3) $\texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ refers to S_j, that is: $\texttt{get}(\texttt{load}(\mathit{absURI}), f) = S_j$,

A closed schema S is well-formed if the graph of the "unguardedly references" relation is acyclic.

For example, the schema S_{loop} above unguardedly references itself (all operators in the path to the subschema $\{\,\texttt{"\$ref"} : \texttt{"here"}\,\}$ are boolean) hence it is not well-formed; instead, the reference in the following schema is guarded by a `"properties"` keyword, hence the schema is well-formed.

```
{  "$id": "http://mjs.ex/gd", "properties": { "data": { "$ref": "http://mjs.ex/gd" }}}
```

Observe that this is an over-conservative criterion: a schema containing an unguarded cycle that is unreachable from the root would be judged ill-formed, but would never generate infinite proofs.

We can extend this definition to Modern JSON Schema as follows. Observe that the new form of (3) means that, while a static keyword $\texttt{"\$ref"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ "unguardedly references" exactly one schema, a dynamic keyword $\texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ "unguardedly references" all the schemas that define f as a dynamic anchor, regardless of their *URI*.

Definition 2 (Unguardedly references for Modern JSON Schema, well-formed). Given a closed Modern JSON Schema schema S, a subschema S_i of S "unguardedly references" a subschema S_j of S if either S_i "unguardedly references" subschema S_j accordingly to Definition 1, or if:

(1) $\texttt{"\$dynamicRef"} : \mathit{absURI}\cdot\texttt{"\#"}\cdot f$ is a keyword of a subschema S'_i of S_i;
(2) every keyword (if any) in the path from S_i to S'_i is a boolean applicator;

(3) `"$dynamicAnchor"` : f is a keyword of S_j.

A closed schema S is well-formed if the graph of the "unguardedly references" relation is acyclic.

For example, consider a closed schema that embeds the two schemas in Figures 2 and 3. Let us use S_{dr} to indicate the subschema that immediately encloses the dynamic reference `"$dynamicRef"` : `"http://mjs.ex/simple-tree#tree"` in Figure 2. The subschema S_{dr} unguardedly references the entire schemas of Figure 2 and of 3, since both contain a dynamic anchor *tree*. However, the schema of Figure 2 does not unguardedly reference itself, nor the schema of Figure 3, since its subschema S_{dr} is *guarded* by the intermediate `"properties"` keyword. Moreover, no other subschema references S_{dr}, hence the graph of the *unguardedly references* relation is acyclic.

When a closed schema S is well formed, then every proof about any subschema of S is finite.

Theorem 3 (Termination). If a closed schema S is well formed, then, for any C that is formed using URIs of S, for any J, r, and κ, there exists one and only one proof tree whose root is $C \vdash^{\hat{S}} J : S \to (r, \kappa)$, and that proof tree is finite.

From now on, we will assume that our rules are only applied to well-formed schemas, so that every proof-tree is guaranteed to be finite. Of course, all schemas that we will use in our examples will be well-formed.

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

Validation is usually regarded as a low-cost test to be embedded in efficient processes such as distributed function invocation: The server declares the expected schema of its parameters and, for every invocation, each parameter is validated by the declared schema. Hence, data-definition languages are usually designed in order to get a high expressive power with a low validation cost. In practice, one would like the validation problem to run in polynomial time in the worst case. This is the case for Classical JSON Schema, whose validation is P-complete [Pezoa et al. 2016]. The input of the validation problem is a couple (J, S), for which we ask whether $[\![\mathit{baseURI}]\!] \vdash^{\hat{S}} J : S \to (\mathcal{T}, \kappa)$, (where *baseURI* is the base URI of S), for some κ; hence, for Classical JSON Schema, the time bound is a polynomial function f whose argument is the total input size, $|J| + |S|$.

Unfortunately, this is not the case for Modern JSON Schema. Dynamic references add a seemingly minor twist to the validation rules, but this twist has a dramatic effect on the computational complexity of validation: we prove here that dynamic references make validation PSPACE-hard. Following [Sipser 2012], we recall here that PSPACE is the class of decision problems that can be solved in polynomial *space* by a deterministic Turing machine; a decision problem B is PSPACE-hard if every problem in PSPACE is polynomial time reducible to B. It is well known that $P \subseteq NP \subseteq PSPACE$. While there is still no proof that the inclusions are proper, no reduction from *PSPACE* to *NP*, or to *P*, is known, hence *PSPACE*-hard problems are currently regarded as "intractable".

We prove that validation is PSPACE-hard by reducing quantified Boolean formulas (QBF) validity, a well-known PSPACE-complete problem [Stockmeyer and Meyer 1973], 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 $[\![\mathit{baseURI}]\!] \vdash^{\hat{S}} J : S_\psi$ returns $(\mathcal{T}, \emptyset)$ if, and only if, the formula ψ is valid.

Observe that the actual value of J is irrelevant: in our encoding, the schema S_ψ is either satisfied by any instance, or by none at all: S_ψ is a *trivial* schema, where *trivial* indicates a schema that returns the same result when applied to any instance J, as happens for the schemas `true` and `false`. Hence, we actually prove that the validation problem is PSPACE-hard even when restricted to trivial schemas only.

```
1 { "id": "urn:psi",
2   "$schema": "https://json-schema.org/draft/2020-12/schema",
3   "allOf": [ { "$ref": "urn:truex1#afterq1" }, { "$ref": "urn:falsex1#afterq1" }],
4   "$defs": {
5     "urn:truex1": {
6       "$id": "urn:truex1",
7       "$defs": {
8         "x1":  { "$dynamicAnchor": "x1", "anyOf": [true] },
9         "not.x1":  { "$dynamicAnchor": "not.x1", "anyOf": [false] },
10        "afterq1":
11           { "$anchor": "afterq1",
12              "anyOf": [ { "$ref": "urn:truex2#afterq2" },  { "$ref": "urn:falsex2#afterq2" }]
13    }}},
14    "urn:falsex1": {
15      "$id": "urn:falsex1",
16      "$defs": {
17        "x1":  { "$dynamicAnchor": "x1", "anyOf": [false] },
18        "not.x1":  { "$dynamicAnchor": "not.x1", "anyOf": [true] },
19        "afterq1":
20           { "$anchor": "afterq1",
21              "anyOf": [ { "$ref": "urn:truex2#afterq2" },  { "$ref": "urn:falsex2#afterq2" }]
22    }}},
23    "urn:truex2": {
24      "$id": "urn:truex2",
25      "$defs": {
26        "x2":   { "$dynamicAnchor": "x2", "anyOf": [true] },
27        "not.x2":  { "$dynamicAnchor": "not.x2", "anyOf": [false] },
28        "afterq2":  { "$anchor": "afterq2", "$ref": "urn:phi#phi" }
29    }},
30    "urn:falsex2": {
31      "$id": "urn:falsex2",
32      "$defs": {
33        "x2":  { "$dynamicAnchor": "x2", "anyOf": [false] },
34        "not.x2":  { "$dynamicAnchor": "not.x2", "anyOf": [true] },
35        "afterq2":   { "$anchor": "afterq2", "$ref": "urn:phi#phi" }
36    }},
37    "urn:phi": {
38      "$id": "urn:phi",
39      "$defs": {
40        "phi": {
41           "$anchor": "phi",
42           "anyOf": [
43             { "allOf":[{ "$dynamicRef": "urn:truex1#x1" }, { "$dynamicRef": "urn:truex2#x2" }]},
44             { "allOf":[{ "$dynamicRef": "urn:truex1#not.x1"}, { "$dynamicRef": "urn:truex2#not.x2"}]}]
45 }}}}
```

Fig. 9. Encoding $\forall x1.\ \exists x2.\ (x1 \wedge x2) \vee (\neg x1 \wedge \neg x2)$.

We start with an example. Consider the following QBF formula: $\forall x1.\ \exists x2.\ (x1 \wedge x2) \vee (\neg x1 \wedge \neg x2)$; Figure 9 shows how it can be encoded as a JSON Schema schema (we use here URIs based on `"urn:"` rather than on `"https:"`, for space reasons).

For each variable xi we define a resource `"urn:truex"`$\cdot i$ (lines 5-13 and 23-29 of Figure 9), which defines two dynamic schemas, one with plain-name `"x"`$\cdot i$ and value `true`, and the other one with plain-name `"not.x"`$\cdot i$ and value `false`[10] (lines 8-9 and 26-27). For each variable xi we also define a resource `"urn:falsex"`$\cdot i$ (lines 14-22 and 30-36), where, on the contrary, `"x"`$\cdot i$ has value `false`, and `"not.x"`$\cdot i$ has value `true`.

The formula ϕ is encoded in the schema `"urn:phi#phi"` (lines 40-45). All variables in `"urn:phi#phi"` are encoded as dynamic references, so that their value depends on the resources that are in-scope when `"urn:phi#phi"` is evaluated. Consider, for example, `"$dynamicRef" : "urn:truex1#x1"` inside

[10]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"`$\cdot i$ is clearly redundant, and is there only for readability.

"urn:phi#phi". A dynamic anchor "x1" is defined in "urn:truex1" and in "urn:false1", hence, if the context of the evaluation of "urn:phi#phi" contains the URI "urn:truex1" before "urn:false1", or without "urn:false1", then "$dynamicRef" : "urn:truex1#x1" resolves to the true subschema defined in "urn:true1". If the context of the evaluation contains "urn:false1" before, or without, "urn:true1", then "$dynamicRef" : "urn:truex1#x1" resolves to the false subschema defined in "urn:falsex1#x1". Observe that a dynamic reference "$dynamicRef" : "urn:falsex1#x1" would behave exactly as "$dynamicRef" : "urn:truex1#x1" — a fundamental feature of dynamic references is that the absolute URI before the "#" is substantially irrelevant, a fact that we indicate by underlining it.

Now we describe how we encode the quantifiers. The first quantifier is encoded in the root schema; if the quantifier is ∀, as in this case, then we apply "allOf" to two references (line 3), one that checks whether the rest of the formula holds when $x1$ is true, by invoking "urn:truex1#afterq1", which sets $x1$ to true by bringing "urn:truex1" in scope, and the second one that checks whether the rest of the formula holds when $x1$ is false, by invoking "urn:falsex1#afterq1", which sets $x1$ to false by bringing "urn:falsex1" in scope.

The formulas "urn:truex1#afterq1" and "urn:falsex1#afterq1", which are identical, encode the evaluation, in two different contexts, of the rest of the formula $\exists x2.\ (x1 \wedge x2) \vee (\neg x1 \wedge \neg x2)$. They encode the existential quantifier (lines 12 and 21) in the same way as the universal one in line 3, with the only difference that "anyOf" substitutes "allOf", so that "urn:truex1#afterq1" holds if the rest of the formula holds for at least one boolean value of $x2$, when $x1$ is true, and similarly for "urn:falsex1#afterq1" when $x1$ is false. This technique allows one to encode any QBF formula with a schema whose size is linear in the size of the formula $Q_1x_1 \dots Q_nx_n.\ \phi$: the size of "urn:phi#phi" is linear in $|\phi|$, and the rest of the schema is linear in $|Q_1x_1 \dots Q_nx_n|$.

Observe that the schema is well-formed: every maximal path in the unguardedly-references graph has shape *r-aq1-aq2-phi-var*, where

(1) *r* is the root schema;
(2) aq1 matches "urn:(true|false)x1#afterq1";
(3) aq2 matches "urn:(true|false)x2#afterq2";
(4) *phi* is "urn:phi#phi";
(5) *var* matches "urn:(true|false)xi#[not.]xi".

We now formalize this encoding.

Definition 4 (S_ψ). Consider a generic closed QBF formula: $\psi = Q_1x_1 \dots Q_nx_n.\ \phi$, 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 $2n + 2$ resources: "urn:psi" (the root), "urn:truex"$\cdot i$ and "urn:falsex"$\cdot i$, for i in $\{\!|\ 1 \dots n\ |\!\}$, and "urn:phi".

The root encodes the outermost quantifiers as follows:

$$\mathit{boolOp} : [\quad \{\text{"\$ref"} : \text{"urn:true1\#afterq1"}\},\quad \{\text{"\$ref"} : \text{"urn:false1\#afterq1"}\}\ ]$$

where *boolOp* = "allOf" when $Q_1 = \forall$, and *boolOp* = "anyOf" when $Q_1 = \exists$.

Every other resource defines a set of named subschemas, each containing a "$anchor" or a "$dynamicAnchor" keyword that assigns it a name, and one more keyword that we call its "body".

For each x_i, the resource "urn:truex"$\cdot i$ contains 3 named subschemas: "x"$\cdot i$, "not.x"$\cdot i$, and "afterq"$\cdot i$. The body of "x"$\cdot i$ is "anyOf" : [true], and the body of "not.x"$\cdot i$ is "anyOf" : [false].

When $i < n$, the body of "afterq"$\cdot i$ encodes $Q_{i+1}x_i.\ \phi$ as follows:

$$\mathit{boolOp} : [\quad \{\text{"\$ref"} : \text{"urn:truex"}\cdot i\cdot\text{"\#afterq"}\cdot i\},\quad \{\text{"\$ref"} : \text{"urn:falsex"}\cdot i\cdot\text{"\#afterq"}\cdot i\}\ ]$$

where *boolOp* = "allOf" when $Q_{i+1} = \forall$, and *boolOp* = "anyOf" when $Q_{i+1} = \exists$. When $i = n$, the body of "afterq"$\cdot i$ is just "$ref" : "urn:phi#phi".

Finally, the "urn:phi" resource only contains a schema "phi", whose body is S_ϕ, which is recursively defined as follows:

$$S_{x_i} = \{\text{"\$dynamicRef"} : \text{"}\underline{\text{urn:truex}}\text{"}\cdot\underline{i}\#\text{"x"}\cdot i\} \qquad S_{\phi_1\vee\phi_2} = \text{"anyOf"} : [\ S_{\phi_1}, S_{\phi_2}\ ]$$
$$S_{\neg x_i} = \{\text{"\$dynamicRef"} : \text{"}\underline{\text{urn:truex}}\text{"}\cdot\underline{i}\#\text{"not.x"}\cdot i\} \qquad S_{\phi_1\wedge\phi_2} = \text{"allOf"} : [\ S_{\phi_1}, S_{\phi_2}\ ].$$

The following theorem states the correctness of the translation.

Theorem 5. Given a QBF closed formula $\psi = Q_1 x_1 \dots Q_n x_n.\ \phi$ and the corresponding schema S_ψ, ψ is valid if, and only if, for every J, $[\![\text{"urn:psi"}]\!]^{\mathcal{S}} \vdash J : S_\psi \rightarrow (\mathcal{T}, \emptyset)$.

Since the encoding has linear size, PSPACE-hardness is an immediate corollary. In our encoding we use the eight operators "$ref", "$anchor", "$dynamicRef", "$dynamicAnchor", "anyOf", "allOf", true, and false, but 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". Hence, only the operators we list below are really needed for our results.

Corollary 6 (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

We present here a polynomial-space validation algorithm, hence proving that the PSPACE bound is tight. 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. This list could be replaced by a static check of well-formedness, but we prefer to employ this dynamic approach, since the list is useful for the complexity evaluation.

For each schema, Algorithm 1 evaluates its keywords, passing the current value of the boolean result and of the *evaluated* children from one keyword to the next. 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 in-place independent applicators ("anyOf"), in-place applicators that update the context ("$dynamicRef"), structural independent applicators ("patternProperties"), and dependent applicators ("unevaluatedProperties").

Function SchemaValidate (*Cont, Inst, Schema, StopList*) applies *Schema* in the context *Cont*, that is, a list of absolute URIs without repetitions, to *Inst*, and uses *StopList* in order to avoid infinite recursion. The *Cont* list is extended by the evaluation of dynamic and static references using the function Saturate (*Cont, URI*) (line 30), which adds *URI* to *Cont* only if it is not already there. The *StopList* records the (Cont, Inst, 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.

Now we prove 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 (Cont, Inst, Schema, StopList) - KeywordValidate (…) - k(...) - SchemaValidate (Cont′, Inst′, Schema′, StopList′), where $k(\dots)$ is the keyword-specific

Algorithm 1: Validation

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

function invoked by `KeywordValidate`. We focus on the sequence of `SchemaValidate` (Cont, Inst, Schema, StopList) tuples, ignoring the intermediate calls. This sequence can be divided in at most n subsequences, if n is the input size, the first one with a context that contains only 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 (Inst, 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, and 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 it invokes, directly and indirectly; the result follows, since these functions are never more than $O(n^3)$ at the same time. This is the basic idea behind the following Theorem, whose full proof can be found in the full version. Since our algorithm runs in polynomial space, the problem of validation for Modern JSON Schema is PSPACE-complete.

Theorem 7. 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 AND POLYNOMIAL TIME DATA COMPLEXITY

While dynamic references make validation PSPACE-hard, annotation-dependent validation alone does not change the P complexity of Classical JSON Schema validation. We prove this fact by restricting our attention to the set of all schemas that contain at most k references, where k is a fixed integer; for this set of schemas we define 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: When, during the computation of $[\![b]\!] \vDash^S J : S \to (r, \kappa)$, we complete the evaluation of an intermediate judgment $C' \vDash^S J' : S' \to (r', \kappa')$, we store this intermediate result. However, while there is only a polynomial number of S' and J' that may be generated while proving $[\![b]\!] \vDash^S J : S \to (r, \kappa)$, there is an exponential number of different C', corresponding to different subsets of URIs that appear in S and to different reordering of these subsets; this phenomenon occurs, for example, in our leading example (Figure 9). We solve this problem in the case of a fixed bound on the number of dynamic references by observing that two different contexts C_1 and C_2 are equivalent, with respect to a specific validation problem $C' \vDash^S J' : S' \to (r', \kappa')$, when the two resolve in the same way any dynamic reference that is actually expanded during the analysis of that specific problem. In the bounded case, this equivalence relation on contexts has a polynomial number of equivalence classes, which allows us to recompute the result of $C' \vDash^S J' : S'$, for a fixed pair J', S', only for a polynomial number of different contexts C'.

In greater detail, our algorithm returns, for each evaluation of S over J in a context C, not only the boolean result and the *evaluated* children, but also a *DFragSet*, that returns the set of fragment ids f such that $\texttt{fstURI}(\_, f)$ has been computed during that evaluation. For each evaluated judgment $C \vDash^S J : S \to (r, \kappa)$, we add the tuple $(C, J, S, r, \kappa, \mathit{DFragSet})$ to 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 are polynomial in size of S, hence that memoization limits the total number of function calls 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 the full version, we also report how `KeywordValidate` is modified.

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(Cont,Inst,Schema,StopList,UpdatableStore)
2      if (Schema == True) then return (True,EmptySet,EmptySet) ;
3      if (Schema == False) then return (False,EmptySet,EmptySet) ;
4      Input := (Cont,Inst,Schema);
5      if (Present (Input,StopList)) then  raise ("error: infinite loop") ;
6      PrevResultSameSchemaInst := UpdatableStore.get(Inst,Schema);
7      for (OldCont,OldRes,OldEval,OldDFragSet) in PrevResultSameSchemaInst do
8          if (Equivalent (Cont,OldCont,OldDFragSet)) then  /* If the current Cont is equivalent to the
               OldCont we reuse the old output                                                    */
9          return (OldOutput);
10     Output := (True,EmptySet,EmptySet);
11     for Keyword in Keywords (Schema) do
12         Output := KeywordValidate (Cont, Inst, Keyword, Output, StopList+Input,UpdatableStore) ;
13     (Res,Eval,DFragSet) := Output;
14     UpdatableStore.addToList((Inst,Schema),(Cont,Res,Eval,DFragSet));
15     if Res == True then  return (Res, Eval, DFragSet) ;
16     else  return (Res, EmptySet, DFragSet);
17
18 Equivalent (Cont,OldCont,DFragSet)
19     Res := True;
20     for f in DFragSet do  Res := And (Res, (FirstURI (Cont,f) ==FirstURI (OldCont,f)));
21     return (Result);
```

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.

Theorem 8. Algorithm 2 applied to $(C, J, S, \emptyset, \emptyset)$ returns (r, κ, d), for some d, if, and only if, $C \vDash^S J : S \longrightarrow (r, \kappa)$.

Theorem 9. 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"` : $initURI$`"#"`$\cdot f$ in S. Then, Algorithm 2 runs on S and J in time $O(n^{k+|D|})$ for some constant k.

Corollary 10. Validation is in P if we fix a constant bound on the maximum number of different fragments f that appear in the argument of a `"$dynamicRef"` : $initURI \cdot$`"#"`$\cdot f$ in S.

P data complexity. There are situations where the schema is fixed and has a very small size by comparison to the instance size, hence it is important to understand how the cost of evaluating $[\![b]\!] \vDash^S J : S$ depends on the size of J, when S is fixed; this is analogous to the notion of *data complexity* that is standard in the database field [Vardi 1982]. When the schema is fixed, then, a

fortiori, also the number of different fragments that are argument of "$dynamicRef" is fixed; hence, by Corollary 10, the problem of validating arbitrary instances using any fixed schema is in P.

Corollary 11 (Fixed-schema complexity). When S is fixed, the validation problem $[\![b]\!] \vdash^S J : S \rightarrow (r, \kappa)$ is in P with respect to $|J|$.

Fixed-schema complexity is similar to data complexity in query evaluation, but the parallelism is not precise: while queries are, in practical cases, almost invariably much smaller than data, there are many situations where JSON Schema documents are bigger than the checked instances, for example, when complex schemas are used in order to validate function parameters.

8 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 substituted with static references, at the price of a potentially exponential increase in the size of the schema.

A dynamic reference "$dynamicRef" : $initURI \cdot$"#"$\cdot f$ is resolved, during validation, to a URI reference $\texttt{fstURI}(C{+}?initURI, f)\cdot$"#"$\cdot f$ that depends on the context C of the validation (Section 4.6), 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 behavior of this subschema depends on the context, as happens with "$ref" : "urn:phi#phi" in our example, which is a static reference, but the behavior of the schema it refers to depends on the context.

To obtain the same effect without dynamic references, we observe that, if the context C is fixed, then every dynamic reference has a fixed behavior, and it can be encoded using a static reference "$ref" : $\texttt{fstURI}(C{+}?initURI, f)\cdot$"#"$\cdot f$. 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 "$dynamicRef" : $initURI\cdot$"#"$\cdot f$ with a static reference to the context-injected version of the schema identified by $\texttt{fstURI}(C{+}?initURI, f)\cdot$"#"$\cdot f$, and (2) substituting every static reference "$ref" : $absURI \cdot$"#"$\cdot f$ with a static reference to the context-injected version of the schema identified by $absURI\cdot$"#"$\cdot f$. 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 $C = [\![$"urn:psi","urn:truex1","urn:falsex2","urn:phi"$]\!]$ and a reference "$ref" : "urn:phi#phi", which refers to the following schema S', which contains four dynamic references, and which can be found inside the resource "urn:phi".

```
{ "$anchor": "phi",
  "anyOf": [ {"allOf": [{"$dynamicRef": "urn:truex1#x1"}, {"$dynamicRef": "urn:truex2#x2"}]},
             {"allOf": [{"$dynamicRef": "urn:truex1#not.x1"},  {"$dynamicRef": "urn:truex2#not.x2"}]}]}
```

The corresponding context-injected schema $\texttt{CI}(C, S')$ is the following. When a schema is identified by $absURI\cdot$"#"$\cdot f$, we identify its context-injected version $\texttt{CI}(C, S')$ using $absURI\cdot$"#"$\cdot \underline{C}\cdot f$, where $\underline{C}$ is an invertible encoding of C into a plain-name.[11]

[11] In the example, we encode a sequence of absolute URIs such as [|"urn:psi","urn:truex1","urn:falsex2","urn:phi"|] as "urn:psi_urn:truex1_urn:falsex2_urn:phi_", that is, we escape any underscore inside the URIs (not exemplified here), and we terminate each URI with an underscore.

```
1 { "$anchor": "urn:psi_urn:truex1_urn:falsex2_urn:phi_phi",
2   "anyOf": [
3     {"allOf":  [{"$ref":  "urn:truex1#urn:psi_urn:truex1_urn:falsex2_urn:phi_x1" },
4                 {"$ref":  "urn:falsex2#urn:psi_urn:truex1_urn:falsex2_urn:phi_x2" }]},
5     {"allOf":  [{"$ref":  "urn:truex1#urn:psi_urn:truex1_urn:falsex2_urn:phi_not.x1"},
6                 {"$ref":  "urn:falsex2#urn:psi_urn:truex1_urn:falsex2_urn:phi_not.x2"}
7     ]}]},
```

In this example, it is interesting to observe how the underlined absolute URI "urn:truex2" of "$dynamicRef" : "urn:truex2#x2" and "$dynamicRef" : "urn:truex2#not.x2" (lines 3 and 4) has been substituted with "urn:falsex2" (lines 4 and 6), which reflects the fact that the context C contains "urn:falsex2" but does not contain "urn:truex2". On the other hand, the underlined absolute URI "urn:truex1" has been substituted with a static reference to "urn:truex1", reflecting the presence of "urn:truex1" in C; the complete unfolding is in the full version.

We can now give a formal definition of the translation process. For simplicity, we assume that all fragment identifiers referred to by "$ref" are plain-names defined using "$anchor", 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 $\mathtt{CI}(C, S)$ without dynamic references, and that maps every pair (C, K) to a keyword $\mathtt{CI}(C, K)$ without dynamic references. This function maps references as specified below, and acts as a homomorphism on all the other operators, as exemplified here with "anyOf".

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

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 is identified by a static or a dynamic anchor (e.g., the subschema identified by "urn:phi#phi" is a fragment). The static translation of S_0, $\mathtt{Static}(S_0)$, is obtained by substituting, in S_0, each fragment S_f identified by absURI"#"$\cdot f$ with many fragments $\underline{C}\cdot f$, one for any context $C \in \mathcal{C}$, where the schema identified by each $\mathit{absURI} \cdot$ "#" $\cdot\, \underline{C} \cdot f$ is $\mathtt{CI}(C, S_f)$, as exemplified in the full version. 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 exponential expansion was to be expected, since this transformation can be used to reduce the validation problem of J using S_0, that is PSPACE-complete with respect to $|J| + |S_0|$, to validation using $\mathtt{Static}(S_0)$, which is P with respect to $|J| + |\mathtt{Static}(S_0)|$.

We can now prove that this process preserves the schema behaviour.

Theorem 12 (Encoding correctness). Let S be a closed schema with base URI b. Then:

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

9 DYNAMIC REFERENCES IN DRAFT 2019-09: "$recursiveAnchor" AND "$recursiveRef"

Dynamic references have first been introduced in Draft 2019-09, in a restricted form, where (1) dynamic anchors have no name, which means that they behave as if they all shared a unique name, and (2) the initial reference of every dynamic reference is the root of its own resource.

In detail, the dynamic references of Draft 2019-09 are based on two keywords, "$recursiveAnchor" : b, with $b \in \{\!|$ true, false $|\!\}$, and "$recursiveRef" : "#". The keyword "$recursiveAnchor" : true,[12]

[12]The keyword "$recursiveAnchor" : false has no effect at all.

placed at the top-level of its resource, has the same effect as `"$dynamicAnchor" : RecRoot` in Draft 2020-12, where *RecRoot* is an arbitrary unique anchor name: it associates a dynamic anchor to the entire resource; please note that a *unique* anchor name *RecRoot* must be used to interpret every `"$recursiveAnchor"` and every `"$recursiveRef"` in the schema and in all schemas that are reachable from it. The keyword `"$recursiveRef" : "#"` is equivalent to `"$dynamicRef" :` $\underline{baseURI} \cdot$`"#"`$\cdot RecRoot$, where *baseURI* is the base URI of the current resource. Hence, `"$recursiveRef" : "#"` is a dynamic reference that *initially* refers to the root of the resource where the keyword is found; for this reason, it is called a dynamic *recursive* reference. Here "initially refers to..." indicates the target of the static interpretation of $\underline{baseURI} \cdot$`"#"`$\cdot RecRoot$, but, at validation time, a dynamic reference is resolved to the outermost resource that defines an anchor with the same name, which will often not coincide with the static "initial" target.

If you consider the schemas of Figures 2 and 3, their Draft 2019-09 versions are obtained by replacing every `"$dynamicAnchor" : "tree"` with `"$recursiveAnchor" : true`, and every `"$dynamicRef" : "...#tree"` with `"$recursiveRef" : "#"` — `"tree"` plays the role of *RecRoot*. Similarly, the Draft 2019-09 version of the metaschema in Figure 4 is obtained by replacing every `"$dynamicAnchor" : "meta"` with `"$recursiveAnchor" : true`, and every `"$dynamicRef" : "...#meta"` with `"$recursiveRef" : "#"`.

Hence, the two restrictions of Draft 2019-09 do not prevent dynamic references from being used for their most important application, the representation of JSON Schema metaschema as a collection of many fragments. However, the presence of only one dynamic anchor name (which we indicate as *RecRoot*) means that every dynamic reference will be resolved, in a fixed dynamic context, to the same target, the root of the first resource in the context that has `"$recursiveAnchor" : true` at its root, which makes it very complicated to mix different uses of dynamic references into a single project, as we crucially do in our encoding of QBF.

The unique-name restriction drastically reduces the expressive power of the mechanism, but it makes validation polynomial: since a schema in Draft 2019-09 corresponds to a schema in Draft 2020-12 with a single dynamic anchor name then, by Corollary 10, its validation time is in P.

Corollary 13. Validation of a JSON instance by a schema that respects Draft 2019-09 is in P.

10 EXPERIMENTS

We implemented Algorithm 2 for the entire JSON Schema language in Scala, applying the rules described here and in the full version.

10.1 Correctness of Formalization

We applied our algorithm to the official JSON Schema test suite [Bergman 2023b].[13] Out of a total of 1,210 tests, we pass all apart from 14 pertaining to schemas with the following unsupported features: special characters in patterns or in URI references, unknown keywords, a vocabulary different from JSON Schema, a decimal with a high precision.

This experiment shows that the rules that we presented, and which are faithfully reflected by our algorithm, are correct and complete w.r.t. the standard test suite.

10.2 Complexity

Validation for Modern JSON Schema is PSPACE-complete in the presence of dynamic references, while it is in P when dynamic references are bound by a constant; this is not something that may be proved by a finite set of experiments, but we already provided a formal proof for this.

In the upcoming experiment, we test a rich set of validators, in order to see (1) whether there exist test cases where the PSPACE-hardness result is reflected by considerable validation times

[13]We only focus on *main* schemas and do not consider the *optional* ones, and use the version with git commit hash 6afa9b3.

with small schemas and instances and (2) whether the difference between schemas with a variable number of variables and with a fixed number of variables is visible, by this set of validators.

Schemas. We define three families of artificial schemas, designed in order to stress-test a generic JSON validator, and we validate the instance *null* against each of them; the schemas are satisfied by any instance. The schemas can be inspected online at https://github.com/sdbs-uni-p/mjs-schemas.

The *dyn schema* family comprises schemas "dyn1.js" up to "dyn100.js" and generalizes our running example; the file "dyn"·i·".js" contains the encoding of

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

as defined in Section 5; schema "dyn"·i·".js", hence, contains i pairs of variables.

The corresponding *stat schema* family comprises Draft-04 schemas "stat1.js" up to "stat100.js", where each keyword "$dynamicRef" is just substituted with the keyword "$ref", without applying the expansion we described in Section 8.

The *dyn_bounded schema* family, from "dyn.bounded1.js" up to "dyn.bounded100.js", encodes

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

Hence, the *dyn_bounded* schemas only contain four dynamic references, a fact that, according to Corollary 10, allows an optimized algorithm to run in polynomial time. Observe that the size of "dyn"·i·".js", "dyn.bonded"·i·".js", and "stat"·i·".js" schemas grows linearly with i.

Validators. For third-party validators, we employ the meta-validator Bowtie [Bergman 2023a], which invokes validators encapsulated in Docker containers. We tested all 16 different open-source validators currently[14] provided by Bowtie that support Draft 4 or Draft 2020. They are written in 11 different programming languages, as detailed in Table 3 of the full version. We also integrated within Bowtie the academic validator from [Pezoa et al. 2016], as well as our own implementation.

Execution environment. Our execution environment is a 40-core Debian server with 384GB of RAM.[15] 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 were measured as GNU time, averaged over five runs, and include the overhead of invoking Bowtie and Docker. We overrode the default timeout setting in Bowtie, to allow for longer-running experiments.

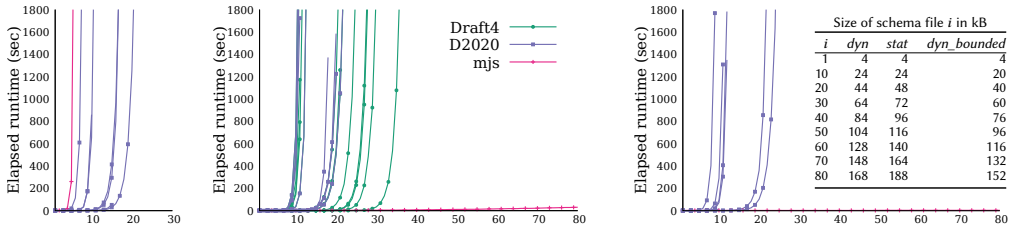
In the figures showing the measured runtimes, plotted lines terminate when the validator produces a logical validation error, a runtime exception (most commonly, a stack overflow), or when Bowtie reports "no response" by the validator.

Results. In Figures 10a-10c we show the results of our evaluation. In all figures, the x-axis indicates the i index of schemas, while the y-axis reports the runtime. We distinguish the runtimes for the Draft-04 and Draft 2020-12 validators, as well as our own prototype implementation, by different line styles. The tics denote the data points, where data points can lie outside the plotted area.

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 versions is extremely visible for 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

[14] As of July 2023, the time of writing this paper.

[15] The 40 cores allow us to run experiments in parallel. However, the experiment may just as well run on a commodity laptop. In fact, in our reproduction package, we evaluate all experiments sequentially, assuming that 2 CPUs and 6MB of memory are assigned to the virtual machine.



(a) *dyn* schemas. (b) *stat* schemas (Draft 4 compatible). (c) *dyn_bounded* schemas.

Size of schema file i in kB

i	*dyn*	*stat*	*dyn_bounded*
1	4	4	4
10	24	24	20
20	44	48	40
30	64	72	60
40	84	96	76
50	104	116	96
60	128	140	116
70	148	164	132
80	168	188	152

Fig. 10. Runtimes of validators on the three schema families, where the tics on the horizontal axes denote the index of the schema, (e.g, 10 is the index of schema "dyn"·10·".js"); the table in Subfigure 10c maps the schema indexes to the actual file sizes in kB. The tested validators support Draft-04 (green lines) or Draft 2020-12 (blue lines). "mjs" (red line) refers to our implementation.

static references. The runtime on the *dyn_bounded* family reflects Corollary 10, which shows the effectiveness of the proposed optimization on this specific example.

The results on the *stat* family strongly suggest that all other validators have chosen to implement an algorithm that is exponential even when there is no dynamic reference 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 somewhat more surprising for the validators for Draft-04, especially for the one published as additional material for [Pezoa et al. 2016]: for Draft-04, as for Classical JSON Schema in general, the validation problem is in P, as proved for the first time in that same paper [Pezoa et al. 2016].

Discussion. This experiment shows that there are families of schemas where the PSPACE-hardness of the problem is visible (Figure 10a), and that the algorithm we describe in Section 7 is extremely effective when dynamic references are replaced with static references (Figure 10b), or limited in number (Figure 10c). In this paper, we focus on worst-case asymptotic complexity, and we do not make claims about real-world relevance of our algorithm, which is an important issue, but is not in the scope of this paper.

11 RELATED WORK

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

Overviews over schema languages for JSON can be found in [Baazizi et al. 2019a,b; Bourhis et al. 2017; Pezoa et al. 2016]. In [Pezoa et al. 2016] 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 and annotation-dependent validation.

In [Bourhis et al. 2017] 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, i.e., checking whether there exists at least one instance

that is validated by the input 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 [val 2023] for a rather complete list of about 50 implementations). Only some of them (about 21), like ajv [ajv 2023] and Hyperjump [hyp 2023], support modern JSON Schema and dynamic references. These validators usually compile schemas to 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 [Martens et al. 2009, 2006], 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 [Henglein and Mairson 1994], 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 analyzes *instances of data structures*.

12 CONCLUSIONS AND OPEN PROBLEMS

Modern JSON Schema introduced annotation-dependent validation and dynamic references, whose exact interpretation is regarded as difficult to understand [Neal 2022], [Jacobson 2021]. The changes to the evaluation model invalidate 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, and it has been discussed with the community of JSON Schema tools developers. The formalization has been expressed as a Scala program, which passes the tests of the standard JSON Schema validation test suite and is available in the full version.

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 polynomial time. We have implemented and experimented with 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.

Many interesting problems remain open, 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.

ACKNOWLEDGMENTS

This work is partly funded by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) grant #385808805.

This work is partly supported by the European Union under the scheme HORIZON-INFRA-2021-DEV-02-01 — Preparatory phase of new ESFRI research infrastructure projects, Grant Agreement n.101079043, “SoBigData RI PPP: SoBigData RI Preparatory Phase Project”.

This work is partly supported by *Ministero dell'Università e della Ricerca* (MUR, Ministry of University and Research) under the PRIN Project "BioConceptum" (grant #2022AEEKXS).

The authors thank Stefan Klessinger and Sajal Jain for integrating academic validators with the Bowtie framework, and Thomas Kirz for assisting with the gnuplot visualizations. The authors thank Julian Bergman, author of the Bowtie framework, for making timeouts configurable for our experiments.

DATA AVAILABILITY STATEMENT

We provide our code, scripts, and data within a reproduction package hosted on Zenodo [Attouche et al. 2023b].

For reuse purposes, the code of our validator is available here [Attouche et al. 2023a].

REFERENCES

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

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

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

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

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

Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023a. *ModernJSONSchemaValidator.* https://gitlab.lip6.fr/jsonschema/modernjsonschemavalidator

Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023b. *Reproduction Package for: Validation of Modern JSON Schema: Formalization and Complexity.* https://doi.org/10.5281/zenodo.10019663

Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023c. Validation of Modern JSON Schema: Formalization and Complexity. arXiv:2307.10034 [cs.DB]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Michael Sipser. 2012. *Introduction to the Theory of Computation - Third Edition.* Cengage.

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

Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber (Eds.). ACM, 137–146. https://doi.org/10.1145/800070.802186

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.

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.

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.

Received 2023-07-11; accepted 2023-11-07