Not Elimination and Witness Generation for JSON Schema

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

Dario Colazzo  
Université Paris-Dauphine, PSL Research University  
dario.colazzo@dauphine.fr

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

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

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

ABSTRACT

JSON Schema is an evolving standard for the description of families of JSON documents. JSON Schema is a logical language, based on a set of assertions that describe features of the JSON value under analysis and on logical or structural combinators for these assertions. As for any logical language, problems like satisfaction, not-elimination, schema satisfiability, schema inclusion and equivalence, as well as witness generation, have both theoretical and practical interest. While satisfaction is trivial, all other problems are quite difficult, due to the combined presence of negation, recursion, and complex assertions in JSON Schema. To make things even more complex and interesting, JSON Schema is not algebraic, since we have both syntactic and semantic interactions between different keywords in the same schema object.

With such motivations, we present in this paper an algebraic characterization of JSON Schema, obtained by adding opportune operators, and by mirroring existing ones. We present then algebra-based approaches for dealing with not-elimination and witness generation problems, which play a central role as they lead to solutions for the other mentioned complex problems.

KEYWORDS

JSON Schema, negation, witness generation

ACM Reference Format:
Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. Not Elimination and Witness Generation for JSON Schema. In Proceedings of 36ème Conférence sur la Gestion de Données (BDA ’20). ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Aim of the paper

JSON Schema [2] is an evolving standard for the description of families of JSON documents. JSON Schema is a logical language, based on a set of assertions that describe features of the JSON value under analysis and on logical or structural combinators for these assertions. As for any logical language, the following problems have a theoretical and practical interest:

- **satisfaction** $J \models S$: does a JSON document $J$ satisfy schema $S$?
- **not-elimination**: is it possible to rewrite a schema to an equivalent form without negation?
- **satisfiability of a schema**: does a document $J$ exist such that $J \models S$?
- **schema inclusion** $S \subseteq S'$: does, for each document $J$, $J \models S \Rightarrow J \models S'$?
- **schema equivalence** $S \equiv S'$: does, for each document $J$, $J \models S \equiv J \models S'$?
- **witness generation**: is there an algorithm to generate one element $J$ for any non-empty schema $S$?

While satisfaction is trivial, all other problems are quite difficult, due to the combined presence of negation, recursion, and complex assertions.

A second aspect that makes the task difficult is the non-algebraic nature of JSON Schema. A language is "algebraic" when the applicability and the semantics of its operators only depends on the semantics of their operands. In this sense, JSON Schema is not algebraic, since we have both syntactic and semantic interactions between different keywords in the same schema object, such as the prohibition to repeat a keyword inside a schema object, or the interactions between the "properties" and "additionalProperties" keywords. For instance, the following schema\footnote{Example taken from the JSON Schema Test Suite (link available in the PDF).} demands that any properties other than foo and bar must have boolean values.

\[
\{ \\
"properties": {"foo": {}, "bar": {}}, \\
"additionalProperties": {"type": "boolean"} \\
\}
\]

Such features complicate the tasks of reasoning about the language and of writing code for its manipulation.

1.2 Main contributions

**JSON Algebra.** We define a core algebra, which features a subset of JSON Schema assertions. This algebra is minimal, that is, no operator can be defined starting from the others.

**Not elimination.** We show that negation cannot be eliminated from JSON Schema, since there are some assertions whose complement cannot be expressed without negation. We enrich the core algebra with primitive operators to express those missing complementary operators, and we give a not elimination algorithm for the enriched
algebra. To our knowledge, this is the first paper where not elimination is completely defined, with particular regard to the treatment of negation and recursion.

Witness generation. We define an approach for witness generation for the complete JSON Schema language, with the only exception of the uniqueItems operator, hence solving the satisfiability and inclusion problems for this sublanguage.

For space reasons, many details and formal aspects presented in the complete report [3] are not reported here, including the extension to uniqueItems for witness generations. The presentation of several steps (especially for witness generation) is driven/based by/on examples.

Also, we would like to stress that results presented in this paper takes part of research activities [3] that are still in progress. So our main aim here is to present existing results, mainly at the definition and formalisation level of algorithms.

1.3 Paper outline
The rest of the paper is organized as follows. In Section 2 we briefly describe JSON and JSON Schema, while in Section 3 we introduce our algebraic framework. In Section 4, then, we show how algebraic expressions can be rewritten so as to eliminate negation. In Section 5, next, we discuss witness generation, while in Section 6 we analyze some related works. In Section 7, finally, we draw our conclusions.

2 PRELIMINARIES
2.1 JSON data model
JSON values are either basic values, objects, or arrays. Basic values B include the null value, booleans, numbers n, and strings s. Objects represent sets of members, each member being a name-value pair (l, V), and arrays represent ordered sequences of values. We will use J to range over JSON expressions and V to range over the values denoted by such expressions, according to the semantics defined below, but the two notions are so similar that we will often ignore this distinction.

We will only consider here objects without repeated names. In JSON syntax, a name is itself a string, hence it is surrounded by quotes; for the sake of simplicity, we avoid these quotes in our notation, that is, we write {name : "John"} rather than {"name" : "John"}.

\[
J ::= B \mid O \mid A
\]

\[
B ::= \text{null} \mid \text{true} \mid \text{false} \mid n \mid s
\]

\[
O ::= \{l_1 : J_1, \ldots, l_n : J_n\}
\]

\[
A ::= [J_1, \ldots, J_n]
\]

Definition 1 (Value equality and sets of values). In the following we denote value equality with the usual notation $J_1 = J_2$, with the expected meaning on base values, while on Objects we have that $O_1 = O_2$ if and only if $O_1 = \{l_1 : J_1, \ldots, l_n : J_n\}$ and $O_2 = \{l'_{\pi(1)} : J'_{\pi(1)}, \ldots, l'_{\pi(n)} : J'_{\pi(n)}\}$ with $\pi$ a permutation over $I = \{1, \ldots, n\}$ and $J_i = J'_i(\pi(i))$ for each $i \in I$. On arrays we have $A_1 = A_2$ if and only if $A_1 = [J_1, \ldots, J_n]$ and $A_2 = [J'_1, \ldots, J'_n]$ with $J_i = J'_i$ for each $i \in \{1, \ldots, n\}$.

Sets of JSON values are defined accordingly: a set of JSON values is a collection with no repetition with respect to this notion of equality, and two sets are equal when they have the same values with respect to this notion of equality.

2.2 JSON Schema
JSON Schema is a language for defining the structure of JSON documents. It is maintained by the Internet Engineering Task Force IETF [1]. Its latest version has been produced on 2019-09 [9] but is not widely used compared to the intermediate Draft-06.

JSON Schema uses the JSON syntax. Each construct is defined using a JSON object with a set of fields describing assertions relevant for the values being described. Some assertions can be applied to any JSON value type (e.g., type), while others are more specific (e.g., multipleOf that applies to numeric values only). The syntax and semantics of JSON Schema have been formalized in [8] following the specification of Draft-04. We limit ourself to an informal discussion revealing the possible constraints associated to each kind of type:

- when defining a string, it is possible to restrict its length by specifying the minLength and maxLength constraints and to define the pattern that the string should match;
- when defining a number, it is possible to define its range of values by specifying any combination of minimum / exclusiveMinimum and maximum / exclusiveMaximum, and to define whether it should be multipleOf a given number;
- when defining an object, it is possible to define its properties, the type of its additionalProperties and the type of the properties matching a given pattern (i.e. patternProperties). It is also possible to restrict the minimum and maximum number of properties using minProperties and maxProperties, and to indicate which properties are required;
- when defining an array, it is possible to define the type of its items and the type of the additionalItems which were not already defined by items, and to restrict the minimum and maximum size of the array; moreover, it is also possible to enforce unicity of the items using uniqueItems.

JSON Schema allows for combining assertions using standard boolean connectives: not for negation, allOf for conjunction, anyOf for disjunction, and oneOf for exclusive disjunction. Moreover, indicating the set of accepted values can be done using the enum constraint.

3 THE ALGEBRA
We opt for a core algebra that is based on a minimal set of operators expressive enough to capture all JSON Schema constraints, including those of the last 2019 specification [9]. We consider two variants of this algebra: one variant making explicit use of negation and another variant where negation is substituted with a set of operators expressing negation implicitly. The syntax of the two algebras is presented in Figure 1.

In $\text{mulOf}(n)$, $n$ is a number. In $\text{betw}^{\#}_{\pi}$ and in $\text{xBetw}^{\#}_{\pi}$, $n$ is either a number or $-\infty$, $M$ is either a number or $\infty$. In $\text{pre}^l$ and in $\#^l_iS$, $i$ is an integer with $i \geq 0$, while in $i \cdot j : S$, $i$ is an integer with $i \geq 1$. 
\[
T ::= \text{Arr} | \text{Obj} | \text{Null} | \text{Bool} | \text{Str} | \text{Num} \\
r ::= \text{JSON Schema regular expression} \\
b ::= \text{true} \mid \text{false} \\
S ::= \text{ifBoolThen}(b) \mid \text{pattern}(r) \mid \text{betw}^m \mid \text{mulOf}(n) \\
| \text{pro}^j \mid r : S \mid i - j : S \mid \#^jS \mid \text{uniqueteks} \\
| \text{root} x_1 = S_1, \text{def} x_2 = S_2, \ldots, \text{def} x_n = S_n \mid x \mid S_1 \land S_2 \\
\text{either:} \mid \neg S \\
\text{or:} \mid \text{notPattern}(r) \mid x\text{Betw}^m \mid \text{notMulOf}(n) \\
| \text{pattReq}(r : S) \mid \text{repeatedItems} \\
| S_1 \lor S_2 \mid \text{type}(T)
\]

Figure 1: Syntax of the core algebras.

In these three operators, \( j \) is either an integer with the same lower bound as \( i \), or \( \infty \).

This algebra features two possibilities for the negation: the core algebra with \( \neg \), which explicitly uses negation \( \neg S \), and the not-eliminated core algebra, in which \( \neg S \) is substituted by the seven operators of the last three lines.

We show below that negation can express the seven operators of the not-eliminated core algebra, and then we prove the opposite direction, that is, the fact that negation can be eliminated using these operators.

**Remark 1.** In this paper we will assume that JSON Schema regular expressions are indeed regular expressions, hence they are closed under negation and intersection, and these operations are decidable. This is actually good enough in practice, but is not true in general [6].

All operators that are related to one specific type, that is, all operators in the first and second line, have an implicative semantics, where the condition is always: “if the instance belongs to the type associated with this assertion”. We say that they are implicative typed assertions (ITAs).

The meaning of each operator is informally given as follows:

- The assertion \( \text{ifBoolThen}(b) \) means: if the instance is a boolean, then it is \( b \).
- The assertion \( \text{pattern}(r) \) means: if the instance is a string, then it matches \( r \).
- The assertion \( \text{betw}^m \) means: if the instance is a number, then it is included between \( m \) and \( M \), extreme included.
- The assertion \( \text{mulOf}(n) \) means: if the instance is a number, then it is a multiple of \( n \).
- The assertion \( \text{pro}^j \) means: if the instance is an object, then it has at least \( j \) properties and at most \( j \).
- The assertion \( r : S \) is two times implicative, since it means: if the instance is an object and if \( k \) is a name of this object that matches the pattern \( r \), then the value associated with \( k \) satisfies \( S \). Hence, it is satisfied by any instance that is not an object and also by any object where no name matches \( r \).
- The assertion \( \text{pattReq}(r : S) \) means: if the instance is an object, then it contains at least one name that matches \( r \) and whose value matches \( S \).

The semantics of a schema \( S \) is the set of JSON instances \([S]_e\), that satisfy that schema, as specified below; the \( e \) parameter is used to interpret variables, and will be explained later.

In the semantics below, \( L(r) \) denotes the regular language generated by \( r \), while \( |A| \) is the number of top-level members of the object \( A \).

Universal quantification on an empty set is true, and the set \( \{1..0\} \) is empty, so that, for example, both \( i - j : S \) and uniqueteks hold on the empty array.

The semantics of a schema \( S \) is the set of JSON instances \([S]_e\), that satisfy that schema, as specified below; the \( e \) parameter is used to interpret variables, and will be explained later.

In the semantics below, \( L(r) \) denotes the regular language generated by \( r \), while \( |A| \) is the number of top-level members of the object \( A \).

Universal quantification on an empty set is true, and the set \( \{1..0\} \) is empty, so that, for example, both \( i - j : S \) and uniqueteks hold on the empty array.

JSON Schema specification says that the “result” of a variable \( x \) is the “result” of the referenced schema, which may be formalized as follows: an equation system is equivalent to the first element, evaluated in an environment \( e \) where every variable is associated to its definition. When a variable is met, it is substituted by its definition in the current environment. We assume that in \( e, x \rightarrow S \) the new binding \( x \rightarrow S \) hides any previous binding for \( x \), so that
As we said before, the seven operators in the last group are re-
we support the usual rule that variables in an inner scope hid-
compatible with that equation, or that we have none. This is a clas-
is not an inductive definition, and actually it may be the case that
three do not correspond to JSON Schema operators, but can sti-
expressed in JSON Schema, through the negation of
The operator type(T) can be expressed using negation as fol-
As we said before, the seven operators in the last group are re-
dundant in presence of negation. type(T) corresponds to the JSON
operator type(T) → ⇒ for readability. In order to stay in the core
exception is the Null type, since we have no typed operators for
null, hence we take the complement of the other five types.

\[\text{type(Str)} \quad = \quad \neg(\text{pattern}(\#) \land \text{pattern(.)})\]
\[\text{type(Num)} \quad = \quad \neg(\text{betw}_{0} \land \text{betw}_{1})\]
\[\text{type(Bool)} \quad = \quad \neg(\text{ifThen}((\text{true}) \land \text{ifThen}((\text{false})))\]
\[\text{type(Obj)} \quad = \quad \neg(\text{pattern} \land \text{pattern}^{1})\]
\[\text{type(Arr)} \quad = \quad (\neg(1 - 1 : f \land \#_{1}^{o}t)\]
\[\text{type(NULL)} \quad = \quad \neg\text{type(Str)} \land \neg\text{type(Num)} \land \neg\text{type(Bool)} \land \neg\text{type(Obj)} \land \neg\text{type(Arr)}\]

The other six operators can be expressed as follows, where we use
the type operator and ⇒ for readability. In order to stay in the core
The definition of \text{pattReq}(r : S) deserves an explanation. The
implies \text{type(Obj)} \quad ⇒ \quad \ldots just describes its implicative nature
in any instance that is not an object. Since \text{r} : \neg\text{S}
means that, if a name matching \text{r} is present, then its value satis-
\neg\text{S}, any instance that does not satisfy \text{r} : \neg\text{S} must possess a
member name that matches \text{r} and whose value does not satisfy \neg\text{S},
that is, satisfies \text{S}. Hence, we exploit here the fact that the nega-
tion of an implication forces the hypothesis to hold.

\[\text{notPattern}(r) \quad = \quad \text{type(Str)} \Rightarrow \neg\text{pattern}(r)\]
\[\text{xBetw}^{M} \quad = \quad \text{type(Num)} \Rightarrow (\neg\text{betw}_{0} \land \neg\text{betw}_{1})\]
\[\text{notMulOf}(n) \quad = \quad \text{type(Num)} \Rightarrow \neg\text{mulOf}(n)\]
\[\text{pattReq}(r : S) \quad = \quad \text{type(Obj)} \Rightarrow (\neg(r : \neg\text{S})\]
\[\text{repeatedItems} \quad = \quad \text{type(Arr)} \Rightarrow \neg\text{uniquetem}s\]
\[\text{S} \land \text{S} \quad = \quad (\neg\text{S} \land \neg\text{S})\]

\[\text{notPattern}(r) \quad = \quad \text{type(Str)} \Rightarrow \neg\text{pattern}(r)\]
\[\text{xBetw}^{M} \quad = \quad \text{type(Num)} \Rightarrow (\neg\text{betw}_{0} \land \neg\text{betw}_{1})\]
\[\text{notMulOf}(n) \quad = \quad \text{type(Num)} \Rightarrow \neg\text{mulOf}(n)\]
\[\text{pattReq}(r : S) \quad = \quad \text{type(Obj)} \Rightarrow (\neg(r : \neg\text{S})\]
\[\text{repeatedItems} \quad = \quad \text{type(Arr)} \Rightarrow \neg\text{uniquetem}s\]
\[\text{S} \land \text{S} \quad = \quad (\neg\text{S} \land \neg\text{S})\]

\[\text{notPattern}(r) \quad = \quad \text{type(Str)} \Rightarrow \neg\text{pattern}(r)\]
\[\text{xBetw}^{M} \quad = \quad \text{type(Num)} \Rightarrow (\neg\text{betw}_{0} \land \neg\text{betw}_{1})\]
\[\text{notMulOf}(n) \quad = \quad \text{type(Num)} \Rightarrow \neg\text{mulOf}(n)\]
\[\text{pattReq}(r : S) \quad = \quad \text{type(Obj)} \Rightarrow (\neg(r : \neg\text{S})\]
\[\text{repeatedItems} \quad = \quad \text{type(Arr)} \Rightarrow \neg\text{uniquetem}s\]
\[\text{S} \land \text{S} \quad = \quad (\neg\text{S} \land \neg\text{S})\]

3.1 Semantics of the negated operators

As we said before, the seven operators in the last group are re-
dundant in presence of negation. type(T) corresponds to the JSON
operator type(T) → ⇒ for readability. In order to stay in the core
The definition of \text{pattReq}(r : S) deserves an explanation. The
implies \text{type(Obj)} \quad ⇒ \quad \ldots just describes its implicative nature
in any instance that is not an object. Since \text{r} : \neg\text{S}
means that, if a name matching \text{r} is present, then its value satis-
\neg\text{S}, any instance that does not satisfy \text{r} : \neg\text{S} must possess a
member name that matches \text{r} and whose value does not satisfy \neg\text{S},
that is, satisfies \text{S}. Hence, we exploit here the fact that the nega-
tion of an implication forces the hypothesis to hold.

Remark 2. The negation of the operator \text{i} - \text{i} : S can be ex-
pressed with no need of a specific negative operator, since it can be ex-
pressed using the same operator plus \text{#}_{\text{n}}^{o}t. The negation of \text{i} - \text{i} : S can also be expressed with no need of a specific negative operator,
since it can be expressed using an exponential number of \text{i} - \text{i} : S and \text{#}_{\text{n}}^{o}t, plus the operator \text{#}_{\text{n}}^{o}S, as described in Section 4.2.

Without the \text{#}_{\text{n}}^{o}S operator, which is our representation of the com-
bination of contains, \text{minContains} and \text{maxContains} (operators
which were introduced in JSON Schema 2019), in order to express the
negation of \text{i} - \text{i} : S we would need at least an operator \exists(\text{i} - \text{j} : S)
that specifies that at least one element between \text{i} and \text{j} matches \text{S}. This operator would be strictly less expressive than the \text{#}_{\text{n}}^{o}S operator,
but this is explained in the next remark.

Remark 3. While \text{#}_{\text{n}}^{o}S can be immediately translated as type(Arr) ⇒
(\neg(1 - \text{i} : \neg\text{S}), the assertions \text{#}_{\text{n}}^{\text{min}}S, \text{min} > 1 and \text{#}_{\text{n}}^{\text{max}}S, \text{Min} <
Max < \infty cannot be expressed for any non trivial S without the \#
operator.
3.2 Representing definitions and references

JSON Schema defines a $\text{Sref}: \text{path}$ operator that allows any sub-schema of the current schema to be referenced, as well as any sub-schema of a different schema that is reachable through a URL, hence implementing a powerful form of mutual recursion. The path $\text{path}$ may navigate through the nodes of a schema document by traversing its structure, or may retrieve a subdocument on the basis of a special id, Sid, or $\text{Sanchor}$ member ($\text{Sanchor}$ has been added in Version 2019/09), which can be used to associate a name to the surrounding schema object. Despite this richness of choices, in most situations, according to our collection of JSON schemas, the sub-schemas that are referred are either the entire schema or those that are collected inside the value of a top-level definitions member. Hence, we defined a referencing mechanism that is powerful enough to translate every collection of JSON schemas, but that privileges a direct translation of the most commonly used mechanisms.

In our referencing mechanism, a schema may have the following structure, where the schema $S_1$ associated to the root definition is the one that is used to validate the instance, and every variable $x_i$ is bound to $S_i$.

$$\text{root} x_1 = S_1, \text{def} x_2 = S_2, \ldots, \text{def} x_n = S_n$$

Such a schema corresponds to a JSON schema whose root contains $S_1$ plus a definitions member, which in turn contains the definitions $\text{def} x_2 = S_2, \ldots, \text{def} x_n = S_n$. In JSON Schema the entire schema can be denoted as $\#$, but we preferred an explicit naming mechanism $\text{root} x_1 = S_1$ for uniformity.

For example, this JSON Schema document:

$$\{ a_1 : S_1, \ldots, a_n : S_n, \text{definitions} : \{ x_1 : S'_1, \ldots, x_m : S'_m \} \}$$

Corresponds to the following expression, where $\text{S}$ is the translation of $S$:

$$\text{root} r = \{ a_1 : S_1, \ldots, a_n : S_n \}, \text{def} x_1 = S'_1, \ldots, \text{def} x_m = S'_m$$

This mechanism is as expressive as the combination of all JSON Schema mechanisms, at the price of some code duplication. In order to translate any JSON document that uses references, in our implementation we first collect all paths used in any $\text{Sref}$ $\text{path}$ assertion. Whenever $\text{path}$ is neither $\#$ nor definitions/k for some $k$, we retrieve the referred subschema and copy it inside the definitions member where we give it a name, and we substitute all occurrences of $\text{Sref}$ $\text{path}$ with $\text{Sref}$ $\text{definitions/name}$, until we reach the shape (1) above. In principle this may increase the size of the schema from $n$ to $n^2$, in case we have paths that refer inside the object that is referenced by another path, in practice we observed a factor of 2 in the worst cases. When we have a collection of documents that refer one inside the other, we first merge the documents together and then apply the same mechanism. It would not be difficult to extend the naming mechanism in order to avoid merging the documents, but we consider this extension out of the scope of this paper.

In our syntax, the root $x_1 = S_1, \text{def} x_2 = S_2, \ldots, \text{def} x_n = S_n$ construct is a first class assertion, hence can be nested. We defined the syntax this way just for uniformity, but in practice we only use schemas where this construct is only used at the outermost level.

3.3 From JSON Schema to the algebra

Our algebra expresses all assertions of Draft 6. The translation rules from JSON Schema to our algebra are provided in Table 1 by omitting symmetric cases (e.g. “maximum”: $M$, “exclusiveMaximum”: M, etc) which can be easily guessed.

| JSON Schema | Algebra |
|-------------|---------|
| “minimum”: $m$ | $\text{betw}_{m}^{\infty}$ |
| “exclusiveMinimum”: $m$ | $\text{xBetw}_{m}^{\infty}$ |
| “multipleOf”: $n$ | $\text{mulOf}(n)$ |
| “minLength”: $m$ | $\text{pattern}(\{, \text{m}, \}$ |
| “pattern”: r | $\text{pattern}(r)$ |
| “uniqueItems”: r | $\text{uniques}(m)$ |
| “minItems”: m | $\text{m}^{\infty}$ |
| “contains”: $S$, “minContains”: m | $\text{items}(S_1, \ldots S_n; \{S\})$ |
| “items”: $\{S_1, \ldots, S_n\}$ | $\text{items}()$ |
| “additionalItems”: $S'$ | $\text{items}(S_1, \ldots, S_n; \{S\})$ |
| “additionalProperties”: $S'$ | $\text{items}()$ |
| “properties”: $\{ i=1..n, k_i : S_i \}$ | $\text{type}(\text{Obj})$ |
| “patternProperties”: | $\{ i=1..m, r_i : PS_i \}$ |
| “additionalProperties”: S | $\text{type}(\text{Obj})$ |

Table 1: Translation rules for JSON Schema
4 NEGATION ELIMINATION

We use not-elimination to indicate the property of a logic to express the negation of every formula with no use of the negation operator.

JSON Schema does not enjoy not-elimination, since it contains some assertions whose negation cannot be expressed without a negation operator, such as uniqueItems. Our algebra with the implicit negated operators is sufficient to rewrite any explicitly negated operator by eliminating the \( \neg \).

We present an algorithm to push negation down the syntax tree of any schema. The algorithm proceeds in three phases:

1. Not-completion of variables: for every variable \( x_n = S_n \) we define a corresponding definition \( \text{not}_x \equiv \neg S_n \).
2. Not-rewriting: we rewrite every expression \( \neg S \) into an expression where the negation has been pushed inside.
3. We first present not-completion and not-rewriting for the case, and we then present the rest of not-rewriting.

4.1 Not-completion of variables

Not-completion of variables is the operation that adds a variable \( \text{not}_x \) for every variable \( x \) as follows:

\[
\text{not-completion}(\text{root}_0, \ldots, \text{root}_n) = \text{root}_0, \ldots, \text{root}_n, \text{not}_x = \neg S_n
\]

After not-completion, every variable has a complement variable defined in the obvious way: \( \text{co}(x_i) = \text{not}_x \) and \( \text{co}(\text{not}_{x_i}) = x_i \).

The complement \( \text{co}(x) \) will later be used for not-elimination.

4.2 Inversion of items(\( )\)

The inversion of items(\( S_1 \cdots S_n; S \)) is the most complex case of not elimination. According to its semantics, only a non-empty array may not satisfy that assertion. For instance, the empty array as well as the string "foo" both satisfy both items(; t) and items(; f), while any array with length 1 or more would violate both types.

More generally, we have the following formula:

\[
\text{items}(S_1 \cdots S_n; S) = \text{type}(\text{Arr}) \land (S_1 \lor \cdots \lor S_n \lor S_{n+1})
\]

that expresses the fact that an array \( [J_1, \ldots, J_n] \) may not satisfy \( \text{items}(S_1 \cdots S_n; S) \) in one of the following \( n + 1 \) ways:

- The array has at least \( 1 \leq i \leq n \) elements and element \( J_i \) does not satisfy \( S_i \):
  \[ S_i = \#_{t_i}^n \land \text{items}(t_1, \ldots, t_{i-1}, \neg S_i; t) \]
- The array has at least \( n+1 \) elements and one element \( J_{n+1} \), with \( i > 0 \), does not satisfy the tail schema \( S \). This case \( N_{n+1} \) is the most complex one and deserves some preliminary discussion.

Concerning \( N_{n+1} \), we consider first the following special cases:

1. The most common case is when \( n = 0 \). In this case one single non-S element is enough to violate \( \text{items}(; S) \), hence we have \( N_{n+1} = \#_1 \neg S \), and the initial sequence \( S_1 \land \cdots \land S_n \) is empty.\(^2\)
2. The second most common case is the one with \( n > 0 \) and \( S = f \). In this case, we violate the tail condition whenever the array has at least \( n + 1 \) elements, hence we have that \( N_{n+1} = \#_{n+1} t \).
3. The third most common case is \( n > 0 \) and \( S = t \). In this case, the tail condition cannot be violated, hence \( N_{n+1} = f \).
4. A last special case is that where the array schema has length 1, that is, \( \text{items}(S_1; S) \), and \( S \) is not trivial. In this case we distinguish two possibilities for the first element of the array, and we define

\[
N_{n+1} = \text{items}(S_1; t) \land \#_1 \neg S \lor \text{items}(S; t) \land \#_2 \neg S
\]

Observe that in the first three cases we can express negation using the operators \#_1 S and \#_2 S that have already been introduced in Draft 06. In the fourth case, however, we need the operator \#_2 S that has been introduced in Version 2019/09.

We have examined a set of ca. 11,000 different schemas, which contain a total of 33,015 instances of \( \text{items}(S_1 \cdots S_n; S) \). Almost all of those instances fall in cases 1 (97%) and 2 (2.5%), but we have 121 examples of 3 (0.4%), while case 4 covers seven cases. We found only one schema that falls out of this classification, since it has 2 item types and a non-trivial \( S \) types.

While these four cases are sufficient in practice, we present here a general formula that is applicable to every case.

Given \( \text{items}(S_1 \cdots S_n; S) \), let us divide any array that does not satisfy it in two parts: the head up to position \( n \), and the tail (which may be empty) after position \( n \). The formula \( N_{n+1} \) specifies that the tail is not empty and contains one element that violates \( S \), that is, we have some tail-non-\( S \)'s. We cannot directly express this, but we can reason by cases on the positions of the elements in the head that violate \( S \) — we call these elements the head-non-\( S \)'s. An array with some tail-non-\( S \) can be described as an array that has \( k \) head-non-\( S \)'s and satisfies \( \#_{k+1} \neg S \). Hence, the formula \( N_{n+1} \) will enumerate all possible distributions of the head-non-\( S \)'s, and ask that one of these distributions holds, with \( k \) head-non-\( S \) and with \( \#_{k+1} \neg S \), which together imply that one tail-non-\( S \) exists.

In order to enumerate all distributions we consider, for the given \( n \), the set of all bitmap's of length \( n \), where a bitmap is a function from \( 1..n \) to \( \{0, 1\} \), and, for a bitmap \( bm \), we use \( \text{sum}(bm) \) for the number of its 1's, that is, for \( \Sigma_{i=1}^{\text{sum}(bm)}(i) \).

We define a function \( \text{NotIf}(\text{Bit}, S) \) such that

\[
\begin{align*}
\text{NotIf}(0, S) &= S \\
\text{NotIf}(1, S) &= \neg S
\end{align*}
\]

Every bitmap \( bm \) will correspond to a possible distribution of head-non-\( S \)'s, as follows that is described by the following schema:

\[
\text{items}(\text{NotIf}(bm(1), S), \cdots \text{NotIf}(bm(n), S); t) \land \#_{\text{sum}(bm)+1} \neg S
\]

The schema is satisfied by any array where the 1's of \( bm \) indicate the positions of the head-non-\( S \)'s, and where at least \( \text{sum}(bm) \) + 1 elements are non-\( S \)'s. Hence any array \( [J_1, \ldots, J_n] \) that satisfies that schema has \( \text{sum}(bm) \) head-non-\( S \)'s and some tail-non-\( S \)'s, and, vice versa, for every array \( A \) that has some tail-non-\( S \)'s, there exists a bitmap \( bm \) such that \( A \) satisfies the corresponding schema.

Hence, \( N_{n+1} \) can be defined by the following disjunction with \( 2^n \) cases:

\[
\bigvee_{bm \in \{0, 1\}^n} \text{items}(\text{NotIf}(bm(1), S), \cdots \text{NotIf}(bm(n), S); t) \land \neg S
\]

\(^2\)This case arises from the translation of \( \text{items} \): \( S \) where \( S \) is not an array and also in those rare situations where \( \text{additionalItems} \) is present and \( S \) is absent
where
\[ R = \#_{\text{sum}(bm)+1} \sim S \]

To sum up we have the four formulas described in Figure 3, where the last one subsumes the first three cases.

### 4.3 Not rewriting

We show now how to push negation down any algebraic expression.

Not-elimination may generate bounds that are trivial or unsatisfiable, hence we should apply bound-normalization rules, such as the following ones. We report the case for betw, \*x. Similar rules exist for betw, \*x.

\[
\begin{align*}
\text{pro}_n &= t \\
\text{pro}_m &= \text{type(Obj)} \Rightarrow f \quad \text{if } n > m \\
\text{pro}_\infty &= \text{type(Obj)} \Rightarrow f
\end{align*}
\]

Not elimination is defined as follows. We do not define the cases for the negative operators (notMulOf etc.) since they follow immediately from their definitions.

\[
\begin{align*}
\neg(S_1 \land S_2) &= (\neg S_1) \lor (\neg S_2) \\
\neg(\neg S) &= S \\
\neg(\text{type}(T)) &= \forall (\text{type}(T') \mid T' \neq T) \\
\neg(\text{pattern}(r)) &= \text{type(Str)} \land \neg\text{Pattern}(r) \\
\neg(\text{betw}_{m}) &= \text{type(Num)} \land (\text{xbetw}_{m,\infty} \lor \text{xbetw}_{m}) \\
\neg(\text{ifBoolThen}(\text{false})) &= \text{type(Boolean)} \land \text{ifBoolThen}(\text{true}) \\
\neg(\text{ifBoolThen}(\text{true})) &= \text{type(Boolean)} \land \text{ifBoolThen}(\text{false}) \\
\neg(\text{mulOf}(n)) &= \text{type(Num)} \land \neg\text{MulOf}(n) \\
\neg(\text{r : S}) &= \text{type(Obj)} \land \text{pattReq}(r : \neg S) \\
\neg(\text{pro}_I) &= \text{type(Obj)} \land (\text{pro}_I \lor \text{pro}_J) \\
\neg(\text{items}(S_1 \cdots S_n; S_{n+1})) &= \text{See Section } 4.2 \\
\neg(\#^I S) &= \text{type(Arr)} \land (\#^I S \lor \#^J S) \\
\neg(\text{uniqueitems}) &= \text{type(Arr)} \land \text{repeatedItems} \\
\neg(x) &= \text{co}(x) \\
\neg(\text{root } x_0 = S_0, \ldots, \text{def } x_0 = S_0, \ldots, \text{def } x_n = S_n, \text{def } \neg \text{not } x_0 = S_{n+1}, \ldots, \text{root } \neg \text{not } x_0 = S_{n+1}, \ldots, \text{def } \neg \text{not } x_n = S_{2n}) &= \text{def } \neg \text{not } x_n = S_{2n} \\
\end{align*}
\]

**Example 1.** For example, assume the following definition.

\[ \text{root } x = \{ a : \text{co}(x) \} \]

where \( \text{co}(x) \) is the complement variable. This is the effect of completion.

\[ \text{root } x = \{ a : \text{co}(x) \}, \text{def } \neg \text{not } x = \neg\{ a : \text{co}(x) \} \]

And this is how not-elimination may now proceed.

\[
\begin{align*}
\text{root } x = (a : \text{co}(x)), \text{ def } \neg \text{not } x = \neg(a : \text{co}(x)) \rightarrow \\
\text{root } x = (a : \text{not } x), \text{ def } \neg \text{not } x = (\text{type(Obj)} \land \text{req}(a) \land a : \neg \text{co}(x)) \rightarrow \\
\text{root } x = (a : \text{not } x), \text{ def } \neg \text{not } x = (\text{type(Obj)} \land \text{req}(a) \land a : x)
\end{align*}
\]

where \( \text{req}(a) \) denotes the fact that \( a \) is required, i.e.

\[ \text{req}(a) = \text{type(Obj)} \Rightarrow \neg(a : f) \]

We can now substitute \( \text{not } x \) with its definition. The definition that we get is not much clearer: if the value is an object with an a member, then the value of that member must be an object with an a member, whose value satisfies the same specification.

\[ \text{root } x = (a : (\text{type(Obj)} \land \text{req}(a) \land a : x)) \]

Some examples of values that match that schema:

\[ \{ a : \{ a : 1 \} \}, \{ a : \{ a : \{ a : 1 \} \} \} \]

### 5 WITNESS GENERATION

#### 5.1 The structure of the algorithm

In order to prove satisfiability, or emptiness, of a schema, we try and generate a witness for the schema. We examine all the possible ways to generate the value hence, if generation is not possible, the schema is not satisfiable.

The basic idea is as follows. Assume, by induction, that you have an algorithm to generate a witness for any assertion \( S \) of size up to \( n \). In order to generate a witness for an ITE of size \( n + 1 \) such as \( \text{pattReq}(r : S) \) one will generate a witness for \( S \) and use it to build an object with a member that matches \( r \) whose value is that witness, and the same approach can be followed for the other ITEs. For the boolean operator \( S_1 \lor S_2 \), we recursively generate witnesses of \( S_1 \) and of \( S_2 \). Negation and conjunction are a problem: there is no way to generate a witness for \( \neg S \) starting from a witness for \( S \) and, given a witness for \( S_1 \), if this is not a witness for \( S_1 \land S_2 \), we may need to try infinitely many others before finding one that satisfies \( S_2 \). Hence, we first eliminate \( \neg \) using not elimination, then we bring all definitions of variables into DNF so that conjunctions are confined to the leaves of the syntax tree, and finally we make conjunction harmless by a technique that is called canonicalization, which is based on the combination of all ITEs having to do with the same type, and which is presented below. After ITEs and boolean operators, we are left with recursive variables. We deal with them by adopting a bottom-up iterative evaluation that mimics the fix-point semantics. We are going to transform this idea into an algorithm.

For the sake of presentation, we will only present our approach by focusing on schemas that do not feature uniquetemms or repeatedItems.

The algorithm consists in six steps.

1. **Translation from JSON Schema to the core algebra and not-elimination.**
2. **Canonicalization:** we split every conjunction into a disjunction of typed groups, where a typed group is a conjunction of typed assertions that contains one type(T) assertion, and where T is the type of all the ITEs in the group.
3. **Variable normalization:** we rewrite the definitions of all variables so that every variable only appears as an argument of a typed operator, as in \( \text{pattReq}(a : x) \), and all \( S \) arguments of typed operator are variables. In this way, no boolean operator has a variable as argument.
4. **Reduction to DNF:** we transform each boolean expression into a Disjunctive Normal Form, that is, into a disjunction of typed groups.
we describe the six steps of the algorithm.

5. Object and array preparation: we rewrite object and array

groups into a form that will simplify recursive generation of

that may need during the next phase.

5.2 Translation to JSON Schema and

Not-elimination

We first translate JSON Schema to the core algebra, but we keep

the items() notation, rather than the i - j : S notation of the core,

since it is notionally more convenient. Not-elimination is then

performed as in Section 4.

5.3 Canonicalization

Canonicalization is a process defined along the lines of [7]. We rely

on the the new notation {S_1, . . . , S_k}, which we call a group, for rep-

resenting the n-ary conjunction of schemas, and we define canoni-

calization of a not-eliminated expression, which may include both

binary conjunctions and group conjunctions, as the following pro-

cess.

1. We first flatten any tree of nested conjunctions, of both forms,

into a single group.

2. For every group (T_1, . . . , T_n, S_1, . . . , S_m), where T_1, . . . , T_n

are the typed assertions and S_1, . . . , S_m the boolean, definition,

or variable assertions, we rewrite it as (T_1, . . . , T_n) Λ

S_1 ∧ . . . ∧ S_m. Thanks to step (1), no S_i is a

conjunct.
(3) For every group \( G \) where at least one type \( T \) assertion is present, we apply and-merging until the group only contains one type \( T \) and a set of ITEs with a compatible type, or collapses to \( f \).

(4) For every group \( G \) where no type \( T \) is present, we rewrite it as the disjunction of six groups, each one starting with a different type \( T \) assertion, one for each core algebra type, and continuing with the subset of \( G \) whose type is \( T \). The group is formed even if this subset is empty.

As an example, the following expression:

\[
(\text{mulOf}(3) \land \text{len}_0^3) \lor ((\text{type}(\text{Num}), \text{mulOf}(2), x \land \text{pattern}(a^\ast)))
\]

is rewritten as follows.

1. \( \{\text{mulOf}(3), \text{len}_0^3\} \lor \{\text{type}(\text{Num}), \text{mulOf}(2), x, \text{pattern}(a^\ast)\} \)
2. \( \{\text{mulOf}(3), \text{len}_0^3\} \lor \{\{\text{type}(\text{Num}), \text{mulOf}(2), \text{pattern}(a^\ast)\} \land x\} \)
3. \( \{\text{type}(\text{Null})\} \lor \{\text{type}(\text{Bool})\} \lor \{\{\text{type}(\text{Num}), \text{mulOf}(3)\} \land x\} \)
4. \( \{\text{type}(\text{Null})\} \lor \{\text{type}(\text{Bool})\} \lor \{\{\text{type}(\text{Num}), \text{mulOf}(3)\} \land x\} \)

We say that a group that contains type \( T \) and a set of ITEs of type \( T \) is a typed group of type \( T \). At the end of the canonicalization phase, any expression has been rewritten as a boolean combination of variables, definitions, and typed groups. For reasons of space, hereafter we will abbreviate a type assertion type(\text{Num}) in a group with \text{Num}, and similarly for the other types, so that we use \text{Null}, \text{Bool}, \text{Num}, \text{Str}, \text{Obj}, \text{Arr} to indicate the six core types. We will also use concatenation to indicate disjunction, so that \{\text{Null} \lor \text{Bool} \lor \text{Num}\} = \{\text{Null} \lor \text{Bool} \lor \text{Num}\}, and we use \text{X} to indicate complement with respect to \text{Null}, \text{Bool}, \text{Num}, \text{Str}, \text{Obj}, \text{Arr}, so that \{\text{Null}\} is the same as \{\text{Null} \lor \text{Bool} \lor \text{Num} \lor \text{Arr}\}. In this way, the expression above can be written as follows.

\[
\{\text{Null}\} \lor \{\text{Bool}\} \lor \{\text{Num}\} \lor \{\text{Str}, \text{len}_0^3\} \lor \{\{\text{type}(\text{Num}), \text{mulOf}(2)\} \land x\} \]

\( = \{\text{Num} \lor \text{Str}\} \lor \{\text{Num}, \text{mulOf}(3)\} \lor \{\text{Str}, \text{len}_0^3\} \lor \{\{\text{Num}, \text{mulOf}(2)\} \land x\}
\]

### 5.4 Variable normalization

Variable normalization is used to reach a form where no boolean operator has a variable as an argument, and all non-boolean schema operators only have variables as argument. This will be crucial for the witness generation phase (Section 5.7). It proceeds in two steps, separation and expansion.

In the separation phase, for every typed operator that has a sub-schema \( S \) in its syntax, such as \#_1^\ast S, when \( S \) is not a variable we add to the global definition a new variable definition \( \text{def } x = S \), and we substitute \( S \) with \( x \).\(^3\) For every variable \( \text{def } x = S \) that we define, we must also define its complement \( \text{def } not \_x = \neg S \), and perform not-elimination and canonicalization on \( \neg S \).

---

\(^3\)In the implementation we make an exception for \( t \) and \( f \), which can appear wherever a variable appears.

In the expansion phase, for any clause \( \text{def } x = S \), we substitute any unguarded occurrence of any variable in \( S \), where an occurrence is guarded if it occurs below a typed operator, with its definition. This process is guaranteed to stop since we do not allow unguarded cyclic definition.

For example, consider the following definitions.

\[
\text{root } x = (\{\text{Arr}, \text{items}(x \land y; t)\} \land y) \lor \{\text{Bool}\}
\]
\[
\text{def } y = (\{\text{Arr}, \text{items}(\{\text{Num}\} \lor \neg x; t)\})
\]

By not-elimination and canonicalization, we obtain the following schema.

\[
\text{root } x = (\{\text{Arr}, \text{items}(x \land y; t)\} \land y) \lor \{\text{Bool}\}
\]
\[
\text{def } y = (\{\text{Arr}, \text{items}(\{\text{Num}\} \lor \neg x; t)\})
\]

\[
\text{def } not \_x = (\{\text{Arr}, \text{items}(\not x \lor \not y; t), \#_1^\ast t\} \lor \{\text{Arr}\}) \land \{\text{Bool}\}
\]
\[
\text{def } not \_y = (\{\text{Arr}, \text{items}(\{\text{Num}\} \lor \neg x; t), \#_1^\ast t\} \lor \{\text{Arr}\})
\]

The typed operator items(); is applied to a non-variable sub-schema \( x \land y \) in items(\( x \land y; t \)) in the first line, and similarly in the other three lines. Hence, during the separation phase we define four new variables in order to separate the non-variant arguments from their guarded operators. We should define four more variables in order to complete the generated equation system, but this is not necessary in this case, since, for example, \( itx \) corresponds already to the negation of \( itx \). Separation produces the following schema.

\[
\text{root } x = (\{\text{Arr}, \text{items}(itx; \neg t)\} \land y) \lor \{\text{Bool}\}
\]
\[
\text{def } itx = x \land y
\]
\[
\text{def } y = (\{\text{Arr}, \text{items}(ity; t)\})
\]
\[
\text{def } ity = (\{\text{Num}\} \lor \neg x) \lor \{\text{Arr}\}
\]
\[
\text{def } not \_x = (\{\text{Arr}, \text{items}(itx; \neg t), \#_1^\ast t\} \lor \{\text{Arr}\}) \lor \{\text{Num}\} \land \not x
\]
\[
\text{def } not \_y = (\{\text{Arr}, \text{items}(ity; \neg t), \#_1^\ast t\} \lor \{\text{Arr}\}) \lor \{\text{Num}\} \land \not y
\]

Now, we must expand all the unguarded variables. The unguarded variables are those that are underlined below.

\[
\text{root } x = (\{\text{Arr}, \text{items}(itx; \neg t)\} \land y) \lor \{\text{Bool}\}
\]
\[
\text{def } itx = x \land y
\]
\[
\text{def } y = (\{\text{Arr}, \text{items}(ity; t)\})
\]
\[
\text{def } ity = (\{\text{Num}\} \lor \neg x) \lor \{\text{Arr}\}
\]
\[
\text{def } not \_x = (\{\text{Arr}, \text{items}(itx; \neg t), \#_1^\ast t\} \lor \{\text{Arr}\}) \lor \{\text{Num}\} \land \not x
\]
\[
\text{def } not \_y = (\{\text{Arr}, \text{items}(ity; \neg t), \#_1^\ast t\} \lor \{\text{Arr}\}) \lor \{\text{Num}\} \land \not y
\]

One can observe that, because of mutual recursion, a process of iterated variable expansion may never stop. However, by the assumption of guarded recursion, the dependencies between the unguarded occurrences are not cyclic. In this case, the longest dependency path is \( itx \) depends on \( x \) that depends on \( y \), and similarly for \( ity, itnx \) and \( itny \).
At this point, variable expansions steps produce the following set.

\[
\text{root } x = (\{\text{Arr}, \text{items}(\text{ix}x; t)\} \land \{\text{Arr}, \text{items}(\text{it}y; t)\}) \lor \{\text{Bool}\}
\]

\[
\text{def } \text{it}x = (\{\text{Arr}, \text{items}(\text{ix}x; t)\} \land \{\text{Arr}, \text{items}(\text{it}y; t)\}) \lor \{\text{Bool}\}
\]

\[
\text{def } \text{it}y = \{\text{Arr}, \text{items}(\text{it}y; t)\}
\]

\[
\text{def } \text{not}_x = (\{\text{Arr}, \text{items}(\text{it}x; t)\} \land \{\text{Arr}, \text{items}(\text{it}y; t)\}) \lor \{\text{Bool}\}
\]

After expansion, variables are only found in guarded positions and their definition is hence a boolean combination of typed groups, hence we are now ready to transform the schema into a Disjunctive Normal Form.

5.5 Transformation in Disjunctive Normal Form

To reach a Disjunctive Normal Form (DNF), we repeatedly apply the following rule, and we apply and-merging and basic boolean reductions to any new conjunction that is generated.

\[
(S_1 \lor \ldots \lor S_n) = \bigvee_{1 \leq i \leq n} (S_i)
\]

For example, we can apply DNF to the last definition of the previous example, followed by and-merging and then by an extended-and-merging.

\[
\text{DNF} : = (\{\text{Num}\} \land (\{\text{Arr}, \text{items}(\text{ix}x; t)\} \land \{\text{Arr}, \text{items}(\text{it}y; t)\} \lor \{\text{Bool}\}))
\]

\[
\text{AndM} : = (\{\text{Num}\} \lor (\{\text{Arr}, \text{items}(\text{ix}x; t)\} \land \{\text{Arr}, \text{items}(\text{it}y; t)\}))
\]

\[
\text{AndM} : = (\{\text{Arr}, \text{items}(\text{ix}x \land \text{it}y; t)\}) \lor \{\text{Bool}\}
\]

Canonicalization ensures that all groups are typed groups, hence that they contain a type(T) assertion. Guarded expressions separation and variable expansion ensure that all guarded schemas are variables (separation invariant) and that no boolean expressions involve variables (expansion invariant). Reduction in DNF preserves these invariants. Unfortunately, the and-merging phase inserts conjunctions of variables in guarded positions, which breaks the separation and expansion invariants. This is not a problem, since object and array preparation have the same effect, hence, after the next step, we have to go back to variable normalization, canonicalization, preparation, until convergence.

Observe that all the arguments of any disjunction left after canonicalization, variable normalization and reduction to DNF are typed groups. Now we need to prepare these typed groups for the witness generation phase.

5.6 The structure of the preparation phase

Before starting witness generation, we must bring the object groups (the typed groups with type Obj) and the array groups in a simplified form where the interactions between the different components are explicit. This is described in detail in the next two sections.

5.6.1 Object group preparation. Object type preparation has similarities with and-merging, but is different. And-merging is an optimization, which performs some easy and optional rewritings that reduce the size of the expression. Type preparation is mandatory, since it provides the type with the completeness and no-overlapping invariants that are needed for witness generation. The similarity between the two phases derives from the fact that they are based on similar equivalences, since they are both applying semantics-preserving transformations to a conjunction of typed assertions.

For this phase and the next one, we introduce a new operator

\[
\text{orPattReq}(r_1 : S_1, \ldots, r_n : S_n)
\]

that represents the disjunction

\[
\text{pattReq}(r_1 : S_1) \lor \ldots \lor \text{pattReq}(r_n : S_n)
\]

and describes \(n\) possible ways of satisfying a single \(\text{pattReq}\) constraints.

The aim of object preparation is to bring object groups into a form where one can easily enumerate all possible ways of satisfying all different assertions, by making all the interactions between different assertions explicit.

More precisely, we rewrite each object group into a constraining set

\[
P_1 : x_1, \ldots, p_n : x_n
\]

and a requiring set

\[
\text{orPattReq}(r_1^1 : y_1^1, \ldots, r_n^1 : y_n^1), \ldots, \\
\text{orPattReq}(r_1^m : y_1^m, \ldots, r_n^m : y_n^m)
\]

which satisfy the following four properties. Hereafter we say that two patterns \(r_1\) and \(r_2\) have a trivial intersection when either \(r_1 \land r_2 = \text{f}\) or \(r_1 = r_2\), so that they are either disjoint or equivalent.

1. Constraint partition: the patterns \(p_i\) in the constraining part are mutually disjoint and cover all names.

2. Constraint internalization in the requiring part: for any pair \(r_k^1 : y_k^1\) in the requiring part, and for each pair \(p_i : x_i\) in the constraining part such that \(r_k^1 \land p_i \neq \text{f}\), we have that \(y_k^1 \subseteq x_i\). In this way, when \(y_k^1\) is satisfied, all the constraints \(p_i : x_i\) that apply to some name that matches \(r_k^1\) are guaranteed to be satisfied, hence they are internalized in the assertion \(y_k^1\).

3. Requirements internal splitting: for any two distinct pairs \(r_j^1 : y_j^1\) and \(r_k^1 : y_k^1\) inside the same \(\text{orPattReq}\), they are split, which means that:

(a) the two patterns have a trivial intersection, and
(b) either the pairs are pattern-disjoint, that is \(r_j^1 \land r_k^1 = \text{f}\),
   or they are schema-disjoint, that is \(y_j^1 \land y_k^1 = \text{f}\).
(4) Requirements external splitting: any two distinct pairs \( r^j_1 : y^j_1 \) and \( r^j_k : y^j_k \) found in two distinct orPattReq are either split, as in the previous definition, or equal in both components.

These invariants depend on inclusion assertion, as in \( y^j_1 \subseteq x_i \), or disjointness, as in \( y^j_1 \cap y^j_k = \emptyset \). During preparation, we are not going to check whether two assertions are included or disjoint, since this would be as hard as checking satisfiability. We are going to build assertions that satisfy this constraints, by adding a factor \( \neg \land x \) when we need a sub-assertion of \( x \), or a factor \( \neg \land \neg(x) \) when we need disjunction from \( x \).

As an example, consider the following group. We use here JSON regular expressions, where \(^\ast\) matches the beginning of a string, \([\ ABC] \) matches any one character different from \( a, b \) and \( a, b \), a dot \( . \) matches any character, \( \$ \) matches the end of the string, so that \( ^\ast[a\,b]^\ast \) matches accccc and acc but does not match ac, because the dot after the \( ^\ast[a\,b]^\ast \) requires a third letter (please look carefully for the dots in the patterns). This is the group.

\[
\{\text{Obj, } \ast a : x1, \ast, b : x2, \text{pattReq}(\ast.d : t), \text{pattReq}(\ast a : x3)\}
\]

Object preparation will first rewrite it as follows, and it will then create new variables to separate and expand all conjunctions such as \( x1 \land x2 \). The variable \( co(x3) \) is the variable whose body is the negation of that of \( x3 \). The step-by-step process that produces this expansion is described in [3], but we show here the final result.

\[
\ast a : x1, \ast, b : x2 \quad \rightarrow \quad ^\ast[a\,b] : x1, \ast ab : x1 \land x2, \\
\text{pattReq}(\ast.d : t) \quad \rightarrow \quad \text{orPattReq}(\ast ad : x1 \land x3, \ast[a\,d] : t) \\
\text{pattReq}(\ast a : x3) \quad \rightarrow \quad \text{orPattReq}(\ast ad : x1 \land x3, ^\ast[a\,bd] : x1 \land x3, \\
\ast ab : x1 \land x2 \land x3).
\]

In the constraining part, the set \( \{a, b\} \) has been divided into three disjoint parts \( \{a, b\}, \ast ab, ^\ast[a\,b] \) by separating the intersection \( \ast ab \) from the two original patterns, and the set is completed with \( ^\ast[a] : t \). The first request pattReq(\(\ast.d : t\)) is split into three different cases. The first \( ^\ast ad : x1 \land x3 \) is in common with the other orPattReq, while the case \( ^\ast ad : x1 \land co(x3) \) is internally and externally split thanks to the \( co(x3) \) factor in the schema, and \( ^\ast[a] \) is pattern-disjoint thanks to the initial \( [a] \). You can also observe that \( ^\ast ad : x1 \land x3 \) internizes the requirement \( ^\ast a : x1 \), the same holds for \( ^\ast ad : x1 \land co(x3) \), while \( [a]d \) only matches the trivial requirement, hence maintains its t schema. The second pattReq is split into three cases as well, in order to bring into view the intersection with the first pattReq, and in order to internalize the constraints of the constraining part.

This splitting effort is needed in order to be able to enumerate and try all the possible ways of satisfying a set of requests. For example, in this case the two orPattReq requests the first component \( ^\ast ad : x1 \land x3 \), and contain two more components each, all of them mutually incompatible, hence having a structure orPattReq(a,b1,b2),orPattReq(a,c1,c2). Hence, we know that there are exactly 5 ways of satisfying both: either by generating a single member that satisfies \( a \), or by generating two members that satisfy, respectively, \((b1,c1),(b1,c2),(b2,c1),(b2,c2)\), and our witness generation algorithm will try to pursue all, and only, these five approaches.

Also array groups need preparation, still because several assertions inside an array group may overlap. For space reason we omit this part in this paper (details are in the full version [3]).

5.7 Recursive witness generation

We illustrate the algorithm by means of an example, by focusing on object groups (other cases are dealt with in the full paper [3]). Consider the following set of equations, which is not complete since we removed all those that are not reachable from the root.

\[
\text{root } x = \\{\text{Obj, orPattReq( } a : l, ^b : y\} \\
\text{def } y = \\{\text{Obj, orPattReq( } a : z, ^b : k, \text{orPattReq( } c : m)\} \\
\text{def } k = \\{\text{Obj, orPattReq( } a : l)\} \\
\text{def } l = \\{\text{Obj, orPattReq( } a : x)\} \\
\text{def } z = \\{\text{Obj, orPattReq( } a : x) \lor \text{Null}\} \\
\text{def } m = \{\text{Num}\}
\]

The algorithm proceeds by passes. Each pass begins with a state where each variable is either Populated with a witness \( J(P(J)) \), Empty (\( \emptyset \)) if we proved that it has no witnesses, or Open (\( ? \)) otherwise. At the beginning each variable is Open:

\[
P_0 : x = ?, y = ?, k = ?, l = ?, z = ?, m = ?
\]

At each pass, we evaluate the body of each variable using the state of the previous pass. If nothing changes, we stop. Otherwise, we continue until a witness is found for the root variable. Here, at pass 1, we are able to prove that \( l \) is empty, and we can provide a witness for \( z \) and \( m \).

\[
P_1 : x = ?, y = ?, k = ?, l = f, z = \text{P(nul1)}, m = P(3)
\]

At pass 2 we use the knowledge of pass 1 to fix both \( y \) and \( k \).

\[
P_2 : x = ?, y = P(\{a : \text{null}, c : 3\}), k = f, l = f, z = \text{P(nul1)}, m = P(3)
\]

And finally, we converge at pass 3.

\[
P_3 : x = P(\{b : \{a : \text{null}, c : 3\}\}), y = P(\{a : \text{null}, c : 3\}), \quad k = f, l = f, \ldots
\]

As a negative example, consider the following system

\[
\text{root } x = \{\text{Obj, orPattReq( } a \}) \\
\text{def } y = \{\text{Obj, orPattReq( } a : z), \text{orPattReq( } b : x)\} \\
\text{def } z = \{\text{Obj, orPattReq( } a : y) \lor \{\text{Num}\}
\]

We report here the trace of a run of the algorithm.

\[
P_0 : x = ?, y = ?, z = ?
\]

\[
P_1 : x = ?, y = ?, z = P(3)
\]

\[
P_2 : x = ?, y = ?, z = P(3)
\]

stop

Here, the algorithm reached a fix point without a value for \( x \) and \( k \), which are therefore empty.
Hence, the algorithm is defined as follows. We first mark all variables that are actually reachable from the root, and delete the others. We associate a state of Open to each variable. We compute a new state for each Open variable on the basis of the current state. If the new state is equal to the previous state, the algorithm returns “no witness”. If the new state has a witness for the root variable, this witness is returned. If the state changed but the root variable is still Open, we execute a new pass.

The computation of the new state proceeds as follows. The definition of each variable is a disjunction of typed groups. The variable asks a witness to each of them. If one group provides a witness, this is a witness for the variable. If one group answers f, it is removed and, if all are removed, the variable answers f. If all groups answer ?., then the variable is still Open. For the typed groups, each runs an algorithm that depends on its type, which is described below. For the base groups, the answer is either $P(f)$ or $f$ during the first pass, and will not change. For the object and array groups the answer will depend on the current state of the variables that appear in them.

Termination of recursive witness generation can be proved using a classical minimal-fixpoint argument, as follows.

Witness generation returns three possible results, Empty, Open, and Populated, and every variable, at the end of each pass, is in one of those three states. If we order these states as Open < Populated and Open < Empty, we observe that witness generation for object and array types is a monotone function on the state. In greater detail, the Open, Empty, or Populated result of witness generation is uniquely determined by the state of all variables, and, whenever one variable increases its state, the result of witness generation either remains equal or increases. As a consequence, the trace of any run, defined as the sequence of tuples that associate each variable with its state, can only increase or remain immobile at every step hence, having a finite number of distinct values, the trace is guaranteed to converge to a fixpoint.

5.8 Witness Generation from Typed Groups

We have finally to specify how each typed group will generate its witnesses starting from the witnesses associated to the different variables. The treatment for these cases is dealt with in the full version [3].

6 RELATED WORK

We are not aware of any formal algebra for JSON Schema. The first effort to formalize the semantics of JSON Schema as by Pezoa et al. in [8] whose goal was to lay the foundations of the JSON schema proposal by studying its expressive power and the complexity of the validation problem. Along the lines of this work, Bouhis et al. [5] characterized the expressivity of the JSON Schema language and investigated the complexity of the satisfiability problem which turns out to be $\text{EXPTIME}$ in the general case and $\text{EXPSPACE}$ when disallowing $\text{uniqueltens}$. None of the above works study the problem of generating an instance of a JSON Schema. The only attempt to solve this problem was investigated by Earle et. al [4] in the context of testing REST API calls but the presented solution, which is based on translating JSON Schema definitions into an Erlang expression, is not formally defined and restricted to atomic values, objects and to some form of boolean expressions.

From the point of view of schema normalization, the closest work to ours is the one in [7] which studies schema inclusion for JSON Schema. To cope with the high expressivity of the JSON Schema language, a pre-requisite step is needed to rewrite the schemas into a Disjunctive Normal Form which has some similarities with the preparation phase of our work. However, compared to our work, the schema normalization in [7] lacks the ability of eliminating negation for all kinds constraints, does not deal with recursive definitions and is not able to decide schema satisfiability which is captured by the inhabited() predicate whose specification is only informally discussed. This has been confirmed in practice by experimenting the tool developed in [7] for parsing real world schemas described in [3]: the tool raised an issue for 21,859 out of 23,480 input schemas. The dominating error is related to constructs not being supported, but many other errors due to the inability to parse recursive schemas or to navigate references are present.

7 CONCLUSIONS

JSON Schema is an evolving standard for the description of families of JSON documents, and is widely used in data-centric applications. Despite the recent interest in the research community related to this schema language, crucial problems like schema equivalence/inclusion and consistency have either been partially dealt with or not explored at all. In this work we present our approach in order to solve these problems, based on our algebraic specification of JSON Schema. We are currently finalising a Java implementation of the presented algorithm, and studying optimisation techniques, by analysing a large repository of JSON Schemas allowing us for determining how often mechanisms that are critical for execution times are used. We are also investigating witness generation techniques able to generate several instances meant be used for testing queries and programs manipulating valid JSON data.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Internet engineering task force. 2020. Available at https://www.ietf.org.
[2] Json schema, 2020. Available at https://json-schema.org.
[3] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Foundations of json schema. In WWW ’16, pages 263–273, 2016.
[4] Clara Benes Earle, Lars-Åke Fredlund, Ångel Herranz, and Julio Marinho. Jsongen, a quickcheck based library for testing json web services. In Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, pages 33–41, 2014.
[5] Pierre Bouhis, Juan L. Reutter, Fernando Suarez, and Domagoj Vrgoc. JSON-data model, query languages and schema specification. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, PODS, pages 123–135. ACM, 2017.
[6] Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. Theory Comput. Syst., 53(2):159–193, 2013.
[7] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. Type safety with json subschema, 2019.
[8] Pierre Bouhis, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoc. Foundations of json schema. In WWW ’16, pages 263–273, 2016.
[9] Pierre Bouhis, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoc. Foundations of json schema. In WWW ’16, pages 263–273, 2016.
[9] A. Wright, H. Andrews, and B. Hutton. JSON Schema validation: A vocabulary for structural validation of json - draft-handrews-json-schema-validation-02. Technical report, Internet Engineering Task Force, sep 2019.