Finding Data Compatibility Bugs with JSON Subschema Checking

Andrew Habib
University of Luxembourg

Avraham Shinnar
Martin Hirzel
IBM Research

Michael Pradel
University of Stuttgart

ISSTA'21 – July 11 - 17, 2021

JSON

- **Data serialization format**
- **Widely supported**

  E.g.: C/C++, C#, Java, JS, Python
- **Pervasively used**

  Web applications, NoSQL databases, machine learning, . . .

Examples:

```
{"First Name": "Joe",
 "Age": 5,
 "State": "CA"}
```

```
{"Ids": [1, 2, 3]}
```

JSON

How to ensure data conforms to a specific format/structure?

Examples:

```
{"First Name": null,
 "Age": -1,
 "State": "CA"}
```

```
{"Ids": [1, 2, "x"]}
```

JSON Schema

Data specification language

JSON Schema

Data specification language

- **Describes**

  structure: objects and arrays

  values: integer, number, string, boolean, and null

JSON Schema

Data specification language

- **Describes**

  structure: objects and arrays

  values: integer, number, string, boolean, and null

- **Enables validation of JSON data**

  JSON document d, JSON schema s

$$\textit{valid}(d, s) \rightarrow \{True, False\}$$

JSON Schema

Data specification language

- **Describes**

  structure: objects and arrays

  values: integer, number, string, boolean, and null

- **Enables validation of JSON data**

  JSON document d, JSON schema s

  $$\textit{valid}(d, s) \rightarrow \{True, False\}$$

- **Many validators available**
- **Widely used**

JSON Schema Example

JSON documents

```
{"First Name": "Joe",
 "Age": 5,
 "State": "CA"}
```

JSON Schema Example

JSON documents

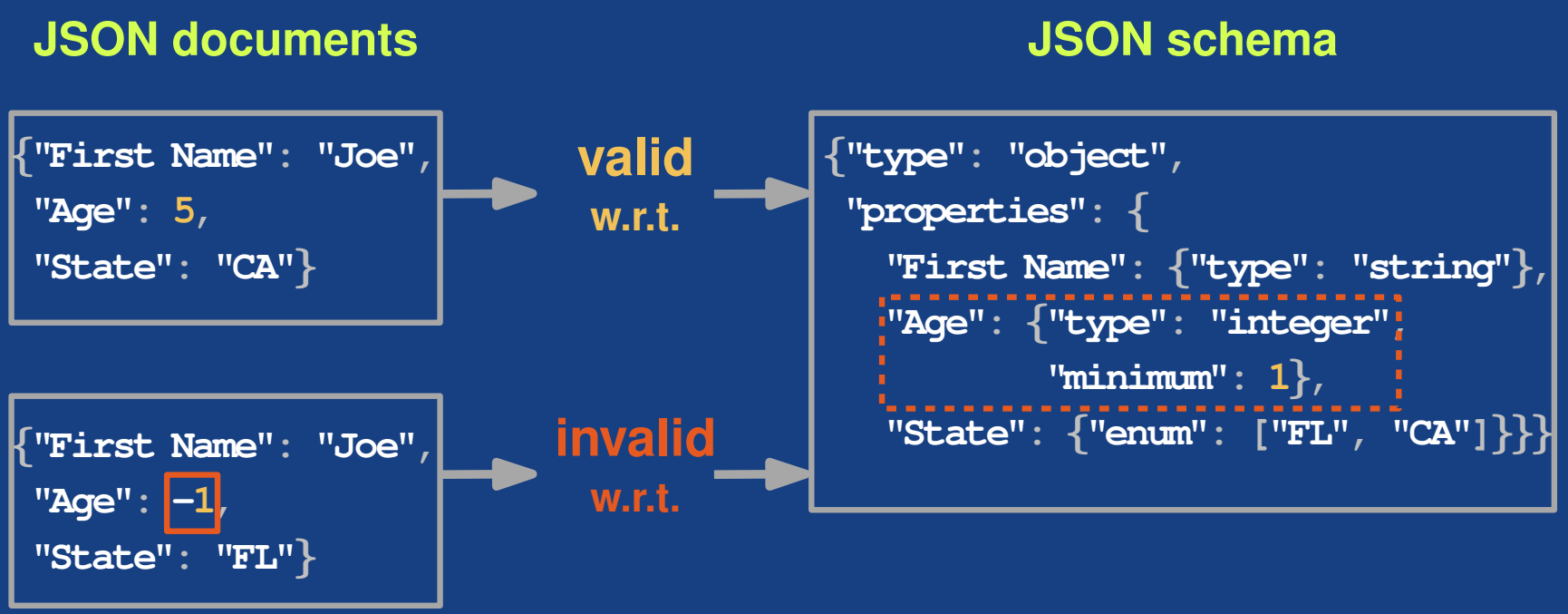
```
{"First Name": "Joe",
 "Age": 5,
 "State": "CA"}
```

valid
w.r.t.

JSON schema

```
{"type": "object",
 "properties": {
   "First Name": {"type": "string"},
   "Age": {"type": "integer",
           "minimum": 1},
   "State": {"enum": ["FL", "CA"]}}}
```

JSON Schema Example



JSON Schema (2)

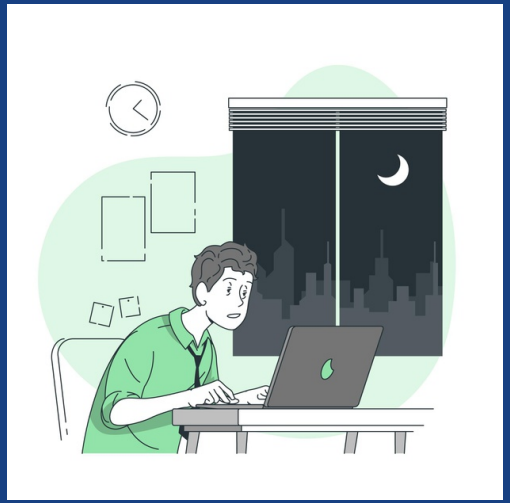
Data validation happens:

- Only at runtime
- When data occurs

JSON Schema (2)

Data validation happens:

- **Only at runtime**
  == Too late?
- **When data occurs**
  == Can’t reason w/out data :(



JSON Schema (3)

Why reason about schemas without data?

JSON Schema (3)

Why reason about schemas without data?

Scenario 1: **Schema evolution, e.g., Web API**

JSON Schema (3)

Why reason about schemas without data?

Scenario 1: **Schema evolution, e.g., Web API**

JSON Schema (3)

Why reason about schemas without data?

Scenario 1: Schema evolution, e.g., Web API

Scenario 2: ML pipeline with schemas

JSON Schema (3)

Why **reason about schemas** without data?

Scenario 1: Schema evolution, e.g., Web API

Scenario 2: ML pipeline with schemas

Data Compatibility Bugs

Given two schemas s_1 and s_2,
are they compatible with each other?

Data Compatibility Bugs 

Given two schemas s_1 and s_2,
are they compatible with each other?

JSON Subschema ($<:$) Problem:

$s_1 <: s_2$ **iff.** $\forall d : \textbf{\textit{valid}}(d, s_1) \implies \textbf{\textit{valid}}(d, s_2)$

Data Compatibility Bugs 

Given two schemas s_1 and s_2,
are they compatible with each other?

JSON Subschema ($<:$) Problem:

$s_1 <: s_2$ **iff.** $\forall d : \textbf{\textit{valid}}(d, s_1) \implies \textbf{\textit{valid}}(d, s_2)$

- Form of subtyping
- Type (schema) as a set of values (documents)

Subschema Example

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
   ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
  ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Subschema Example

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
   ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
  ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Subschema Example

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
    ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
   ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Subschema Example

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
   ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
  ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Subschema Example

Schema evolution bug:

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
   ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

$\not<:$

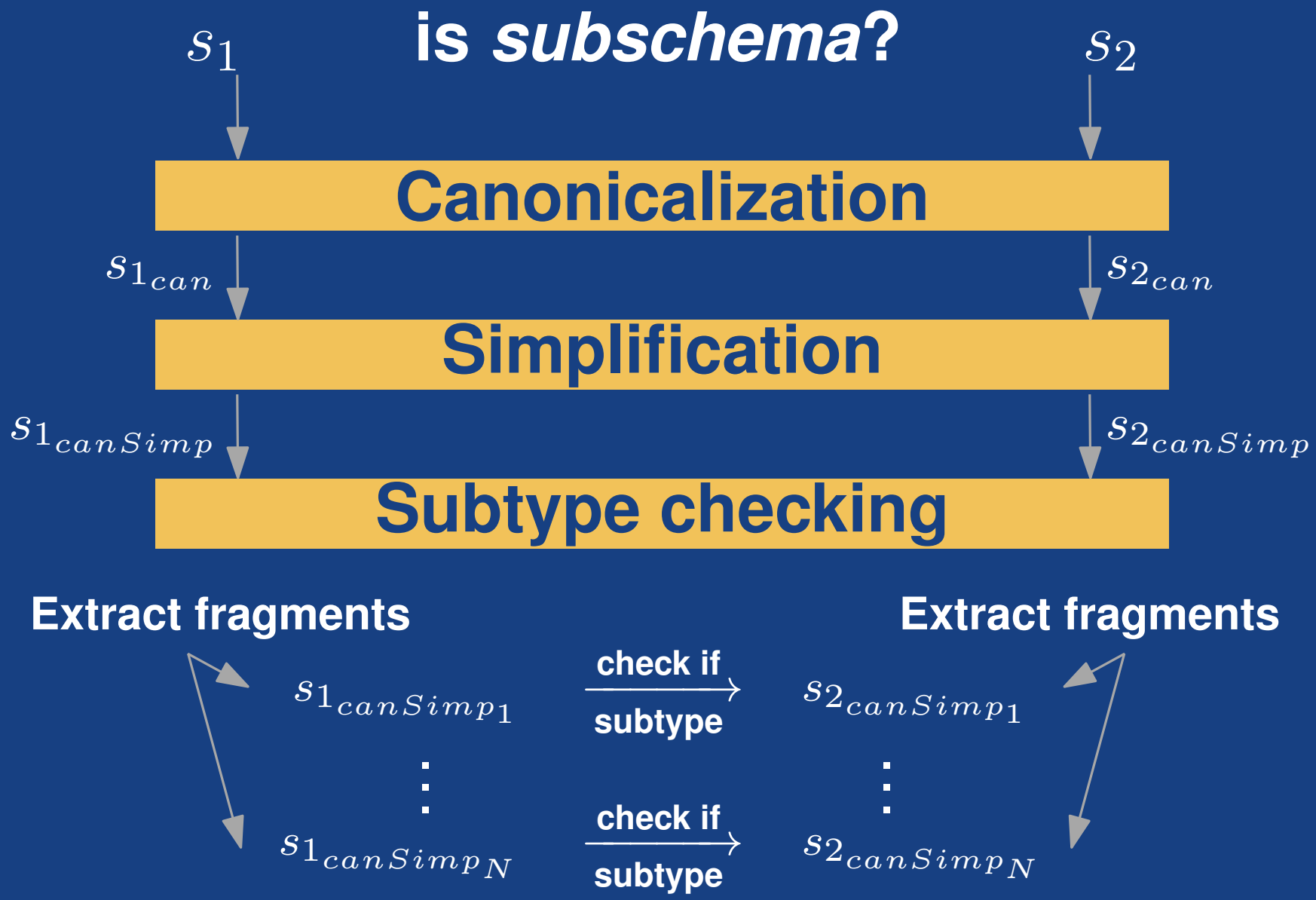
$:\not>$

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
  ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Breaking change! Wrong versioning.

Subschema Approach



Canonicalization & Simplification

- Reduce the ways the different features of JSON Schema are used.

Canonicalization & Simplification

- **Reduce the ways the different features of JSON Schema are used.**
- **Rewrite schemas into:**
  - Equivalent
  - More tractable

Canonicalization & Simplification

- **Reduce the ways the different features of JSON Schema are used.**
- **Rewrite schemas into:**
  - Equivalent
  - More tractable

E.g.: Explicate implicit disjunctions

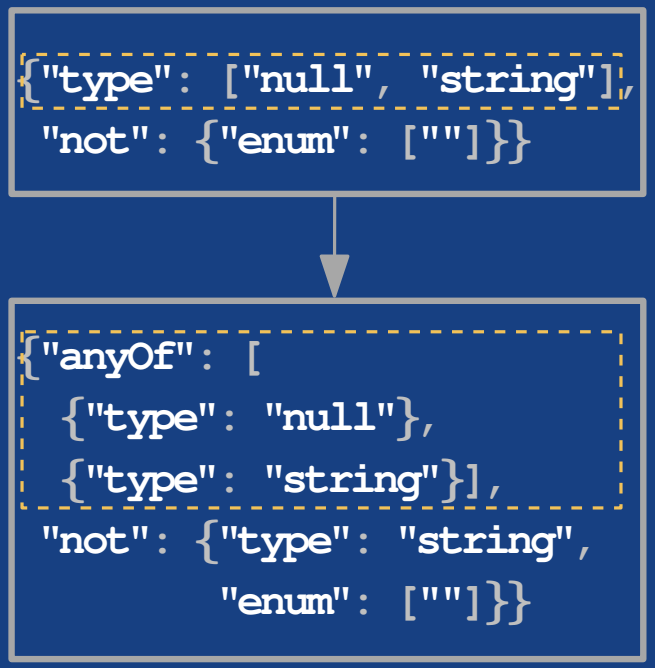
```
{"type": ["null", "string"]}
```

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]}
```

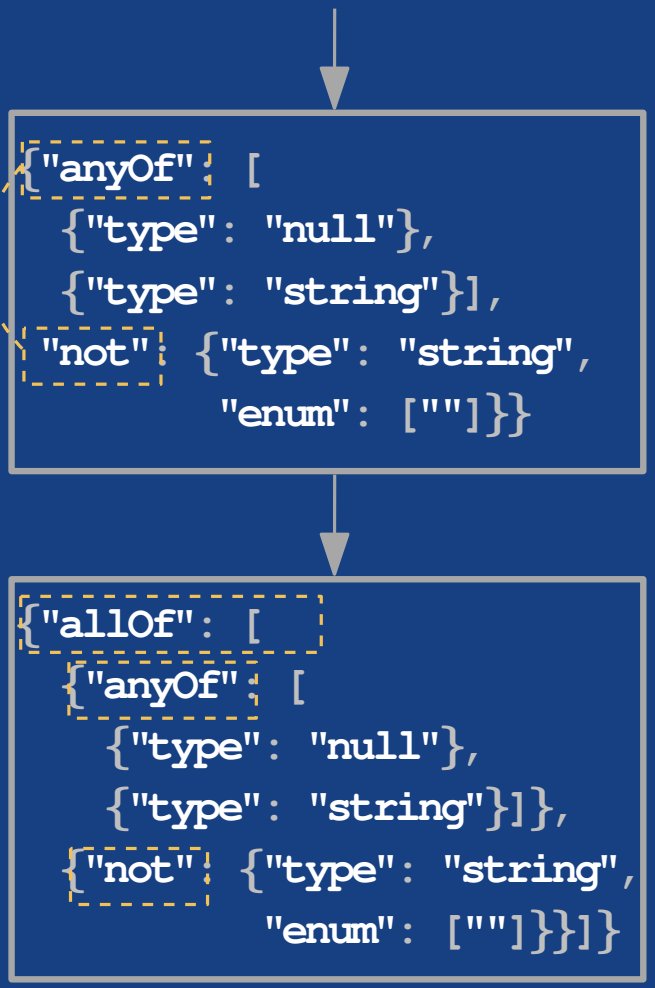
Example

```
{"type": ["null", "string"],
 "not": {"enum": [""]}}
```

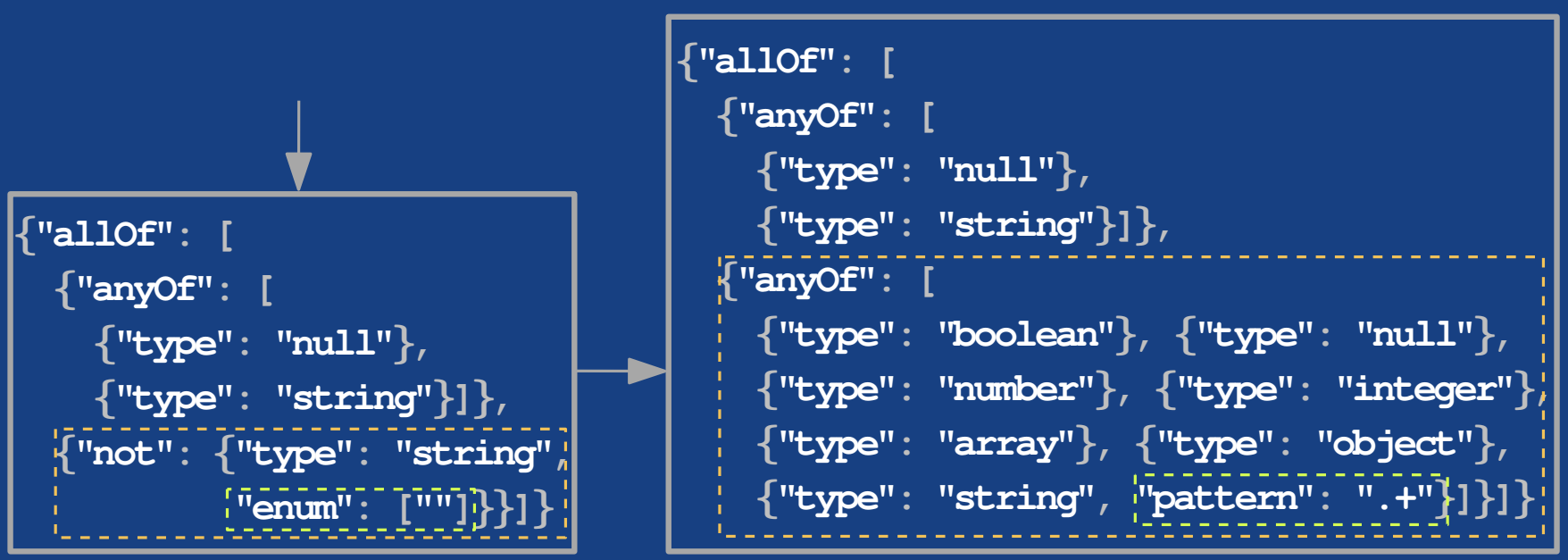
Example


{"type": ["null", "string"],
"not": {"enum": [""]}}
{"anyOf": [
{"type": "null"},
{"type": "string"}],
"not": {"type": "string",
"enum": [""]}}

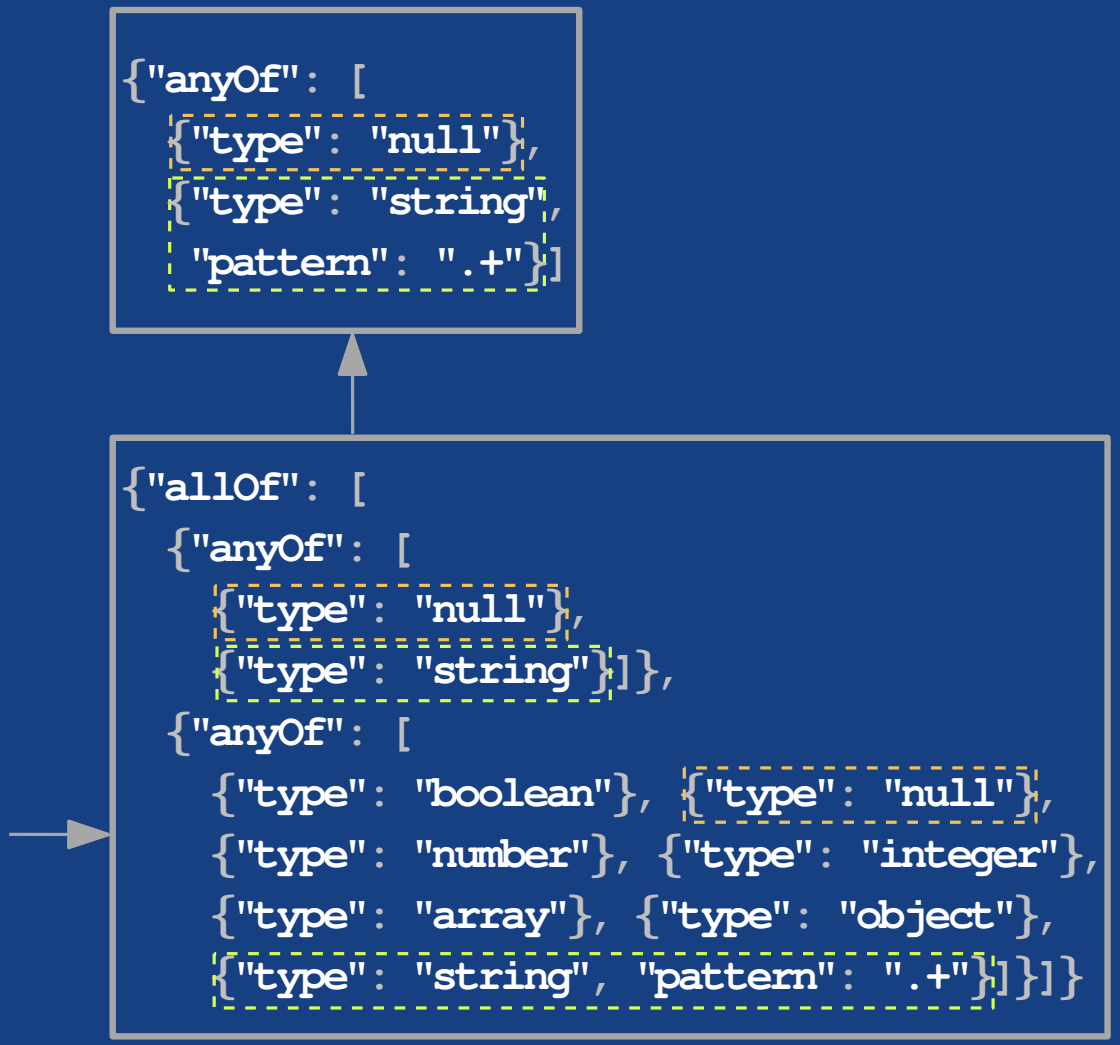
Example


{"anyOf": [
{"type": "null"},
{"type": "string"}],
"not": {"type": "string",
"enum": [""]}}
{"allOf": [
{"anyOf": [
{"type": "null"},
{"type": "string"}]},
{"not": {"type": "string",
"enum": [""]}}]}

Example

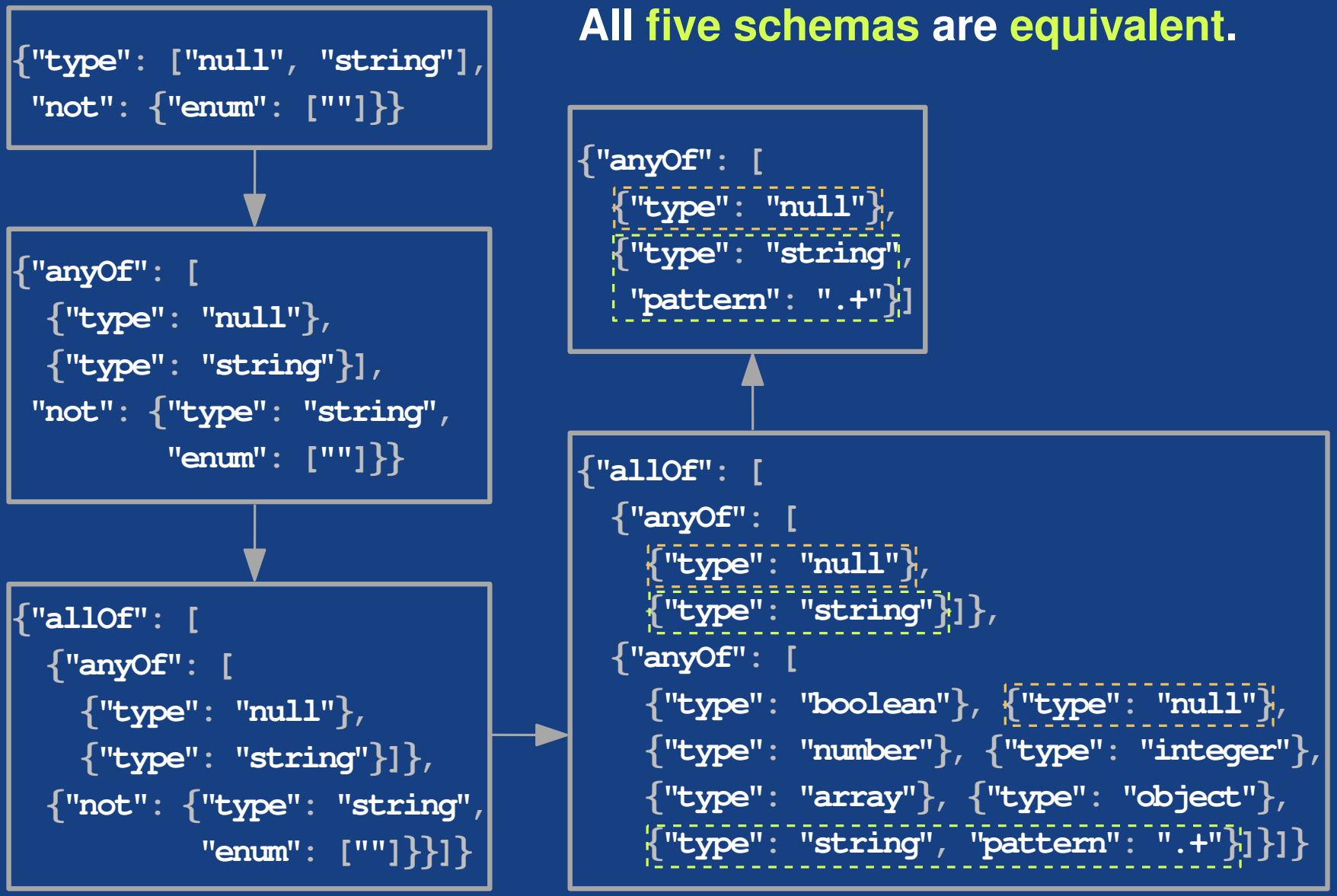

{"allOf": [
{"anyOf": [
{"type": "null"},
{"type": "string"}]},
{"not": {"type": "string",
"enum": [""]}}]}
{"allOf": [
{"anyOf": [
{"type": "null"},
{"type": "string"}]},
{"anyOf": [
{"type": "boolean"}, {"type": "null"},
{"type": "number"}, {"type": "integer"},
{"type": "array"}, {"type": "object"},
{"type": "string", "pattern": ".+"}]}]}

Example


{"anyOf": [
{"type": "null"},
{"type": "string",
"pattern": ".+"}]
{"allOf": [
{"anyOf": [
{"type": "null"},
{"type": "string"}]},
{"anyOf": [
{"type": "boolean"}, {"type": "null"},
{"type": "number"}, {"type": "integer"},
{"type": "array"}, {"type": "object"},
{"type": "string", "pattern": ".+"}]}]}

Example

All five schemas are equivalent.



Subschema Checking

Type-specific rules, e.g.:

$$\textbf{null}\ \frac{s_1.\texttt{type} = \texttt{null} \qquad s_2.\texttt{type} = \texttt{null}}{s_1 <: s_2}$$

Subschema Checking

Type-specific rules, e.g.:

$$\textbf{null}\ \frac{s_1.\texttt{type} = \texttt{null} \qquad s_2.\texttt{type} = \texttt{null}}{s_1 <: s_2}$$

$$\textbf{boolean}\ \frac{\begin{array}{c} s_1.\texttt{type} = \texttt{boolean} \qquad s_2.\texttt{type} = \texttt{boolean} \\ s_1.\texttt{enum} \subseteq s_2.\texttt{enum} \end{array}}{s_1 <: s_2}$$

Subschema Checking

Type-specific rules, e.g.:

$$\textbf{null}\ \frac{s_1.\texttt{type} = \texttt{null} \quad s_2.\texttt{type} = \texttt{null}}{s_1 <: s_2}$$

$$\textbf{boolean}\ \frac{s_1.\texttt{type} = \texttt{boolean} \quad s_2.\texttt{type} = \texttt{boolean} \quad s_1.\texttt{enum} \subseteq s_2.\texttt{enum}}{s_1 <: s_2}$$

$$\textbf{string}\ \frac{s_1.\texttt{type} = \texttt{string} \quad s_2.\texttt{type} = \texttt{string} \quad s_1.\texttt{pattern} \subseteq s_2.\texttt{pattern}}{s_1 <: s_2}$$

Subschema Checking (2)

Non-type-specific rules, e.g.:

non-overlapping anyOf

$$\frac{\forall i \in \{1,\ldots,n\}, \exists j \in \{1,\ldots,m\}, s_i <: t_j \quad \textit{nonOverlapping}([t_1,\ldots,t_m])}{\{\texttt{anyOf}:[s_1,\ldots,s_n]\} <: \{\texttt{anyOf}:[t_1,\ldots,t_m]\}}$$

Subschema Checking (2)

Non-type-specific rules, e.g.:

non-overlapping anyOf

$$\frac{\forall i \in \{1,\ldots,n\}, \exists j \in \{1,\ldots,m\}, s_i <: t_j \quad \textit{nonOverlapping}([t_1,\ldots,t_m])}{\{\texttt{anyOf}:[s_1,\ldots,s_n]\} <: \{\texttt{anyOf}:[t_1,\ldots,t_m]\}}$$

uninhabited

$$\frac{\neg\, inhabited(s_1)}{s_1 <: s_2}$$

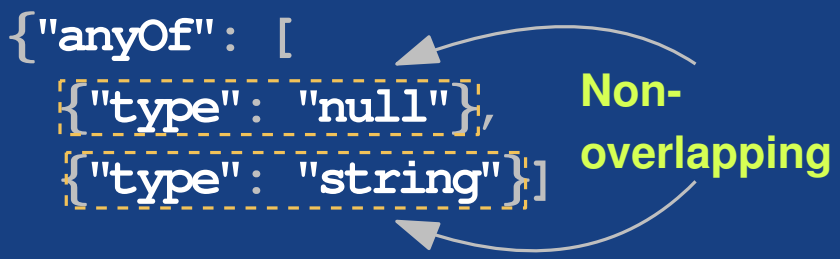
Example

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]
```

Example

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```



Example

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```

<:

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]
```

Example

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```

<:

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]
```

Example

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```

$<:$

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]
```

Example

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```

$<:$

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]
```

```
{"anyOf": [
  {"type": "null"},
  {"type": "string",
   "pattern": ".+"}]
```

$:\not>$

```
{"anyOf": [
  {"type": "null"},
  {"type": "string"}]
```

Evaluation Setup

Dataset

- **Four projects:**
  - Web: Snowplow, Kubernetes (k8), Washington Post (WP)
  - Machine learning: Lale
- **More than 3,000 schemas**
- **More than 100,000 schema pairs**

Effectiveness in Detecting Bugs

Five schema evolution bugs in Snowplow, e.g.:

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
   ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
  ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Effectiveness in Detecting Bugs

Five schema evolution bugs in Snowplow, e.g.:

```
{"properties": {
  "event": {"type": "object"},
  "error": {"type": "string"},
   ...}
 "required": ["event", "error"],
 "additionalProperties": false
}
```

version 1.0.0

```
{"properties": {
 "payload": {"type": "object"},
 "failure": {"type": "string"},
  ...}
 "required": ["payload", "failure"]
 "additionalProperties": false
}
```

version 1.0.1

Breaking change! Wrong versioning.

All bugs are confirmed.

Effectiveness in Detecting Bugs (2)

38 ML schema bugs in Lale, e.g.:

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

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

Effectiveness in Detecting Bugs (2)

38 ML schema bugs in Lale, e.g.:

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

Array of numbers or strings (wrong)

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

Array of numbers *or* array of strings (correct)

All bugs are confirmed and fixed.

JSON subschema is used actively within Lale as a static type checker.

Correctness in Practice

300 ground truth pairs (manually)

- **Precision** $= \frac{TP}{TP+FP}$
- **Recall** $= \frac{TP}{TP+FN+?_{<:}}$

  ($?_{<:}$ for undecided cases by our subschema approach)
- **Correctness** $= \frac{TP+TN}{TP+TN+FP+FN}$

Correctness in Practice

300 ground truth pairs (manually)

	Ground truth		
Dataset	Pairs	$<:$	$\not<:$
Snowplow	50	17	33
WP	50	35	15
K8	50	31	19
Lale	50	22	28
Unrelated	100	3	97
Total	300	108	192
Precision			
Recall			
Correctness			

Correctness in Practice

300 ground truth pairs (manually)

	Ground truth			*jsonsubschema*					
Dataset	Pairs	$<:$	$\not<:$	TP	TN	FP	FN	$?_{<:}$	$?_{\not<:}$
Snowplow	50	17	33	14	29	0	0	3	4
WP	50	35	15	31	4	0	0	4	
K8	50	31	19	31	19	0	0	0	0
Lale	50	22	28	22	28	0	0	0	0
Unrelated	100	3	97	3	97	0	0	0	0
Total	300	108	192	101	177	0	0	7	15
Precision						100%			
Recall						93.5%			
Correctness						100%			

Limitations

- **Unsupported features** (returns: ***unknown***)
  - Recursive schemas through `"$ref"`
  - Negation, disjunction and enumeration of arrays and objects
  - Non-regular regex patterns
- **Not very fast**

  Several seconds for large schemas

Conclusion

Detecting data compatibility bugs with JSON subschema

- **Principled approach:**

  canonicalization, simplification, subtype checking

- **Used actively by IBM Lale**

  (`github.com/IBM/lale`)

- **Open-source tool**

  (`github.com/IBM/jsonsubschema`)