LIFE ON THE WEB

- A very common paradigm in Web development: APIs
  - I send some query to the server
  - The server replies with some data

What's the weather like?

It's windy mate 😊
How does this data look like?

- More often than not it is in JSON

```json
{"city": "Chicago", "when": "now"}
```

```json
{
    "city": "Chicago",
    "timestamp": "15-05-17-08-08",
    "temp": 22,
    "description": "windy"
}
```
BOTTOM LINE

- JSON is used a lot:
  - Data exchange on the Web (APIs and the like)
  - Part of programming languages (Python, Ruby, JavaScript)
  - NoSQL databases (MongoDB, Neo4j)

- The how come:
  - Almost no research on the JSON data format and its usage
  - Very few studies available
  - No formal data model/query language
  - Some preliminary work on schema specification
JSON: Data Model, (Query Languages) and Schema Specification

Pierre Bourhis¹, Juan L. Reutter², Fernando Suárez² and Domagoj Vrgoč²

¹ CNRS CRIStAL Lille and INRIA Lille
² PUC Chile and Center for Semantic Web Research

Thank to Domago Vrgoc for his slides
WHAT IS JSON?

- A fully compositional data format with:
  - Strings and numbers as atomic elements (and some other stuff)
  - Arrays allowing grouping
  - Objects/dictionaries allowing nesting

```json
{
  "name": {
    "first": "John",
    "last": "Doe"
  },
  "age": 32,
  "hobbies": ["fishing","yoga"]
}
```
DATA MODEL FOR JSON – JSON TREES

- JSON document = a set of key-value pairs
  - Each value is again a JSON object
- Naturally suggests tree structure

```
{
  "name": {
    "first": "John",
    "last": "Doe"
  },
  "age": 32,
  "hobbies": ["fishing", "yoga"]
}
```
SOME SPECIFICS OF JSON TREES

- Inherent typing
  - Each node is either an object or an array
  - Leaves are atomic values

- Labelled edges
  - Allows retrieving values deterministically (e.g. Doc["name"])

- Array nodes have random access to their children

- Fully compositional
  - Value of a node is always a valid JSON tree
Main differences:
- JSON trees are deterministic
  - Allows direct access
- Value of a node is again a JSON tree
  - Comparing values amounts to comparing subtrees, not atoms
- JSON trees mix ordered and unordered data
  - Random access for arrays

Can we code JSON as XML?
- Sure, but this might be an overkill
- One of the main reasons JSON got popular = not to do this
SCHEMA DEFINITION FOR JSON

- We want to specify what sort of data our JSON has

- IETF has a draft proposal for JSON Schema

- Heavily studied in

- A friendly introduction available at:
  http://cswr.github.io/JsonSchema/
SCHEMA DEFINITION FOR JSON

```json
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "age": { "type": "integer" },
    "club": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "founded": { "type": "integer" }
      },
      "required": ["name"]
    }
  },
  "required": ["first_name", "last_name", "age", "club"]
}
```
To capture JSON Schema we introduce JSON Schema logic (JSL)

\[ \varphi, \psi := \top | \neg \varphi | \varphi \land \psi | \varphi \lor \psi | \psi \in \text{NodeTests} | \square_e \varphi | \square_{i:j} \varphi | \Diamond_e \varphi | \Diamond_{i:j} \varphi \]

- NodeTests take care of basic stuff (typing and matching):
  - E.g. \( \text{Int} \land \text{Min}(i) \) – a number greater than \( i \)

- Modal operators take care of arrays, objects and nesting:
  - E.g. \( \square_{1:+\infty} \text{Str} \) – all elements of my array are strings
HOW DOES THIS WORK?

```
{
    "type": "object",
    "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" },
        "age": { "type": "integer" },
        "club": {
            "type": "object",
            "properties": {
                "name": { "type": "string" },
                "founded": { "type": "integer" }
            },
            "required": ["name"]
        }
    },
    "required": ["first_name", "last_name", "age", "club"]
}
```

\( \text{Obj} \land \Diamond \text{first\_name} \land \text{Str} \land \Diamond \text{last\_name} \land \text{Str} \land \Diamond \text{age} \land \text{Int} \land \Diamond \text{club} (\text{Obj} \land \Diamond \text{name} \land \text{Str} \land \Box \text{founded} \land \text{Int}) \)
WHAT CAN OUR LOGIC DO?

- In terms of expressive power:

**Theorem:**

JSL captures the core JSON Schema
- For every schema there is an equivalent formula and vice versa

- In terms of algorithmic properties:

**Theorem:**

a) Testing if JSL formula is true on a JSON is in $O(|T|^2 \times |\varphi|)$
- Drops to $O(|T| \times |\varphi|)$ when uniqueness constraint is not permitted
b) The satisfiability problem for JSL is in EXPSPACE
- It is PSPACE-complete when uniqueness constraint is not permitted
CAPTURING FULL JSON SCHEMA

- Full JSON Schema allows references and definitions
  - Can get crazy if not well-formed
  - To capture this we need recursion

- We add this to JSL by allowing datalog-like definitions
  - Example: every path from the root to the leaves is even:

\[
\begin{align*}
\gamma_1 & = \Box_{\Sigma} \ast \gamma_2 \\
\gamma_2 & = (\Diamond_{\Sigma} \ast T) \land (\Box_{\Sigma} \ast \gamma_1) \\
\gamma_1 &
\end{align*}
\]
We can now capture all of (well-formed) JSON Schema

**Theorem:**
Recursive JSL captures well-formed JSON Schema

Algorithmically we get a jump:

**Theorem:**
- Evaluating a recursive JSL formula is PTIME-complete
- The satisfiability problem for recursive JSL is in 2EXPTIME
  - It is EXPTIME-complete when uniqueness constraint is not allowed
Navigational Queries Over JSON

- Inspired by:
  - Python (dictionaries), MongoDB (find function)
  - JSONPath (non-determinism, recursion)

- Compare values (and retrieve):
  ```
  Doc["hobbies"][1] == {"first": "John", "last": "Doe"}
  ```
A query language capturing this

$\alpha, \beta := \langle \varphi \rangle \mid X_w \mid X_i \mid \alpha \circ \beta \mid \varepsilon$

$\varphi, \psi := T \mid \neg \varphi \mid \varphi \land \psi \mid \varphi \lor \psi \mid [\alpha] \mid EQ(\alpha, A) \mid EQ(\alpha, \beta)$

Our previous queries:

- $X_{\text{hobbies}} \circ X_1$
- $EQ(X_{\text{name}}, \{\"first\": \"John\", \"last\": \"Doe\"\})$
Main problems:

- **Evaluation:**
  - Input: A JSON tree $T$, a node $n$, a formula $\varphi$
  - Question: Will $\varphi$ return $n$ when evaluated over $T$?

- **Satisfiability:**
  - Input: A JNL formula $\varphi$
  - Question: If there a JSON tree where $\varphi$ will return at least one node?

**Theorem:**

a) Evaluation of JNL can be solved in time $O(|T| \times |\varphi|)$
b) Satisfiability is NP-complete (even with no negation or EQ).
EXTENDING JNL

- JSONPath allows non-determinism and recursion – JNL*

\[ \alpha, \beta \ := \ \langle \varphi \rangle \ | \ X_e \ | \ X_{i:j} \ | \ \alpha \circ \beta \ | \ (\alpha)^* \ | \ \varepsilon \]

\[ \varphi, \psi \ := \ T \ | \ \neg \varphi \ | \ \varphi \land \psi \ | \ \varphi \lor \psi \ | \ [\alpha] \ | \ EQ(\alpha, A) \ | \ EQ(\alpha, \beta) \]

**Theorem:**

a) Evaluating JNL* is $O(|T|^3 \times |\varphi|)$
b) Without $EQ(\alpha, \beta)$ evaluating JNL* is $O(|T| \times |\varphi|)$

c) Satisfiability of JNL* is undecidable

d) Without $EQ(\alpha, \beta)$ satisfiability of JNL* is EXPSPACE-complete.
e) Without $EQ(\alpha, \beta)$ and without recursion it is PSPACE-complete.
NAVIGATIONAL QUERIES OVER JSON

- Can be shown same to JSON Schema logic
- Check our paper for this 😊
Most interesting questions are driven by practical use-cases:

- MongoDB – a full fledged JSON database
- Streams
- Generating API documentation
THANK YOU!