# Typescript-like Types in Lean

## 1 Motiviation

When working with JSON Schema, either to do run-time validation or to autogenerate Lean `structures`, constructing and working with JSON objects becomes cumbersome and difficult. For instance, many plotting configs in VegaLite end up in the form:

```
structure Config where
  param1 : String ⊕ ExprRef
  param2 : Float ⊕ ExprRef
  param3 : Int ⊕ ExprRef
  ...
```

Constructing such objects would then require `.inl _` every time, which is annoying to use. Constructing substructures requires remembering the names and structure of parameters which are unnecessary for 99% of use cases. Most plots do not need the "backdoor" into Vega that `ExprRef` allows you. One option would be to manually construct alternative types which have default coercions from `String`. However, there are 60,000 lines of VegaLite types, so an alternative target for automatic generation is quite attractive.

## 2 Goals

We would like succinct typesafe JSON with TypeScript-like behavior in Lean.

- Capable enough schema description: intersections, unions, objects, literals, objects, arrays, etc.
- Runtime checking
- Natural JSON syntax
- Enable compile-time checking of automatically generated schemas.
- Propagate JSON type information for field-accessors, discriminant-based narrowing, and object construction.
- Subtyping relation
- (Optional) be able to convert from JSON Schema (however, maybe not the full spec yet)

## 3 Non-goals

- Advanced typescript types: type parameters, conditional types, recursive types
- Compatability with Typescript
- Non-json types

## 4 Alternatives Considered

- GADTs do not solve the problem of `.inl` everywhere and cannot represent type unions easily.
- Runtime type checking does not provide enough safety to guide use-cases like a large plotting library.

## 5 General Approach

```
structure JsonType where
  | literal ...
  | inter ...
  | union ...
  | ...

-- Runtime checking
def JsonType.check (t : JsonType) (x : Json) : Bool := ...
```

```
-- TypedJson as Subtype
def TypedJson (t : JsonType) := Subtype (t.check · = true)
```

Assuming `schema` is a known constant, we can use that information using `decide` or `native_decide`. Using the proposition `schema.check obj = true`, then you can use various decidable facts to extract information:

- Given `schema.check obj = true`
- We check that `schema.hasField "param1" = true` and let `fieldType := schema.getField "param1"`
- Then `fieldType.check (obj.getObjVal! "param1") = true`.

Since `schema` will generally be available as a constant, we can compute information about it at compile-time using `decide`/`native_decide`.

## 5.1 Other potentially pain points

- **Inference during construction**: It may be helpful to automatically assign types for simple constructions. It is not easy to make Json objects in Lean by default.
- **Accesing properties and indices**: Accessing properties of Json objects is already difficult right now. Maybe macros can include the necessary proof plumbing, so `typedJson @. param1` checks that `param1` is in the type.
- **Schema Construction**: Constructing a schema either automatically or even within Lean will likely be difficult.

# 6 Type System

## 6.1 Notation

$$\text{JSON Values}\quad v ::= \textbf{null} \mid \textbf{true} \mid \textbf{false} \mid n \mid s \mid [v_1, ..., v_n] \mid \{f_1 : v_1, ..., f_n : v_n\}$$

$$\begin{aligned} \text{JSON Types}\quad \tau ::= {} & \textbf{null} \mid \textbf{bool} \mid \textbf{number} \mid \textbf{string} \mid \textbf{any} \\ & \mid (\text{literal} :: \text{String}) \mid (\text{number} :: \text{Nat})... \\ & \mid \{f_1 : \tau_1, ..., f_n : \tau_n, o_1 : \sigma_1, ..., o_m : \sigma_m\}\ (\text{Object}) \\ & \mid [\tau_1, ..., t_n]\ (\text{Tuple}) \\ & \mid \tau[]\ (\text{Array}) \\ & \mid (\tau_1 \mid \tau_2)\ (\text{Union}) \\ & \mid \tau_1 \& \tau_2\ (\text{Intersection}) \\ & \mid \textbf{never} \end{aligned}$$

## 6.2 Type Checking

We will denote Lean typing as $s$ :: String, which will be rarely used. While we will use $s$ : **string** to indicate typing in our subtype system `stringType.check s = true`.

### 6.2.1 Basic Types and Literals

$$\frac{}{\textbf{null} : \textbf{null}} \qquad \frac{}{\textbf{true} : \textbf{bool}} \qquad \frac{}{\textbf{false} : \textbf{bool}} \qquad \frac{\text{s} :: \text{String}}{\text{s} : \textbf{string}}\ \text{fromString}$$

$$\frac{\text{i} :: \text{Nat}}{\text{i} : \textbf{number}}\ \text{fromNat} \qquad \frac{\text{i} :: \text{Int}}{\text{i} : \textbf{number}}\ \text{fromInt} \qquad \frac{\text{i} :: \text{Float}}{\text{i} : \textbf{number}}\ \text{fromFloat}$$

**Literals**

$$\frac{}{\text{s} : (s :: \text{String})} \quad \frac{}{\text{x} : (x :: \text{Nat})} \quad \frac{}{\text{x} : (x :: \text{Int})} \quad \frac{}{\text{x} : (x :: \text{Float})}$$

**any**

$$\frac{}{\text{v} : \textbf{any}}$$

### 6.2.2 Schema combinators

**Arrays**

$$\frac{v_i : \tau \quad \dots \quad v_n : \tau}{[v_1, \dots, v_n] : \tau[]}$$

**Tuples**

$$\frac{v_i : \tau_1 \quad \dots \quad v_n : \tau_n}{[v_1, \dots, v_n] : [\tau_1, \dots \tau_n]}$$

**Unions**

$$\frac{\text{v} : \tau_1}{\text{v} : \tau_1 \mid \tau_2} \quad \frac{\text{v} : \tau_2}{\text{v} : \tau_1 \mid \tau_2}$$

**Intersections**

$$\frac{\text{v} : \tau_1 \quad \text{v} : \tau_2}{\text{v} : \tau_1 \ \& \ \tau_2}$$

**Objects**

Note that $\{\}$ should be taken to be unordered. For notation, $o_i$ are the optional field names, $w_i$ the optional values, and $\sigma_m$ the optional types. heading

$$\frac{v_1 : \tau_1 \quad \dots \quad v_n : \tau_n \quad w_{i_1} : \sigma_{i_1} \quad \dots \quad w_{i_k} : \sigma_{i_k}}{\left\{f_1 : v_1, \dots, f_n : v_n, o_{i_1} : w_{i_1}, \dots, o_{i_k} : w_{i_k}\right\} : \{f_1 : \tau_1, \dots, f_n : \tau_n, o_1? : \sigma_1, \dots, o_n : \sigma_n\}}$$

This formulation does not require $k$ to be positive, and it implies some sort of subtyping to be consistent.

## 6.3 Subtype Checking

Even if we know that $v : \tau_1$ as a hypothesis, we may be unable to pass it safely to a function that takes $\tau_2$. We need a way to reliable coerce values during compile time. We can implement this with a decidable subtyping relation <: which is proven true.

So if $v : \tau_1$ implies $v : \tau_2$, then this will not imply that $\tau_1 <: \tau_2$, so subtyping is complete. Rather, we'll target a subset of easily checkable rules so that $\tau_1 <: \tau_2$ and $v : \tau_1$ imply that $v : \tau_2$.

As in type checking, the judgements below show how the algorithm works. We will recursively check different conditions, moving up the tree, and backtracking if necessary.

**Trivial subtyping**

$$\frac{}{\tau <: \tau} \quad \frac{}{\tau <: \textbf{any}} \quad \frac{}{\textbf{never} <: \tau}$$

**Arrays**

$$\frac{\tau_1 <: \tau_2}{\tau_1[] <: \tau_2[]}$$

**Tuples**

$$\frac{\tau_1 <: \top \quad \dots \quad \tau_n <: \top}{[\tau_1, \dots, \tau_n] <: \top[]} \qquad \frac{\tau_1 <: \sigma_1 \quad \dots \ \tau_n <: \sigma_n}{[\tau_1, \dots \tau_n] <: [\sigma_1, \dots \sigma_n]}$$

**Unions**

$$\frac{\top <: \tau_1}{\top <: \tau_1 \mid \tau_2} \qquad \frac{\top <: \tau_2}{\top <: \tau_1 \mid \tau_2}$$

**Intersection**

$$\frac{\top <: \tau_1 \quad \top <: \tau_2}{\top <: \tau_1 \ \& \ \tau_2} \qquad \frac{\tau_1 <: \top}{\tau_1 \ \& \ \tau_2 <: \top} \qquad \frac{\tau_2 <: \top}{\tau_1 \ \& \ \tau_2 <: \top}$$

**Literals**

Literal types are subtypes of their base types:

$$\frac{}{(\text{string literal } s) <: \textbf{string}} \qquad \frac{}{(\text{number literal } n) <: \textbf{number}} \qquad \frac{}{(\text{bool literal } b) <: \textbf{bool}}$$

**Objects**

For objects with required and optional fields, we write $v_1 = \{f_1 : \tau_1, \dots, f_n : \tau_n, o_1? : \sigma_1, \dots, o_m? : \sigma_m\}$ where $f_i$ are required and $o_i$ are optional. For all required fields in $\tau_2$, they are required in $\tau_1$ with a subtype. For all optional fields, if they are in $\tau_1$ too, then they must have a subtype.

$$\frac{\forall\ f_i \in \tau_2\ ,\ f_i \in \tau_1.\text{required} \land \tau_1.f_i <: \tau_2.f_i \qquad \forall\ o_i? \in \tau_2,\ o_i \in \tau_1 \to \tau_1.o_i <: \tau_2.o_i}{\tau_1 <: \tau_2}$$

This ensures width subtyping (extra fields allowed) and depth subtyping (covariant field types), while preventing incompatible types for shared fields.

The subtyping is also similar to record subtyping as in Software Foundations subtyping section.

### 6.3.1 Normalization

We will also need various normalization procedures, which will only be applied once, that will make subtyping more powerful.

**Key lemma:** Normalization must preserve the set of values that check against a type:

$$(\text{norm } \tau).\text{check}(v) \Leftrightarrow \tau.\text{check}(v)$$

This equivalence is critical for the soundness of using normalization as a preprocessor:

$$\frac{\texttt{norm}\ \tau_1 <: \texttt{norm}\ \tau_2}{\tau_1 <: \tau_2}$$

**Literals**

$$\textbf{null} \mapsto \textbf{null}, \textbf{bool} \mapsto \textbf{bool}, \textbf{number} \mapsto \textbf{number}, \textbf{string} \mapsto \textbf{string}, \textbf{any} \mapsto \textbf{any}, \textbf{never} \mapsto \textbf{never}$$

**Objects**

Objects fields should be sorted and all optional fields $o_i$ should be separate.

$$\{f_1 : \tau_1, \dots, f_n : \tau_n, o_1 : \sigma_1, \dots, o_m : \sigma_m\} \mapsto \{f_1 : \texttt{norm}\ \tau_1, \dots, f_n : \texttt{norm}\ \tau_n, o_1 : \texttt{norm}\ \sigma_1, \dots, o_m : \texttt{norm}\ \sigma_m\}$$

**Tuples**

$$[\tau_1, \dots \tau_n] \mapsto [\texttt{norm}\ \tau_1, \dots, \texttt{norm}\ \tau_n]$$

**Arrays**

$$\tau[] \mapsto (\texttt{norm}\ \tau)[]$$

**Unions and Intersections**

$$\tau_1 \mid \tau_2 \mapsto (\texttt{norm}\ \tau_1) \mid (\texttt{norm}\ \tau_2) \quad \tau_1 \& \tau_2 \mapsto (\texttt{norm}\ \tau_1) \& (\texttt{norm}\ \tau_2)$$

All $n$-length unions and intersections should be sorted, and the formulas should be put into disjunctive normal form (this is to enable narrowing later):

$$\tau_1 \& (\tau_2 \mid \tau_3) \mapsto (\tau_1 \& \tau_2) \mid (\tau_1 \& \tau_2)$$

Shared fields in intersections should be merged and extra fields should be concatenated:

$$\{f_1 : \tau_1, ..., f_n : \tau_n, ...\} \& \{f_1 : \sigma_1, ... f_n : \sigma_n, ...\} \mapsto \{f_1 : \tau_1 \& \sigma_1, ..., f_n : \tau_n \& \sigma_n, ...\}.$$

**Never**

Every **never** in a union should be removed. A single **never** in a tuple, array, or intersection should turn the whole type into **never**. A **never** in the required params of an object type should turn to **never**, while optionals should be removed.

## 7 Type "Inference"

### 7.1 Making objects

Given known objects, it may be appropriate to use the type rules in Section 6.2 for real construction:

- We should able to construct a typed null, typed objects, etc.

This mirrors more directly a GADT approach.

### 7.2 Accessing fields

Just as almost all languages provide the ability to type access of structures and arrays, so `x.property` has a known type, we need to be able to provide types easily for accessed properties.

Since we can always decide whether $\tau <: \{f : \sigma\}$ and in fact, we can *get* $\sigma$ using a function fieldType $\tau$, we need only work with known $\{f : \sigma\}$. In this case, we only need to prove

$$\frac{\mathrm{v} : \{f : \tau\}}{\mathrm{v.f} : \tau}$$

### 7.3 Narrrowing

In typescript, one central part of typing is how to discriminate unions. Unlike many other languages with similar features, tags are relatively easy to apply. Typescript calls this narrowing.

```
function padLeft(padding: number | string, input: string): string {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;
    // padding: number
  }
  return padding + input;
  // padding: string
}
```

Typescript will automatically apply narrowing across controw-flow for

- `typeof obj === "number`
- `obj instanceOf Supertype` (this one will turn into intersection)

- `shape.kind === "rect"`
- `x === y`
- `"swim" in animal`

In Lean terms, we can implement this by using macros which inspect the available context instead of analyzing control-flow. We can construct specific types that look for these hypotheses when trying to infer a type. Like subtyping, these rules are search paths which are proven correct. Unlike subtyping, we will be trying to produce the $\tau$ in the conclusion, so these are *forward* rules instead of *backward* rules. Unfortunately, these are far more likely to end up as macros, since requirements for each rule are in the Lean context.

$$\frac{\text{typeof v} = \tau}{\text{v} : \tau} \quad \frac{\text{v} : \tau_1 \quad \text{v} : \tau_2}{\text{v} : \tau_1 \ \& \ \tau_2} \quad \frac{\text{v.f} = \text{value}}{\text{v} : \{f : \text{value}\}} \quad \frac{\text{v} = \text{w} \quad \text{v} : \tau_1 \quad \text{w} : \tau_2}{\text{v} : \tau_1 \ \& \ \tau_2} \quad \frac{\text{f in v}}{\text{v} : \{f : \textbf{any}\}}$$

Since types are expected to always be in context, intersections are expected to normalize to much more convenient types.

## 8 Potential Extra Conveniences

- Being able to turn JSON Schema into such types
- Additional type constructors
- Convenient syntax (also for JSON objects and typed JSON objects). The current notation is bad.
- Convenient special macros to do the above algorithms, instead of merely proofs. Perhaps `simp` sets, `grind` rules, or `aesop` collections.

## 9 Side note on Flow Typing

Whenever you have a single type which will be used in many different ways, this is a "convenient" way to get flow-sensitive typing.

Your set of properties *should* form a lattice, which makes this especially convenient, and there are even ways to handle recursion by using fixed-point theorems in lattice theory.

After reading Why Don't More Languages Offer Flow Typing, I became aware that I really want *flow types* defined over any lattice.

Given Lean's pseudo-SMT solver `grind`, combining and constructing little flow types for a type could be very easy. As an example, perhaps row-types or dataframes are best implemented in this way.