

# TypeScript-like Types in Lean

## 1 Motivation

When working with JSON Schema, either to do run-time validation or to autogenerated 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.
- **Accessing 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

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

JSON Types  $\tau ::= \text{null} \mid \text{bool} \mid \text{number} \mid \text{string} \mid \text{any}$

$$\begin{aligned} & | (\text{literal} :: \text{String}) \mid (\text{number} :: \text{Nat}) \dots \\ & | \{f_1 : \tau_1, \dots, f_n : \tau_n, o_1 : \sigma_1, \dots, o_m : \sigma_m\} \text{ (Object)} \\ & | [\tau_1, \dots, t_n] \text{ (Tuple)} \\ & | \tau[] \text{ (Array)} \\ & | (\tau_1 \mid \tau_2) \text{ (Union)} \\ & | \tau_1 \& \tau_2 \text{ (Intersection)} \\ & | \text{never} \end{aligned}$$

## 6.2 Type Checking

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

### 6.2.1 Basic Types and Literals

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

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

### Literals

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

**any**

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

### 6.2.2 Schema combinators

#### Arrays

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

#### 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

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

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 reliably 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

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

#### Arrays

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

## Tuples

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

## Unions

$$\frac{\tau <: \tau_1}{\tau <: \tau_1 \mid \tau_2} \quad \frac{\tau <: \tau_2}{\tau <: \tau_1 \mid \tau_2} \quad \frac{\tau_1 <: \tau \quad \tau_2 <: \tau}{\tau_1 \mid \tau_2 <: \tau}$$

## Intersection

$$\frac{\tau <: \tau_1 \quad \tau <: \tau_2}{\tau <: \tau_1 \& \tau_2} \quad \frac{\tau_1 <: \tau}{\tau_1 \& \tau_2 <: \tau} \quad \frac{\tau_2 <: \tau}{\tau_1 \& \tau_2 <: \tau}$$

## Literals

Literal types are subtypes of their base types:

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

## 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 must be required in  $\tau_1$  with a subtype. For all optional fields in  $\tau_2$ , they must exist (either as required or optional) in  $\tau_1$  with a subtype.

$$\frac{\begin{array}{l} \forall f_i \in \tau_2.\text{required}, f_i \in \tau_1.\text{required} \wedge \tau_1.f_i <: \tau_2.f_i \\ \forall o_i \in \tau_2.\text{optional}, o_i \in (\tau_1.\text{required} \cup \tau_1.\text{optional}) \wedge \tau_1.o_i <: \tau_2.o_i \end{array}}{\tau_1 <: \tau_2}$$

This ensures width subtyping (extra fields allowed in  $\tau_1$ ) and depth subtyping (covariant field types), while preventing incompatible types for shared fields.

**Soundness note:** The second condition is critical for soundness with open objects. Without it,  $\{\} <: \{x? : \text{string}\}$  would hold, but an object satisfying  $\{\}$  could have  $\{x : 42\}$  which violates  $\{x? : \text{string}\}$ .

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:

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

## Literals

`null`  $\mapsto$  `null`, `bool`  $\mapsto$  `bool`, `number`  $\mapsto$  `number`, `string`  $\mapsto$  `string`, `any`  $\mapsto$  `any`, `never`  $\mapsto$  `never`

## Objects

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

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

## Tuples

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

## Arrays

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

## Unions and Intersections

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

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):

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

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

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

## 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 be 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  $x.\text{property}$  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  $\text{fieldType } \tau$ , we need only work with known  $\{f : \sigma\}$ . In this case, we only need to prove

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

### 7.2.1 More Details

Suppose that we have a JSON type  $\tau$  and consider a key  $k$ . We will first consider the case when we are looking for a field we know exists.

Suppose that  $\sigma = \tau[k]$ . We can compute  $\sigma$  as following:

- for objects, we can lookup  $\sigma$  directly in the required and optional fields.

If a field is both required and optional, then we take the intersection of those types.

- for unions, we need the key to exist in both sides, and then we can take the union of both sides.

- for intersections, we take the type from both sides, and then intersect them. If they only exist on one side, then that's fine.

These are true, since if  $j : \{k : t, \dots\}$ , then  $j[k] : t$ . If  $j : t_1 \mid t_2$ , then  $j : t_1$  or  $j : t_2$ , so by induction  $j[k] : t_1[k]$  or  $j[k] : t_2[k]$ . If  $j : t_1 \& t_2$ , then  $j : t_1$  and  $j : t_2$ . If  $k \in t_1$  and  $k \in t_2$ ,  $j[k] \in t_1[k]$  and  $j[k] \in t_2[k]$  respectively.

So the important theorem of  $k \in \tau$  is that  $j[k] : \tau[k]$ . Now  $j[k]$  contains two facts. One,  $j[k] ? =$  some  $v$ , and two  $v : \tau[k]$ .

For optional access, then we look up in opt, but otherwise the process is identical.

### 7.3 Narrowing

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 control-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.

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

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.