1 Points

- Criterion for config language design
- Exposition of designed language
- JSON examples and comparison with JSON Schema
- Future work: definitional equality, simple proof search, better JSON embedding, more examples

2 Story

Once apon a time, applications had configurations, but the configurations were messy and ill defined, having to choose between static languages with stronger guarantees, or computational languages with less guarantees. Enter our hero, Type Theory, who gives us a small but powerful language that lets us combine the static of power of JSON Schema with the computational power of Lua! With this new found power, order can be brought to configurations by statically defining the space of valid configurations for an application and then using the power of computation to define specific configurations in a concise way. And so, using this approach, applications and their configurations lived happily ever after.

3 Introduction

As programmers we have this vague notion of a configuration, the sort of knobs and dials that can be adjusted to change the behaviour of an application without actually changing the application itself. In this sense, the configuration for an application is a set of parameters, almost as if the application was a function with these parameters as its domain. A common problem with a configuration is that what exactly consists of a valid configuration is ill defined. Another problem which goes along with this is that the configuration is hidden inside the source code.

Try to backup these claims with evidence

4 Existing Configuration Languages

While many different configuration languages exist, two stick out as waypoints in the possible design space. JSON, which exemplifies a language with no computation but with strong static guarantees when combined with JSON Schema, and Lua which can express arbitrary computations but gives almost no static guarantees. When looking at current configuration languages, one has to make a decision on the line defined by these two endpoints. Either choosing a language which lets one precisely define what a valid configuration is, but without the ability to create abstractions and express higher order configurations, or

a language where these two are readily done but without the capacity to give precise static guarantees.

Expound upon these languages some more, give some examples

5 Our Solution

We propose a configuration language which doesn't fall on the previously defined line, but which combines the best of both endpoints, chiefly the ability to precisely define the space of valid configurations while using computation to concisely pick a point in this space. Our language, is that of Martin-Löf type theory (MLTT) with the usual type formers, along with non-dependent record types. We give an implementation of a type checker along with an embedding into Racket so that an application written in Racket can easily import a configuration

6 Examples

Our first example exemplifies the ability of MLTT to express arbitrary static constraints. In Figure 1 we show an example taken from the JSON Schema website which describes a space of JSON values representing an address. The address can contain seven fields, only three of which are actually required to be present. However, another condition is present where if either the "postOfficeBox" or "extendedAddress" fields are given then the "streetAddress" field must also be present.

reference

In Figure 2 we give a translation of this schema into MLTT. We split-up the schema into two parts. First the data, which we represent as a record where each required field is a string, and the other fields are maybe a string. The second part is a predicate on the first part which captures the "dependentRequired" part of the schema. We encode this validity condition using the "So" type which is a predicate on a boolean asserting that it is true. Its only element is called "Oh" of type "So true". We then use the "SuchThat" type which takes a type a predicate on that type, which can be thought of as a dependent sum where the second part is erased at runtime. Finally we give an example instance of an address. The most interesting part here is that since the data part of the Address only contains normal forms, the validity conditions can be fully

```
{
  "$id": "https://example.com/address.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "description": "An address similar to http://microformats.org/wiki/h-card",
  "type": "object",
  "properties": {
    "postOfficeBox": {
      "type": "string"
    },
    "extendedAddress": {
      "type": "string"
    },
    "streetAddress": {
      "type": "string"
    },
    "locality": {
      "type": "string"
    },
    "region": {
      "type": "string"
    },
    "postalCode": {
      "type": "string"
    },
    "countryName": {
      "type": "string"
    }
  },
  "required": [ "locality", "region", "countryName" ],
  "dependentRequired": {
    "postOfficeBox": [ "streetAddress" ],
    "extendedAddress": [ "streetAddress" ]
  }
}
```

Figure 1: JSON Schema Address example

computed automatically and so we simply need to give "Oh" as the element of "AddressValid".

Our second example shows the ability of MLTT to encode more dynamic configurations than simple records. We give a very minimalist version of the "Shake" build language from

insert shake reference

7 Future Work

In which I expound upon how proof search would be nice so we don't have so many explicit type applications, and that definitional equality is annoying and currently doesn't let us reason about primitives internally to the language.

```
;; Boolean implication
(def [=> : (-> Bool Bool Bool)]
  (λ a b
    (ind-Bool lzero a
      (λ _ Bool)
      b
      true)))

(def [AddressData : (Type 0 lzero)]
  (Rec
    [postOfficeBox (Maybe String)]
    [extendedAddress (Maybe String)]
    [streetAddress (Maybe String)]
    [locality String]
    [region String]
    [postalCode (Maybe String)]
    [countryName String]))

(def [AddressValid : (-> Address (Type 0 lzero))]
  (λ a
    (So
      (and
        (=> (some? String (! a postOfficeBox))
            (some? String (! a streetAddress)))
        (=> (some? String (! a extendedAddress))
            (some? String (! a streetAddress)))))))

(def [Address : (Type 0 lzero)]
  (SuchThat AddressData AddressValid))

((An AddressData AddressValid
   (rec
     [postOfficeBox (Some String "123")]
     [extendedAddress (None String)]
     [streetAddress (Some String "456 Main St")]
     [locality "Cityville"]
     [region "State"]
     [postalCode (Some String "12345")]
     [countryName "Country"])
   Oh)
 : Address)
```

Figure 2: MLTT Address example