Configuration languages - A comparison of Dhall and CUE

Olof Gren, Lukas Gutenberg
olofgr@kth.se, lukasgut@kth.se

April 2022

Contents

1 Introduction

Imagine that you have created a program. This program makes use of several different dependencies to work and has several different input parameters, therefore it has a configuration file, in JSON or YAML, to keep track of everything. Then the unthinkable happens and a bug has appeared but it is nowhere in the code, it is somewhere in the configuration. Now you have to find, in your thousands of lines long config file, where the bug is with a 10 pages long error message to go from. The worst part is that you have no way to automatically error check the file which means you have to go through the entire thing manually, even for a small typo. Wouldn't it be great if we had a way to automatically find errors in your config file while you are writing it?

1.1 Configuration languages

Configuration languages are, for the purposes of this paper, languages that compile code into data for some other program to use as configuration files. There are not many languages for this purpose, but both CUE and Dhall fulfill these criteria, being able to compile to both JSON and YAML. With the rise of the YAML engineer as a profession[8], the right tools of the trade are following closely behind.

The term configuration language has also been used differently in the past [1], as short for configuration programming languages, but since both Dhall and CUE describe themselves as configuration languages, so shall we.

1.2 Purpose

Since there are several configuration languages available today, anyone who wishes to use one has to choose, something that can be hard as the difference between them can be large. In this paper we want to give an overview of the differences between two different configuration languages: CUE and Dhall. For this we will present what we think are desired properties for a configuration language, so we can evaluate how CUE and Dhall fulfill these properties and see which language is better at what.

1.3 Desired properties

These are properties that we consider to be of key importance for any configuration language. Some of these properties can be considered more important than others which will be discussed later.

1. Backwards compatibility
2. Clarity
3. Ease of use
4. Error Identification

5. Schema validation
6. Scaling
7. Don't repeat yourself

Backwards compatibility is often important, as it means we can use our old configuration files as a basis for our new configuration in our system. It also allows adding language features a little at a time, adding them as they are needed.

Another important point is clarity - is it possible to read the configuration source and understand what it meant? It would be optimal here for someone with no prior experience with using the language to still be able to read and understand the files.

It should, of course, be fairly easy to use the language, both in terms of syntax and complexity. There is, however, a trade-off of how easy it is to use and how much we can gain from it. If the program is very easy to use it probably does provide much to prevent mistakes from happening.

By Error identification we mean that the configuration language should not just reject an invalid configuration but also be able to give some hints on how to debug it so that the error can be found.

We also think that it is important for the configuration language to represent any given schema. If it is known that the configuration must follow some specified schema, the configuration language should be able to represent the schema and be able to validate if the configuration follows the schema.

A configuration language should work well no matter the size of the project, from small-scale single-person projects, to large-scale multi-billion systems. We should expect at most linear growth of complexity and performance cost.

Finally, a configuration language should offer powerful features to generate a lot of output data so that the programmer does not have to write a lot of text. If the language can generate text for you there is a lower chance of something going wrong.

2 Dhall

Dhall is, as its authors call it, a programmable configuration language. When an expression is evaluated, it reduces it down to JSON or YAML. Some systems can also accept Dhall directly [3]. Dhall is a total functional language - it guarantees that typechecking can be done in finite time, and that programs that pass typechecking will also run in finite time [9, 7]. This means it unlikely that evaluating a Dhall file will take much time, and that Dhall is not Turing complete.

Dhall has a syntax inspired by JSON, but it is not a superset of nor a subset of JSON. It is strongly typed, and one of the design goals is to make configuration files more strongly typed to prevent mistakes [2]. Invalid states should also not be valid Dhall, and writing a well-typed correct configuration file should be the path of least resistance.

2.1 Features

Dhall is fairly minimalist language, but it supports functions and bindings to minimize code repetition and type checking and schema validation to prevent errors.

2.1.1 Type checking

Typechecking can catch common errors automatically.

```
[
    { name = "Bill"
    , amount = 3664
    , email = "billDoe@emailprov.com"
    }
    , { name = "Alice"
    , amount = 2089
    , emal = "alicedefaoe@emailprov.com"
    }
]
```

This listing is not valid Dhall. There is a typo (email has been spelled emal once), which means the two list elements do not have the same type. These checks are part of the language, strong typing is enabled by default in Dhall.

2.1.2 Schema validation

Schema validation in Dhall can be accomplished by doing an explicit type check of the data.

```
[
    { name = "Bill"
    , amount = 3664
    , email = "billDoe@emailprov.com"
    }
    , { name = "Alice"
    , amount = 2089
    , email = "alicedefaoe@emailprov.com"
    }
]: List {name : Text, amount : Natural, email : Text}
```

Here we check that the objects in our list are not just of the same type, but that the objects have a specific format. This means schema validation is the same as explicit typing in Dhall.

2.1.3 Bindings and functions

To further prevent mistakes due to repetition Dhall offers let-bindings. This allows to write once, and reuse common parts of the code.

```
let Record = { name : Text
, amount : Natural
, email : Text
}
let EmailDomain = "@emailprov.com"
 in
[
    { name = "Bill"
    , amount = 3664
    , email = "billDoe" ++ EmailDomain
    }
    , { name = "Alice"
    , amount = 2089
    , email = "alicedefaoe" ++ EmailDomain
    }
]: List Record
```

In this listing the type of our records has been bound to a name , and we can reuse a text string for completing the email addresses.

Dhall also offers functions to complete records and reduce typing.

```
let Record = { name : Text
, amount : Natural
, email : Text
}
let EmailDomain = "@emailprov.com"
let genRecord = \(n: Text) -> \(a : Natural) -> \(user : Text) ->
{ name = n
, amount = a
, email = user ++ EmailDomain
}: Record
 in
[
    genRecord "Bill" 3664 "billDoe"
    , genRecord "Alice" 2089 "alicedefaoe"
]: List Record
```

This allows us to make a constructor for our data to shorten the input data so there is no need to repeat ourselves when entering records. The function syntax in Dhall uses currying for multiple arguments.

3 CUE

CUE is a Turing-incomplete data constraint language and was created with a focus on large-scale engineering and automation [4, 6, 10]. CUE describes itself as a superset of JSON where any valid JSON file is also a valid CUE file

and it is also capable of converting CUE files to *and* from JSON and YAML files. Furthermore, it also supports interactions with Go, OpenAPI, and Kubernetes. CUE has put extra effort to work well with Kubernetes and there is a tutorial specifically to show how CUE can be used with it [5]. Apart from creating configuration files CUE is also capable of data validation, data templating, querying, code generation, and scripting. We will focus on the configuration part in this text.

3.1 Features

One of the most important part of CUE is that it is associative, commutative, and idempotent, making it so that no matter the order things are written in you always get the same result. It also allows for assigning a field the same value multiple times, but it is not allowed to change it to a new value. CUE emphasises that it merges the notion of schema and data, making it so that the same definition can be used for validating data and act as a template at the same time, allowing us to create structure, constrain it, and define it in one *or* multiple files:

```
book: {
title: string
year:  int
available:  int
}
```

```
book: {
title: string
year:  <1900
available:  >0
}
```

```
book: {
title: "Emma"
year:  1815
available:  3
}
```

This allows us to create an easy to organize structure of constraints and definitions to prevent assigning bad values or names to data.

Another important feature of the constraints is their ability to reduce boilerplate, i.e. repeating code, by defining the structure of fields:

```
jobs: {
  foo: acmeMonitoring & { /* ... */ }
  bar: acmeMonitoring & { /* ... */ }
  baz: acmeMonitoring & { /* ... */ }
}
```

```
jobs: [string]: acmeMonitoring
```

```
jobs: {
    foo: { /* ... */ }
    bar: { /* ... */ }
    baz: { /* ... */ }
}
```

The above code, taken from [4], shows how *acmeMonitoring* can be moved out of the expression by making the constraint that it must be included, this implicitly makes CUE include it if it isn't already.

CUE also has support for mathematical expressions with everything from addition and multiplication to more advanced concepts such as if statements and even regular expressions. This means that instead of writing a value in a field you can for example multiply the values of two other fields or even do some kind of expression on a list:

```
import "list"


Value: {
    list: [1, 2, 3, 4, 5]
    sum: [for x in list {x*2}]
}
```

The above expression gives the resulting JSON below.

```
{
    "Value": {
        "list": [
            1,
            2,
            3,
            4,
            5
        ],
        "sum": [
            2,
            4,
            6,
            8,
            10
        ]
    }
}
```

4 Comparing CUE and Dhall

4.1 Writing a cue example in Dhall

Since both Cue and Dhall compile to JSON (and YAML) it is revealing to show the design differences between the two configuration languages by seeing how to make source code in the two languages that compile to the same JSON. The example below is taken from the CUE examples [10]. We have written a Dhall file with the same JSON output, and a similar schema.

CUE:

```
package hierarchy

import "list"

#Schema: {
hello: string
life:  int
pi:    float
nums: [...int]
struct: {...}
}

#Constrained: #Schema & {
hello: =~"[a-z]+"
life:  >0 | *42
nums:  list.MaxItems(10)
}

Value: #Constrained & {
hello: "hello"
life:  35
pi:    3.14
nums: [1, 2, 3, 4, 5]
struct: {
a: "a"
b: "b"
    }
}
```

Dhall:

```
let Schema =
{ hello : Text
, life : Natural
, pi : Double
, nums : List Natural
, struct : { a : Text
    , b : Text
    }
}
let leq = \(i : Natural) -> \(j : Natural) ->
 Natural/isZero  (Natural/subtract j i)
let constrainone = \(s : Schema) ->
 leq 1 s.life
let constraintwo = \(s : Schema) ->
```

```
 leq (List/length Natural s.nums) 10

let value =
{ hello = "world"
, life = 42
, pi = 3.14
, nums = [1, 2, 3, 4, 5]
, struct = { a = "a"
    , b = "b"
    }
} : Schema
let asserts = assert : constrainone value && constraintwo value === True
in
value
```

Quite a few things needed to be changed for the Dhall version. The constraints on the text field "hello" could not be maintained in Dhall. As the authors of Dhall have noted [2], there is very little text manipulation in Dhall by design. The intended way to use Dhall is to use enums as the compute values, and have a function to translate them into text. Also problematic is Dhall's lack of comparison operators. This makes writing conditions for variables more complicated.

A third interesting comparison is that the Dhall file is longer and has more boilerplate. From this experience it can be concluded that Dhall does not support evaluating any arbitrary schema as well as CUE does. It also shows that Dhall is not a language that is ready for general use. There are several features that are still missing for the language to not feel unnecessarily limiting.

4.2 Comparision in desired properties

By comparing Dhall and CUE with regards to our requirements we can see that CUE is superior in many fields. CUE is backwards compatible with YAML and JSON while Dhall needs new implementations. It also has clearer syntax that is easier to read and follow. Furthermore, CUE is also easier to use but in return it is not as strict which risks lowering the quality of the error identification. We also noticed that CUE has a larger support for implementing any desired schema compared to Dhall, especially in strings. Finally, we believe that they both have the potential to scale fairly well and both have support to prevent boilerplate but in different ways with no clear winner.

5 Conclusion

According to our experience in trying to write CUE and Dhall, we think that CUE is in most aspects a better language than Dhall. The only point where Dhall compares favourably and not neutrally or negatively with CUE, is in error identification. Thus we think that Dhall may be a better option than CUE only

when backwards compatibility is not important, there is ample time to learn a new language just for configuration, and being able to check arbitrary schemas is less important than finding common errors quickly and thoroughly.

References

[1] Judy M Bishop and R Faria. "Languages for configuration programming: a comparison". In: *IEEE Trans. Soft Eng. to appear, also UP CS Tech Report* 94.04 (1994).

[2] *Design Choices*. Apr. 2022. URL: https://docs.dhall-lang.org/discussions/Design-choices.html.

[3] *Dhall in production*. Apr. 2022. URL: https://docs.dhall-lang.org/discussions/Dhall-in-production.html.

[4] Google LLC. *CUE Documentation*. Apr. 2022. URL: https://cuelang.org/docs.

[5] Google LLC. *Kubernetes tutorial*. Apr. 2022. URL: https://github.com/cue-lang/cue/tree/v0.4.2/doc/tutorial/kubernetes.

[6] Google LLC. *The CUE Data Constraint Language*. Apr. 2022. URL: https://github.com/cue-lang/cue.

[7] *Safety Guarantees*. Apr. 2022. URL: https://docs.dhall-lang.org/discussions/Safety-guarantees.html.

[8] *Seriously, are we all yaml engineers now?* Apr. 2022. URL: https://www.reddit.com/r/ProgrammerHumor/comments/9thtqf/seriously_are_we_all_yaml_engineers_now/.

[9] D. A. Turner. "Total Functional Programming". eng. In: (2004). ISSN: 0948-695X.

[10] Tony Worm et al. *Learn you some CUE for a great good!* Hofstadter Inc, Apr. 2022. URL: https://cuetorials.com.