University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Stanisław Barzowski
Student no. 359012

Improving Jsonnet ecosystem

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr hab. Aleksy Schubert prof. UW
Instytut Informatyki

September 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Jsonnet is an open source, purely functional, dynamically typed programming language, designed for programmatically creating complex configuration files. In this thesis I summarize my work on tools and libraries for Jsonnet.

In the first part of the thesis I describe my contributions to creating and improving a new implementation of the interpreter, which later became the primarily recommended version.

In the second part, I introduce a new tool for finding mistakes in Jsonnet programs (the linter). It performs multiple checks, but its most important functionality is finding the descriptive types which can be used for identifying the expressions which always fail. A type discovery algorithm specifically designed for this case is presented.

In the third part I present two new, small utility libraries for Jsonnet. The first library, *jsonnet-modifiers*, is designed to provide powerful convenience functions for modifying complex, nested values. The second library, *jsonnet-parser-combinators*, is an implementation of parser combinators. Both of them are publicly available.

Keywords

Jsonnet, dynamically typed languages, domain specific languages, interpreter, type discovery, type inference, static analysis, linter

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatyka

Subject classification

Software and its engineering — Domain specific languages

Tytuł pracy w języku polskim

Ulepszanie narzędzi dla jezyka Jsonnet

Contents

Chapter 1

Introduction

Jsonnet is a specialized programming language designed for generating configuration data. The most typical use case involves configuring multiple services in a *Kubernetes*[1] cluster, but Jsonnet is not tied in any way to a specific application and can be used to produce configuration files in any form. It is especially helpful for centralizing configuration, i.e. using the same information to configure multiple programs.

The language was originally designed by Dave Cunningham, who also created the first implementation which was made public in 2014 [6, 9]. My involvement with the project began in mid-2017, when I brought the new interpreter implemented in Go to feature parity with the C++ implementation. By October it was already used in third party projects such as *ksonnet* [26]. Since then the new implementation is being continuously improved and it became the primarily recommended version. Besides working on a new interpreter, I was also involved in improving the old implementation, automatic code formatter, standard library and documentation. Recently I created two new Jsonnet libraries and a new tool — a linter, that is a tool which automatically detects erroneous or otherwise undesirable code and provides clear feedback to the user.

The rest of this chapter describes Jsonnet in more detail and provides references for finding additional information about the language. Next, in the second chapter, I present the work on the new interpreter, its scope, challenges and solutions. After that, in the third chapter, I introduce the linter. There, I describe the goals and design decisions as well as explanation of the inner workings of the new tool, with the focus on the custom-designed algorithms. The fourth chapter briefly describes the recently created utility libraries for Jsonnet. Finally, in the last chapter I summarize the work done.

[1] *Kubernetes* [20] is a very popular [4] open source container orchestration system. It originates from Google and builds upon their experiences with *Borg*, the internal cluster management system described in [27],

Its primary principle of operation is that the user describes the desired state of the entire cluster, consisting of multiple machines and services and the system automatically tries match it, even in case of hardware failures. *Kubernetes* can gracefully handle rollouts on the massive scale and it can make increasing the number of instanced of a given service as easy as changing one number in the configuration file. From the perspective of this thesis, its most important aspect is that it is configured using YAML and that its configuration files are often very large and repetitive. Jsonnet provides a good way to build such configuration files programmatically. Interestingly, for configuring *Borg* internally a Turing-complete programming language, BCL, is used. I believe this solidifies the case for Jsonnet in *Kubernetes* context. BCL also happens to be one of the inspirations for Jsonnet.

1.1. The case for a universal configuration language

It is obvious that some software requires extensive configuration. The most relevant examples for the case of Jsonnet are (1) deployment tools such as *Terraform*, (2) cluster management, especially when combined with service-specific configuration as is common with *Kubernetes* and (3) complex dashboards such as *Grafana*.

There is more than one way to approach the complex configuration needs. For example, it is tempting to start with what is supposed to be a simple declarative format, then add more and more features as they are requested. The expansion may start with adding references or variables to avoid duplication. This in turn makes conditionals quite useful. The result of fulfilling the needs as they arise is often a full, Turing-complete programming language, but designed in an ad hoc manner. This is likely to mean that the language does not follow the best practices of language design. Moreover the effort is largely duplicated between projects.

Jsonnet and similar software provide an alternative solution. The programs may consume a very simple hierarchical format for configuration, such as JSON [7], which is generated by a separate program. It may be implemented in a completely transparent way to the end user, i.e. the program may use Jsonnet sources as its configuration and evaluate it to JSON for consumption.

This kind of configuration architecture has another very important advantage – it is possible to easily centralize configuration parameters across many programs. It is for example possible to use common information sources for generating configuration for *Prometheus* to collect metrics and for *Grafana* to display them. This allows maintaining a single source of truth, even in very complex cases.

Additionally the organization can enforce the use of a single configuration language for all their internal needs. This allows them to largely eliminate the effort required for learning a separate configuration language for each tool.

It is worth noting separately, that configuration languages are not limited to producing just one format. Jsonnet outputs JSON by default, and there are some convenience features for YAML, but it can produce the data in any format. This allows using it to configure all kinds of existing applications, even those which actually define their own format.

1.2. Quick introduction to Jsonnet

Jsonnet is a dynamically typed, purely functional, side-effect free[2] programming language.

[2] There are no side-effects whatsoever in the semantics of the language. All Jsonnet programs are just computing a value. See the further sections for a detailed rationale.

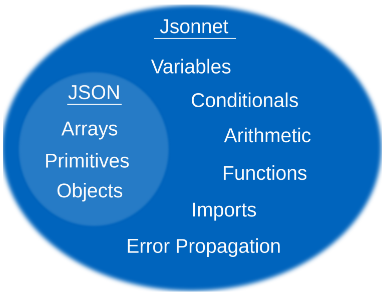

Figure 1.1: A visualization of Jsonnet features from the official website [9].

Jsonnet programs consist entirely of expressions and in fact the whole program in fact is just one big expression. The simplest form of an expression is JSON, unsurprisingly evaluating to exactly the same JSON structure.

```
{
    "null": null,
    "string": "any_utf8_string",
    "number": 42,
    "bool": true,
    "array": [1, 2, "test"],
    "object": { "a": 1, "b": 2 }
}
```

Of course Jsonnet allows not only providing the data directly, but also calculations:

```
    [
        2 + 2,
        [1, 2, 3][2],
        {a: 1, b: 2}.b,
        if 2 + 2 == 5 then "?" else "ok",
        std.sort([1, 3, 2]),
    ]
```

Variables can be introduced using `local` expressions, which have similar semantics to Haskell's `let`. The variables are best thought of as simply giving an expression a name. They are immutable and lazily evaluated. They can be self-recursive and mutually-recursive within one `local` expression.

```
    [
        local two = 2; two + two,
        local arr = [1, arr]; arr[1][1][1][0],
        local a~= [b[0], 1], b = [2, a[1]]; a,
    ]
```

The objects in Jsonnet are more than simple mappings – they are objects in the object-oriented programming sense as well. This is achieved in a somewhat unusual way, based exclusively on mixins. There are no classes, only objects.

1.2.1. Operator "+" and inheritance

Each element of the inheritance chain is a valid object on its own and they are only combined using an overloaded operator `+`. So for example adding two objects like `A + B` results in an object which results from overriding A with B. Notably, the inheritance operation is associative, so it is reasonable to talk about chain of inheritance rather than tree of inheritance.

```
    local A = {
        x: 1,
        xRef: self.a,
    };
    local B = {
        x: 2,
        xSuperRef: super.a,
        y: self.xRef,
    };
    A + B,
```

1.2.2. Input data

Jsonnet does not allow IO operations such as opening and writing to files or accessing the network for reasons outlined in the further sections. There are only a few specific ways in which Jsonnet may take input from "the outside world", namely imports, external variables and top-level arguments. Imports allow splitting a Jsonnet program into multiple files. The imported paths must be string literals (no calculations are allowed), so it is statically possible to gather all the files directly or indirectly imported. External variables are actually not variables in the same sense as those introduced by `local`. These are global values accessible in the entirety of the Jsonnet program, including imported code. They have their own namespace and they are available using `std.extVar` function. The third mechanism, top level arguments, allows calling a Jsonnet program evaluating to a function with externally provided arguments. Importantly, all three methods are defined in a way independent from the underlying operating system. Even imports can load files from user-defined sources rather than the file system.

1.2.3. Getting more information about Jsonnet

Of course, this a very minimal presentation of Jsonnet features. Further parts of this work relate to the implementation of the tools for Jsonnet and necessarily dive into the technical details of the language. Having in mind that Jsonnet is still a fairly niche technology, I will try to briefly explain the relevant concepts and design decisions whenever they appear. These explanations would necessarily need to be very brief, so for the readers unfamiliar with Jsonnet, I recommend the tutorial on the Jsonnet website (see [25]). Conveniently, the tutorial also includes interactive examples, which can be modified and used as a playground for using Jsonnet without the need of installing it locally. It may also be helpful to have a look at some real world example such as grafonnet-lib library [5].

1.3. Core ideas

Jsonnet was designed with a number of explicit design goals in mind. Below, I included the summary of the goals from the official website:

Hermeticity: Code may be treated as data. The same JSON should be generated regardless of the environment, i.e. without non-deterministic or system-dependent behaviors.

Templating language: Code should be interleaved with verbatim data so that it is easy to maintain the two in synchrony. The success of templating languages proves their effectiveness.

Variants: It should be simple and intuitive to derive variants of any existing configurations that override attributes for special or ad-hoc purposes.

Modularity: As configurations grow, it must be possible to manage the complexity using standard techniques for programming in the large. Configurations may span many files and many developers. Errors should provide stack traces that describe the nested context.

Familiarity: Raw data should be specified as JSON. Computation constructs should behave in standard ways and compose predictably.

Powerful yet simple: Trivial cases should be trivial. More complex cases should be approachable, with localized additional complexity. Everything must be possible, yet the language must still have a small footprint for learning and future tooling.

Wide scope: The same language should be able to configure anything. Ideally, all components of a system should be managed via a well-maintained centralized configuration, written in a common language.

Formal rigor: There must be an authoritative specification that is complete and simple enough to understand, and a comprehensive set of tests. This allows new implementations to be developed without compatibility hurdles. The language should not be defined by its first implementation.

(from the official website of Jsonnet [14])

Most of these traits are self-explanatory, but two important ones are worth discussing in more detail.

1.3.1. About hermeticity

Jsonnet programs have no side effects and do not depend on the environment in any way. So for example it is not possible to read arbitrary files in the file system. Jsonnet programs can only access data which was specifically passed to them. All input mechanisms are well-defined and independent from the operating system concepts.

This is of course a big limitation and the users sometimes request providing ways to dynamically read files. There are, however, great benefits to the hermeticity. It eliminates the whole class of problems caused by differences in the setup.

There are also less fundamental operational benefits. So for example programs and Jsonnet can always be locally debugged and tested with confidence that the result will be the same on any machine. Within the language, error handling is greatly simplified because with no user-controlled IO, errors are either bugs or failures of the external system to provide data to Jsonnet. In both cases the calculation can be simply aborted.

1.3.2. About formal rigor

Jsonnet has a much more precise specification than most languages. The specification contains almost[3] complete big step operational semantics for the language. Together with hermeticity this means that Jsonnet programs are very close to being functions in the mathematical sense.

1.4. Popularity of Jsonnet

Jsonnet is definitely a niche technology compared to any mainstream general purpose languages. Nevertheless, it seems to be one of the strongest contenders in the area of programming languages for building configuration. It is hard to precisely quantify the popularity of

[3] The most important area in which the specification is not fully mathematically precise is floating point arithmetic where it defers to IEEE754 and to the standard meaning of arithmetic operations. The specification for importing other files is quite precise, but also is not fully formal.

Jsonnet, or any other configuration language for that matter, because configuration is typically not shared outside of the organization. It is also not very likely to be talked about publicly, because from the organization perspective it is just an operational detail[4].

However, there are still some indirect ways in which we can observe interest in the project. On *Github*, the developers can mark a repository with a star to express interest. The repository google/jsonnet has 3369 stars[5] while google/go-jsonnet has 513. Slack channel `#jsonnet` on *Kubernetes*.slack.com has 535 members and the mailing list has 164.

There are a few companies which share their experiences with Jsonnet publicly. The best example is Databricks which has a complete configuration written in Jsonnet, containing over 100 000 lines of Jsonnet code [28].

While the number of users is difficult to pinpoint, there is clearly a sizeable audience which can benefit from the improvements described in this thesis

1.5. Where to find the project and my contributions

The work described in this thesis is either already in the main branch or currently waiting for review. The contributions can be found in the following repositories:

- Go implementation of Jsonnet interpreter and the linter:
  https://github.com/google/go-jsonnet
- C++ implementation of Jsonnet linter, shared resources and documentation:
  https://github.com/google/jsonnet
- A new library for modifying deeply nested values:
  https://github.com/sbarzowski/jsonnet-modifiers
- A new library for parsing in Jsonnet:
  https://github.com/sbarzowski/jsonnet-parser-combinators

[4] Ideally configuration within a project or organization would be a boring detail, it would not require much thought and it would work effortlessly with any software.

[5] *Github* also provides "used by" metric, but it is not actually very meaningful in our case. Effectively, it accounts for repositories installing Jsonnet as a Python package. This is a somewhat unusual (but officially supported) way of using Jsonnet. On the other hand it also counts many repositories which actually have little to do with Jsonnet, because some fairly popular packages such as *AllenNLP*) depend on Jsonnet for optional functionality.

Chapter 2

New Interpreter in Go

The work on the new interpreter in Go was started by Joe Beda[1], who built a mostly complete lexer and parser for Jsonnet in Go. I took over from this stage and I brought the implementation into a complete specification-compliant state. This new implementation is being continuously improved and it is gaining popularity ever since.

There are many advantages of the new implementation:

- Go is a very popular language among the projects which are most likely to be benefit from Jsonnet, i.e. projects related to deployment of services. A native implementation is easier to use with other Go code. It may also make it easier for the users to contribute to the project.
- Go is a safe language, i.e. ordinary code cannot cause undefined behavior[2]. This eliminates some classes of security issues. Additionally, it also means more context (usually full stack traces) available in crash reports.
- Go has a high-quality garbage collector, allowing a more straightforward implementation. In C++ a custom, simple Mark-and-Sweep garbage collection was used and it was necessary to carefully manage the references to Jsonnet values.
- A reimplementation is a chance to improve some aspects of the design. For example, providing efficient, compiled implementations of standard library functions is much easier in the new implementation.
- A reimplementation is an opportunity to very thoroughly check the specification and the test suite.

Even taking all these benefits into account, creating a new interpreter may be seen as very controversial decision. Such effort usually carries a risk of fragmenting the community and splitting the development effort. Fortunately, it was not nearly as problematic as in a typical case of reimplementation. There were a few reasons for that:

- The scope of the interpreter is mostly fixed. The language and its standard library are slowly evolving, but these changes are usually quite easily replicated across implementations.

[1] Joe Beda is best known for being one of the few people who started the Kubernetes project.

[2] It is possible to write unsafe code in Go, but since it is limited to the cases where it actually necessary, the attack surface is much smaller.

- A very precise specification is available in the form of formal big step operational semantics. Many other languages which are specification-defined have multiple compatible[3] implementations. Examples include some of the most popular languages: C, Java, C++.
- There exists a test suite which checks the behavior in detail.
- The amount of work required to get the new version to the usable state was relatively small. It took only a few person-months to make it good enough to become the new primary focus for development. However both versions still need to be maintained and kept up to date with the changes to the language, which requires a lot of effort.. The performance improvements and other deeper changes are at this point mostly limited to the Go implementation.

Further in this chapter I describe some of the decisions and issues related to the new implementation. For convenience I will use the term *cpp-jsonnet* for the older implementation in C++ and *go-jsonnet* for the newer implementation in Go[4].

2.1. Status of the new interpreter and the migration

2.1.1. Lack of error parity

Jsonnet generally strives for a fully byte-for-byte specified output, leaving no room for implementation differences. This is important for a configuration language, where predictability is perhaps the most important property. In particular the output should not depend in any way on the environment in which it is executed. This also extends to replacing the interpreter. Due to the precise specification and a healthy test suite, it is feasible to maintain multiple fully compatible implementations. The test suite also covers error cases, so it may seem natural to make sure the errors are also the same.

However, the specification does not describe the precise behavior in case of any errors. In the semantics, no rule is provided for handling errors, effectively making the run of the program invalid and leaving the details for the implementation. This is practical, because all errors are fatal in Jsonnet[5].

In the early stage of the work on the interpreter in Go it was unclear whether we should try to make the errors exactly the same as well. We eventually decided against it for a few reasons. The most important one was that stack traces (which were a part of the output in case of an error) exposed implementation details, such as the internals of the standard library, which make keeping the both versions synchronized difficult and limiting. The details of caching behavior, the precise order of evaluation also affect the stack traces. In addition to that, keeping both versions the same would make improving the presentation of the error messages more difficult.

2.1.2. Effect on the code formatter

The original C++ implementation, in addition to the interpreter, contains a code formatting tool.

[3] Usually there are implementation differences, but for most kinds of programs it is possible to use only the standard parts which are supported by all implementations and most programs can be reasonably expected to require little or no porting effort.

[4] These shorter forms are commonly used in the Jsonnet community.

[5] Having only fatal errors is reasonable, because Jsonnet by itself has no IO and therefore it is not exposed to any external errors. If errors arise during imports, it is seen as a problem outside of the responsibility of the Jsonnet program.

Maintaining compatibility between *go-jsonnet* and *cpp-jsonnet* for the interpreter is largely based on the precise specification. The situation of formatting is different, because it is implementation-defined and at the same time it is beneficial that all users format their code in the same way[6], so duplication would be highly problematic.

We decided that the best way forward is to keep the additional tools usable separately from the interpreters. It was not obvious how to handle the formatter, since it was already available in the main command (as `jsonnet fmt`) and formatting functions were available in `libjsonnet.h` and there was a concern of breaking the setup of existing users. The problem was resolved in 2019 when Dave Cunningham separated formatter into its own tool [22] and library [18]. Any new tools, including those which are based on *go-jsonnet* such as the linter, will be designed to be usable separately from the interpreter.

2.1.3. C bindings

All library interfaces for *cpp-jsonnet* are based on C-compatible library. In particular, official C++ and Python public APIs and the `jsonnet` command are implemented as a wrapper around C API. There also exist unofficial bindings for other languages including Lua [23], JavaScript [12], PHP [24], Ruby [10] and Rust [15].

It is important for the new interpreter to provide a compatible C interface to become a full drop-in replacement. I implemented the most important parts of the C library using *go-jsonnet*. The current version of the C API for *go-jsonnet* does not include the support for import callbacks functions and native functions.

When using Go, creating C libraries is possible by with *cgo*, which allows exposing C functions, values and types to Go and exporting Go functions as C functions. Predictably creating such an interface can be sometimes problematic when it comes to memory allocation as Go uses a garbage collector and in C the memory is manually managed. To make interoperation possible *cgo* defines specific rules concerning pointers:

> Go code may pass a Go pointer to C provided the Go memory to which it points does not contain any Go pointers. The C code must preserve this property: it must not store any Go pointers in Go memory, even temporarily. [...]
>
> C code may not keep a copy of a Go pointer after the call returns. [...]
>
> A Go function called by C code may not return a Go pointer (which implies that it may not return a string, slice, channel, and so forth). A Go function called by C code may take C pointers as arguments, and it may store non-pointer or C pointer data through those pointers, but it may not store a Go pointer in memory pointed to by a C pointer. A Go function called by C code may take a Go pointer as an argument, but it must preserve the property that the Go memory to which it points does not contain any Go pointers.
>
> (*cgo* documentation [2])

These rules make it illegal to pass a pointer to the interpreter structure (containing cache and configuration for the execution) directly or indirectly to the user of C library. I worked around this limitation, by passing an artificial identifier to the user of the C library – an integer instead of a pointer. A global mapping is created between such identifiers and the requested interpreter structures. This way it was possible to emulate manual allocation of structures

[6] One of the primary motivations for having an automatic formatter is keeping consistency both within and across projects.

in C, which are backed by complex representations in Go, without violating interoperation rules.

2.2. Interpreter performance improvements

Jsonnet was not designed for speed. Clearly, building configuration files is not a computationally intensive task. Generally the amount of resources required can be expected to be roughly linear relative to the output size, with most of the logic involving simply combining larger structures from smaller parts and then returning the whole big structure.

Accordingly, the new implementation of Jsonnet in Go did not have an explicit goal of improving performance. Situation changed when with the growing popularity of the language, larger and larger codebases were created which required a lot of time to evaluate. The users reported evaluation times of the order of minutes and sometimes even hours. This was clearly not an ideal situation.

When *go-jsonnet* reached feature completeness it was significantly slower than the C++ implementation. This situation reversed over time through the means explained further in this section.

Improving performance is an ongoing effort, which will be relevant as long as Jsonnet itself is relevant. Currently, all implementations of Jsonnet are still slow when compared with the more mainstream interpreted languages and it seems that there are still many easy opportunities for optimization.

2.2.1. Changes in the design

Jsonnet is a lazily evaluated language and as such has an internal concept of a *thunk* – a thing which can be evaluated to a value when requested. The original design of *go-jsonnet*, unconcerned with performance, used Go interfaces as thunks. Multiple implementations of specialized thunks were created for various operations (e.g. a thunk for a function call containing a target thunk and argument thunks or a thunk for regular evaluation containing an AST node and its evaluation environment). The thunks were treated as completely opaque objects, with evaluation as the only operation. This facilitated construction of arbitrary lazy values in a way that mirrors Jsonnet code. The described design may be considered elegant in some ways, but it came at a significant cost – each evaluation required an additional vtable lookup. Moreover, the implementation often created lazy values when it was not strictly necessary, but allowed presenting things in a more modular and uniform manner.

When it became clear that it is a source of performance problems for the users, Dave Cunningham stripped the additional layers of abstraction. Currently there is only one type of thunks – evaluating the AST in some environment[7] and it is always used directly. The functions in the interpreter operate on concrete values rather than thunks when possible, avoiding the need for additional wrapping.

2.2.2. Object field caching

Object fields in Jsonnet are closer to computed properties[8] than simple values. They are not calculated when the object is created, but only when they are accessed – but that is normal in a lazy language. In their definitions they can refer to the current object, in a way supporting inheritance and that means they are fundamentally calculations rather than values.

[7] This required expressing some operations as artificial AST nodes.

[8] Computed properties in the sense of Python's `@proprety` and C# properties.

Originally, every time a field was accessed, its value was computed from scratch. This was problematic, especially for users who were not aware of this behavior. It is easy to devise an example where this could lead to exponential behavior:

```
local nextFib = {
    a: super.b,
    b: super.a + super.b,
};
{a: 0, b: 1} + nextFib + nextFib + nextFib + ...
```

A less spectacular, but just as important case is deep indexing of big nested objects — without caching, every time all of the objects on the path are created, with all their fields (shallowly, of course, but it still requires time and memory proportional to the number of fields of all the objects on the path and results in multiple memory allocations).

The original reason for the decision to avoid caching is not completely clear to me, but it was related to the concerns about the memory usage. Current consensus is that the object fields in Jsonnet should be cached for the reasons mentioned above. This, combined with abandoning the requirement of performance parity[9], allowed me to finally implement the field caching, which became available the 0.14.0 release of *go-jsonnet*.

The caching only happens for a single concrete object, so the fields are still just calculations for inheritance purposes. In other words, an object created by inheritance starts with a clear cache, independent from the cache of objects from which it was constructed. The reason for this is that in general overriding even a single field may change the values of all fields. Of course it is potentially valid to "opportunistically" reuse cache of fields which are not dependent on self. We may add this optimization in some future version.

2.2.3. Native implementations of standard library functions

Probably the most straightforward way of improving performance is reimplementing functionality from the standard library in the language of the interpreter. Natively implemented parts of the standard library are referred to as *builtins* in the Jsonnet community. Importantly, these functions are still expected to be pure and they do not add any new capabilities to the language[10]. It is recommended that new additions to the standard library have an implementation in Jsonnet first.

Jsonnet is very ill-suited for efficiently performing some types of computation. Due to lazy evaluation and lack of static types, there are usually at least two levels of indirection for each operation. This, combined with using the AST directly for interpretation[11] means that the basic building blocks are extremely slow. While we can work on improving the situation, executing many simple operations on small chunks of data is always going to be slow. Builtins mitigate this problem, by allowing the user to use bigger building blocks and making the overhead less relevant. For example, it helps tremendously with the performance of searching for a word in a string – Another extreme example is the comparison operation. Originally there

[9] When I started working on *go-jsonnet* project, we wanted to avoid significant differences in performance characteristics across implementations, which made it difficult to implement any high-level optimizations. Fortunately, this decision was later changed.

[10] There are some builtins which actually provide otherwise unavailable functionality. Some builtins `std` ↪ `.extVar`, `std.native` and `std.thisFile`. There are also primitive operations which are partially implemented as builtins, for example arithmetic. Finally there are functions which technically could be implemented in Jsonnet, but it would be very impractical like numerical functions.

[11] Mainstream interpreted languages are usually compiled to byte code before execution. Python, Ruby, PHP and Perl all use an intermediate bytecode.

was one (builtin) function for primitive comparisons which was wrapped in a more general comparison function written in Jsonnet which also supported arrays and objects. This meant that every time the user used an operator <, it involved at least two function calls and a few Jsonnet-level conditionals. It was replaced with a single efficient builtin.

Yet another category of functions benefiting from builtin implementations are those which can take advantage of mutable data structures. For example, a function joining multiple arrays in Jsonnet is necessarily much slower because simply appending element by element to an array is actually quadratic[12]. A smarter solutions are often possible, e.g. creating a list or joining it in a tree-like fashion. These approaches are often slower in practice due to overhead of the operations involved. Using a builtin allows us to simply allocate and then fill in the array.

This is very easy in the Go implementation, because basic runtime operations are available as Go functions. These operations include evaluating a value, indexing an array, calling a function or even creating an anonymous function containing Go code It is therefore possible to "write Jsonnet in Go", that is program using Go syntax and control structures, but at the same type have full access to Jsonnet runtime concepts[13].

In the C++ implementation doing so is quite difficult, because it requires manipulating the stack directly. Moreover when transitioning to a different stack frame, it is necessary to manually save the state. An additional challenge is the operation of the garbage collector, which requires keeping the Jsonnet values properly registered and connected.

2.2.4. "Opportunistic" data structures

Since we are interested in building configuration, we must expect our users not to pay too much attention to sophisticated programming in Jsonnet. In the context of configuration clarity is extremely important and we do not want to force the users to implement complicated algorithms and such. We definitely do not expect the users to optimize the code in a way that is common in C++ or even Python.

Fortunately some other characteristics of our niche put Jsonnet in a unusually good position to make naively written code behave reasonably. First of all, all Jsonnet programs are working in batch mode, that is no intermediate interaction is expected, the program gets its inputs and it produces the answer some time later. This means that we only care about the performance of the execution as a whole, so we can optimize just the amortized complexity of any algorithms. Second, the overhead of most operations is already quite high, so adding additional logic is not going to degrade performance so much. This allows us to do things like dynamically switching the representation of a value based on how it is used.

So far, this resulted in an implementation of lazily concatenated strings, which provide a way to address a well-known inconvenience with immutable strings, that combining a large number of short strings into a single long one, by using concatenation results in quadratic behavior. Languages often provide library facilities to help with that, for example in Java there is StringBuilder and in Haskell DiffLists can be used for this purpose. It is also often possible, in any language, to create a list of them and merge them efficiently in one go.

In Jsonnet, ideally such operations would just work fast enough when written in a naive manner, so that Jsonnet users do not need to worry about such things. We can achieve that

[12] In subsection 2.2.4, I present how I improved the performance of such naive joining for strings. The same technique is planned to be applied to arrays as well. Using a standard library function for joining is still much faster when the values to be joined are already in an array.

[13] At least as long as no lazy values need to be created, which became more difficult due to the changes described in subsection 2.2.1.

by internally having two representations of the strings. The first is simple flat representation, just as before (an array of Unicode codepoints). The second representation is a tree of concatenations, with a constant size node for each concatenation. When two strings are concatenated, the new string is using the second representation, referencing the two parts. When a string is accessed (e.g. indexed) the second representation is flattened to the first, in time proportional to the string length[14] This results in a linear complexity when the user when building the long string naively, but they are not accessing the intermediate values. This has already been implemented, but it is not a part of any release yet.

The same reasoning applies to arrays as well. More advanced algorithms can also be added, which could allow worst case amortized $\mathcal{O}(n \log n)$ complexity, independently from how it is used. I come back to this issue in the subsection 2.3.2.

2.2.5. Sharing import cache between evaluations

Sometimes multiple Jsonnet files are used to create parts of configuration. In such case it is likely that they use the same libraries, inputs and intermediate calculations. Until recently, each evaluation of a Jsonnet program started from a clean state, which depending on the setup, can result in parsing, preprocessing and evaluation the same files over and over again. With the changes to the API, it is not possible to evaluate multiple Jsonnet programs while reusing the cached evaluations of imported files. Currently this ability is restricted to the library use and it is not possible to achieve this effect from the command line.

Importantly the caching is implemented in the safe way, meaning that it will never change the result of evaluations. This is possible, because all calculations are deterministic and pure and additionally input parameters, such as external variables, are controlled and the cached is invalidated if they change.

2.3. Proposed further optimizations

2.3.1. Static expression caching

Some expressions are static meaning that, they are independent from any function arguments, `self` and `super`. An extremely common example of such expression is indexing a library to get one of its fields, like `std.map`. This often means that an object (e.g. '`std`') is unnecessarily indexed over and over again, even though the result is always the same. Of course this problem could be mitigated by the end user by assigning each such subexpression to the variable, but it is an unnatural way to write code and I do not recommend such style.

This issue can be handled systematically by memoization of all static expressions. Whether a given expression is static can be determined before evaluation, during preprocessing step. The only concern is the effect on memory usage – values of all evaluated static expressions will be stored until the end of the execution. This may warrant restricting the caching to only some forms of expressions or to some computed values.

2.3.2. More "opportunistic" data structures

As mentioned in the previous section, in *go-jsonnet* the strings can be merged lazily, which improves the performance by an order of magnitude when a long string is constructed naively

[14] It is linear, because we do not add nodes for empty strings and therefore the number of nodes is at most the same as the length of the string.

from a large number of short strings. The obvious next step is applying the same technique for arrays.

Later, we could consider more advanced underlying data structures which are maintaining good performance even when used in a less regular way. For brevity from this point forward I will only consider arrays, but the same consideration may apply just as well to strings.

There are many operations and multiple trade-offs to consider. Below I included some natural goals to consider:

1. Fast naive concatenation of n constant length arrays – $O(n)$.
2. Total cost of indexing an array of length n, however it is constructed, m times where $m > n$ is $O(m)$.
3. Indexing an array constructed in one go (e.g. with `std.makeArray` or `std.map`) – amortized $O(1)$.
4. Fast indexing in general – amortized $O(\log n)$.
5. Fast concatenation in general – amortized $O(\log n)$.
6. Fast slicing – amortized $O(\log n)$.
7. Fast updating of elements – amortized $O(\log n)$.

Lazy concatenation satisfies conditions 1, 2, 3 and 5. Representation based exclusively on balanced trees (e.g. AVL) can easily satisfy goals 4, 5, 6, 7. If instead of keeping individual elements of an array in the nodes of a tree, we allowed larger flat subarrays to be stored, it would be possible to satisfy condition 3 and reduce the memory usage (by a constant factor, but very meaningfully), but property 6 would be lost. That would also allow flattening the trees which are indexed enough (e.g. after $O(\frac{n}{\log n})$ indexing operations) which would allow achieving 2. It is also possible to achieve property 1, for example by introducing lazy concatenation to this representation (effectively creating a tree of trees).

Perhaps the problem of slicing can be solved by keeping the reference to the original array with indexes to the beginning and end of the slice. This decision helps a lot with the design of data structures, by making slicing a trivial constant time operation, but it has memory usage implication, because it keeps the whole underlying array from being freed by the garbage collector.

2.3.3. Lexer and parser optimization

For some users of the language, despite the slow evaluation, parsing and lexing remains a significant portion of the overall execution time. This is clearly the case for users who use large libraries for relatively small configurations. Historically, even parsing and preparing standard library, which has around 1300 lines of code, was a large factor in the time needed to evaluate small Jsonnet programs. This problem was alleviated by saving the preprocessed AST directly in the binary[15], but this solution is not easily applicable to the code which is not bundled with the interpreter[16].

[15] This was achieved by automatically generating a Go file containing the whole, ready to execute the standard library AST. This improvement was contributed by Liang Mingqiang.

[16] A related idea of saving a pre-parsed AST in binary format is worth consideration in the future. A similar approach is used in Python with `.pyc` files containing the cached bytecode.

Both parsing and lexing is performed by a hand-written lexer and recursive descent parser. I am unaware of any obvious significant inefficiencies of the implementation and the improvements are most likely to be made with the use of concrete real-world benchmarks and profiling.

Chapter 3

Linter

3.1. Motivation

It seems widely accepted that the length of the feedback loop and clarity of the reported issues are important for a programmer's productivity[1]. One of the quickest form of feedback is provided by the compiler errors and warnings in statically typed languages. For interpreted languages usually additional tools, called linters[2] are necessary to perform static checking. Scope of reported problems for such software is usually a bit broader than for compilers, since the messages include not only errors, but also the issues of coding style.

Jsonnet interpreter catches some problems statically, most notably the use of undeclared variables[3]. Still, most of the time, even trivial mistakes result in a runtime error. To make things worse, the errors are often appearing in unexpected places (i.e. far from the actual mistake) and the stack trace is not always very helpful. These problems are common across dynamically typed programming languages. However Jsonnet combines a few characteristics which exacerbate them:

- It is lazily evaluated, so the stack trace informs the user where the invalid value was used, which may be a completely different place from where it was lexically defined. It also means that problems may easily go unnoticed in simple tests, where some values are not used, but crash in complex cases.

- It has no user-defined types, instead relying on arbitrarily nested structures of ad-hoc objects and arrays.

- It is expected to be used by programmers with very little preparation, due to its role as a configuration language.

In essence the goal of the linter is to partially bridge the gap in error reporting between Jsonnet and languages with a static type system. I expect such checks to be especially useful for the new users of the language, especially those who had little prior experience with lazy semantics.

[1] I could not find any specific studies for this claim, but the popularity of IDEs showing errors immediately for the user supports it

[2] The term linter originates from a Unix utility called Lint (see [8]), designed for finding technically valid, but undesirable constructs in C programs.

[3] Checking for undeclared variables is mandated by the specification.

3.2. Design goals

The purpose of the linter is to aid development by providing quick and clear feedback about simple problems. With that in mind I defined the following goals:

- It should find common problems, especially the kinds resulting from typos, trivial omissions and issues resulting from misunderstanding of the semantics.

- It should find problems in the parts of code which are not reached by the tests (especially important due to the lazy evaluation).

- It must be practical to use with the existing Jsonnet code, without any need for modification.

- It must be fast enough so it is practical to always run the linter before execution during development. The overhead required to run the linter prior to running the program in real world conditions should be comparable with parsing and desugaring.

- It must be conservative regarding the reported problems. False negatives are preferable to false positives. False positives are allowed as long as they relate to code which is going to be confusing for humans to read or if they can be worked around easily while preserving readability.

- Its results must be stable, i.e. trivial changes such as changing the order of variables in `local` expressions should not change the result nontrivially.

- It must preserve the abstractions. Validity of the definitions should not depend on their use. In particular calling functions with specific arguments or accessing fields of objects should not cause errors in their definitions.

- It should be possible to explicitly silence individual errors[4], so that occasional acknowledged false positives do not distract the users. This is also necessary to make enforcing a clean pass in Continuous Integration.

The above goals naturally lead to the some more specific code-level rules which all analyses must obey:

- All expressions should be checked, even the provably dead code.

- Always consider both branches of the `if` expression possible (even if the condition is trivially always true or always false).

- Correctness of a function definition should not depend on how it is used. In particular when analyzing the definition assume that function arguments can take arbitrary values.

- Correctness of an object definition should not depend on how it is used. In particular when analyzing the definition assume that the object may be part of an arbitrary inheritance chain.

[4] Not implemented yet.

3.3. Scope

There are clearly more useful analyses than I am able to implement in a reasonable time span. Therefore it was necessary to choose some analyses which provide the most value to the users of the language. The following list summarizes the kinds of problems which I consider most important to catch in the first version of the linter:

- Problems that are usually caught before execution in a compiled language:
  - Unused variables
  - "Type" problems
    - Accessing nonexistent fields
    - Calling a function with a wrong number of arguments or named arguments which do not match the parameters
    - Trying to call a value which is not a function
    - Trying to index a value which is not an object, array or a string
- Endlessly looping constructs, which are always invalid, but often appear as a result of confusion about language semantics (e.g. `local x = x + 1`).

In the following sections I will present in more detail the techniques used to implement the automatic detection of these problems.

3.4. Binding variable use to definition

I will start with a very simple, yet very useful analysis – connecting the variable use and its definition. This is easy in Jsonnet, because it is impossible to dynamically introduce variables with unknown names or import unknown names[5]. All variable names originate from `local` expressions within the same file. It is therefore enough to traverse the AST once, keeping track of the environment, that is the mapping between variable names and their definitions within the currently analyzed scope. Almost identical step, used for detecting use of undeclared variables, is already a mandatory part of any compliant Jsonnet interpreter[6].

The data from this step can be directly used for finding unused variables. Due to the generality of `local` in Jsonnet, this check also finds unused functions and objects, since `local` is used to introduce all local names and there is no separate syntax for declarations.

Just as importantly, the results from this step are used in other checks. Due to referential transparency, having a definition available is enough to reason about the value of the variable.

[5] This is not the case for other languages. For example, in Python it is possible to introduce statically unknown names with `from library import *`.

[6] See *Static Checking* section of the Jsonnet specification [11].

3.4.1. Examples

Jsonnet source	Linter output

`local x = 1; {}`	`unused1.jsonnet:1:7-12 Unused variable: x`
	
	`local x = 1; {}`

`local x = 1, y = x; {}`	`unused2.jsonnet:1:14-19 Unused variable: y`
	
	`local x = 1, y = x; {}`

`{`
`    local x = 42`
`}`
	`unused3.jsonnet:2:11-17 Unused variable: x`
	
	`    local x = 42`

`local obj = {};`
`local f(arg) = 42;`
`true`
	`unused4.jsonnet:1:7-15 Unused variable: obj`
	
	`local obj = {};`
	
	` Unused variable: f`

3.5. Checking types

Probably the most useful ability of the linter is checking for the kind of problems which are in other languages prevented with static typing, such as using a nonexistent object field or calling a function with a wrong number of arguments. This was also by far the most challenging part of the linter, in both design and implementation. The difficulty arises mainly from the need to handle recursive definitions, which makes a simple traversal of the AST insufficient.

At its core, the check is concerned with the sets of possible values of each expression. This information can be used to verify the validity of operations. For example, knowing that expression `x` is a number, we can say that `x()` is definitely invalid (since only functions can be called).

This result in an algorithm which has two stages. In the first stage, for each expression we find the *upper bound* for a set of its possible values. In the second step the AST is traversed and all the expressions are checked using a simple criterion of whether an operation can succeed for *any* values of subexpressions allowed by the upper bounds. This is different from how typical type checking works, which is closer to checking that an expression is valid in some sense for *all* values of subexpressions.

Notably, *lower bounds* are not necessary at any point of the algorithm. During checking, lower bounds are not helpful in proving that there are no possible values for which a given expression is correct, because for that it is necessary to know what values the expressions definitely cannot have. For finding the upper bounds, the design decision forbidding reasoning about the definition by its usage eliminates the need for using the lower bound. For example, function arguments are always considered to be potentially anything, since within the definition and its environment there is nothing that constrains them.

Before getting into the description of the details, it may be helpful to consider an example and the relevant output from the linter:

Jsonnet source	Linter output

`local a = [b[0], 1],`
`      b = ["str", a[1]],`
`      c = [a, b, c];`
 	
`local lib = {`
`    local foo(x) = [`
`        c[2][2][2][0][0],`
`        c[2][1][1]`
`    ],`
`    bar: foo,`
`};`
 	
`[`
`    lib.nonexistent,`
`    lib.bar(1, 2),`
`    lib.bar(),`
`    lib.bar(2)[0](),`
`    lib.bar(2)[1](),`
`]`
	`types.jsonnet:14:5-20 Indexed object`
	`    ↪  has no field "nonexistent"`
	
	`    lib.nonexistent,`
	
	`types.jsonnet:15:16-17 Too many`
	`    ↪ arguments, there can be at`
	`    ↪ most 1, but 2 provided`
	
	`    lib.bar(1, 2),`
	
	`types.jsonnet:16:5-14 Missing`
	`    ↪ argument: x`
	
	`    lib.bar(),`
	
	`types.jsonnet:17:5-20 Called value`
	`    ↪ must be a function, but it is`
	`    ↪ assumed to be string`
	
	`    lib.bar(2)[0](),`
	
	`types.jsonnet:18:5-20 Called value`
	`    ↪ must be a function, but it is`
	`    ↪ assumed to be number`
	
	`    lib.bar(2)[1](),`

3.5.1. Types classification

It is clearly impossible to represent the precise sets of possible values of every expression in the linter. We need to be able to describe any such set using a reasonable amount of memory[7] and to be able to answer questions about validity of the operations involving elements of these sets. Therefore it is necessary to choose a smaller *domain*, that is a family of sets that can be used to describe upper bounds of sets of values. The elements of the domain will correspond to what is intuitively perceived as types (e.g. all numbers or all arrays containing numbers). The choice of the domain is relatively free, with many options offering various trade-offs.

A natural starting point is Jsonnet specification, which defines exactly seven types[8]: null,

[7] Preferably using at most hundreds of bytes.

[8] With the exception of *function* all of them directly correspond to JSON types.

boolean, number, string, object, array and function. Contents are not distinguished in this type system, there is only one object type and one array type. Similarly parameters and result type are not a part of a function type — there is only one type for functions. No user defined types exist and complex values are constructed by nesting values inside objects and arrays.

For the purposes of the linter, during the verification step, we need more specific information about the values that expressions can take. In particular, fields of objects and their types are required for checking if libraries are used correctly. Thus, a more advanced representation is necessary. Nevertheless, the simple classification into seven types is still useful. To avoid ambiguities when referring to types from the Jsonnet specification I will use the term *basic types.*

It is quite common Jsonnet programs to have an expression which can have value of two or more different basic types. The most common example is having nullable fields. Many algorithms can also operate just as well on arrays and strings[9].

Finally, the representation within each basic type is chosen. Some of them are considered primitive — no further distinction is made, while some are more complex and have multiple variants, referencing other types. I will call these referenced types *internal types* of a given type. For example, number would be an internal type of array of numbers.

Basic type	**Extended type description**
Null	Primitive
Boolean	Primitive
Number	Primitive
String	Primitive
Function	Complex: • Result type. • Parameter names and optionality if available. • Minimum and maximum number of parameters if available. *No constraints of argument values are recorded, only the information available at the definition.*
Objects	Complex: • Known types of fields (if a field with a given name exists in the mapping its type must be as specified). • Whether other fields with unknown names may exist and if they do, a shared type for all of them. *The internal inheritance structure is not kept.*

[9] Strings in most cases behave just like an array of one letter (codepoint) strings.

Basic type	**Extended type description**
Array	Complex: • Types of at most five first elements, if known. • A single type for all remaining elements. *This representation supports using arrays as both tuples and collections.*

3.5.2. Types as sets of values

In this analysis the types are nothing more than a compact description of sets of values that expressions can be assumed to have. This is a slightly different concept from types in a typical programming language. For example, traditionally, both branches of the `if` expression must have types which are in some sense compatible[10]. In our case we simply take the sum type[11]. In fact the step of finding the upper bounds for value sets will always succeed (after all we can always take *all* values as an upper bound.). We can say that the types we are looking for are descriptive, as opposed to more commonly used prescriptive type systems which forbid mixing some values within one type. A very similar idea, in the context of Prolog is described in detail in the work of Pietrzak et al [19].

3.5.3. Unions and widening

Previously described types are not closed under union. In other words, the union of sets of values described by two types is not necessarily representable as a type. For example, there is no way to directly represent a union of an array of numbers and an array of strings. Hence, the only option is to find a representable upper bound of the union. In the case of our example, that would be an array which can contain both strings and numbers[12] .

3.5.4. Support for objects

Objects are central to Jsonnet. Even when not using the OOP features of Jsonnet, they are used for organizing libraries, grouping data together and producing JSON objects in the output.

The way in which Jsonnet handles OOP concept is somewhat unusual compared to currently popular languages which are advertised as object-oriented. There are no declarations whatsoever and no user-defined types, so the concept of classes does not fit naturally in Jsonnet. There is no fixed hierarchy of inheritance. Instead, the objects can be freely combined in an ad-hoc matter as mixins. The ordinary `+` operator is overloaded on objects, such that `a + b` results in an object created by `a` inheriting form `b`.

This approach can be considered very elegant and fits well in Jsonnet, but it is quite problematic for static analysis. Since any field can be replaced and new fields may be added, it is not known within the definition of an object, what `self` and `super` are referring to. Therefore it is impossible to say in general whether their usage is right or wrong or what types they are producing (more precisely than "any object")

[10] The precise rules of what types are compatible for this purpose vary by language. Sometimes it is quite complicated, but they always forbid using completely unrelated types. For example, in all C, C++, Haskell and Ocaml having one branch of integer type and the other of type string is invalid.

[11] Widened if necessary, see subsection 3.5.3.

[12] It is different from the union, because it also allows values like `[42, "a"]`.

For these reasons I decided that all references to `self` and `super` will be completely opaque, that is treated as if they were referring an unknown object.

3.5.5. Catching the problems – weak checking

In the analysis the linter is checking only a very weak condition — whether each expression can potentially succeed at all. One way to think about it, is that it is roughly equivalent to having 100% code coverage without any actual assertions.

It is possible to find any violations in a single pass over the AST, with each expression checked independently. For example, for an indexing expression `expr1[expr2]` the linter will produce a warning if neither of the following is possible:

- Expression `expr1` is an object and `expr2` a string.
- Expression `expr1` is an array or a string and `expr2` is a number.

At this point, it is worth noting that the linter uses only shallow type information in this step – it does not access any internal types. More precisely, only a single type is accessed for each expression. For example, for a given expression `expr` the linter may check if it can be an object and even if it has a field `foo`, but it will not access the type of `expr.foo` through the type of `expr`. If the expression `expr.foo` appears independently in the source code, of course its type can be used, but the upper bound on the values it can take is calculated separately in the previous stage, not extracted the type of `expr`.

Having a separate step for actually finding the problems allows for using only the subset of source files to emit warnings[13], even though for finding the types, the entirety of the source code is necessary. In particular it is useful to derive the types for third party libraries, so that the code using the library can be checked, but it is undesirable to display warnings about the internals of the library. The library authors may be following a different convention or even have errors in the parts of code that do not concern the user. By keeping only the relevant warnings we make it much easier to notice the problems in user's code and even make it feasible to require a fully warning-free pass for all the merged code.

There are a few ways in which this solution is superior to simply testing. First of all, it provides the results without the need to write any additional code, which is useful given that writing tests for configuration files is not a common practice. The library code is more likely to be tested, at least to some extent, but even then quick feedback during development is likely to be useful. Second, it always verifies the whole code, including branches which are unlikely to be executed in normal operation, and therefore likely to be overlooked during testing. The importance of this point is greatly increased due to lazy evaluation, which makes coverage much less predictable for the programmer. Third it provides the information about the problem in a much more clear and actionable form, compared to a runtime error with a stack trace. Moreover, multiple issues can be presented at once to speed up the development process even more.

3.5.6. Finding the types – the naive approach

The most obvious approach to this analysis is to traverse the AST, calculating the types of the children before the parent. This works in the simplest cases and even that would probably be enough to create a useful tool. Unfortunately, using this method types for self-referential

[13] This is not implemented yet, but it is planned to be added soon, once some changes outside of linter get merged.

structures cannot be calculated. This is problematic since recursion is very common in Jsonnet programs.

The locals in Jsonnet can be self-recursive and mutually recursive within one local declaration. This means that multiple variables can be introduced into the scope at once. In even as simple case as below the naive algorithm cannot conclude that `bar` is a number:

```
local bar = foo, foo = 42; bar
```

Because of this limitation I turned my attention towards more sophisticated methods.

3.5.7. Finding the types – the abstract interpretation approach

Abstract interpretation [3] is a standard, well-studied technique for determining possible values in a program. One of the applications for the method is type discovery. For example, it was used for augmenting Hindley-Milner type inference by Monseuez in [17, 16]. More in line with the goals of this thesis, abstract interpretation was also used for finding descriptive types, for example by Pietrzak et al in [19] in the context of Prolog.

For Jsonnet it is possible to find the descriptive types using abstract interpretation in a mostly generic way with the domain of types from subsection 3.5.1. There are only a few things that require clarification.

First, it is necessary to define the mapping between the abstract domain (types) and concrete domain (values). This is given by the descriptions of the types in subsection 3.5.1 – for each value it is clear what is the smallest type that contains it and for each type what values it contains.

Second, contexts must be defined. In textbook abstract interpretation these correspond to positions in the control flow graph. Here, since the whole program is an expression, it is possible to simply use sets of possible values for each expression.

Third, the relationship between contexts which describe how the sets can be iteratively extended. These can be derived directly from concrete semantics and abstraction function, with the exception of two special considerations, derived from design goals in section 3.2. One is that the type of functions should be calculated in general, without assumptions . All function calls The other is that both branches of conditional are always considered possible[14]. These calculations can be performed with AST nodes used directly for defining each context.

The main problem with this approach is that the types are potentially infinite and commonly very big. Even considering that only fields with literal names are allowed, there is still nesting. The algorithm in the presented form will enter an infinite loop when presented with the following code `local arr = [arr]; true`. The problems of this kind are usually solved by introducing additional widening which makes it converge more quickly toward the fixed point. For example, we could limit the maximum depth of the type or its total size - replacing the nested parts with special representation of any value. The performance characteristics of such algorithm are still not immediately clear.

For this reason I decided to more deeply investigate various ways of describing the relationship between types (especially recursive relationships), hoping to find a better representation. This in turn lead to a satisfying solution, without going back on abstract interpretation[15]. Nevertheless, in the future versions of the linter approaches based abstract interpretation may be worth revisiting.

[14] It is actually a consequence of the way the domain is defined, but it is worth mentioning explicitly.

[15] I cannot exclude the possibility that a reader more familiar with the relevant theory will be able to show how the other presented techniques are actually a form of abstract interpretation.

3.5.8. Finding the types – the equation approach

Another way of looking at the problem originates from my attempts to derive the types in a systematic way manually, on paper. We have a complex structure which defines potentially recursive relationships between types. It is very natural to introduce a variable for each of the types in question and write down their relationships explicitly, independently from concrete, value-level semantics. We can also see this approach as eliminating all the irrelevant data from the AST, leaving only the information relevant for calculating types. For example, we can describe the types of the following snippet:

Jsonnet source	**Type equations**

`local a = if x then 1 else b,`
`    b = if x then a else 2;`
	$ta \subseteq \textbf{number} \cup tb$
	$tb \subseteq \textbf{number} \cup ta$

Notably, the equations constructed in this way always have a solution – all variables equal to *any* (the whole domain, all possible values). We would like to find a practically small upper bound, but not necessarily the smallest one (which may be difficult to describe in a compact way or to calculate at all).

For finding the solution we can use the usual algebraic transformations, similarly to solving a system of equations. In our example, we can start by simply substituting the *tb* variable in the first equation:

$$\begin{aligned} ta &\subseteq \textbf{number} \cup \textbf{number} \cup ta \\ tb &\subseteq \textbf{number} \cup ta \end{aligned}$$

Since $x = y \cup x$ describes exactly the same constraint on x as $x = y$, we can now eliminate the recursion.

$$\begin{aligned} ta &\subseteq \textbf{number} \cup \textbf{number} \\ tb &\subseteq \textbf{number} \cup ta \end{aligned}$$

This procedure can be generalized. For now, I will assume that all the starting equations have the form $x_i = x_{j_1} \cup x_{j_2} \cup \ldots \cup x_{j_m} \cup c_1 \cup \ldots c_k$, where x_i describe type variables and c_k are concrete types. This is enough to describe programs which use only flat types (no objects, arrays or functions) and no overloaded operator "+"[16].

In this case, it is easy to design an algorithm which goes over variables and eliminates them the right side one by one. For each variable, first the self-reference is eliminated as in the example above and then the variable is eliminated from all other equations by replacing it with its definition. This is reminiscent of the simplest algorithms for solving systems of linear equations.

The situation becomes much more interesting once we consider indexing (of object, array and string types), like in the example below:

[16] Operator "+" requires special handling, because it is heavily overloaded, very common and used in some fundamental operations, such as inheritance for which we would like to reason about the result, depending on the specific arguments. For all other functions and operators we simply assume that they return values of one specific type.

Jsonnet source	Type equations

`local a = [1, b[1]],`
`      b = [a[0], 1];`
	$ta \subseteq [\mathbf{number} \cup tb[]]$
	$tb \subseteq [ta[] \cup \mathbf{number}]$

We can start by analyzing what happens when a simple substitution based procedure is executed, in a manner similar to the previous cases.

$$\begin{aligned} ta &\subseteq [\mathbf{number} \cup tb[]] \\ tb &\subseteq [[\mathbf{number} \cup tb[]][] \cup \mathbf{number}] \end{aligned}$$

Indexing a literal array can be then simplified as follows.

$$\begin{aligned} ta &\subseteq [\mathbf{number} \cup tb[]] \\ tb &\subseteq [\mathbf{number} \cup tb[] \cup \mathbf{number}] \end{aligned}$$

In this case we can still eliminate the self-recursion, but it is not as simple as before. This points to a possible problem. Can we eliminate the self-reference in general when indexing is allowed?

Jsonnet source	Type equations
`local a = [1, a];`	$ta \subseteq [\mathbf{number} \cup ta]$

In this case it is impossible to eliminate the self-recursion in a compact (i.e. not infinite) way. One way to handle this issue is to replace ta on the right-hand side with some reasonable upper bound. Obviously it can lead to a larger set of values in ta than it is strictly necessary. However, it is not immediately clear what is a reasonable upper bound in such cases.

There is an attractive alternative, though. We can allow recursion in internal types, because they are not accessed during the checking stage[17]. This information is only necessary for the purpose of finding a shallow type for each expression, where one variable may index another. This approach has nice properties, for example indexing ta any number of times will only lead to arrays and numbers.

In order to achieve this we can introduce a new type variable for each occurrence of indexing. To illustrate, I will go back to the previous example:

$$\begin{aligned} ta &\subseteq [\mathbf{number} \cup tb[]] \\ tb &\subseteq [ta[] \cup \mathbf{number}] \end{aligned}$$

We can introduce a new type variable for $ta[]$:

$$\begin{aligned} ta &\subseteq [\mathbf{number} \cup tb[]] \\ tb &\subseteq [ta_{[]} \cup \mathbf{number}] \\ ta_{[]} &\subseteq \mathbf{number} \cup tb[] \end{aligned}$$

[17] See subsection 3.5.5.

And following the same procedure we can eliminate $tb[]$, resulting in an index-free representation:

$$\begin{aligned}
ta &= [\mathbf{number} \cup tb_{[]}] \\
tb &= [ta_{[]} \cup \mathbf{number}] \\
ta_{[]} &= \mathbf{number} \cup tb_{[]} \\
tb_{[]} &= ta_{[]} \cup \mathbf{number}
\end{aligned}$$

The example concerns indexing of functions and assumes only one kind of indexing. This can obviously be generalized to handle objects, with multiple kinds of indexing – one for each known known field and one for "generic" indexing. Conveniently, we can also handle functions and function calls using this method. In our representation, the function result type is not parametrized by the arguments. Therefore any function call results in the same type, with arguments having no effect on the upper bound calculations – just as if the function was a container which can be indexed to get a value of the result type. So, function calls can be thought of as yet another kind of indexing.

The final issue is handling of operator "+", which needs to be somehow handled for the purposes of indexing. The general approach is to approximate it as a set union[18] when the arguments do not have a form which allows calculating it outright.

3.5.9. Finding the types – the graph approach

While the equation approach seems like a good way to perform the calculation of types on paper, a direct implementations of the algorithms presented is neither very convenient nor efficient. A better way to handle the inference step is to construct a graph-like structure representing the relationship between types and transform it to a form which makes it easy to get upper bounds of the set of possible values for each expression.

In this representation a *placeholder* is associated with each expression in the AST. Placeholders play a similar role to type variables in the equation approach. Each placeholder is a node in the graph, connected to others in different ways depending on its definition. There are four kinds of such nodes[19]:

1. Direct node – a placeholder with a specific type. For example, it can be known to be a number or a function which returns a value of a type given by another placeholder.

2. Union nodes – a placeholder is defined as a union of other placeholders. For example, a placeholder for `if b then x else y` will be union of placeholders for `x` and for `y`. I will use the term *trivial reference* to union nodes of a single placeholder. For example, such nodes will be created in local definitions (a placeholder for variable is a trivial reference to its body).

3. Indexing nodes – a placeholder is defined as a result of indexing another placeholder (*target*) in a particular way (*kind of indexing*). There are multiple kinds of indexing, explained in more detail in subsection 3.5.9. For example, placeholder for `arr[0]` indexes the target `arr` and the indexing kind is "indexing with a small number 0".

[18] The fact that we can do this depends on the specific representation of types that I chose. This would not be the case for example, if we had special representations for small integers.

[19] In the implementation mixed nodes (e.g. having both indexing and direct parts) are allowed by the representation, but do not appear at any stage of the computation in the current version.

4. Plus nodes[20] – a placeholder defined as the result of applying a plus operator to two placeholders. The operation is not commutative when it operates on any value other than numbers, so the left and the right arguments are distinguished.

Edges in the graph correspond to *shallow dependencies*, that is to the other nodes to which the definition refers, other than the internal types. There is an edge coming from union nodes to each of its summands, from indexing node to its target and from each plus node to its arguments. There are no edges coming from direct nodes.

At its core, the idea for processing the graph is exactly the same as in equation approach, but this representation is much more suitable for a practical implementation.

Building the graph of placeholders

First step is to build the graph of placeholders, which will capture the entirety of the information necessary for calculating types. This is achieved a simple traversal of the AST, which creates a placeholder for each node of the AST and calculates the definition for it.

Importantly creating a placeholder is separate from providing its definition. In particular, potentially recursive `local` expression is handled in two steps. First, a new placeholder is created for each variable. Afterwards definitions for the placeholders are calculated one by one for each variable — with all new variables available in the environment. This solves the problem of representing recursive definitions, because only identifiers of placeholders of other variables – not their full definitions are needed for this step.

Representation of direct types

The representation of direct types follows the description in subsection 3.5.1. Primitive types are represented as simple boolean fields — either a placeholder contains a number or it does not. Complex representations often refer to other types. These references are represented as placeholders, not directly available types. More precisely in all cases of one type being a part of the definition of another, a list of placeholders is kept, with the meaning that the referred type is a (widened) union of types represented by the placeholders in the list.

This fact is critical for the algorithm as it allows to refer to a yet unknown type. For example, it is not necessary to know the precise type of `x` to describe that `[x]` is an array of the type of `x`. This ability is what allows us to describe recursive definitions.

An important advantage of this representation is that it effortlessly and naturally allows for representation of self-recursive structures. For example, a placeholder p can directly include an array of type p, representing `local arr = [arr]`.

Eliminating indexing

Just like in the case of the equation approach, it is useful to eliminate indexing in favor of creating element placeholders for indexed types. The linter distinguishes multiple kinds of indexing:

- Indexing with a literal string (a separate kind of indexing for each string).
- Indexing with a literal small number (a separate kind of indexing for each number).

[20] This may be generalized to support other operations, with some restrictions. Each such operation would require explicit support from the linter.

- Generic square bracket indexing (of array, string or object with an unknown index value).
- Calling a function[21].

The types of indexing are derived from syntax alone and they are fully known when the graph is constructed.

At the start of this stage, the graph contains some placeholders which are indexing others. For each target placeholder and each kind of indexing applied to it, the linter creates a new *element* placeholder. The new placeholders will be union nodes, defined by unpacking the information from the target nodes. Afterwards the linter also modifies the existing indexing nodes to simply contain element placeholders, transforming them to trivial references. After this stage of the algorithm there are only direct, union and plus nodes in the graph.

The unpacking requires more explanation. It depends on the kind of *target* node.

In case of direct nodes it is enough to take the types from the representation. For example, if an array of numbers and strings is indexed using generic square brackets, we add int and string to the new type.

For union nodes it is necessary to follow the inclusion relationship. If a target type T includes type T', the linter creates another element placeholder for type T'. Then the element placeholder for T will include the indexed placeholder for T'.

Handling of target nodes which are also indexing nodes is the most complex. Let us assume that target T is indexing another placeholder T'. In such case, first the deeper element type I is created (for T indexing T'). Note that I is describing the same set of values as T, but it is a union node. So we can now use I instead of T for indexing, which results in a new element type I' which is an element type of both I and T.

Plus placeholders are simply approximated as union types of the argument types for the purposes of indexing. This is obviously a very crude approximation, akin to using a union as an upper bound for intersection. This is one of the simplifications which allow good performance. It may be further improved in the future in cases when it is not recursive, that is the argument types do not depend on the type of the plus expression.

It may seem that some of the element types are created unnecessarily, but a new node is always created for a reason – handling of cycles in the graph. The linter at first creates identifiers for the new placeholders, which can be immediately used for references and only then fills in the definitions. This solves the problem, because we are carefully never referring to the definitions of the element types during the calculations.

There leads to a more subtle problem — naive implementation of the above procedure would result in a large number of trivial nodes. It is critical for the performance of the algorithm that whenever definition is trivial, that is the placeholder contains a single placeholder P all further references to it are replaced with P. This allows for path compression which helps avoid endless indexing of trivial references. It is equally important that the summand placeholders in a union node are represented as a set, that is, deduplicated[22].

Determining the upper bounds without plus nodes

After the indexing is eliminated the graph contains only direct nodes, union nodes and plus nodes. It is ready for calculating the direct representation of upper bounds. Let us assume for

[21] See subsection 3.5.8 for explanation why a function call is considered an indexing operation

[22] Early version of the implementation simply concatenated arrays, without deduplication, resulting in explosion of the number of identical elements and crashes due to excessive memory usage.

a moment that there are no plus nodes. In such case all edges in the graph are representing inclusion.

The most important observation at this stage is that the nodes in each strongly connected component (SCC) in such a graph have the same type. It is obvious when we consider that each cycle in the graph is a cycle of inclusions, therefore the placeholders must represent the same set.

On the other hand, in acyclic graphs it is possible to simply propagate the upper bound through the inclusions in topological order.

The complete algorithm for this stage is therefore to analyze the strongly connected components in topological order. For each such component, a widening of all direct types within it is accumulated and further widened by upper bounds of any contained placeholders from outside the component. These upper bounds must have already been calculated, because the processing is performed in topological order.

Determining the upper bounds in general case

The algorithm described previously can be generalized to also allow plus nodes. The idea is the same, the only difference is that the edges represent now a more general *dependence* relationship. During the first step, strongly connected components are found. Each components corresponds to a set of placeholders with mutually dependent definitions. Using the same approximation of plus nodes as in subsection 3.5.9 the linter assumes that all placeholders in a component describe the same set.

Same as before, the strongly connected components are analyzed in a topological order. When processing a plus node in a component, if at least one of its arguments belongs to the same SCC, the plus node is treated as a union node. When the arguments are from another SCC, their upper bounds are already (shallowly) known and a detailed type calculation is performed.

The imports

Jsonnet has two constructs which allow dividing the functionality between files, `import` and `importstr`. For the purposes of type analysis `importstr` is completely trivial – it imports the file as a string[23]. On the other hand `import` can return value of any type[24] and its type is very important – this is needed for the linter to check if the library is used correctly. The semantics of import are such that it would be correct to simply replace import with the code inside. The imports can be recursive[25], but fortunately the linter already has a standard way of handling that — first a placeholder for each imported file is created and only then the analysis happens in the same way as for local variables.

[23] It is used for importing raw data, rather than a Jsonnet program.

[24] By convention it is usually an object which contains all the "exported symbols", but it is not a language requirement. It may also be very useful to have it return a function to create a parametrized module.

[25] While recursive imports are valid, I consider their use a bad practice and perhaps a future version of linter will warn about them.

3.5.10. Examples

Jsonnet source	Linter output
local f = function() [f, 1]; f() ↪ [0]()[0]()[1]()	types0.jsonnet:1:30-48 Called value ↪ must be a function, but it is ↪ assumed to be number local f = function() [f, 1]; f() ↪ [0]()[0]()[1]()
local a = [1, b[1]], b = [a[0], 1]; a[1]()	types1.jsonnet:3:1-7 Called value ↪ must be a function, but it is ↪ assumed to be number a[1]()

local obj = {
    a: 1
};
(obj + {b: 7}).c
	types2.jsonnet:4:1-17 Indexed object
	↪ has no field "c"
	
	(obj + {b: 7}).c

local foo(x) = x;
[
    foo(),
    foo(y=7),
    foo(1, 2),
]
	types3.jsonnet:3:5-10 Missing
	↪ argument: x
	
	    foo(),
	
	types3.jsonnet:4:11-12 function has
	↪ no parameter y
	
	    foo(y=7),
	
	types3.jsonnet:5:12-13 Too many
	↪ arguments, there can be at
	↪ most 1, but 2 provided
	
	    foo(1, 2),

Jsonnet source	Linter output

`(function() 17)[3]`	`types4.jsonnet:1:1-19 Indexed value`
	`    ↪ is neither an array nor an`
	`    ↪ object nor a string`
	
	`(function() 17)[3]`

`local x1 = {a: 1, b: 2};`
`local x2 = if true then {a: x1} else`
`    ↪  {b: x1};`
`local x3 = if true then {a: x2} else`
`    ↪  {b: x2};`
`// [...]`
`local x32 = if true then {a: x31}`
`    ↪ else {b: x31};`
 	
`x32.a.b.a.b.a.b.a.b.a.b.a.b.a.b.a.b`
`   .a.b.a.b.a.b.a.b.a.b.a.b.a.b.a.b()`
	`types5.jsonnet:(34:1)-(35:38) Called`
	`    ↪  value must be a function, but`
	`    ↪  it is assumed to be number`
	
	`x32.a.b.a.b.a.b.a.b.a.b.a.b.a.b.a.b`
	`   .a.b.a.b.a.b.a.b.a.b.a.b.a.b.a.b()`

3.6. Endless loop detection

Jsonnet is designed primarily for building configuration, and as such it is likely to be used by people with little experience with it, who just need to make a small local change. Many of these users are more familiar with an imperative paradigm and are tempted to write code of the following kind:

```
local x = 17;
local x = x + 1;
...
```

The user may expect that `x` in the second line refers to the previous value, but in fact it is self-recursive and evaluation is stuck in an endless loop[26]. The semantics of Jsonnet's `local` are similar to Haskell's `let` and Ocaml's `let rec`, i.e. the variables are immediately visible in the scope (this allows for recursive definitions which are most useful with functions and objects).

I implemented an automatic detection mechanism for the common cases of invalid self-recursive definitions. My hope that this functionality will help the new users of the language. For the example mentioned in the beginning the following output is produced:

```
example_loop.jsonnet:2:11-12 Endless loop in local definition
```

The checking mechanism is limited to detecting endless loops in `local` definitions. Similar situation in objects is intentionally ignored since it is not clear at the location of usage

[26] The actual Jsonnet implementations have limits of recursion depth to avoid taking too much of the user's memory. In such case Jsonnet will crash with a "stack limit exceeded" message.

whether one of the fields was not intended to be overridden, which could break the loop. The situation in case of objects is also less of a problem, since the late-binding semantics of object methods/computed properties are quite similar to the semantics of the most mainstream languages.

3.6.1. Checking mechanism

This check can be seen as a form of strictness analysis. Jsonnet locals consist of multiple *binds*, each introducing a single variable. Variables introduced in a single `local` may be mutually recursive as well as self recursive. For each bind we can identify direct, strict dependencies – the expressions which will need to be unconditionally evaluated when a body of the bind's variable is evaluated. Having calculated the dependencies it is enough to check if there exists a cycle. If it does, we know that there is an endless loop.

In practice, the implementation traverses the AST using a depth first search (DFS). The operation depends on the type of the current node. If it is a variable reference, the algorithm enters the definition of the variable. When it enters any other type of node it transparently enters all of its *direct* children. If in a single run of the DFS starting in one of the locals the same node is reached twice, a loop was found. This is a standard way of finding cycles in graphs.

The children of a node must fulfill a few conditions to be considered direct. First they need to be evaluated when their parent gets evaluated or not at all. For example, this means that in case of an `if` expression, the condition and both branches are considered direct, but elements of an array are not. Second, they need to get evaluated in the same environment (i.e. the same variables available) as their parent. Third, they may not get evaluated while evaluation of any non-direct children is in progress.

3.6.2. Examples

Jsonnet source	Linter output

`local x = x + 1;`
`true`
	`looping0.jsonnet:1:11-12 Endless`
	`↪ loop in local definition`
	
	`local x = x + 1;`

`local x = y.foo,`
`      y = if x then {foo: true} else`
`    ↪ {foo: false};`
`true`
	`looping1.jsonnet:1:11-12 Endless`
	`↪ loop in local definition`
	
	`local x = y.foo,`

Jsonnet source	Linter output

`local x = if true then x else x;`
`true`
	`looping2.jsonnet:1:24-25 Endless`
	    ↪ `loop in local definition`
	
	`local x = if true then x else x;`

`local bool = true;`
`local x =`
    `if bool then`
        `if !bool then`
            `x`
        `else`
            `42`
    `else`
        `if !bool then`
            `42`
        `else`
            `x;`
`true`
	`looping3.jsonnet:5:13-14 Endless`
	    ↪ `loop in local definition`
	
	            `x`

3.7. Future work

3.7.1. Gradual typing

Detecting some kinds of problems is very hard if not impossible without type declarations from the user. There are also other reasons for having some form of type declarations. For example, users requested providing a way of specifying a shape for an object for the purpose of data validation. There are also plans to create a documentation generator and types are an excellent form of documentation.

Many of the problems related to adding the gradual typing are related to designing the new syntax for them in a way which fits naturally within the current language and does not introduce incompatibilities. One source of such problems is lack of any declarations in the language. If we kept the assumption that every AST node is an expression, the types would need to be described by values and have value semantics - i.e. if the same definition appears twice in the code, even in different files, both occurrences would need to describe the same type.

On the other hand introducing declarations would require heavy changes to the language, especially the way imports are handled. The imports are expressions which evaluate to the value of the expression within the imported file. Libraries are organized as files containing objects, fields of which correspond to exported symbols in other programming languages. Adding declarations would require a complete redesign of how they work.

Of course there is also a question of the syntax of type annotations. One way to handle it is through assertions. This method would be quite verbose. but it allows to specify type requirements within existing syntax. The alternative is to add new syntax, which would allow a more readable specification.

Such gradual typing would be very different from the types introduced for the purposes of the linter. The types from the declarations are expected to behave like prescriptive types (see subsection 3.5.1). So the existing algorithm could still be used for narrowing the possible values which gradual typing mechanism would consider “any” or “unknown”, which will stay relevant for unannotated code[27].

3.7.2. Function requirements without gradual typing

If the gradual typing direction proves not feasible, the linter can be strengthened in a different way. A separate step could be added between type-building and checking passes, which would analyze the way in which the function uses its arguments (within its definition) and determine the requirements of the function. This is consistent with the design goals, since it derives information about the function from the function definition and from how the function is used. In the most basic form it would only deduce information from usage in expressions which are strictly evaluated every time the function is called. Unfortunately, this falls short in case of simple recursive functions which have a trivial branch, such as the following implementation of the factorial function:

```
local factorial(n, acc) =
    if n <= 1 then
        acc
    else
        factorial(n - 1, n * acc)
```

Technically, it is correct a string as `acc` and 0 as `n`. However, clearly the intention was for that argument to be an integer.

I decided to leave out this kind of analysis for now, because of its limited capabilities relative to complexity it introduced, and since it is clearly inferior to even the most basic approach based on gradual typing.

[27] I expect unannotated code to be common even after such gradual typing is introduced

Chapter 4

New utility libraries for Jsonnet

The quality of the ecosystem of a programming language depends not only on tooling, but also on the availability of the libraries. Many libraries were created for Jsonnet, most of them concerned with helping to generate a specific type of configuration. There even exists an experimental package manager for Jsonnet, called jsonnet-bundler [13].

In this chapter I briefly present the new utility libraries that I created, in response to the needs reported by some users.

4.1. Modifier library

Generally, Jsonnet is designed to work with big, nested structures — the configuration so big and complex that it is impractical or at least annoying to write by hand. With Jsonnet it is easy and convenient to build such structures from scratch and to fetch the data out of given structures. However, it is somewhat tedious to perform deep modifications, i.e. given a deep structure, return a modified version of it with some deeply nested field changed.

To understand the problem, it is best to see an example. Assume that we want to change "xxx" to a different value in the following code:

```
local obj = {
    a: {
        b: {
            c: [
                {
                    d: "xxx"
                },
                {
                    d: "yyy"
                }
            ]
        }
    },
    something: "foo",
    moreThings: "bar"
};
```

As it turns out, it is not really difficult, but requires quite a lot of code, with inheritance and a higher order function for changing an element of an array.

```
obj + { a~+: { b +: {
        c: std.mapWithIndex(
            function(i, e) if i == 0 then e + {d: "CHANGED"} else e,
            super.c
        )
    }
}}
```

This kind of code stands in sharp contrast with how such modification[1] would look like in a typical imperative language:

```
obj.a.b.c[0].d = "CHANGED"
```

To alleviate this problem I wrote a new library that allows a much easier and more concise way to perform such modifications in Jsonnet. For example the same modification as above can be expressed as follows:

```
m.change(["a", "b", "c", 0, "d"], "CHANGED")(obj)
```

4.1.1. Features showcase

The abilities of the library go far beyond simple indexing. It provides a very composable way to combine arbitrary indexing and modifications.

Performing many modifications at once

Jsonnet snippet	Jsonnet output

`local obj = {`
`    arr: ["x", "x", {"a": "b"}]`
`}`
`m.changeWith(`
`    ["arr"], m.many([`
`        m.change([0], "CHANGED-1"),`
`        m.change([2, "a"], "CHANGED`
`            ↪ -2")`
`    ])`
`)(obj)`
	`{`
	`    "arr": [`
	`        "CHANGED-1",`
	`        "x",`
	`        {"a": "CHANGED-2"}`
	`    ]`
	`}`

[1] This is a slightly different thing, since this is an in-place modification as opposed to creating the new value in Jsonnet. I think that the comparison is still fair, since it is how it would typically be written in a language such as Python.

Changing a field in all objects in an array

Jsonnet snippet

```
m.change([m.map, "a"], "foo")([
    {"a": 0},
    {"a": 1},
    {"a": 2}
])
```

Jsonnet output

```
[
    {"a": "foo"},
    {"a": "foo"},
    {"a": "foo"}
]
```

Custom selector example - indexing elements at even positions

Jsonnet snippet

```
local changeEvenPositions = function(modifier) function(arr)
    std.mapWithIndex(function(index, elem) if index % 2 == 0 then modifier(
        ↪ elem) else elem, arr)
    ;

m.changeWith([changeEvenPositions], function(x) x * 2)([1,2,3,4,5,6])
```

Jsonnet output

```
[2,2,6,4,10,6]
```

Custom selector example - transparently indexing within serialized JSON

Jsonnet snippet

```
local reparseJson = function(modifier) function(str)
    std.manifestJson(modifier(std.parseJson(str)))
    ;

local obj = {
    "foo": '{"a": {"b": "x"}}'
};

m.change(["foo", reparseJson, "a", "b"], 'CHANGED')(obj)
```

Jsonnet output

```
{
    "foo": "{\n \"a\": {\n \"b\": \"CHANGED\"\n }\n}"
}
```

4.1.2. Design

There are two basic concepts:

Selector is a function which takes a modifier and a part of the structure, extracts a part of the provided part[2] (e.g. a field, an element, all elements) and applies a modifier.

Modifier is a function which takes a part of and returns a new version of it. Technically it can be any one argument function.

We can express it more precisely in type notation[3], assuming that the type for the parts of the structure that we're indexing is P (short for part):

```
Modifier :: P -> P
Selector :: Modifier -> Modifier = (P -> P) -> P -> P
```

With such design two important operations are very natural:

- Nested indexing is simply function composition of Selectors.
- Applying multiple modifications is just function composition of (curried) Modifiers.

The library consists of utility functions and the most common selectors and modifiers. For notational convenience the library allows using numbers as selectors indexing arrays and strings and strings as selectors for choosing a field of an object.

4.2. Parser combinator library

Sometimes it is useful to parse a piece of text to validate it or extract data from it. A simple example is parsing an URL into components such as "scheme", "host", "path" and "query"[4]. One of the approaches to parsing is using *parser combinators* – functions which allow composing parsers in various ways to create more complex parsers.

Notably, Jsonnet does not provide support for regular expressions at the time of writing[5] leaving users with no solution better than parsing "manually" even in simple cases. The work on the library started as an effort to fill this gap, but even when the regular expression become available, parser combinators have other benefits. For example they are obviously they are more powerful, in terms of the languages they can describe. In particular any structures which

[2] The provided part can itself be a complex structure.

[3] Of course it is just a notation, there are no types like that in Jsonnet.

[4] This particular feature was requested by a user in a Github issue [1].

[5] The reason for lack of support for regular expressions was that the users expected extended regexp syntax and at the same time we needed to ensure precise definition and compatibility. This meant choosing a solution which has good documentation and compatible implementations in many languages. Relatively recently RE2 [21] was accepted as good enough and David Coles submitted the proposed implementation.

can be arbitrarily nested can be easily parsed with parser combinators, but not necessarily with regular expressions[6]. This may be important even in fairly simple cases. Solutions based on parser combinators can also be much more readable, since it is easy to define meaningful parts and then combine them in straightforward way. Unfortunately, these advantages come at a high performance cost in the context of Jsonnet implementation, since it is likely that multiple Jsonnet functions will need to be called for each character parsed. This may still be acceptable for small scale, occasional parsing[7].

4.2.1. Example: arithmetic expressions

As a showcase for the library, I present a complete implementation for parser and evaluator of arithmetic expressions, with four operators and parentheses:

```
local pc = import '../parser-combinators.libsonnet';

local operator1 = pc.capture(pc.any(["*", "/"]));
local operator2 = pc.capture(pc.any(["+", "-"]));

local ws = pc.in_whitespace; // make it shorter

local funcs = {
    "+":: function(x, y) x + y,
    "-":: function(x, y) x - y,
    "/":: function(x, y) x / y,
    "*":: function(x, y) x * y,
};

local calc(exprVal) =
    // expects [value, null] or [value, [operator, value]]
    if exprVal[1] == null then
        exprVal[0]
    else
        funcs[exprVal[1][0]](exprVal[0], exprVal[1][1])
    ;

local applyCalc(p) = pc.apply(p, calc);

local in_paren = pc.apply(["(", expr2, ")"], function(x) x[1]),
      expr0 = ws(pc.any([in_paren, pc.int])),
      expr1 = ws(applyCalc([expr0, pc.optional([operator1, expr1])])),
      expr2 = ws(applyCalc([expr1, pc.optional([operator2, expr2])]))
      ;
local expr = expr2;
```

[6] Some practical implementations of “regular expressions”, extend beyond regular languages. For example newer versions of Perl support recursive patterns. They are probably still not the best tool for these cases.

[7] Larger scale parsing is probably not a good fit for Jsonnet anyway. In the standard setup Jsonnet code has all the data provided as JSON and its role is to combine and present in the target form.

```
{
   expr:: expr,
}
```

4.2.2. Design

A parser is a function which given state, produces an optional value and a new state. State describes the current position in the given text and optionally a parsing error. We can express that using type notation[8]:

```
Parser :: State -> (Value, State)
State = [Pos, Text, Err or null]
```

The text can be either a string or an array, the library does not make an explicit distinction and the generic combinators should work on both.

In order to bring the benefits of readability to the users, I needed to design the API in a way which matches Jsonnet concepts and feels natural within the language. In particular, Jsonnet does not have support for operator overloading or user-defined types, which means that the interface needed to be different from some established libraries for other languages. A very simple, but important idea was to allow arrays and strings when parsers (functions) are expected. Arrays are interpreted as sequences (`pc.seq`) and strings as constant parsers (`pc.const`). This reduced the number of necessary parenthesis, which appeared to be critical for readability when I analyzed the first examples.

[8] As before, it is just a notation, there are no types like that in Jsonnet.

Chapter 5

Summary

At the time of writing this thesis, it has been over two years since I started my involvement in the Jsonnet project. During this time I implemented multiple improvements and did my best to help the community. A large part of my efforts is described in detail in the previous chapters, including the following:

- Bringing a new interpreter, implemented in Go, to a fully functional state and later improving its performance through high-level optimizations. This interpreter is now the primarily recommended version for most use cases[1].

- Creating a new tool, the linter, which allows the users to easily check for mistakes in their code.

- Authoring two small utility libraries which may help with text processing and modification of complex objects.

This was not all of the work that maintaining a real world programming language required. Besides the efforts described before, I also prepared many smaller changes. These included fixing the reported bugs and disparities between implementations, expanding the standard library with a few simple, but useful functions, adding new capabilities to the automatic code formatter, improving the test suite and keeping the documentation up to date. During the whole time I also worked directly with the community.

5.1. Directions for future work

Below I briefly describe the efforts which I think may have high, positive impact on the project[2]:

- Bringing C bindings for the Go implementation of Jsonnet to full compatibility. This is the primary obstacle for deprecating the C++ implementation. See chapter 2.

- Improving performance of Jsonnet implementation This seems to be to be the main point of complaint from the users. Currently Jsonnet is very slow, even compared with other interpreted languages. Fortunately there are many easy gains possible in this area. More details in chapter 2.

[1] C++ version may still be right for users depending on C bindings which are still very much a work in progress in the Go version.

[2] Not all of these ideas are mine and some may even predate my involvement in the project.

- Introducing a form of gradual typing, which fits well in the existing design of the language. More details in chapter 3.

- Creating an official Jsonnet test corpus – a single shared test suite for all implementations of Jsonnet interpreter and other tools which process Jsonnet code. It should contain the existing tests from Go and C++ implementation of the interpreter, together with benchmarks and real world examples.

- Creating a documentation generator. This could help create a healthy ecosystem of libraries. It will require establishing conventions for what can be documented and in what way. This is not obvious, because there are no declarations or user-defined types in Jsonnet. For the same reason, collecting this data will involve some form of static analysis.

References

[1] *Add std.parseUrl (Github issue for Jsonnet).* URL: `https://github.com/google/jsonnet/issues/630` (visited on 09/25/2019).

[2] *Command cgo.* URL: `https://golang.org/cmd/cgo/` (visited on 09/25/2019).

[3] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: `10.1145/512950.512973`. URL: `http://doi.acm.org/10.1145/512950.512973`.

[4] *Currents: A quarterly report on developer trends in the cloud.* URL: `https://www.digitalocean.com/currents/june-2018/` (visited on 09/25/2019).

[5] *grafonnet-lib.* URL: `https://github.com/grafana/grafonnet-lib`.

[6] *Initial commit (Jsonnet implementation in C++).* URL: `https://github.com/google/jsonnet/commit/55002bc60e` (visited on 09/25/2019).

[7] *Introducing JSON.* URL: `https://json.org/` (visited on 09/25/2019).

[8] Stephen C. Johnson. "Lint, a C Program Checker". In: *COMP. SCI. TECH. REP.* 1978, pp. 78–1273.

[9] *Jsonnet - The Data Templating Language (official website).* URL: `https://jsonnet.org/` (visited on 09/25/2019).

[10] *Jsonnet library for Ruby.* URL: `https://github.com/yugui/ruby-jsonnet` (visited on 09/25/2019).

[11] *Jsonnet Specification (Jsonnet official website).* URL: `https://jsonnet.org/ref/spec.html` (visited on 09/25/2019).

[12] *Jsonnet wrapper for Node.js.* URL: `https://github.com/yosuke-furukawa/node-jsonnet` (visited on 09/25/2019).

[13] *jsonnet-bundler: a Jsonnet package manager.* URL: `https://github.com/jsonnet-bundler/jsonnet-bundler` (visited on 09/25/2019).

[14] *Language Design (Jsonnet official website).* URL: `https://jsonnet.org/articles/design.html` (visited on 09/25/2019).

[15] *libjsonnet bindings for Rust.* URL: `https://github.com/anguslees/rust-jsonnet` (visited on 09/25/2019).

[16] Bruno Monsuez. "Polymorphic Types and Widening Operators". In: *Proceedings of the Third International Workshop on Static Analysis.* WSA '93. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 267–281. ISBN: 3-540-57264-3. URL: `http://dl.acm.org/citation.cfm?id=647164.717822`.

[17] Bruno Monsuez. “Polymorphic Typing by Abstract Interpretation”. In: *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK, UK: Springer-Verlag, 1992, pp. 217–228. ISBN: 3-540-56287-7. URL: `http://dl.acm.org/citation.cfm?id=646830.707693`.

[18] *Move fmt stuff out of libjsonnet.h (Jsonnet implementation in C++)*. URL: `https://github.com/google/jsonnet/commit/520961df765b1744b` (visited on 09/25/2019).

[19] Pawel S. Pietrzak et al. “A Practical Type Analysis for Verification of Modular Prolog Programs”. In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM ’08. San Francisco, California, USA: ACM, 2008, pp. 61–70. ISBN: 978-1-59593-977-7. DOI: `10.1145/1328408.1328418`. URL: `http://doi.acm.org/10.1145/1328408.1328418`.

[20] *Production-Grade Container Orchestration - Kubernetes*. URL: `https://kubernetes.io/` (visited on 09/25/2019).

[21] *RE2, a regular expression library*. URL: `https://github.com/google/re2/wiki/Syntax` (visited on 09/25/2019).

[22] *Separate jsonnet fmt into its own executable (Jsonnet implementation in C++)*. URL: `https://github.com/google/jsonnet/commit/08875537f6e350a8d` (visited on 09/25/2019).

[23] *The Google Jsonnet for LuaJIT*. URL: `https://github.com/yuduanchen/luajit-jsonnet` (visited on 09/25/2019).

[24] *The Google Jsonnet for PHP*. URL: `https://github.com/Neeke/Jsonnet-PHP` (visited on 09/25/2019).

[25] *Tutorial (Jsonnet official website)*. URL: `https://jsonnet.org/learning/tutorial.html` (visited on 09/25/2019).

[26] *Update go-jsonnet dependency to include the entire runtime (ksonnet)*. URL: `https://github.com/ksonnet/ksonnet/commit/098f5fc17c` (visited on 09/25/2019).

[27] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[28] *Writing a Faster Jsonnet Compiler (Databricks blog post)*. URL: `https://databricks.com/blog/2018/10/12/writing-a-faster-jsonnet-compiler.html` (visited on 09/25/2019).