Rust types from JSON samples

Approximating type providers with procedural macros in Rust

Erik Vesteraas

Thesis submitted for the degree of
Master in
Informatics: Programming and Networks
60 credits

Institute of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017

Rust types from JSON samples

Approximating type providers with procedural macros in Rust

Erik Vesteraas

Rust types from JSON samples

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Contents

List of Listings

List of Figures

Abstract

When programmers access external data in a statically typed programming language, they are often faced with a dilemma between convenient and type-safe access to the data.

In the programming language F#, a concept called type providers has been proposed as a solution to this problem by having compiler support for libraries with the capability to generate types at compile time.

This thesis presents *json_typegen*, a project which aims to show the feasibility of similar solutions in the Rust programming language. The project uses compile-time metaprogramming along with alternative interfaces to the same code generation implementation to achieve convenient, type-safe access to data in the JSON data format. While JSON is chosen as the format for the presented library, the approach also applies to other data formats and sources.

Acknowledgements

I would like to thank my supervisors, Volker Stolz and Martin Steffen, for their guidance and feedback, as well as for their trust in me from the moment I approached them with my own idea, and throughout my forays into exploring other ideas within Rust for this thesis.

I would also like to thank my family for their support, and in particular my wife, Anette, for her valuable feedback, and her wonderful drawings of Ferris, the Rust mascot, which make this thesis a little more pleasant to look at.

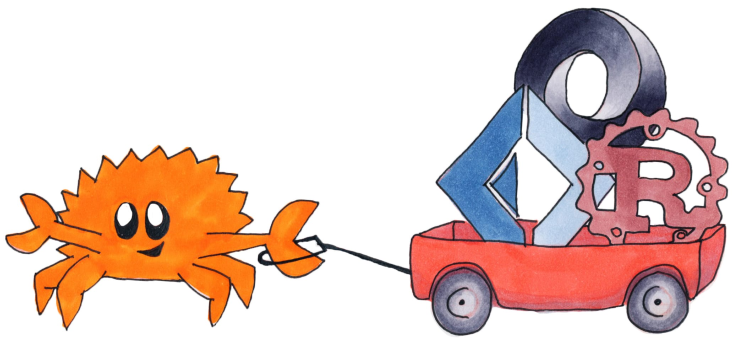
Finally, I would like to thank the Rust community, as every interaction with someone from the Rust community, whether over IRC, online forums or in person, has continually shown me that beyond the Rust language, the community that is *around* Rust in itself makes Rust worth investing time and energy into.

Reading notes

The most recent version of this document can be found at http://erik.vestera.as/thesis/ along with links to the various implementation parts of the project. The LaTeX source can be found at https://github.com/evestera/thesis. If you come across any typos while reading this, or have any feedback in general, feel free to open an issue at https://github.com/evestera/thesis/issues or send me an email at erik@vestera.as.

Typographic conventions

Clickable link	Rust Programming Language
Inline code	`true || false`
Project or library name	*json_typegen*



Chapter 1

Introduction

Whether communicating with an API, reading configuration files, or using a static dataset, most modern programs at some point have to interact with external data sources. And with the rise of micro-services it is not uncommon to have to access data from many such external data sources in a single program. When programmers write code that accesses external data in a statically typed programming language, they are often faced with a dilemma between convenience and type-safety. For type-safe access to external data, significant boiler-plate in the form of code with abundant checks or a large amount of custom types must usually be written. While approaches that avoid this boiler-plate often abandons many of the benefits of static types.

In the programming language F#, a concept called type providers has been proposed as a solution to this problem by having compiler support for libraries with the capability to generate custom types, based on the external data, at compile time. With this method, a type provider offers access to external, potentially complex resources, in a way that is both convenient and type-safe.

This thesis presents *json_typegen*, a project which aims to show the feasibility of similar solutions in the Rust programming language. The project uses compile-time metaprogramming, along with alternative interfaces to the same code generation implementation, to achieve convenient, type-safe access to data in the JSON data format, in a way similar to the type providers of F#. The presented project provides support for the very common JSON format, but the approach, and much of the actual code used by *json_typegen*, can be applied to create similar tools for other data formats and sources.

In this introductory chapter we will look at the background for the *json_typegen* project. First, in section 1.1, we will look at the Rust programming language. Then, in section 1.2, we will look at the JSON data interchange format, and the challenges associated with accessing data in JSON from statically typed programming languages, and Rust in particular. Finally, in section 1.3 we will look at how these challenges have been attempted solved with type providers in the programming language F#.

In chapter 2, the project and its interfaces is presented in detail. And in chapter 3 we will look at how the code inference and code generation used in the project works, and ways in which it could be extended.

1.1 The Rust programming language

The project presented in the thesis, *json_typegen*, creates types for programs written in the Rust programming language, and is itself written in Rust. The Rust programming language is a modern, open source programming language with C-like surface syntax. It is a statically typed language with type inference, with a focus on memory safety and zero-cost abstractions.

Rust does not use a garbage collector and the core library can be used without access to heap allocation. This allows Rust to be used for what is often referred to as "systems programming", e.g. for writing drivers, operating systems and code for microcontrollers. Spaces which has so far been occupied mainly by C and C++[1]. Rust tries to combine the low-level control of these languages with modern syntax and an advanced type system that statically prevents whole classes of memory safety issues [16].

While the basic syntax of Rust can be said to be C-like it also has a lot of functionality and syntax reminiscent of functional languages and in particular languages of the ML-family. Features such as closures, pattern matching and monadic error handling are available and a significant part of idiomatic Rust code.

What follows is a very basic introduction to the parts of Rust we need to talk about

[1]There are *some* other languages that can be said to compete in this area, like Ada, Objective-C and D. I won't go into the details of how these languages compare to Rust, but for various reasons C++ is the main competitor and point of comparison for most use cases.

in this thesis. Many of Rusts more advanced language features are not necessary to understand the basic concepts of this thesis, and are as such not explained here. For more details on the language features of Rust, see the Rust book[2].

1.1.1 Structs

```
struct Person {
    name: String,
    age: i64,
}
```

Listing 1: A basic Rust struct

The primary language construct for complex types in Rust are structs. A struct declaration, as shown in Listing 1, is a sequence of field names, along with their types. Structs are product types and are analogous to records in some languages. Rust does not have classes or objects, and is not object oriented in the classical sense. In other words is there no inheritance between structs, and polymorphism in Rust is instead supported by other language features, such as enumerated types, and traits.

1.1.2 Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

Listing 2: The enumerated type **`Option`** in Rust

An enum in Rust is a data type that is one alternative from a number of variants, i.e. an enumerated type, sum type or union. A declaration of an enum, as seen in Listing 2 is given as a list of variant names, used as constructors. Each variant can optionally carry some associated data.

[2]https://doc.rust-lang.org/book/

Rust supports pattern matching on enum types and verifies that any match is exhaustive.

1.1.3 Traits

A trait specifies methods that a type needs to provide to implement the trait. On a superficial level, traits in Rust are similar to interfaces in languages like Java, and type classes in Haskell.

```rust
trait Clone {
    fn clone(&self) -> Self;
}

#[derive(Debug)]
struct Bag {
    label: String
}

impl Clone for Bag {
    fn clone(&self) -> Self {
        Bag {
          label: self.label.clone()
        }
    }
}
```

Listing 3: The Rust trait **`Clone`** and examples of implementation

Listing 3 shows the trait **`Clone`**, which is a slightly simplified version of a trait in the Rust standard library. The trait defines the function **`clone`**, which takes as its argument a reference to an instance of the type implementing the trait – **`&self`** – and returns a new copy of the same type – **`Self`**.

The listing also shows a struct, **`Bag`**, which implements two traits. **`Bag`** implements the trait **`Clone`** with an **`impl Clone`** block, providing an implementation of the required function.

In addition to the manual implementation of **`Clone`**, the trait **`Debug`**, is "derived", i.e. automatically implemented by the compiler, using an annotation, **`#[derive(Debug)]`**. The code for this automatic implementation is included with the compiler, but it is also possible to write libraries that can derive custom traits. Deriving a trait usually requires that the constituent types also implements the trait. E.g. to derive **`Clone`** for **`Ty`** you need to be able to clone all parts of **`Ty`**.

1.1.4 Macros

A macro system is in short a language feature that allows a programmer to write code that does source-to-source transformations, also known as metaprogramming. In Rust there are two categories of macros: declarative macros and procedural macros, which are both expanded at compile time.

Macro declaration:
```
macro_rules! double {
    ($e: expr) => ({
        let temp = $e;
        temp + temp
    })
}
```

Macro usage:
```
let ten = double!(2 + 3);
```
Listing 4: A simple declarative macro in Rust

Declarative macros in Rust are also known as "macros by example" or "pattern-based macros". These are syntactic, hygienic macros that with a syntax similar to pattern matching, match input patterns to expanded source code. Listing 4 show declaration and usage of a very basic declarative macro. The macro **`double`** has a single rule transforming a single expression to a block expression. A macro can have multiple rules and each pattern can be a combination of tokens and metavariables.

Procedural macros are macros where the expansion is done by running a procedure rather than by evaluating rules. To create a procedural macro in Rust you create a library that exposes a function that takes a **`TokenStream`** as input and outputs a **`TokenStream`**. Since this library and function can use arbitrary Rust code to generate its output, it can do things that are outside the scope of the normal compiler. There are several examples of powerful procedural macro libraries in Rust already. Two prominent examples are:

- *diesel* [1], an ORM and query builder for Rust, has a macro which for generating a type-safe DSL (at compile time) for communicating with an SQL database, by inspecting said database.
- *vulkano* [8], which wraps the Vulkan graphics API, can compile graphics shaders at Rust compile time.

1.1.5 Cargo

Cargo is the package manager and build tool for the Rust ecosystem. While not a part of the Rust programming language itself, it is provided alongside the compiler in every normal installation. Cargo makes it easy to create new packages – “crates” in Rust terminology – build them, and manage other crates as dependencies.

While users of some programming languages may be accustomed to build systems with good dependency management this is not the case for users of the traditional systems level languages, C and C++. The part of *json_typegen* that comes closest to working like a type provider is a library the user adds a dependency to their project. Thanks to the availability of Cargo, this is an almost insignificant barrier for potential users of the project.

For a perspective on the difference in the development experience it is useful to look at how the experience of going from nothing to a minimal project with a dependency is in Rust and C++.

In appendix D I have included a comparison of setting up a project using both Cargo and what I currently consider to be the best alternative in the C++ ecosystem. While the specifics are not important, suffice to say that setting up a project and adding dependencies in Rust using Cargo is significantly easier and more easily reproducible than the equivalent scenario using C++.

1.2 JSON

JSON is a text-based data format for structured data. It is very commonly used as a data interchange format in modern HTTP-based architectures, but also in various other use cases like configuration files. JSON was originally based on a subset of the JavaScript programming language, but is designed to be language-independent and is now used in the interaction between applications written in practically all programming languages.

$$\begin{aligned}
\mathit{value} &::= \mathit{object} \mid \mathit{array} \mid \mathit{number} \mid \mathit{string} \mid \texttt{\textbf{true}} \mid \texttt{\textbf{false}} \mid \texttt{\textbf{null}} \\
\mathit{object} &::= \texttt{\textbf{\{}}\ [\ \mathit{string}\ \texttt{\textbf{:}}\ \mathit{value}\ {*}(\ \texttt{\textbf{,}}\ \mathit{string}\ \texttt{\textbf{:}}\ \mathit{value}\ )\ ]\ \texttt{\textbf{\}}} \\
\mathit{array} &::= \texttt{\textbf{[}}\ [\ \mathit{value}\ {*}(\ \texttt{\textbf{,}}\ \mathit{value}\ )\ ]\ \texttt{\textbf{]}} \\
\mathit{number} &::= [\ \texttt{\textbf{-}}\ ]\ \mathit{int}\ [\ \mathit{frac}\ ]\ [\ \mathit{exp}\ ] \\
\mathit{string} &::= \texttt{\textbf{"}}\ {*}\mathit{char}\ \texttt{\textbf{"}}
\end{aligned}$$

Listing 5: The JSON grammar from RFC 7159. It is somewhat simplified as the actual specification is very precise. See the full specification for the exact definitions of *int*, *frac*, *exp* and *char*.

The format as described by the two official JSON specifications – IETF RFC 7159 [9] and ECMA 404 [19] – is intended to be very simple to write and to parse. As can be seen from the grammar in Listing 5 a JSON value can only be an object, array, number or string, or one of the literals **`true`**, **`false`** and **`null`**. In addition, there is no way to define new types, references or to specify external resources or definitions.

A JSON object is a collection of mappings from keys (that are JSON strings) to JSON values. As this closely corresponds to what is usually referred to as a map or a dictionary in most programming languages, the term object may seem like a bit of a misnomer here, arising from JSONs background as a subset of JavaScript. Whether or not the fields of an object are ordered is explicitly unspecified by RFC 7159, but it states [9, p. 6] that "implementations whose behavior does not depend on member ordering will be interoperable in the sense that they will not be affected by these differences."

A JSON array is an ordered collection of JSON values. It is worth noting that JSON places no restriction on the types of the values of the array, and as such, it can sometimes be a heterogeneous collection.

```json
{
  "name": "Bob",
  "age": 24,
  "phoneNumbers": [
    {
      "areaCode": 456,
      "number": 80931
    }
  ]
}
```

Listing 6: An example JSON object

Listing 6 shows a basic example of a JSON object. Since JSON objects and arrays can themselves contain objects and arrays, data serialized as JSON can be arbitrarily deeply nested, but as mentioned JSON itself has no support for any kind of references, so there is no possibility of loops or infinite types.

In RFC 7159 it is stated that "the representation of numbers is similar to that used in most programming languages." However, there are some very significant differences to the most normal number types in most programming languages. A number in JSON can be arbitrarily large, and arbitrarily precise. The only actual rule for a number to be valid JSON is that it satisfies the grammar given in the specifications. As this shows, while the specifications are quite simple, this simplicity leads to some complications and pitfalls.

A JSON string resembles string literals in many programming languages, with characters and escape sequences, like `\n` or `\u000a`, enclosed in quotation marks. As seen from the grammar, strings serve double duty in JSON as both object keys and general values. This means that many possible JSON keys can be quite challenging to map to valid identifiers in a programming language. It is for example perfectly valid to have an object with the keys `""`, `"  "` and `"µ"`.

1.2.1 *Reading JSON in dynamically typed languages*

```
var parsed = JSON.parse(jsonString);
console.log(parsed.phoneNumbers[0].areaCode);
```

Listing 7: Printing the first areaCode in JavaScript

```
parsed = json.loads(jsonString)
print(parsed["phoneNumbers"][0]["areaCode"])
```

Listing 8: Printing the first areaCode in Python

In dynamically typed programming languages reading data from deserialized JSON is relatively straightforward, as seen in Listings 7 and 8, showing parsing and reading in JavaScript and Python[3] respectively. The code in these Listings can, however, fail at runtime with exceptions or errors if the deserialized data does not match expectations, e.g. if the field **`"phoneNumbers"`** is missing, or contains a number instead of an array. So in many real world use cases the code will be a bit more complex, to deal with such problems without crashing.

1.2.2 *Reading JSON in statically typed languages*

In statically typed programming languages reading data from deserialized JSON often requires a bit more effort. There are several possible (and relatively common) approaches.

For my examples I will use Rust, but first I need to introduce *Serde*:

Serde

Serde [5] is a Rust framework for serialization and deserialization. The core of the framework is independent of the source/target data format, and the input/output of specific data formats is provided by separate crates.

[3]It is also possible in Python to get the JSON deserialized into such forms that its syntax become more like the direct field access of JavaScript, but the code shown uses the default behaviour.

Serde works by providing two traits, **`Serialize`** and **`Deserialize`**, that Rust types can implement. The core library has already implemented these traits for the most common data types like **`i64`**, **`String`**, **`Vec`** and **`HashMap`**. In addition, the crate *serde_derive* provides code for deriving **`Serialize`** and **`Deserialize`** for custom – i.e. your own – types.

Once a Rust type implements the necessary trait, data can be converted with the various format specific crates, that provide implementations for one or both of the traits *Serializer* and *Deserializer*. *serde_json* [6] provides implementations of these types for JSON. It also provides some other functionality that is useful for working with JSON, such as helper functions, a macro for creating arbitrary JSON, and a catchall type for JSON values.

Catchall types

```rust
enum Value {
    Null,
    Bool(bool),
    Number(Number),
    String(String),
    Array(Vec<Value>),
    Object(Map<String, Value>),
}
```

Listing 9: An enumerated type in Rust for JSON values

Listing 9 shows the enumerated type **`Value`** from the library *serde_json*, which is this catchall type. In other words can any legal JSON value can be represented by this single type. Similar types can be created in most statically typed programming languages. However, working with such types in a type-safe manner can be both tedious and error-prone, when trying to get some specific data. The code in Listing 10 does approximately the same thing as the code in Listings 7 and 8, using pattern matching[4]. While the code is type-safe and will not fail at runtime, and the code verifies our assumptions about the structure of the data while unwrapping, it is not particularly convenient to write.

[4]And for now, ignoring the helper functions provided by *serde_json*

```rust
let parsed: Value = serde_json::from_str(json_str)?;

use Value::*;
if let Object(map) = parsed {
    if let Some(Array(vec)) = map.get("phoneNumbers") {
        if let Some(Object(map)) = vec.get(0) {
            if let Some(Number(num)) = map.get("areaCode") {
                println!("{}", num);
            }
        }
    }
}
```

Listing 10: Printing the first areaCode in Rust using pattern matching

More weakly typed approaches

One way to make the catchall types easier to work with is to create helpers that essentially skips individual checking of each field access. Listing 11 shows two alternative ways to read from the parsed data.

```rust
let parsed: Value = serde_json::from_str(json_str)?;

println!("{}", parsed["phoneNumbers"][0]["areaCode"]);

if let Some(num) = parsed.pointer("/phoneNumbers/0/areaCode") {
    println!("{}", num);
}
```

Listing 11: Printing the first areaCode in Rust using indexing expressions and JSON pointers

The first way shows how it is possible to use indexing syntax – **`container[index]`** – with the **`Value`** type, due to the fact that it implements the **`Index`** trait. In other words, very similarly to the syntax one would use with e.g. the default Python solution. While index expressions in Rust *can* panic, the shown expression will actually not do so, as the **`Value`** implementation of the **`Index`** trait instead returns **`Value::Null`** if indexing fails. This does, however, conflate when indexing fails, or even makes no sense at all, and when the data contains an actual **`Null`** value. E.g. trying to read the fourth element of **`false`** would return **`Null`**, behavior that is not normally expected in a strongly typed language.

The second approach shown instead uses a string, which encodes a path[5], to combine multiple **`get()`** calls into a single function call. This call returns an **`Option`** to distinguish between **`Null`** values and lookup failure, but is otherwise quite similar to the indexing method.

Neither of these approaches, nor the first approach with the individual **`get()`** calls actually gives us a number in the form of a Rust integer (**`i64`**), though. What we actually get is[6] a **`Value::Number(Number { ... })`** which means that if we wanted to do something else than just printing it, our code would still have some unwrapping to do. And with all this, if we misspell a field, we may not even know that the code failed at runtime because of a typo and not some problem with the data[7].

Custom types

What we actually want are custom types, that accurately reflects the actual data. This way, any mismatches between our assumptions and the actual structure of the data is revealed at the time of JSON parsing, and any misspelling of field names is discovered at program compile time.

Listing 12 shows an example of using such custom data types with the *serde* framework. However, even for this small example this approach required 9 significant lines of code for the types. And with real-world data sources, actual JSON data can be significantly larger and more complex. So while custom types are preferable once written, actually writing them can often be a significant hurdle.

[5]The path is encoded as a JSON Pointer. We will come back to JSON Pointers in more detail in section 2.6.3.

[6]Assuming the input was actually the string in Listing 6.

[7]This encoding of data access in strings rather than in the type system, leads some to jokingly refer to this as a “stringly” typed, rather than a strongly typed, approach.

```rust
#[derive(Deserialize)]
struct Person {
    name: String,
    age: i64,
    phoneNumbers: Vec<PhoneNumber>,
}

#[derive(Deserialize)]
struct PhoneNumber {
    areaCode: i64,
    number: i64,
}

let parsed: Person = serde_json::from_str(json_str)?;

println!("{}", parsed.phoneNumbers[0].areaCode);
```

Listing 12: Printing the first areaCode in Rust using custom types

1.3 Type providers

Solving the problem of convenient, type-safe access to external resources is of course something that has been thought about before this thesis. F# is an open source, functional programming language for the Common Language Runtime. Like Rust it is statically typed, and draws inspiration from the ML-family of languages. In version 3.0 of F# a concept called “type providers” was introduced.

As the name suggests the type provider “provides” the necessary custom types to the compiler, letting the programmer access the resource without having to spell out the type information manually.

Listing 13 shows an example of a type provider, **`JsonProvider`**, in use. The type provider is given a static parameter, in this case a string of JSON. A type is generated from this sample at compile time, and the type is then used to parse another instance, with the same structure, at runtime. The parsed object is then available to be used in a type-safe manner, just like any other instance of a custom type.

```
open FSharp.Data

type Simple = JsonProvider<""" { "name":"John", "age":94 } """>

[<EntryPoint>]
let main argv =
    let simple = Simple.Parse(""" { "name":"Jane", "age":4 } """)
    printfn "%s %d" simple.Name simple.Age
    // prints "Jane 4"
    0
```

Listing 13: Minimal example of the use of a type provider in F#

To reiterate: the expression `JsonProvider<...>` is evaluated at compile time and provides a type to the compiler. This lets the compiler know the type of, and type check, expressions like `simple.Age`, even though the type, or even the existence, of the field `Age` is not visible anywhere in the actual F# code. This is somewhat reminiscent of how type inference lets the compiler lets the compiler know the type of a variable like `name` in an expression like `let name = "Bob"`, even if the type is not visible in code.

The particular type provider shown, `JsonProvider`, comes from the library *F# Data* [2], which has type providers for multiple formats, and the example is adapted from its documentation[8]. Both the sample string and the string providing the instance could have been replaced by paths to either local or remote resources:

```
type Simple = JsonProvider<"http://example.com/person.json">
type Simple = JsonProvider<"samples/person.json">
```

The compiler also provides functionality that makes it easy for implementors of editors to provide autocomplete and similar functionality for types generated by type providers. Thus, when writing code that talks to a JSON based API, it is often possible to just point the type provider to an endpoint or a sample from some documentation, and get the types one needs to start working. Possibly even without looking at the JSON, and instead using autocomplete to explore the data.

[8] http://fsharp.github.io/FSharp.Data/library/JsonProvider.html

1.3.1 *Type providers from an implementation standpoint*

So what *is* a type provider? How do they work? From an implementation standpoint, support for type providers can be boiled down to two features [18]:

- Compile-time metaprogramming with access to side effects[9], with the expressiveness of a full programming language.
- Support for thunks, i.e. lazy or delayed evaluation, of at least parts of the representation of these types, and letting such thunks be made by compile-time metaprogramming.

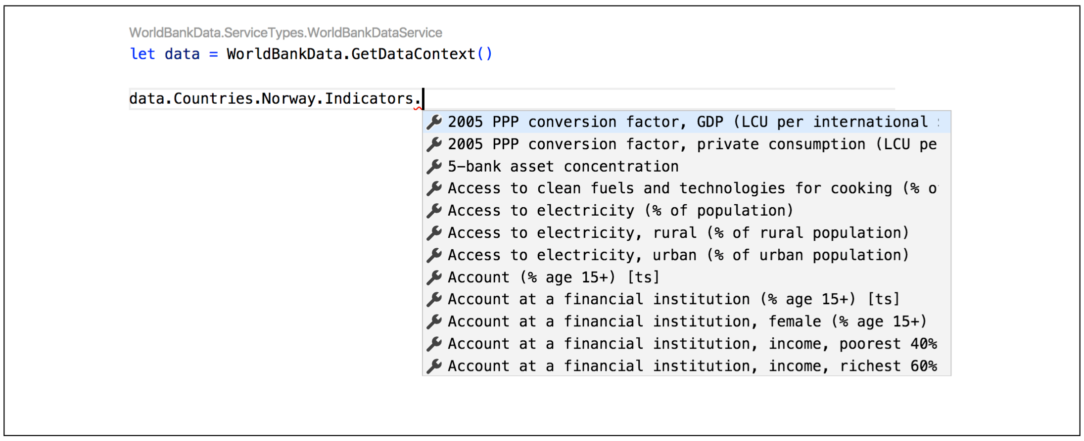
Additionally, for a type provider to make sense, the language has to be statically typed. In a dynamically typed language, the types are associated with the runtime instances of values, so providing extra type information before runtime does not provide much benefit. Since type providers can create a large number of types it is also beneficial if the language has type inference so the actual names of the sub-types are less significant.

As we looked at in section 1.1.4, Rust does have compile-time metaprogramming with the flexibility of a full programming language, in the form of procedural macros. It is also a statically typed language with type inference. Rust does however not have any ability for procedural macros to let parts of the generated code be created lazily.

In F# type provider implementations it is possible to add type members as thunks, which are then only evaluated as required by the compiler. This makes it possible for type providers to provide access to data which whose type spaces would otherwise be too large to create custom types for. Just like lazy evaluation makes infinite lists possible, and practical, in languages like Haskell, thunks as parts of the generated types makes arbitrary large or infinite type hierarchies possible, and practical, in F#.

A good example of a type provider which makes use of this functionality is the World-Bank type provider in *F# Data*. As the name suggests, it provides access to the World

[9]The paper *Dependent Type Providers* [12], which shows the use of dependent types in the Idris programming language to create type providers, shows that "compile-time metaprogramming with access to side effects" could possibly be replaced with "type creation abilities with side effects".



```
WorldBankData.ServiceTypes.WorldBankDataService
let data = WorldBankData.GetDataContext()

data.Countries.Norway.Indicators.
```

```
2005 PPP conversion factor, GDP (LCU per international
2005 PPP conversion factor, private consumption (LCU pe
5-bank asset concentration
Access to clean fuels and technologies for cooking (% o
Access to electricity (% of population)
Access to electricity, rural (% of rural population)
Access to electricity, urban (% of urban population)
Account (% age 15+) [ts]
Account at a financial institution (% age 15+) [ts]
Account at a financial institution, female (% age 15+)
Account at a financial institution, income, poorest 40%
Account at a financial institution, income, richest 60%
```

Figure 1.1: Autocomplete from the WorldBank type provider

Bank data catalog via its web API. Figure 1.1 shows a screenshot of the kind of autocomplete that is available when using the type provider. Having such functionality makes it quick and easy to dive into the data, and it would be very hard, or impossible, to replicate without some sort of strategy for only generating what is necessary.

While thunks as parts of generated types is clearly a very powerful feature, many of F#s type providers are not reliant on it. If it is feasible to have all the necessary data available at once, and the generated type is not too large, delayed production of the code is not required. For example the **`JsonProvider`** from *F# Data* which creates a single type hierarchy from a limited set of samples, does not need thunks. In other words, something like **`JsonProvider`** *should* be possible in Rust. In chapter 2 I will show how I have attempted to achieve this, but let us first look at the advantages and disadvantages of the type providers it is inspired by.

1.3.2 Advantages of F#s type providers

The most significant benefit of type providers is that they enable type-safe access to external resources, significantly improving type checking for sections of code using data from a type provider.

As mentioned, such type-safe access can also be achieved by manually writing the custom types, but the ease of use of a type provider makes it very easy to add a new external resource and start experimenting. Not only is the barrier of having to write boilerplate code reduced or removed, but type information gleaned through e.g. autocomplete facilitates exploration of the resource.

When working with external data, the code we write encode our assumptions about the data we are working with. E.g. if I have parsed some data into a variable **`p`** and I write code like **`let real_name: String = p.name`**, the code encodes my assumptions that the data has a field with the name **`name`**, and that said field is a string. When the types are in sync with the external data the type checking will thus check that our assumptions match the actual structure of the data. Additionally, the type checking will *only* check the assumptions we state through our use of the data, which are likely to be the ones we care about.

Since these assumptions are re-checked when we re-compile this can help us not just when our assumptions change, but also when the things we are making assumptions about change. In this way a type provider like **`JsonProvider`** can help alert us about changes to remote resources while we are developing or in a continuous integration setting.

The most advanced type providers can thanks to the delayed evaluation of AST encode information that would otherwise be entirely impractical to otherwise encode in the type system. We will be hard pressed to recreate this in Rust since we will not have access to delayed evaluation, but there may be other ways that we may be able to make different trade-offs and thus explore new possibilities.

1.3.3 Disadvantages of F#s type providers

While F#s type providers have significant advantages, they also have some disadvantages. Type providers are essentially a code generation tool. However, type providers do not give the user access to the generated code. For type providers that use thunks, it is of course not a real option to generate the full code to be able to show it, as the “full” code could in fact be infinite. And if you just show the code that is already forced by the program usage, no new information is provided.

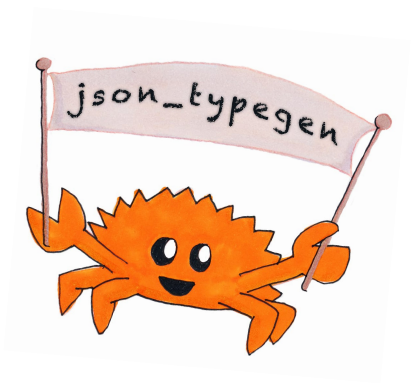
Since the difference between type providers that use delayed evaluation and those that don’t is completely abstracted away from the users of the type providers, this means that there is no option to look at the generated code to understand what is going on and to inspect the structure of the parsed data. While some of the same understanding can be achieved by looking at the autocomplete suggestions it is often easier to understand the structure by looking at the actual types.

No access to the generated code also causes a certain amount of lock-in. Without access to the code, trying to migrate a project using type providers to manually written types forces a complete rewrite of the types provided by the type provider. In the case of parsing JSON and generating types from it there are many trade-offs that have to be made, and thus there might be reasons for such a migration as a project develops. I will come back to these trade-offs in section 2.6.

A completely different issue with certain type providers is that they provide the greatest benefit with concern to the type checking when verifying the use of an external resource. However, if access to this external resource requires network access, this causes the compilation of the program to require network access for every build. Sometimes requiring network access to build is not a big concern, but other times it can be a big annoyance or even be completely out of the question.

Another concern many people have with type providers is that since the type checking checks if the code matches the data, the build can break if the structure of the data changes. In other words, the advantage of having our assumptions checked by the type checking also means that the build can break without any changes to the things we manage with version control. Such breakage runs contrary to the concept of reproducible builds which is a very common goal for build systems. One may counter that an external data source like an API is part of the system when it is used in a program, and as such a build failure is the correct behavior, but it is nevertheless a contentious issue.

While these very real concerns are not something that dissuade us from pursuing the significant benefits that type providers can give us, we should at the same time keep them in mind as we go forward.



Chapter 2

Presentation of the project

Type providers have significant benefits that are worth exploring in other programming languages like Rust. Inspired by *F# Data* I have created a project, *json_typegen*, that aims to approximate the benefits of type providers in Rust. In particular I have focused on *F# Datas* type provider for JSON, **`JsonProvider`**.

The *json_typegen* project is divided into five crates, as well as a crate demonstrating the use of the main crate. Most important of these are one library – a procedural macro – and two binaries – a command-line interface and a web interface. Figure 2.1 shows how the crates depend on each other.

$$\begin{tikzcd}
 & \boxed{\text{json\_typegen\_shared}} & \\
\boxed{\text{json\_typegen\_web}} \arrow[ur] & \boxed{\text{json\_typegen\_cli}} \arrow[u] & \boxed{\text{json\_typegen\_derive}} \arrow[ul] \\
 & & \boxed{\text{json\_typegen}} \arrow[u] \\
 & & \boxed{\text{json\_typegen\_demo}} \arrow[u]
\end{tikzcd}$$

Figure 2.1: Internal dependencies in the project

2.1 The procedural macro

The crate *json_typegen* provides a procedural macro of the same name, **`json_typegen`**. This macro provides an interface very similar to *F# Datas* **`JsonProvider`**.

```rust
#[macro_use]
extern crate json_typegen;
extern crate serde_json;

json_typegen!("Point", r#"{ "x": 1, "y": 2 }"#);

fn main() {
    let mut p: Point =
        serde_json::from_str(r#"{ "x": 3, "y": 5 }"#).unwrap();
    println!("deserialized = {:?}", p);
    p.x = 4;
    let serialized = serde_json::to_string(&p).unwrap();
    println!("serialized = {}", serialized);
}
```

Listing 14: Usage of the procedural macro

In Listing 14 a minimal example of usage of the procedural macro **`json_typegen`** is shown. This example is provided in the project source as a demo crate, *json_typegen_demo*. It is worth noting that the calls to **`unwrap`** unwraps result values assuming success, and will crash the program in the event of a serialization or deserialization failure. In a real world use case, we would replace these calls with error handling code.

As can be seen in the example, the procedural macro defines a type – **`Point`** – which is then available to be used by the programmer as any other type. The type gives type-safe access to its fields, in the example **`p.x`**. Attempts to access an invalid field – e.g. **`p.z`** – would be a type error and thus be caught at compile time.

Like **`JsonProvider`**, the procedural macro supports inline samples as shown, samples stored as local files and URLs that point to remote samples:

```rust
json_typegen!("Point", "http://example.com/point.json");
```

```
json_typegen!("Point", "samples/point.json");
```

Invocation of procedural macro:

```
json_typegen!("Point", r#"{ "x": 1, "y": 2 }"#);
```

Generated code:

```
#[derive(Default, Debug, Clone, PartialEq,
         Serialize, Deserialize)]
struct Point {
    x: i64,
    y: i64,
}
```

Listing 15: The code generated by a macro invocation

Based on the JSON samples, types are inferred and Rust code with type declarations is generated. Listing 15 shows the code generated by the macro shown in the example in Listing 14. Further examples of the resulting generated code from different JSON inputs can be seen in appendix C.

The types generated by *json_typegen* are all built up either by other generated types, or from a small set Rust types we will refer to as our base types:

- **`Vec<T>`**, growable arrays, and **`Option<T>`**, optional values, with the type parameter **`T`** always being either another of our base types or an earlier generated type.
- **`i64`**, 64-bit integers, and **`f64`**, 64-bit floating point numbers
- **`String`**, heap-allocated strings
- **`bool`**, boolean values
- **`serde_json::Value`**, *serde_jsons* catchall type, as a fallback when the inference gets either not enough or conflicting information

In chapter 3 we will look in more detail at how these types are inferred and how the types are generated.

2.1.1 Limitations compared to `JsonProvider`

While the procedural macro looks syntactically similar to, and in many ways works like **`JsonProvider`**, there are some significant differences in actual use. For some features, further support from the compiler and the tooling would be necessary.

At the moment there is no tooling solution for Rust that provides autocomplete for types generated by procedural macros. There is however, no fundamental limitation at play here, and is more likely connected to how young the Rust ecosystem is. As such, there is reason to believe that this may change in the future. However, how long it may be until such a tooling solution exists is mostly guesswork at this point.

Procedural macro hack

Another limitation of the current implementation is that the macro can only be used once per scope. This is a consequence of the fact that, at the time of writing, function-like procedural macros are currently not enabled on the stable version of the Rust compiler.

To work around the fact that function-like procedural macros are not yet fully enabled **`json_typegen`** is actually a normal pattern-based macro. The invocation of this macro expands to the declaration of a dummy type that uses a custom derive that is implemented in *json_typegen_derive*[1]. Due to a limitation in current pattern-based macros the macro is unable to create new names for each dummy type. Because of this two invocations of the macro would create two (unused) types with the same name, which would result in a compilation error.

In my testing thus far I have not encountered a use case where I needed to work around this limitation. If necessary the easiest way to work around is to wrap the macro invocation in module scopes, as shown in Listing 16, and if desired import the created types. Another workaround is to write the code the pattern-based macro expands to, and use *json_typegen_derive* directly. However, in the near future, no hack should be needed at all as function-like procedural macros become available

[1]This hack is demonstrated in isolation, and described in more detail at https://github.com/dtolnay/proc-macro-hack

```
mod point {
    json_typegen!("pub Point", "point_sample.json");
}

mod vector {
    json_typegen!("pub Vector", "vector_sample.json");
}

use point::*;
use vector::*;
```

Listing 16: Workaround for the single use limitation imposed by the procedural macro hack

on the stable compiler[2]. For users of *json_typegen* this change will most likely be unnoticeable, with no change to the exposed API.

2.2 The command-line interface

Invocation of procedural macro:

```
json_typegen!("Point", r#"{ "x": 1, "y": 2 }"#);
```

Equivalent run of the CLI:

```
json_typegen -n Point '{ "x": 1, "y": 2 }'
```

Listing 17: Equivalent uses of CLI and macro

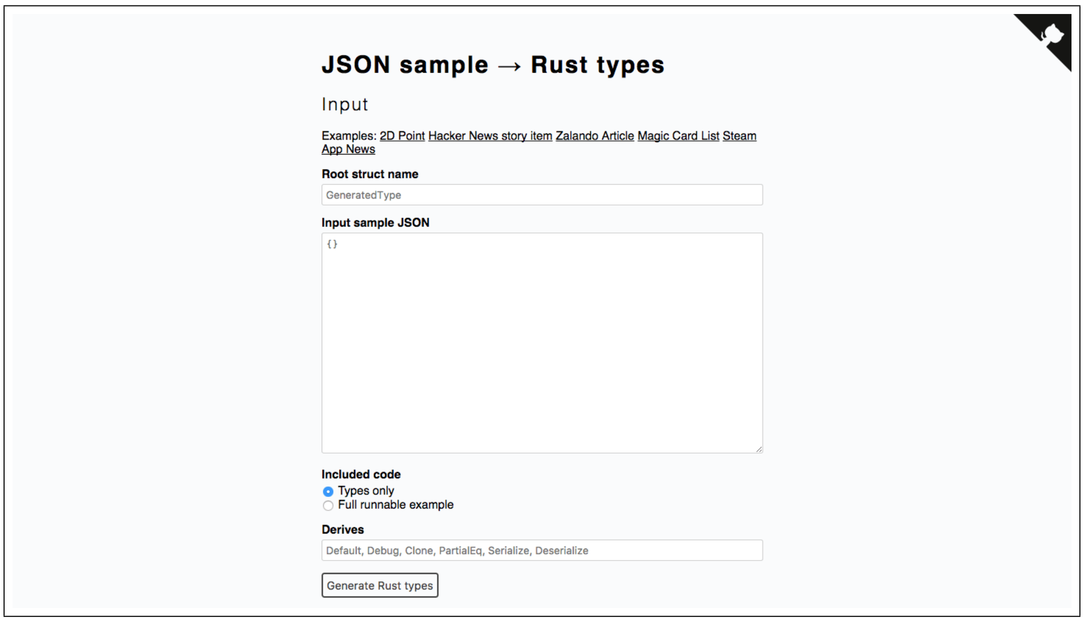
While the code generation in `JsonProvider` and the other type providers in *F# Data* is only available as a type provider, I have chosen to explore the opportunity multiple interfaces to the *json_typegen* project. The crate *json_typegen_cli* provides a binary, `json_typegen`, which is a command-line interface (CLI) to the same code generation that is used by the procedural macro. In other words, running the binary `json_typegen` and invoking the macro `json_typegen` will output the same code if the same input sample and options are given.

[2]Tracking issue for procedural macros: https://github.com/rust-lang/rust/issues/38356

Like the macro, the CLI accepts either a sample directly as a command-line argument, as a path to a remote source or a local file:

```
json_typegen -n Point 'http://example.com/point.json'
json_typegen -n Point samples/point.json
```

2.3 *The web interface*



JSON sample → Rust types

Input

Examples: 2D Point Hacker News story item Zalando Article Magic Card List Steam App News

Root struct name

GeneratedType

Input sample JSON

{}

Included code

[✓ Types only]

Full runnable example

Derives

Default, Debug, Clone, PartialEq, Serialize, Deserialize

Figure 2.2: Screenshot of the web interface

The third interface I have made to *json_typegen* is a web interface. The crate *json_typegen_web* provides a binary that both provides a web API for the code generation, as well as hosting the static HTML/JavaScript files providing a frontend to this API[3]. This interface is currently deployed and available at http://vestera.as/json_typegen/.

The procedural macro and the command line interface both require the user to download and compile a somewhat significant amount of code. In small projects where the

[3]In a sense, this is means that this is actually *two* interfaces, but as the web API is not intended for public consumption, I generally only count them as one.

sample can be assumed not to change, this initial cost may for a lot of users seem to outweigh the benefits of the code generation.

The web interface provides a convenient way to do quick one-time generation of types from samples, or for new users to test the project before potentially deciding to use one of the other interfaces.

Having a web interface also provides some nice user experience benefits. A web form has good potential for making visible the various configuration options for the code generation, and quickly testing them out. It can also provide documentation for each feature directly inline, e.g. by using descriptive labels and placeholders.

2.4 Shared code

As seen in Figure 2.1 the macro, web interface and CLI all depend on a common crate, *json_typegen_shared*. This crate contains the actual inference and code generation logic. The following general data flow is common to all three interfaces:

1. Retrieve an actual JSON sample, based on the input
2. Parse the JSON text into JSON values
3. Infer type shapes from the JSON values
4. Generate Rust code from the type shapes

In the CLI and the web interface, there is additionally a phase of formatting the generated code before the code is output to the user.

A simple heuristic is used to decide how the JSON sample should be retrieved: If the input starts with **`"http://"`** or **`"https://"`** it is interpreted as a remote sample. If it starts with **`"["`** or **`"{"`** it is interpreted as an inline sample. If neither of these apply, it will be attempted used as a local path.

The JSON text is parsed using *serde_json* into the catchall type **`serde_json::Value`**, which is then used as the input for the inference and code generation. We will look at these last two steps in detail in chapter 3.

As mentioned in section 1.1.4 procedural macros are **`TokenStream`** → **`TokenStream`** functions. However, a **`TokenStream`** can both be parsed from a string, and written into a string. In other words, we have access to both a **`String`** → **`TokenStream`** and a **`TokenStream`** → **`String`** function.

The different interfaces we want to provide place some requirements on what signatures we need to be able to use our code with. As mentioned, procedural macros need to have the signature **`TokenStream`** → **`TokenStream`**. A web API, on the other hand, must at least externally have an interface that boils down to **`String`** → **`String`**. That leaves us with two alternatives, (using → for a conversion and ⇒ for the transformation):

$$\begin{aligned} \texttt{String} \rightarrow \texttt{TokenStream} \Rightarrow \texttt{TokenStream} \rightarrow \texttt{String} &\qquad \text{web} \\ \texttt{TokenStream} \Rightarrow \texttt{TokenStream} &\qquad \text{macro} \end{aligned}$$

$$\begin{aligned} \texttt{String} \Rightarrow \texttt{String} &\qquad \text{web} \\ \texttt{TokenStream} \rightarrow \texttt{String} \Rightarrow \texttt{String} \rightarrow \texttt{TokenStream} &\qquad \text{macro} \end{aligned}$$

At the time of writing, **`TokenStream`** is a mostly opaque type, and we are thus forced to choose the second alternative, but this may change in the future. In the real project things are a bit more complicated, as more types are involved, but the central point that we can consolidate different interfaces to a shared core that does the actual transformation thanks to conversion methods still stands.

2.5 Synergy

Having these different interfaces provides us with some benefits that would be very difficult or impossible to achieve with only a single one, i.e. some emergent synergy.

With the three interfaces, there is a clear migration path for most use cases. Users can start with the web interface to test the code generation and see if the results fit their use case. Since it is just a normal website they can do this without having to install anything or add any dependencies, but just copy and paste the output given by the web interface.

If the code generation is suitable for their use case, and the code generation is frequently used, copying and pasting from a website can get tedious. The convenience of automatic code generation combined with the additional benefits with regards to verification will hopefully lead people to try the procedural macro.

As mentioned in section 2.1.1, the procedural macro does have some limitations, and as mentioned in section 1.3.3 even the original type providers have their disadvantages. With a combination of the interfaces in this project, some interesting solutions or workarounds for several of these issues are possible.

Perhaps most obviously, we now have a way, several in fact, to see the code the procedural macro expands to. The two additional interfaces in *json_typegen* both make it easy to see this generated code. On nightly versions of Rust it is in fact also possible to see the resulting code after macro expansion without any additional tools. However, since this code is fully expanded[4] it is primarily useful for debugging purposes. Usually, a user only wants a single “step” of expansion, e.g. only to see what *json_typegen* itself does.

The two additional interfaces also make lock-in almost a non-issue. If a user wants to stop using *json_typegen* entirely, they can just do a final one-time generation of the code. They can do this using either the web or command line interface, and replace the macro invocation with the generated code.

One reason why someone might not want to use the procedural macro at the moment is the current lack of autocomplete. However, switching from the procedural macro to manually generated code means abandoning the verification against the external resource that the procedural macro gives.

2.5.1 Conditional compilation

To make the generated source available to editors while still keeping the benefits of the procedural macro it is possible to use conditional compilation. Conditional

[4]Fully expanded, in the sense that every file in the crate has been inlined, every derive implementation has been made, and every **`println`** invocation has been expanded.

compilation is the concept of automatically removing parts of the source code before the main compilation, depending on some condition. E.g. removing Windows-specific code when compiling on Linux and vice versa.

In the source code:
```
#[cfg(not(feature = "online-samples"))]
#[derive(Default, Debug, Clone, Serialize, Deserialize)]
struct Point {
    x: i64,
    y: i64,
}

#[cfg(feature = "online-samples")]
json_typegen!("Point", "http://example.com/point.json");
```

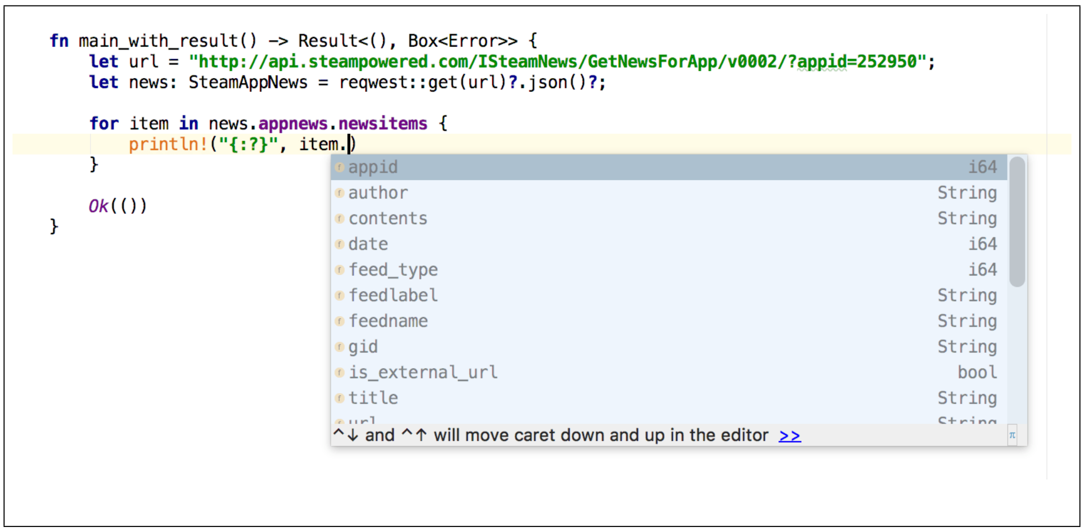
Normal build:
```
$ cargo build
```

Type checking against online samples:
```
$ cargo check --features "online-samples"
```
Listing 18: Conditional compilation

In Rust conditional compilation is possible with annotations like **`#[cfg(condition)]`**. Listing 18 shows an example where a feature flag **`"online-samples"`** is used to enable use of the procedural macro. In the default build the pre-generated type is used, but if the feature flag is enabled, the pre-generated type is ignored and the procedural macro used instead.

Since this method makes the types available to the editor, autocomplete based on the types is now more easily possible. Figure 2.3 shows autocomplete results that are based on conditionally compiled types in the IntelliJ IDE with the IntelliJ Rust plugin. The JSON sample and generated types for this example can be found in appendix C.3.

Using conditional compilation in this way also lets us get around the issue of requiring network access to build. If network access is the only issue we care about we don't even need to pre-generate the code, and can instead use two macro invocations – one with a local or inline sample and one with the remote one – as our two options.



```
fn main_with_result() -> Result<(), Box<Error>> {
    let url = "http://api.steampowered.com/ISteamNews/GetNewsForApp/v0002/?appid=252950";
    let news: SteamAppNews = reqwest::get(url)?.json()?;

    for item in news.appnews.newsitems {
        println!("{:?}", item.)
    }

    Ok(())
}
```

```
appid                 i64
author             String
contents           String
date                  i64
feed_type             i64
feedlabel          String
feedname           String
gid                String
is_external_url      bool
title              String
url                String
```

Figure 2.3: Autocomplete from generated types in IntelliJ Rust

Since the normal build in such a setup does not rely on an external resource we also get reproducible builds this way. Even if the external resource changes, it will still be possible to check out an old version of our source code and compile it. This can actually be very useful e.g. for tracking down bugs with tools like **`git bisect`**.

2.6 *Configurability*

There is one final problem that does not just apply to type provider like tools, but to code generation in general: Code generation almost always has to make some assumptions, and is as such hard pressed to cover every use case and every requirement a programmer might have.

Listing 19 shows a simple example of a case where the inferred type falls short of what the programmer would write by hand. There is no **`Date`** type in the Rust standard library, and as such, even with inspection of the **`String`**, it would not be natural for the code generation to infer a **`Date`** type by default. While the **`Date`** type specifically may conceivably be added to the standard library in the future, the same example could apply to all manner of domain-specific types.

Sample:
```
{
  "registered": "2016-08-19",
  "..."
}
```

Inferred type:
```
#[derive(...)]
struct Order {
    registered: String,
    ...
}
```

Desired type:
```
#[derive(...)]
struct Order {
    registered: Date,
    ...
}
```

Listing 19: A simple example of how generated code can fall short

In such cases where the generated code falls short the programmer has a few alternatives:

- They can simply use the generated code as is, and convert from `String` to `Date` as necessary. If the suboptimal type is used frequently in the code though, littering the code with snippets like `Date::from(order.registered)` becomes frustrating pretty fast.
- They can customize the generated code by hand, and reapply the customizations whenever the code has to be generated again. Since this requires access to the generated code this alternative means abandoning any procedural macro or type provider. As it also makes the customized code incompatible with the generated code, conditional compilation will not help here either.
- Finally, they can customize the generated code by hand, and completely abandon the code generation tool.

All of these alternatives have significant downsides, so if configuration of the code generation is possible, and does not come with significant downsides of its own it would in most cases be clearly preferable.

While it is clear that *json_typegen* should have at least *some* configuration options, it is not obvious to what extent. It is probably not a good idea to try to cover every edge case, as such a goal could quickly lead the code to become too complex to maintain and expand. We can, however, make an effort to cover the most common cases, and thus increase the usefulness of the project for most people.

In the next sections, I will discuss the configuration options that currently available in *json_typegen* and some proposals for future expansions of these options. For a quick overview of the implementation state of these, and other features of *json_typegen*, see section 4.1.

2.6.1 Visibility

Perhaps the simplest configuration option in *json_typegen* is type visibility. By default, the generated types have no visibility specification. In Rust this means that the types are private, i.e. only accessible from the module[5] they are defined in, and any modules it may contain.

Private visibility for types from *json_typegen* was chosen as the default because of the inherent volatility of generating types from external resources and the fact that any change to a public type is considered a breaking API change. Thus, if the types were public by default, any change in an external sample would cause a breaking API change in users crates. If this is what the users desire, they should be free to choose so, but it is not something that should happen by accident.

To set a more public visibility for the generated types, a visibility specifier may be given along with the name for the root type. E.g. **`json_typegen!("pub Point", ...)`** would create the type **`Point`** as a public type, and if the type name **`"pub(crate) Point"`** was given, it would be created as a type visible within the current crate.

By default in *json_typegen* struct fields "inherit" the visibility specifier (if any) of the containing struct. This should usually be the desired behavior, but if not it can be

[5]A file in Rust is itself a module

overridden with a visibility specifier set using the option **`field_visibility`**.

2.6.2 Derive list

As explained in section 1.1.3 common behavior is specified in traits, and many of these traits can be derived – automatically implemented – by annotating the type with a list of traits to derive.

By default *json_typegen* annotates each generated type with the following derive list:

```
#[derive(Default, Debug, Clone, PartialEq,
    Serialize, Deserialize)]
```

In other words will every type generated by *json_typegen* implement every trait in this list. The derive list can be overridden with the **`derives`** option. None of the traits are mandatory, but a generated type with an empty derive list would be close to useless. Since each trait has a significant effect we will look at each of the default derived traits separately. Additionally, we will look at some further traits that can be derived that a user might want to add for different use cases.

Any unused traits and the accompanying code is eliminated by compiler optimizations when doing release builds, and so do not contribute anything to binary size. The added code generation of unused traits added very slightly to compile time in my testing, but not enough so as to be noticeable.

Since the cost for additional derived traits is so small, all the traits that are implemented for our base types[6] are in our default list.

Serialize, Deserialize

`Serialize` and **`Deserialize`** are as explained in section 1.2.2 the traits that let *serde*-compatible libraries convert to and from their various formats. It may seem like you would always want these traits, as creating types that can be serialized and deserialized is the primary purpose of *json_typegen*. Indeed, one would almost always

[6]As a reminder, our base types are **`Vec<T>`**, **`Option<T>`** **`i64`**, **`f64`**, **`String`**, **`bool`** and **`serde_json::Value`**, hereafter referred to as **`Value`**.

want to have at least one of these traits, and sometimes both, depending on the use case. As such, in the default list both are included, but when customizing, often one can eliminate one of them.

Default, Debug, Clone

The *Default* trait provides a no-argument constructor, creating the default value for the type. Deriving this trait creates the no-argument constructor filling in the field values with their respective default values.

This trait is in the default list mostly because it is useful for quick testing and for examples, and as such is a good aid in explorative programming. For handwritten types, it usually makes more sense to manually set the default values.

Implementing the **`Debug`** trait makes the type printable via the **`?`** and **`#?`** debug format specifiers. E.g. an instance **`v`** of a type that implements the **`Debug`** trait is a valid argument in a statement like **`println!("{:?}", v);`**. Without implementing this trait, instances of a type can not be directly printed.

The **`Clone`** trait makes it possible to make a copy of an instance of the type by calling the **`clone`** method. For the generated types the derived implementation will result in a deep copy, which relatively speaking will be expensive. This should usually be avoided in high-performance code, but is sometimes unavoidable. In quick testing and explorative programming, however, it can often be quicker and easier to clone rather than try to satisfy Rusts borrow checker.

Eq, PartialEq

`Eq` and **`PartialEq`** provide methods for comparisons that are, respectively, full and partial equivalence relations. Implementing **`PartialEq`** makes it possible to use the **`==`** and **`!=`** operators. **`Eq`** provides no additional functionality, and is just a marker trait declaring the relation to be a full equivalence relation.

`PartialEq` is implemented for all of our base types, and is as such included in the default derive list. **`Eq`**, on the other hand, is not implemented for **`f64`**, floating point numbers, and **`Value`**, as they themselves can contain floating point numbers.

`f64` does not implement **`Eq`** because one of the requirements of a full equivalence relation is that it is reflexive, i.e. that every value is equal to itself. The possibility of **`NaN`** values makes **`f64`** not satisfy this property, as **`NaN != NaN`**.

While **`f64`** should not be **`Eq`**, JSON numbers are actually not allowed to be NaN or Infinity according to RFC 7159 [9, p. 7]. This means that if we could guarantee that only values from JSON data are stored in any fields that are inferred as **`f64`** we could safely make them **`Eq`**. Unfortunately, this is not something we can guarantee in Rust as immutability is a property of variable bindings, and not on fields.

`Value` actually enforces that its numbers don't contain **`Infinite`** or **`NaN`** and thus *could* be **`Eq`**.

Users that know they won't have any floats or inference issues that lead to **`Value`** values can add **`Eq`** to the derive list. If they have floats or **`Value`** values, but are sure that any floats will come from JSON, an **`Eq`** implementation can be manually added with a single line – **`impl Eq for TheType {}`** – for each generated type. If this is something many people would do, generating these lines is something that could somewhat easily be added to *json_typegen*.

Hash

The **`Hash`** trait provides the methods necessary for producing hashcodes for a type. Like with **`Eq`**, **`Hash`** is implemented for all of our base types except **`f64`** and **`Value`**. The problem here again is **`NaN`**, but it is not as clear cut as with **`Eq`**. There are several reasonable, but no perfect solution to how floats should be hashable, so the Rust standard library provides no **`Hash`** implementation for **`f64`**. Unfortunately, this leaves us with users only being able to add **`Hash`** manually to the derive list when they have no floats or **`Value`** values (or implementing **`Hash`** manually).

Some of the collections in the standard library depend on types implementing **`Hash`** and **`Eq`**. The fact that our default generated types lack implementations for these traits means that they can not be entered into any **`HashSet`** and can only be used as values, not keys, in a **`HashMap`**.

Ord, PartialOrd

`Ord` and **`PartialOrd`** implement methods for comparisons on types that form a total and partial order respectively. Like **`HashMap`** and **`HashSet`** depend on **`Hash`** and **`Eq`**, **`BTreeMap`**, **`BTreeSet`** and **`BinaryHeap`** depend on **`Ord`**.

An implementation for **`PartialOrd`** is missing for **`Value`** and **`Ord`** is missing for both **`f64`** and **`Value`**. Again, this means that by default the types can not be used in **`BTreeSet`** and **`BinaryHeap`**, and only as values in **`BTreeMap`**.

It may be tempting to add these derives based on the fields of the types we generated, but this could easily lead to some annoying and confusing situations for users. E.g. having working code, without any configuration, break because a field that is not even used is added to the sample. While the limitations on the default generated types is unfortunate, adding potential for such unpredictable breakage is in my opinion even worse.

New

In addition to the derives provided the standard library and *serde* (which we already assume as a peer dependency) there are other custom derive libraries that can be used to add functionality to the generated types from *json_typegen*. One simple such example is the *derive-new* crate, which derives a **`new`** constructor for the type with all fields as arguments. To add this functionality to all the generated types, all a user of *json_typegen* has to do is to add *derive-new* as a dependency, and add **`New`** to the derive list.

Frunk: Generic, LabelledGeneric

I won't go into detail on all the different functionality that can be added by custom derives, but one library that is of particular interest to us is *frunk* [3]. *frunk* is a library for doing functional generic programming in Rust. Among the things that it provides are two traits with accompanying derive code: **`Generic`** and **`LabelledGeneric`**.

Rust is a *nominally typed* programming language. This means that if we create types that have the same shape, but different names, they can not be used in place of each other, and converting between them must be done by mapping each field from one type to its corresponding field in the other. While we will look at other ways around this issue as well, **`Generic`** and **`LabelledGeneric`** provide a way to get a semblance of *structural* typing in Rust.

Types that implement the **`Generic`** trait from *frunk* can be converted between each other if they are structurally the same, i.e. if the types have the same number of fields, with the same types, in the same order. **`LabelledGeneric`** will additionally require the fields to have the same names to allow the conversion.

2.6.3 JSON Pointers

While some configuration options like type visibility and the derive list make sense to specify globally, for the whole JSON document at once, a lot of what would be beneficial to configure has to be done on a much more granular level.

The initial case in Listing 19, where we would want the type of a specific field to be **`Date`** rather than the inferred **`String`** is an example of where we would need such granularity.

In the example, we would want to target “the field **`registered`** of the root object”. In our original JSON example of section 1.2, we extracted “the field **`areaCode`** of the first element of the **`phoneNumbers`** field of the root object”. One of the ways we did this was with the **`pointer`** method from *serde_json* which takes as an argument something called a JSON Pointer.

The JSON Pointer specification [11] was originally developed along with the specification for JSON Patch [10] for the HTTP PATCH method, which needed a way to specify specific elements of a JSON document. A JSON Pointer is simply a string consisting of “reference tokens” each prefixed with a forward slash (to separate the tokens). A reference token is interpreted as either a field name or an array index, depending on whether an object or an array is encountered. So to specify “the field **`registered`** of the root object” we would write the JSON Pointer **`"/registered"`**. And the JSON Pointer we used in section 1.2 to extract “the field **`areaCode`** of the first element of the **`phoneNumbers`** field of the root object” was **`"/phoneNumbers/0/areaCode"`**.

Since JSON Pointers are representable as simple strings they are easy to use as arguments to both macros or in command line invocations. Since they are already used by *serde_json* it also makes sense to use JSON Pointers rather than introduce a completely new way to reference data originating from JSON. Many of the alternatives are also unnecessarily complex for our use case.

```
json_typegen!("Order", "samples/order.json", {
    "/registered": {
        use_type: "Date",
    },
});
```

Listing 20: Macro invocation with JSON Pointer configuration

Listing 20 shows how a macro invocation with an option specified with a JSON Pointer could look. Some examples of possible options that could be specified via JSON Pointers are:

- **`use_type`** Specifies a type for the code generation to use, with varying levels of specificity. E.g. as shown in Listing 20.
- **`same_as`** Specifies that the type of one part of a JSON document should be inferred (and if necessary, coerced) to the same type as another part. E.g. **`"/a/b": { same_as: "/c/d" }`**.
- **`type_name`** Gives a type name for the code generation to use when generating the types. Useful when the field name would not reflect the type. E.g. **`"/top_left": { type_name: "Point" }`**

In section 3.4.3 we will look at how the addition of these options, and JSON Pointers in general, affect the logic behind the code generation.

As JSON Pointers are a “stringly” typed way to specify paths, it may seem strange to suggest the use of them in a project that is so focused on type safety. There is, however, a significant distinction in using such code at compile time and at runtime. The criticism of “stringly” typed code in a strongly typed language is that it ends up reverting the programming to a state where errors are discovered at runtime instead of at compile time. If configuration for *json_typegen* specified through a JSON Pointer does not apply to any of the sample, this can be known when generating code, i.e. at compile time for the user, and a warning or an error can be shown at that point in time.

Due to the original use case the JSON Pointer specification was developed for, the specification only specifies how to target single elements in JSON documents. There is no wildcard for e.g. specifying every field in an object, or every element in an array. While wildcards adds some complexity to implementation, it is still something that can be quite useful for our use case. For arrays[7] wildcards may not be needed, as any option applied to one element, should, due to how the inference works, be applied to all of them.

JSON Pointer does however have syntax for "the (nonexistant) member after the last array element", **`"-"`**, which I propose as a simple solution for an explicit wildcard for array indexes and map keys.

2.6.4 Cost of configurability

While extensive configurability has significant benefits, there are also significant costs. Every configuration option a project adds increases its exposed API surface. The more exposed API surface we have, the more difficult maintenance and continued development becomes with making breaking changes.

With multiple interfaces, each added configuration option in the shared code also needs to be exposed and handled multiple times. E.g. for just the web interface, each option needs to be added to the HTML form, added to the communication with the backend and be handled by the backend. Then similar work has to be done for the command line interface and the procedural macro.

A lot of this code is, however, the same for each option of the same type. It is as such possible that code generation could be used to minimize the additional work associated with the additional interfaces. Internally, *json_typegen* uses a struct **`Options`** which groups all configuration options. It is not implausible that it would be practical to write code generation, that took such a struct as input and created interface code. This would in itself be a useful project, and there is in fact a relatively new project, *structopt* [7], which attempts to do this for command line parsing.

[7]And with extensions, other collections.

2.7 Improving the synergy

While the synergy between the different interfaces is already quite good there are ways in which it could be improved.

Since many of the use cases involve moving between the interfaces, making the transition between them as smooth as possible is an important part of the total user experience of the project.

While the ”native” user experiences of the interfaces are quite different, with different strengths and weaknesses, the central functionality they provide, including configurability, should be the same. However, if a user has taken extensive advantage of this configurability, having to replicate the same configuration settings from one interface in another can erase the convenience the other interface would provide.

For example is one of the use cases for the web interface to easily be able to see the generated code when using the procedural macro. However, an option in the macro is expressed as code, while the natural interface for the website is a form. For this specific use case, it would be much nicer to just be able to copy the macro invocation as text and get the output without having to use the form. In other words, it would be convenient if the web interface was able to read the macro syntax as an additional input format. Use cases for the CLI would benefit much the same way if a macro invocation could be used directly to generate code. What I propose is essentially to use the macro syntax as an ad hoc configuration interchange format.

If the web interface additionally can output in the macro syntax we get easy transitions that cover what I think are the most likely user stories:

- Just discovered the project, and having tested in the web interface, want to start using the macro: web $\rightarrow$ macro.
- Using the macro, but want to see the generated code for quick debugging: macro $\rightarrow$ web.
- Using the macro, but want to use CLI for autocomplete or other reasons for conditional compilation: macro $\rightarrow$ CLI.

The way I've described how the procedural macros work in section 2.4 it may seem like we could get at least parsing of the macro “for free”. Unfortunately, this is currently

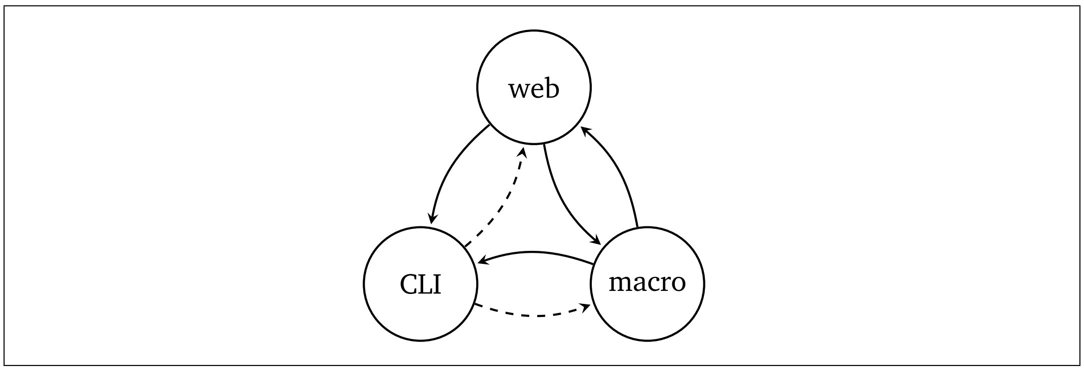


Figure 2.4: Transitions enabled by the CLI and the web interface being able to read, and web interface being able to write, macro syntax. The dashed lines show transitions that require the use of the macro syntax as the configuration format, and not just as the interchange format.

not the case. Since the current macro is not a procedural macro, we do not ever use the macro code itself as input, but rather what it expands to. However, any parsing code written for configuration interchange may very well be directly usable once normal procedural macros are available on the stable compiler and it is time to transition the *json_typegen* macro.[8]

Due to how beneficial conditional compilation currently is, it could also be quite useful if the code required for using conditional compilation could also be generated automatically. If the code generation was able to output macro syntax, this should be quite easily implementable.

2.8 Editor plugins

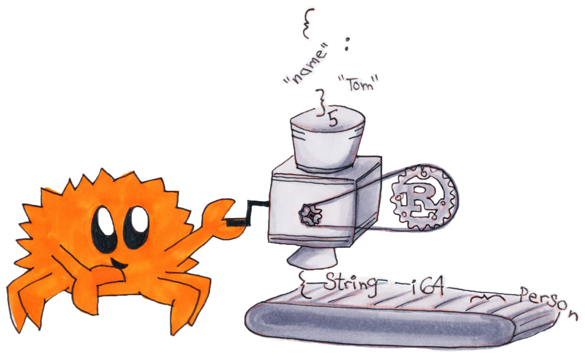
One thought I have had while working on this project is to what extent it matters that it is actually the compiler, and not something else, that actually inserts the provided types into the code of the program. To explore this idea I have created a prototype plugin[9] for the Atom editor, which expands a macro invocation to the resulting code

[8]I am also considering a refactor of the macro part of the project before then, which might enable the use of the macro input more directly by itself.

[9]https://github.com/evestera/atom-json-typegen

inside the editor. The plugin just comments out the macro invocation, sends it to the web interface backend, and inserts the result into the buffer. The generated code can be hidden with code folding, so that the macro invocation is the only thing visible, if so desired. Since this process is entirely outside the compiler, no dependencies are added to the project.

Since the generated code is available to the editor, just like it would be if the user had used the command line interface, the editor can use the code for code completion etc. While the plugin is very rudimentary, it illustrates how simple of a concept tooling support for procedural macros could be. If the code did not have to be inserted into the buffer for the editor to be aware of it, such a plugin could just provide what was necessary for full editor support for the procedural macro. Full tooling support for macros, procedural or otherwise, just depends on being able to expand macros to the resulting code, and making this expanded code available to the editor.



Chapter 3

Code generation

Once a JSON sample has been retrieved and parsed using *serde_json* into the catchall type **`serde_json::Value`**, *json_typegen* is ready to begin inference and code generation. In this chapter we will look at the details of these three stages:

- First, a generalized "shape" is inferred from the JSON values.
- Then this inference is enhanced in intermediary passes.
- Finally, Rust code is generated based on the inferred shapes.

To begin with I will present a basic version of the inference algorithm and code generation, before looking at how this system can be extended in various ways.

3.1 *Shape inference*

The shape inference is based on the algorithms used in *F# Data* as presented in the paper *Types from Data: Making Structured Data First-class Citizens in F#* [17]. As done in this paper I will use the term shape for the intermediate representation of the inference data, to avoid confusion with actual programming language types.

The goal of the shape inference is to infer generalized shapes from the samples which can then later be used for generating code. These shapes are intended to be general enough to not be tied to inference from JSON – *F# Data* also does inference for

CSV and XML – and not tied to the generation of any particular type of programming language. Rust code for the basic version of the shape inference can be seen in appendix A.

3.1.1 Values and shapes

The inference thus works with two types. The JSON sample, in our case parsed into a **`serde_json::Value`** – as was shown in Listing 9 – and the shape type. An abstract version of **`serde_json::Value`**, with value variables ω, and our shape type, with shape variables σ, can be written as follows:

$$\begin{aligned}\omega ::= &\ \texttt{Null} \mid \texttt{Bool(}b\texttt{)} \mid \texttt{Number(}n\texttt{)} \mid \texttt{String(}s\texttt{)}\\ &\mid \texttt{Array(}[\omega_1, \cdots, \omega_n]\texttt{)} \mid \texttt{Object(}\{k_1 : \omega_1, \cdots, k_n : \omega_n\}\texttt{)}\end{aligned}$$

$$\begin{aligned}\sigma ::= &\ \texttt{any} \mid \bot \mid \texttt{bool} \mid \texttt{string} \mid \texttt{int} \mid \texttt{float}\\ &\mid \texttt{optional(}\sigma\texttt{)} \mid [\![\sigma]\!] \mid \{\!\{k_1 : \sigma_1, \cdots, k_n : \sigma_n\}\!\}\end{aligned}$$

To be clear about how notation is disambiguated: $[a, b, c]$ is an actual sequence of items, while $[\![a]\!]$ is the *shape* representing a sequence/list.[1] Likewise $\{k_1 : v_1, \cdots, k_n : v_n\}$ is a map-like collection, while $\{\!\{k_1 : \sigma_1, \cdots, k_n : \sigma_n\}\!\}$ is the *shape* representing a record/object/struct.

For an intuitive understanding of the more abstract shapes it is helpful to think of what kind of knowledge each shape represents. $\bot$ (read as "bottom") represents the complete lack of knowledge about what type should be inferred. **`optional(`**σ**`)`** represents the knowledge that a type may sometimes be **`null`** or a field may not always be there, i.e. that it is nullable or optional. So **`optional(`**$\bot$**`)`** tells us that a type is optional, but that we know nothing else of about its type. The **`any`** shape, on the other hand, represents conflicting information. If we end up with the **`any`** shape it means that the information we have received is not representable by any of the other alternatives. E.g. a type that can be both a **`string`** and an **`int`**.

[1] In *Types from Data* the notation $[\![a]\!]$ is used for a different purpose in a section of the paper we will not look at here.

Unlike in *Types from Data*, to preserve information and let any choice be made in the code generation step, lists[2] are not considered nullable. In other words, the inference can infer the shape **`optional(`**$[\![\sigma]\!]$**`)`**, but it is up to the code generation to decide how to handle the shape. In the default code generation, **`Option<Vec<T>>`** will not be generated, but this can be enabled by configuration.

With this change, a separate **`null`** shape, which the original algorithm has, is no longer necessary. Instead, $\bot$ is not considered nullable either, and **`optional(`**$\bot$**`)`** serves the purpose of **`null`**.

3.1.2 From values to shapes

$$
\begin{aligned}
\text{vts}(\texttt{Null}) &= \texttt{optional(}\bot\texttt{)} \\
\text{vts}(\texttt{Bool(}b\texttt{)}) &= \texttt{bool} \\
\text{vts}(\texttt{Number(}n\texttt{)}) &= \begin{cases} \texttt{int} & \text{if } n \in \mathbb{Z} \\ \texttt{float} & \text{otherwise} \end{cases} \\
\text{vts}(\texttt{String(}s\texttt{)}) &= \texttt{string} \\
\text{vts}(\texttt{Array(}[\omega_1, \cdots, \omega_n]\texttt{)}) &= [\![\text{fold}(\text{csh}, \bot, [\text{vts}(\omega_1), \cdots, \text{vts}(\omega_n)])]\!] \\
\text{vts}(\texttt{Object(}\{k_1 : \omega_1, \cdots, k_n : \omega_n\}\texttt{)}) &= \{\!\{k_1 : \text{vts}(\omega_1), \cdots, k_n : \text{vts}(\omega_n)\}\!\}
\end{aligned}
$$

Figure 3.1: vts(ω), the function converting JSON values to shapes

The main inference function vts(ω), shown in Figure 3.1 takes as its input a value and produces a shape. Values containing other values are converted by applying vts recursively.

fold(*function*, *base*, *sequence*) is the fold operator [15], common in functional programming (and available on iterators in Rust). In this case, it reduces a sequence of shapes to a single shape by finding a common shape with the csh(σ_1, σ_2) function,

[2]… and with extensions, other collections.

which we will look at in the next section. The initial shape is $\bot$, which means that an empty **Array** will be represented as the shape $[\![\bot]\!]$.

3.1.3 Finding common shapes

$$\begin{aligned}
\text{csh}(\sigma, \sigma) &= \sigma && (eq)\\
\text{csh}(\sigma, \bot) &= \sigma && (bottom)\\
\text{csh}(\texttt{int}, \texttt{float}) &= \texttt{float} && (num)\\
\text{csh}(\sigma_1, \texttt{optional(}\sigma_2\texttt{)}) &= \text{opt}(\text{csh}(\sigma_1, \sigma_2)) && (opt)\\
\text{csh}([\![\sigma_1]\!], [\![\sigma_2]\!]) &= [\![\text{csh}(\sigma_1, \sigma_2)]\!] && (arr)\\
\text{csh}(\sigma_1 = \{\!\!\{\cdots\}\!\!\}, \sigma_2 = \{\!\!\{\cdots\}\!\!\}) &= \text{cfs}(\sigma_1, \sigma_2) && (obj)\\
\text{csh}(\sigma_1, \sigma_2) &= \texttt{any} && (any)
\end{aligned}$$

Figure 3.2: csh(σ_1, σ_2), the common shape function

Sometimes multiple elements must be represented by the same shape, e.g. a single shape representing all the elements of an array. Figure 3.2 shows the function csh(σ_1, σ_2) which for two shapes finds a common shape which can represent them both. Order of arguments does not matter, so for every rule csh(a, b) the rule csh(b, a) is the same.

csh(σ_1, σ_2) defines a partial order for the set of shapes, where if csh$(\sigma_1, \sigma_2) \vdash \sigma_3$, $\sigma_1 \sqsubseteq \sigma_3$ and $\sigma_2 \sqsubseteq \sigma_3$. The relation $\sqsubseteq$ can be understood as the “can be represented by”-relation. E.g. **float** $\not\sqsubseteq$ **int**, but **int** $\sqsubseteq$ **float** (and also **int** $\sqsubseteq$ **int**).

Figure 3.3 shows an incomplete Hasse diagram for the partially ordered set given by $\sqsubseteq$ on the set of shapes. The internal ordering of list and record shapes, $[\![\sigma]\!]$ and $\{\!\!\{\cdots\}\!\!\}$, is not shown.

When we are working with optional shapes, we want optionality to be a boolean property, i.e. either something is optional, or it is not. For any shape σ that already encodes the possibility of values that are missing or **null**, opt(σ), as shown in Figure 3.4, is simply a no-op.

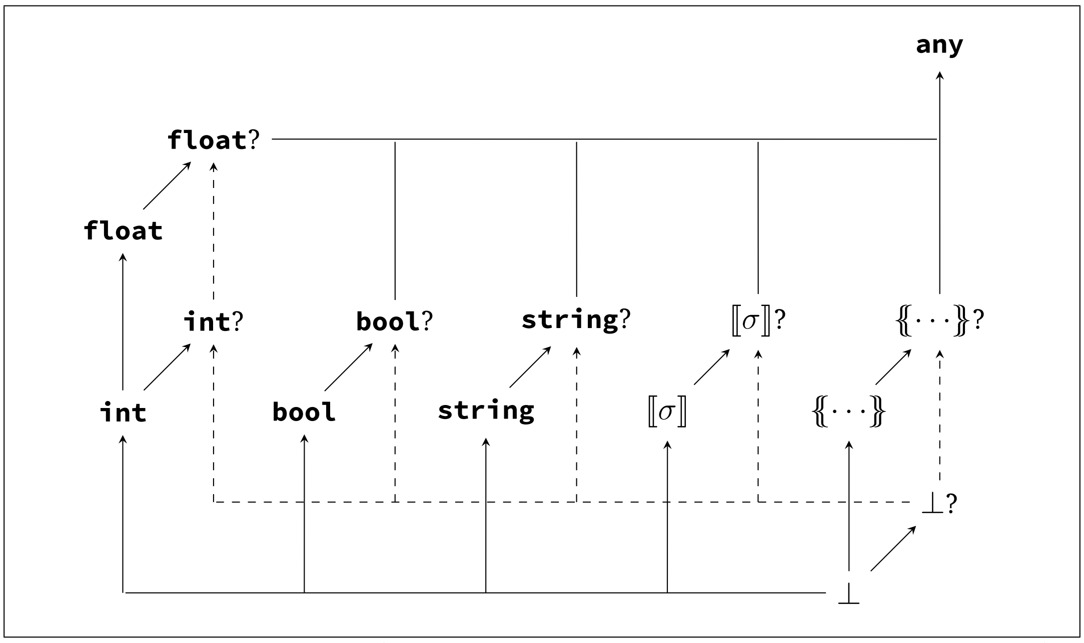


Figure 3.3: Hasse diagram for the partial order. For compactness σ? is used for **`optional(`**σ**`)`**. Dashed lines have the same meaning as solid lines, and are only used for visual clarity.

$$
\begin{aligned}
\mathrm{opt}(\texttt{\textbf{any}}) &= \texttt{\textbf{any}} \\
\mathrm{opt}(\texttt{\textbf{optional(}}\sigma\texttt{\textbf{)}}) &= \texttt{\textbf{optional(}}\sigma\texttt{\textbf{)}} \\
\mathrm{opt}(\sigma) &= \texttt{\textbf{optional(}}\sigma\texttt{\textbf{)}}
\end{aligned}
$$

Figure 3.4: opt(σ), the function ensuring optionality/nullability of shapes

Figure 3.5 shows how the common shape of two record shapes is found by finding the common shapes of its fields. For keys that are not present in both records, the shape that is present is optional/nullable. Note that in *Types from Data*, records have row variables [20] so that all records can be considered to have the same keys. To minimize the number of new concepts needed to understand the basic algorithm I have chosen this slightly less elegant notation in Figure 3.5 instead. Informally, every key present in either of the input record shapes will be present in the output record

$$\text{cfs}(\sigma_1 = \{\!\{k_1 : v_1, \cdots, k_n : v_n\}\!\}, \sigma_2 = \{\!\{k_1 : v'_1, \cdots, k_n : v'_n\}\!\}) =$$
$$\left\{\!\!\left\{ \forall\, k_n \in \sigma_1 \cup \sigma_2.\, k_n : \begin{cases} \text{csh}(v_n, v'_n) & \text{if } k_n \in \sigma_1 \cap \sigma_2 \\ \text{opt}(v_n) & \text{if } k_n \notin \sigma_2 \\ \text{opt}(v'_n) & \text{if } k_n \notin \sigma_1 \end{cases} \right\}\!\!\right\}$$

Figure 3.5: cfs(σ_1, σ_2), the function for finding the common shape of two records

shape. For every field, if the field is found in both record shapes, the shape of the field in the output is the common shape of the shapes for the corresponding fields in the input. If the field is found in only one of the input shapes, the corresponding field in the output shape must be optional.

3.2 Intermediary passes

In the basic version of the algorithm, there are no intermediary passes. These passes are mainly a result of extensions of the algorithm. The extensions come about for two main reasons: Improving the code to more closely resemble hand-written code, and adding configurability of the inference and code generation.

We will look at these extensions and their consequences for the algorithm in section 3.4.

3.3 Generating Rust types

Once a shape tree has been inferred from the sample and intermediate passes have been run to improve the inferred shapes, the shape tree is used to generate Rust code. The code generation is a simple recursive procedure.

The main function takes as its input some path information, used for naming any generated types, and a shape. The function pattern matches on the shape and produces as its output a tuple: The name of the type representing the shape, and potentially some code generated to declare the returned type.

Appendix B contains a (very) naive implementation of this algorithm. Among other things, this implementation is not configurable and assumes all fields names can be used as Rust field names and Rust type names if the first letter is uppercased. Except for such simplifications, the implementation is similar to the actual code generation in *json_typegen*.

In the actual implementation, code is generated via quasiquotation, using a macro **`quote!`** from the *quote* [4] library. This is quite similar to the use of format strings in the naive implementation, except that the code to generate code is simpler, and that it generates tokens, rather than strings. For our use case, this is actually not entirely positive. When working with just tokens, there is no need to format the code nicely, so when the tokens from *quote* is output as a string, it becomes just one long line. For this reason, a separate formatting step is done, if the code is to be shown to a user. In the future it may be better to choose a simple, custom solution for the equivalent role of *quote*, as *json_typegen* generates a rather small subset of Rust.

3.3.1 Type and field names

Up to the point of code generation, JSON field names have been preserved as they were in the original JSON. These field names are used to create both field names for the generated types, as well as type names for any nested types. However, as mentioned in section 1.2, the JSON field names can be completely arbitrary strings, while the field names and type names we generated must conform to Rust rules and conventions for identifiers.

The generated names are restricted to consist of "words" of alphanumeric ASCII characters, and the complete name must start with a letter. Type names are "PascalCased", i.e. concatenated, with the first letter of each word upper-cased, and field names are "snake_cased", i.e. all lower case, with words separated by an underscore. For some fields, nothing of the original name remains after restrictions are applied, and we have to resort to fallback names like **`GeneratedType`**.

Additionally the generated types cannot collide with Rust keywords or each other. Collisions with keywords are solved by appending a word describing its use, like

Sample:
```
{
  "one two": { " ": 5, "?": 2 }
}
```

Generated types:
```
struct Root {
    #[serde(rename = "one two")]
    one_two: OneTwo,
}

struct OneTwo {
    #[serde(rename = " ")]
    field: i64,
    #[serde(rename = "?")]
    field2: i64,
}
```
Listing 21: Field and type renaming

`field`, so the field `"type"` becomes `type_field`. Collisions between generated names are solved by adding an incrementing counter to the name.

For field names, if the Rust field name does not match the JSON field name an annotation to relay this information to *Serde* is added to the field. Listing 21 shows an example with several of these issues at once.

3.3.2 Generation of a runnable program

In addition to just generating the types themselves, the code generation can also generate a complete, runnable Rust program. In addition to the generated types, the runnable program contains the necessary imports and a main function showing how to use *serde_json* to deserialize and serialize into the generated types.

The generation of a full program is mainly for documentation and demonstration purposes in the web interface. Using a button in the web interface the generated code can be directly compiled and run in the online Rust playground.

3.3.3 Use of derivable traits

As outlined in section 2.6.2 a lot of the functionality of *json_typegen* is founded on the fact that many traits can be derived – i.e. that they can be implemented by just adding their name to a list – and that this works even for complex generated types.

The fact that this is possible relies on two important preconditions:

1. That our generated types are composed of either our base types, or other generated types. I.e. that every leaf in our generated type tree is one of our base types.
2. That every trait in our derive list is derivable and implemented for each of our base types.

Both of these preconditions can be broken by the user if the right configuration option is provided. As explained in section 2.6.2, the derive list can be overridden. This can obviously break our second precondition, either if a trait is not implemented for one of our base types, or for that matter if the trait is not derivable at all. There are also ways configuration can introduce new, essentially opaque, types into the code generation. These new types essentially become new base types, and as such can easily break our first precondition.

If a user breaks our preconditions in this way and this leads to a compiler error. The messages when a derive fails are clear and should make it quite obvious to the user what the problem is. With this in mind, I think letting the user break these preconditions is an acceptable trade-off for the benefit these configurations provide.

While letting the user break these preconditions is acceptable, care has to be taken to preserve these preconditions when extending the basic system. To make the source of any errors obvious, no derive errors should be possible that does not directly mention a trait or type explicitly specified by the user. E.g. if the system is extended with sets, enabling this extension should not be possible without explicitly choosing a target type if doing so breaks the preconditions.

3.3.4 Strictness of deserialization

While the types we generate should match the samples they are generated from, and any JSON text that matches their structure, there is always the possibility that the actual JSON we encounter at runtime does not perfectly match what we inferred.

Mismatched field types

If a field has entirely the wrong type, e.g. an object where we expect a number, deserialization has to fail with an error, as discovering such type issues as early as possible is a big part of the reason to deserialize to strong types. There are, however, some types we *could* conceivably coerce between. E.g. the string value **`"3.5"`** could be parsed and coerced into a **`f64`** field. While I have some ideas for extensions for specific instances like strings to number, I have no plans for supporting coercing data at runtime in an ad-hoc fashion.

When it comes to how we handle fields that are either missing, or that were never observed during inference, there are some choices to be made.

Missing fields

In the inference we detect potentially missing fields and encode this information into the shapes. By default, the code generation maps **optional(σ)** to **`Option<T>`**, and if any field that was not inferred to be optional is missing, deserialization fails. However, another way to handle missing fields that *Serde* supports is to use the default value provided by an implementation of the **`Default`** trait. I.e. if a string is missing, use the empty string, if an integer is missing use 0, and so on. By enabling this functionality for every field a user could make sure deserialization never fails due to a missing field.

Additional fields

By default, if *Serde* tries to deserialize and the data has an additional field that the target type does not have, the additional field is just ignored. This matches well with our intent with type providers, in that we only want to check the assumptions we actually make. However, it is also possible to make such additional fields cause deserialization to fail, which is behavior users of *json_typegen* can enable through configuration.

3.4 Extensions

As have been mentioned earlier, there are several ways to extend this basic setup to better align with what handwritten code would look like. We will now look at a few such extensions. For the sake of simplicity, we will mostly not go into the details of how these extensions interact or the complete extended algorithms, focusing instead on each extension by itself.

3.4.1 Tagged `any` shapes

In addition to the basic inference algorithm described in section 3.1, *Types From Data* [17] also describes an extension it calls “labelled top shapes” for providing a better fallback than the general **`any`** shape.

In the basic algorithm, if the least upper bound given by $\mathrm{csh}(\sigma_1, \sigma_2)$ is **`any`**, we throw away the information we have about why we had to use **`any`**, i.e. σ_1 and σ_2. The basic concept of this extension is to instead incorporate this information into a top shape **`anyof(`**$\sigma_1, \cdots, \sigma_n$**`)`**. This can then be used in the code generation to e.g. create enumerated types with custom types inside.

This extension is absolutely vital for making the algorithm work well with XML and HTML. In these formats heterogeneous collections with easily disambiguated types are abundant. For *json_typegen*, the extension would still be useful, but not essential, as **`serde_json::Value`** provides a decent fallback type as long as *json_typegen* is specific to JSON.

3.4.2 *Detecting distinct object shapes*

One of the reasons why tagged **any** shapes are not as useful for JSON as for e.g. XML and HTML is, as mentioned in the previous section, that JSON provides no directly obvious way of disambiguating objects representing different types. There are, however, ways we could infer that two different JSON objects should be interpreted as having different types.

Intersection of keys

One indicator that two objects may represent different types is if they have no fields in common, i.e. if the intersection of the sets of keys of two objects is empty. Listing 22 shows how two such objects, with no fields in common could be represented as an enumerated type rather than a struct, if tagged **any** shapes were also implemented.

JSON sample input:

```
[
  { "a": 1 },
  { "b": 1 }
]
```

Two alternative representations:

```
struct S {
    a: Option<i32>,
    b: Option<i32>,
}

enum E {
    Variant1 { a: i32 },
    Variant2 { b: i32 },
}
```

Listing 22: Two ways of representing objects with no overlap

This does, however, have the potential to make the inference less intuitive, as different order of objects in an array can lead to completely different inferred types.

Take for instance the JSON array **`[{ "a": 1 }, { "b": 1 }, { "a": 1, "b": 1 }`**. If the array is folded from left to right, i.e. the single-field objects are combined first, there is no intersection of keys, and one might infer a shape corresponding to an enumerated type. It is not obvious how one should proceed after this either, as the two-field object matches both of the existing alternatives equally well. If the array is instead folded from right to left, or the array is re-ordered, a shape corresponding to the struct in Listing 22 would be inferred.

Discriminators

Another way object types can be disambiguated is if one of the fields acts as a discriminator field or type tag. This is a relatively common pattern in real-world JSON, but neither the existence of a discriminator field, nor the placement nor name of such a field is something one can rely on for inference. The use of a discriminator is thus something that would have to rely on configuration through JSON Pointers.

When discriminator fields *are* available they provide a reliable way to figure out which objects should correspond to which variants, even if instances of a variant do not have the exact same fields. Additionally, they provide an intuitive way to name each variant, providing good usability for the eventually generated types.

3.4.3 JSON pointer hints and configuration

As outlined in section 2.6.3 it would be beneficial if we could use JSON pointers to specify configuration options and hints (hereafter referred to as just options) to the inference and code generation.

Actually applying these options require some modifications to the basic version of the algorithm. However, the options are not all applied the same way or at the same time. Some have to be applied during inference, some are best applied as intermediate transformations between inference and code generation, and some have to be applied during the code generation. This means that while we start with a simple list of options, we have to do some pre-processing of the options before running the algorithm with them.

JSON sample with discriminators:

```
[
  {
    "type": "foo",
    "a": 1
  },
  {
    "type": "bar",
    "b": 1
  }
]
```

Enumerated type using discriminator values:

```
enum E {
    Foo { a: i32 },
    Bar { b: i32 },
}
```

Listing 23: Enumerated type created from sample with discriminator field/type tags

The pointer options must then be threaded through our inference and code generation functions, and applied as they match.

A pointer option is a 3-tuple of a pointer, an option name, and an option value. For applying the option we transform the pointer into a list of tokens. E.g. the pointer **`"/field1/field2"`** is converted to the list of tokens [**`"field1"`**, **`"field2"`**]. For each recursive call of e.g. the vts function, the first token of each pointer is removed and tested on e.g. the field name to see if the hint should be carried on to the inner call. If a list of keys is empty, then the hint applies to the current value.

For a perspective of how handling of such pointer options can quickly become complex we will look briefly at the three examples in section 2.6.3: **`use_type`**, **`same_as`** and **`type_name`**.

On the surface, **`use_type`** seems very simple. The user specifies a type, and the code generation outputs the type for a field. Looking closer, there are actually several possibilities which all have to be handled:

- The specified type may be a recognized shape, like **`number`**, **`map`** or **`tuple`**. In this case the information is relevant to the inference, and may lead to some specific behavior therein.

- The specified type can be a concrete type, which also corresponds to a shape (or for that matter, a shape with inner shapes). E.g. **`VecDeque`**, which corresponds to the $[\![\sigma]\!]$ shape. Now, the shape information must be used in the inference, but the specific type information must be used in code generation.

- Finally, the specified type may be something the inference knows nothing about, an opaque type. In this case it may be best to infer as normal, but then ignore the inferred shape in code generation, but log or insert it into a generated comment for debugging purposes.

For **`same_as`**, handling the simple case may actually be quite straightforward. As an intermediate phase, extract two shapes, find a common shape using $\mathrm{csh}(\sigma_1, \sigma_2)$, annotate the result in some way so that the code generation knows to only create one type, and write the result, or some kind of reference back to the original locations. But what happens if there are two or more **`same_as`** declarations? Now, any potential overlap must be correctly handled. And if wildcards are supported even a single **`same_as`** is no longer so simple.

`type_name` seems perhaps simplest of all. Just a type name that should be used in code generation. There is, however, some hidden complexity here as well. Consider the possibility of two pointer options assigning the same name. Perhaps it should be considered an error, but it could also be interpreted as a declaration that the two shapes pointed to should be the same, which brings us back to the complexity of **`same_as`**.

3.4.4 More number types

By default, our code generation only uses **`i64`** for integers and **`f64`** for floating point numbers. While these are good defaults, there are several other number types that could be used. In particular, the Rust standard library provides types for unsigned numbers, e.g. **`u64`**, and smaller numbers, e.g. **`i8`** and **`f32`**. While rare in practice, JSON numbers can also be arbitrarily large and precise, so it is also worth considering support for big integers and arbitrary precision floats.

Adding support for just unsigned numbers could be done by adding another shape **unsigned** to our set of shapes, changing the rule for $\text{vts}(\texttt{Number(}n\texttt{)})$ to use **unsigned** if possible and adding some rules to $\text{csh}(\sigma_1, \sigma_2)$. However, extending the inference in this fashion scales very poorly. For n number shapes, we would need $n!$ rules in $\text{csh}(\sigma_1, \sigma_2)$.

What I propose instead is to replace **int** and **float** with a single number shape, $\texttt{number(}m, n, f\texttt{)}$, which tracks three properties:

- The minimum observed value, m
- The maximum observed value, n
- Whether any of the observed values was floating point, f

The rules for **float** and **int** in vts and csh and can then be replaced with the following rules:

$$\text{vts}(\texttt{Number(}n\texttt{)}) = \texttt{number(}n, n, n \in \mathbb{Z}\texttt{)}$$

$$\text{csh}(\texttt{number(}m_1, n_1, f_1\texttt{)}, \texttt{number(}m_2, n_2, f_2\texttt{)}) = \\ \texttt{number(}\ \text{min}(m_1, m_2), \text{max}(n_1, n_2), f_1 \vee f_2\texttt{)}$$

If it is desirable to infer arbitrary precision floats one should also track the maximum observed precision. And while in theory, the shape and rules as outlined above could work for arbitrarily large integers this is not something that can be easily implemented in *json_typegen*. The reason for this is that the **Value** type from *serde_json* currently only supports numbers that can be represented with the base rust types. In other words, to add support for detecting such numbers, we would have to replace *serde_json* with our own parser.

While arbitrary precision is out of the question to begin with, the extension should be sufficient for all the number types in the Rust standard library. By default we should still only generate **i64** and **f64**, though. The primary benefit of unsigned numbers is to disallow negative values (the slightly extended positive range is usually a secondary concern), but while it is common to write floating point numbers with the decimal point even when the value is an integer, there is no such hint for numbers which may be negative. As such the risk/reward is just not good enough to justify it as a default. In the same way, using **i8** instead of **i64** only restricts the possible values, and e.g. documentation samples can often have artificially small numbers, so the risk of assuming small ranges from limited samples should not be the default.

While unsigned and smaller numbers are not enabled by default, like **optional(**$[\![\sigma]\!]$**)**, a flexible number shape just preserves additional information that can be used if enabled through configuration.

3.4.5 Maps

One common issue with JSON is that its simplicity means that some common data structures are missing. This drives people to use the same data structures with different intentions as different ad-hoc data structures. Perhaps the most common such pattern is the use of JSON objects as maps.

While JSON has no concept of a map, maps with strings as keys can be encoded in JSON as objects, and there is no loss of fidelity inherent in this encoding. The only issue for us is that there is no good way to infer the difference between an object used to encode a structure that will persist across e.g. API calls, and an object used to encode a mapping from arbitrary keys to values.

While the intention that an object is used as a map can not be directly inferred from just a sample, with inference hints from the user, code using maps can still be inferred and generated.

$$\text{vts}(\texttt{Object(}\{k_1 : \omega_1, \cdots, k_n : \omega_n\}\texttt{)}, [\cdots, []\ \texttt{use\_type map}, \cdots]) =$$
$$\texttt{map(}\ \text{fold}(\text{csh}, \perp, [\text{vts}(\omega_1), \cdots, \text{vts}(\omega_n)])\texttt{)}$$
$$\text{vts}(\omega, [\cdots, []\ \texttt{use\_type map}, \cdots]) = \texttt{error!}$$

Figure 3.6: Extending the hinted vts to support maps

To be able to infer maps we would first need to add **map(**σ**)** as an alternative to our set of possible shapes. In the notation I have not included the key type, as JSON only

$$\text{csh}(\texttt{map(}\sigma_1\texttt{)}, \texttt{map(}\sigma_2\texttt{)}) = \texttt{map(}\ \text{csh}(\sigma_1, \sigma_2)\texttt{)} \qquad (map)$$

Figure 3.7: Extending csh to support maps

supports string keys for object fields, and as such to assume strings as map keys in the generated code should suffice. Figure 3.6 shows how the function vts already extended with hints could be extended to infer maps. The shown rules should take priority over the existing rule matching on **`Object`**.

csh can be extended by adding a simple rule shown in Figure 3.7 before the existing rule (any). One may argue that map values are already nullable, since **`map.get()`** or any equivalent will return some nullable type, and that we should thus take care to not infer a shape for the map values which could be lowered to a non-nullable shape (or rather, to lower such types when we infer them).

However, the only nullable shape we currently have that can be lowered to something else is **`optional(`**a**`)`**, which can be lowered to its wrapped shape a. The only way for the algorithm to infer map values that are **`optional(`**a**`)`** is if the map sample contains explicit **`Null`** values. For a map to contain such values would be quite rare, and if it were to happen, those null values are likely to carry meaning. With these things in mind, I consider the best option to be to *not* lower the map value shapes.

In most programming languages there is also the consideration of which map type to use. The Rust standard library provides two map types, **`HashMap`** and **`BTreeMap`**. In addition to these alternatives there are various map types in published in crates in the Rust ecosystem. As an example, *json_typegen* itself uses a **`LinkedHashMap`** internally. From the perspective of a user, giving a hint **`use_type`** t should work with any of the types mentioned above for t, as well as just **`map`**, letting *json_typegen* choose the map type.

For the default map type, whatever we choose ends up as essentially a new base type. Both **`HashMap`** and **`BTreeMap`** are good candidates, but as the default I have chosen **`HashMap`**, as it is the recommendation for general maps [13, std::collections]. As mentioned, we assume strings as keys, so the full new base type is **`HashMap<String, T`**. With **`String`** fixed as the key type, **`HashMap`** implements all the traits in our derive

list and can thus be safely added to our set of base types.

3.4.6 *Tuple types*

Another common pattern of data structure adaption in real world JSON is the use of arrays as tuples. E.g. serializing the tuple **`("a", 1)`** as the JSON text **`["a", 1]`**. Unfortunately, this does not interact very well with the basic algorithm. The JSON text **`["a", 1]`** would be inferred as the shape $[\![\textbf{any}]\!]$. Rust does, however, have tuples, and could represent the type as **`(String, i64)`**, so if it is possible to do so without breaking the original algorithm it makes sense to try to infer tuples.

$$\text{vts}(\texttt{Array(}[\omega_1, \cdots, \omega_n]\texttt{)}) =$$
$$\begin{cases} \langle\!\langle \text{vts}(\omega_1), \cdots, \text{vts}(\omega_n), 1 \rangle\!\rangle & \text{if } 1 < n \leq tmax \\ [\![\text{fold}(\text{csh}, \bot, [\text{vts}(\omega_1), \cdots, \text{vts}(\omega_n)])]\!] & \text{otherwise} \end{cases}$$

Figure 3.8: vts(ω), rule modification for tuples

To be able to infer tuples we add the tuple shape $\langle\!\langle \sigma_1, \cdots, \sigma_n, \kappa \rangle\!\rangle$ to our set of shapes, where $\sigma_1, \cdots, \sigma_n$ are field shapes and κ is the count of samples this shape is based on. Figure 3.8 shows how vts(ω) can be modified to infer tuples. The shown rule replaces the original rule for **`Array`**.

A tuple automatically implements all the traits in our default derive list if all the types inside it does [13, primitive std::tuple], and so can be safely added to our set of base types. This derive-like functionality only works for tuples with at most 12 fields, so the max inferred tuple arity – $tmax$ – should never be set higher than this. In my own opinion, anything beyond 2-tuples and 3-tuples should not be automatically inferred.

Figure 3.9 shows how csh(σ_1, σ_2) can be extended to accommodate for tuples. The two rules are pure additions, and none of the original rules are affected.

When we add tuples to the inference, while some tuples can be inferred automatically, our code generation should not be *too* eager to use tuples. We do not want every short list in our samples to be interpreted as a tuple. Figure 3.10 shows how tuple shapes

$$\mathrm{csh}(\langle\!\langle \sigma_1, \cdots, \sigma_n, \kappa_1 \rangle\!\rangle, \langle\!\langle \sigma'_1, \cdots, \sigma'_m, \kappa_2 \rangle\!\rangle) =$$
$$\begin{cases} \langle\!\langle \mathrm{csh}(\sigma_1, \sigma'_1), \cdots, \mathrm{csh}(\sigma_n, \sigma'_m), \kappa_1 + \kappa_2 \rangle\!\rangle & \text{if } n = m \\ [\![\mathrm{csh}(\mathrm{fold}(\mathrm{csh}, \bot, [\sigma_1, \cdots, \sigma_n]), \mathrm{fold}(\mathrm{csh}, \bot, [\sigma_1, \cdots, \sigma_n]))]\!] & \text{otherwise} \end{cases}$$
$$\mathrm{csh}(\langle\!\langle \sigma_1, \cdots, \sigma_n, \kappa \rangle\!\rangle, [\![\sigma_m]\!]) =$$
$$[\![\mathrm{csh}(\mathrm{fold}(\mathrm{csh}, \bot, [\sigma_1, \cdots, \sigma_n]), \sigma_m)]\!]$$

Figure 3.9: $\mathrm{csh}(\sigma_1, \sigma_2)$, rule additions for tuples

$$\mathrm{tolist}(\langle\!\langle \sigma_1, \cdots, \sigma_n, \kappa \rangle\!\rangle) = [\![\mathrm{fold}(\mathrm{csh}, \bot, [\sigma_1, \cdots, \sigma_n])]\!]$$

$$\mathrm{test}(\sigma_t = \langle\!\langle \sigma_1, \cdots, \sigma_n, \kappa \rangle\!\rangle) =$$
$$\kappa \geq tsamplemin \vee ((\exists \sigma \in [\sigma_1, \cdots, \sigma_n].\sigma \neq \textbf{any}) \wedge \mathrm{tolist}(\sigma_t) = [\![\textbf{any}]\!])$$

Figure 3.10: Removal of tuple shapes

can be converted to lists, and the test for whether a tuple shape should be used or converted to a list shape. Intuitively, a tuple shape should only be used if we have either seen enough instances to believe it is not just a short list, or if the use of a tuple type will prevent us from combining shapes into the fallback shape **any**. Note that the clause $(\exists \sigma \in [\sigma_1, \cdots, \sigma_n].\sigma \neq \textbf{any})$ will be redundant if we consider 2 to be enough samples to use a tuple.

If automatic inference of tuples is disabled, tuple shapes can still be created, and every tuple shape encountered in code generation can be converted to a list unless locally enabled by pointer configuration.

3.4.7 Combining identical subtypes

When we generate types with the basic inference and generation, we can end up with many types with exactly the same fields. In appendix C.2 we can see a real world example of such identical types[3]. As mentioned in section 2.6.2, one of the problems with this is that nominally different types can't be used interchangeably, but it also makes the types harder to understand, and contributes to code bloat.

One way to combine identical shapes to a single type is to let the user specify with JSON Pointer options, if two parts of the sample should correspond to the same type. However, for trivially equal shapes, this is something we should be able to do automatically.

Finding identical subtypes is the task of finding identical subtrees in our shape tree. A naive approach needs $\binom{n}{2} = \frac{n(n-1)}{2} \in O(n^2)$ comparisons, where n is the number of nodes in the tree, so care must be taken on implementation. The shape tree can be considered to be a rooted unordered labelled tree. Unordered, as we want types with the same fields in different order to be considered to be the same. An optimal algorithm for finding identical subtrees of such a tree can be implemented with complexity $O(n)$ [14].

For considering whether two shapes represent the same field type, the default derived equality operator should not be used. For instance should the shapes $[\![\bot]\!]$ and $[\![\texttt{int}]\!]$ be considered equal.

In addition to finding identical types for a single shape tree i.e. from a single sample, it could also be useful if it was possible to combine identical types across several samples. E.g. if a program interacts with an external API, the API may have several endpoints with different requirements, but with reuse of types. As an example of this, an API for web commerce might choose to inline product data both in search results and in endpoints for order data.

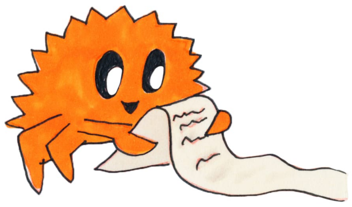
As we saw in section 2.6.2 it is already possible to use the *frunk* crate to get a level of structural typing and thus convert between types which are nominally different but

[3]In the example there are two sets of identical types: $\{$**Indexed**, **Start**, **Created**, **Deposited**$\}$ and $\{$**PublishedPrint**, **Start2**, **End**, **Issued**$\}$. Additionally, the shape of the second set can be considered to be covered by the shape of the first set. Handling such covered shapes automatically is not discussed in this section.

arose from identical shapes. This of course also applies between types generated from different samples.

It is also possible to combine multiple samples into a single sample by wrapping them in a JSON object with field names corresponding to the desired type names of the root of each sample. This will, however, only work for inline samples if done by the user. Thus, if such combining of identical subtypes is implemented, it would likely also be useful if *json_typegen* provided a macro which could combine samples in this manner internally, as this combining could then be done after parsing each sample into a **`serde_json::Value`**.

Chapter 4



Conclusions and future work

In this final chapter, I will look at the state and future of *json_typegen* itself, as well as how the ideas explored in this project can be continued further.

4.1 Implementation state

While the core of *json_typegen* – and several of the configuration options and extensions outlined in this thesis – can be considered complete, there is a tremendous amount of features and extensions that *could* be implemented. As *json_typegen* is a project I have use for myself, and I see (from the very limited logging on my server, the statistics on the official crate repository[1] and the direct feedback I have received) that it is also of interest to others, I will continue to implement many of these features and extensions to the project myself.[2]

The table included below shows the current state of the implementation. At its current state it clearly shows the cost of multiple interfaces. Several extensions are implemented, but are only available through direct invocation of the code generation, as the necessary interface code is extensive and time consuming.

[1]https://crates.io

[2]Though I *do* hope this thesis may also be of use to any potential contributors in the future.

Feature	Relevant sections	Inference	Codegen	Macro	CLI	Web
Basic functionality	2	✓	✓	✓	✓	✓
Type visibility	2.6.1	✓	✓	✓	✓	✓
Field visibility	2.6.1	✓	✓	-	-	-
Derive list	2.6.2, 3.3.3		✓	-	-	✓
Nullable collections	3.1.1		✓	-	-	-
Missing fields	3.3.4		✓	-	-	-
Unknown fields	3.3.4		✓	-	-	-
Tagged **`any`** types	3.4.1	-	-			
JSON Pointers	2.6.3, 3.4.3	✓	-	-	-	-
`use_type`: u64, i8, etc.	3.4.4	-	-	-	-	-
`use_type`: shape	3.4.3	-				
`use_type`: opaque	3.4.3	-	-			
`use_type`: map	3.4.5	✓	✓			
`use_type`: tuple	3.4.6	✓	✓	-	-	-
`type_name`	2.6.3	-	-			
`same_as`	2.6.3	-	-			
Detection of equal types	3.4.7	-	-			
Macro syntax input	2.7				-	-
Macro syntax output	2.7				-	-

Legend: Complete: ✓ Missing: - Not applicable: *(blank)*

Despite how clear this cost is, I am convinced the benefits of having these interfaces outweighs the cost, and that multiple interfaces is the strongest benefit *json_typegen* has over a pure type-provider-like interface. Each interface is useful in itself even without the synergy, and having to rewrite the (far from trivial) code generation for each interface would make it all the more likely that an interface might not exist at all.

As mentioned in section 2.6.4, it may be possible to generate interface implementations, at least to a certain extent. If this is achieved, both extending *json_typegen* itself and expanding to other formats beside JSON should be substantially easier.

4.2 Expanding to other formats

While JSON is perhaps the most popular serialization format at the moment, it is obviously far from the only one. *F# Data*, which was the main inspiration for *json_typegen*, has type providers for XML, HTML and CSV in addition to JSON. These type providers have a common core of inference and type generation logic. While some of the choices and extensions of *json_typegen* are somewhat specific to JSON, like *F# Data*, it should be possible to use the same core for multiple formats. For types that have both a *serde* implementation and either a **`Value`** type of its own, or somehow can map into **`serde_json::Value`**, adding basic support for the format should be somewhat straight forward. For some formats it may however be better to create entirely separate libraries, as not every format maps straightforwardly to the set of shapes used by *json_typegen*.

4.3 JSON Schema

JSON itself is a simple, schemaless data format. There is however a separate schema format – JSON Schema[3] – that is in development (and has been for some years). It is likely that similar code generation techniques to those used in this project could be applied to JSON Schema, and it may seem like a natural progression from "types from samples" is "types from schemas". There are however some issues with JSON Schema that make me a bit hesitant to start (or recommend) a code generation project for JSON Schema:

- While it is only a personal observation, it seems to me as though a lot of the people/websites actually using JSON Schema (often in Swagger/OpenAPI) don't actually use it according to the specification. As an example; many completely omit the "required" field that specify which fields of an object are not nullable.

[3]http://json-schema.org/

In my opinion this defeats some of the point of a schema, as it leaves us unable to generate code that is both safe and ergonomic without inspecting the data itself and making assumptions based on it.

- JSON Schema also has some challenges for code generation in that it can easily represent types which are very hard to represent with *serde*-compatible types, so it may require actually writing the deserialization code more or less specifically for any hypothetical JSON Schema library. The references in JSON Schema also means that a simple, recursive tree walk is not enough to parse a schema into shapes.
- The specification also lacks straightforward ways to represent usage patterns that are already common in JSON, like objects as maps and arrays as tuples.

Another interesting possibility is the use of the inferred shapes of *json_typegen* to *create* JSON schemas. While this would certainly be possible, I am a little skeptical of it's usefulness. However, as it should be a rather straightforward addition of another output format to *json_typegen*, even it is only slightly useful, that may be enough for someone to do it when the inference is already in place.

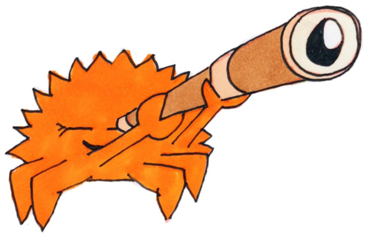
4.4 Towards a true type provider experience for Rust

In closing, I want to touch on the question of why languages with full support for type providers are as rare as they are. As explained in section 1.3.1, support for type providers is only two relatively simple to understand concepts: The ability to create types based on some external resource, and the ability to delay the creation of parts of these types via e.g. thunks.

While full procedural macros are not all that common, they are not exceedingly rare either. A simple version of procedural macros can be implemented with a simple preprocessing step. Especially if they are only to be used for basic type provider support. For most type providers, a simple string or two is enough input to do something useful. The fact that procedural macros are in a sense external to the core of a language is also part of why tooling support for procedural macros lag behind. They have to be expanded, evaluated, eliminated, for tools to do their job.

Delayed evaluation of parts of types, however, is something that is much rarer, as this concept has effects into how the very type system is evaluated inside a compiler. So, would this even be possible in Rust? I'm not sure. If one were to explore this, a good starting point would perhaps be to implement a toy language which does little more than support thunks in types.

However, I believe both *F# Data* and *json_typegen* show that this second part of type provider support is not essential to create something useful. And while it may not be necessary to add a code generation tool onto every compiler, for a language such as Rust, which have the necessary tools for basic type providers already available, a future where projects that look like type providers are exceedingly common seems almost inevitable. As soon as tooling catches up.



Appendix A

Rust implementation of inference algorithm

```rust
extern crate linked_hash_map;
extern crate serde_json;

use linked_hash_map::LinkedHashMap;
use serde_json::{Value, Map};

#[derive(Debug, PartialEq, Clone)]
pub enum Shape {
    Any,
    Bottom,
    Bool,
    StringS,
    Int,
    Float,
    List { elem_type: Box<Shape> },
    Recd { fields: LinkedHashMap<String, Shape>, },
    Optional(Box<Shape>),
}
```

```
pub fn value_to_shape(value: &Value) -> Shape {
    match *value {
        Value::Null => Shape::Optional(Box::new(Shape::Bottom)),
        Value::Bool(_) => Shape::Bool,
        Value::Number(ref n) => {
            if n.is_i64() {
                Shape::Int
            } else {
                Shape::Float
            }
        }
        Value::String(_) => Shape::StringS,
        Value::Array(ref values) => array_to_shape(values),
        Value::Object(ref map) => object_to_shape(map),
    }
}

fn array_to_shape(values: &[Value]) -> Shape {
    let inner =
        values.iter().fold(Shape::Bottom, |shape, val| {
            let shape2 = value_to_shape(val);
            common_shape(shape, shape2)
        });
    Shape::List {
        elem_type: Box::new(inner),
    }
}

fn object_to_shape(map: &Map<String, Value>) -> Shape {
    let inner = map.iter()
        .map(|(name, value)| {
            (name.clone(), value_to_shape(value))
        })
        .collect();
    Shape::Recd { fields: inner }
}
```

```rust
fn common_shape(a: Shape, b: Shape) -> Shape {
    if a == b {
        return a;
    }
    use self::Shape::*;
    match (a, b) {
        (a, Bottom) | (Bottom, a) => a,
        (Int, Float) | (Float, Int) => Float,
        (a, Optional(b)) | (Optional(b), a) => make_optional(
            common_shape(a, *b),
        ),
        (List { elem_type: e1 }, List { elem_type: e2 }) => {
            List {
                elem_type: Box::new(common_shape(*e1, *e2)),
            }
        }
        (Recd { fields: f1 }, Recd { fields: f2 }) => Recd {
            fields: common_field_shapes(f1, f2),
        },
        _ => Any,
    }
}

fn make_optional(a: Shape) -> Shape {
    use self::Shape::*;
    match a {
        Any | Optional(_) => a,
        non_nullable => Optional(Box::new(non_nullable)),
    }
}
```

```rust
fn common_field_shapes(
    f1: LinkedHashMap<String, Shape>,
    mut f2: LinkedHashMap<String, Shape>,
) -> LinkedHashMap<String, Shape> {
    if f1 == f2 {
        return f1;
    }
    let mut unified = LinkedHashMap::new();
    for (key, val) in f1.into_iter() {
        match f2.remove(&key) {
            Some(val2) => {
                unified.insert(key, common_shape(val, val2));
            }
            None => {
                unified.insert(key, make_optional(val));
            }
        }
    }
    for (key, val) in f2.into_iter() {
        unified.insert(key, make_optional(val));
    }
    unified
}
```

Appendix B

Naive implementation of generation of Rust types from shapes

```
pub fn shape_to_code(
    name: &str,
    shape: &Shape,
) -> (String, Option<String>) {
    match *shape {
        Shape::Any | Shape::Bottom => (
            "::serde_json::Value".into(),
            None,
        ),
        Shape::Bool => ("bool".into(), None),
        Shape::StringS => ("String".into(), None),
        Shape::Int => ("i64".into(), None),
        Shape::Float => ("f64".into(), None),
        Shape::List { elem_type: ref e } => {
            let (inner, inner_defs) = shape_to_code(name, e);
            (format!("Vec<{}>", inner), inner_defs)
        }
```

```rust
        Shape::Recd { fields: ref map } => {
            let type_name = uppercase_first_letter(name);
            let mut inner_defs = String::new();

            let mut struct_def =
                format!("struct {} {{\n", type_name);
            for (key, val) in map.iter() {
                let (field_type, field_defs) =
                    shape_to_code(key, val);
                if let Some(defs) = field_defs {
                    inner_defs += &defs;
                }
                struct_def +=
                    &format!("  {}: {}\n", key, field_type);
            }
            struct_def += "}}\n";

            (type_name, Some(struct_def + &inner_defs))
        }
        Shape::Optional(ref e) => {
            let (inner, inner_defs) = shape_to_code(name, e);
            if **e == Shape::Bottom {
                (inner, inner_defs)
            } else {
                (format!("Option<{}>", inner), inner_defs)
            }
        }
    }
}

fn uppercase_first_letter(s: &str) -> String {
    match s.chars().next() {
        None => String::new(),
        Some(c) => c.to_uppercase().to_string() + &s[1..],
    }
}
```

Appendix C

Examples of generated Rust types

This appendix contains examples of the result of using the code generation of *json_typegen* on various real-world JSON samples.

The JSON shown may be abridged, where such changes do not affect the inference results. E.g. an array with 50 elements may have been reduced to only 2 or 3.

C.1 Launch Library Launch List

API documentation: https://launchlibrary.net/1.2/docs/api.html
Sample source: https://launchlibrary.net/1.2/launch?next=2

JSON Sample

```
{
  "total": 2,
  "launches": [
    {
      "id": 1329,
      "name": "Vega | OptSat 3000 & VENμS (VENUS)",
      "net": "August 2, 2017 01:58:00 UTC",
      "tbdtime": 0,
      "tbddate": 0
    },
    {
      "id": 1233,
      "name": "Long March 3B/E | Alcomsat-1",
      "net": "August 5, 2017 00:00:00 UTC",
      "tbdtime": 1,
      "tbddate": 1
    }
  ],
  "offset": 0,
  "count": 2
}
```

Generated code

```
#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct LaunchList {
    total: i64,
    launches: Vec<Launch>,
    offset: i64,
    count: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Launch {
    id: i64,
    name: String,
    net: String,
    tbdtime: i64,
    tbddate: i64,
}
```

C.2 CrossRef DOI API

API documentation: https://github.com/CrossRef/rest-api-doc
Sample source: https://api.crossref.org/works/10.1145/2908080.2908115/

JSON Sample

```
{
  "status": "ok",
  "message-type": "work",
  "message-version": "1.0.0",
  "message": {
    "indexed": { "date-parts": [[2017, 7, 25]],
     ↪   "date-time": "2017-07-25T04:31:32Z",
     ↪   "timestamp": 1500957092169 },
    "publisher-location": "New York, New York, USA",
    "reference-count": 26,
    "publisher": "ACM Press",
    "license": [
      {
        "URL":
         ↪   "http://www.acm.org/publications/poli-
         ↪   cies/copyright_policy#Background",
        "start": { "date-parts": [[2016, 6, 13]],
         ↪   "date-time": "2016-06-13T00:00:00Z",
         ↪   "timestamp": 1465776000000 },
        "delay-in-days": 164,
        "content-version": "vor"
      }
    ],
    "content-domain": { "domain": [ ],
     ↪   "crossmark-restriction": false },
    "short-container-title": [ ],
    "published-print": { "date-parts": [[2016]] },
    "DOI": "10.1145/2908080.2908115",
    "type": "proceedings-article",
    "created": { "date-parts": [[2016, 6, 2]],
     ↪   "date-time": "2016-06-02T19:23:42Z",
     ↪   "timestamp": 1464895422000 },
    "source": "Crossref",
    "is-referenced-by-count": 0,
    "title": ["Types from data: making structured
     ↪   data first-class citizens in F#"],
    "prefix": "10.1145",
    "author": [
      { "given": "Tomas", "family": "Petricek",
       ↪   "affiliation": [{ "name": "University of
       ↪   Cambridge, UK" }] },
      { "given": "Gustavo", "family": "Guerra",
       ↪   "affiliation": [{ "name": "Microsoft,
       ↪   UK" }] },
      { "given": "Don", "family": "Syme",
       ↪   "affiliation": [{ "name": "Microsoft
       ↪   Research, UK" }] }
    ],
    "member": "320",
    "reference": [
      {
        "key": "key-10.1145/2908080.2908115-1",
        "unstructured": "L. Cardelli and J. C.
         ↪   Mitchell. Operations on Records. In
         ↪   Mathematical Foundations of
         ↪   Programming Semantics, pages
         ↪   22&#8211;52. Springer, 1990.",
        "DOI": "10.1007/BFb0040253",
        "doi-asserted-by": "crossref"
      },
      {
        "key": "key-10.1145/2908080.2908115-2",
        "unstructured": "A. Chlipala. Ur:
         ↪   Statically-typed Metaprogramming with
         ↪   Type-level Record Computation. In ACM
         ↪   SIGPLAN Notices, volume 45, pages
         ↪   122&#8211;133. ACM, 2010."
      }
    ],
    "event": {
      "name": "the 37th ACM SIGPLAN Conference",
      "location": "Santa Barbara, CA, USA",
      "sponsor": ["SIGPLAN, ACM Special Interest
       ↪   Group on Programming Languages"],
      "acronym": "PLDI 2016",
      "number": "37",
      "start": { "date-parts": [[2016, 6, 13]] },
      "end": { "date-parts": [[2016, 6, 17]] }
    },
    "container-title": [
      "Proceedings of the 37th ACM SIGPLAN
       ↪   Conference on Programming Language
       ↪   Design and Implementation - PLDI 2016"
    ],
    "original-title": [ ],
    "deposited": { "date-parts": [[2017, 6, 24]],
     ↪   "date-time": "2017-06-24T15:39:00Z",
     ↪   "timestamp": 1498318740000 },
    "score": 1,
    "subtitle": [ ],
    "short-title": [ ],
    "issued": { "date-parts": [[2016]] },
    "ISBN": ["9781450342612"],
    "references-count": 26,
    "URL":
     ↪   "http://dx.doi.org/10.1145/2908080.2908115",
    "relation": { "cites": [ ] }
  }
}
```

Generated code

```rust
#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct CrossRefMetadata {
    status: String,
    #[serde(rename = "message-type")]
    message_type: String,
    #[serde(rename = "message-version")]
    message_version: String,
    message: Message,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct Message {
    indexed: Indexed,
    #[serde(rename = "publisher-location")]
    publisher_location: String,
    #[serde(rename = "reference-count")]
    reference_count: i64,
    publisher: String,
    license: Vec<License>,
    #[serde(rename = "content-domain")]
    content_domain: ContentDomain,
    #[serde(rename = "short-container-title")]
    short_container_title: Vec<::serde_json::Value>,
    #[serde(rename = "published-print")]
    published_print: PublishedPrint,
    #[serde(rename = "DOI")]
    doi: String,
    #[serde(rename = "type")]
    type_field: String,
    created: Created,
    source: String,
    #[serde(rename = "is-referenced-by-count")]
    is_referenced_by_count: i64,
    title: Vec<String>,
    prefix: String,
    author: Vec<Author>,
    member: String,
    reference: Vec<Reference>,
    event: Event,
    #[serde(rename = "container-title")]
    container_title: Vec<String>,
    #[serde(rename = "original-title")]
    original_title: Vec<::serde_json::Value>,
    deposited: Deposited,
    score: i64,
    subtitle: Vec<::serde_json::Value>,
    #[serde(rename = "short-title")]
    short_title: Vec<::serde_json::Value>,
    issued: Issued,
    #[serde(rename = "ISBN")]
    isbn: Vec<String>,
    #[serde(rename = "references-count")]
    references_count: i64,
    #[serde(rename = "URL")]
    url: String,
    relation: Relation,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct Indexed {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
    #[serde(rename = "date-time")]
    date_time: String,
    timestamp: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct License {
    #[serde(rename = "URL")]
    url: String,
    start: Start,
    #[serde(rename = "delay-in-days")]
    delay_in_days: i64,
    #[serde(rename = "content-version")]
    content_version: String,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct Start {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
    #[serde(rename = "date-time")]
    date_time: String,
    timestamp: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct ContentDomain {
    domain: Vec<::serde_json::Value>,
    #[serde(rename = "crossmark-restriction")]
    crossmark_restriction: bool,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct PublishedPrint {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct Created {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
    #[serde(rename = "date-time")]
    date_time: String,
    timestamp: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct Author {
    given: String,
    family: String,
    affiliation: Vec<Affiliation>,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪  Serialize, Deserialize)]
struct Affiliation {
    name: String,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Reference {
    key: String,
    unstructured: String,
    #[serde(rename = "DOI")]
    doi: Option<String>,
    #[serde(rename = "doi-asserted-by")]
    doi_asserted_by: Option<String>,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Event {
    name: String,
    location: String,
    sponsor: Vec<String>,
    acronym: String,
    number: String,
    start: Start2,
    end: End,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Start2 {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct End {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Deposited {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
    #[serde(rename = "date-time")]
    date_time: String,
    timestamp: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Issued {
    #[serde(rename = "date-parts")]
    date_parts: Vec<Vec<i64>>,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Relation {
    cites: Vec<::serde_json::Value>,
}
```

C.3 Steam Store News

API documentation: https://developer.valvesoftware.com/wiki/Steam_Web_API
Sample source: http://api.steampowered.com/ISteamNews/GetNewsForApp/v0002/?appid=252950&count=2&maxlength=150

JSON Sample

```
{
  "appnews": {
    "appid": 252950,
    "newsitems": [
      {
        "gid": "2079952210065770645",
        "title": "DreamHack Atlanta Rocket League
          ↪  Championship Preview",
        "url": "http://store.steampow-
          ↪  ered.com/news/externalpost/steam_com-
          ↪  munity_announce-
          ↪  ments/2079952210065770645",
        "is_external_url": true,
        "author": "Dirkened",
        "contents": "https://rocketleague.me-
          ↪  dia.zestyio.com/DreamHack-
          ↪  Atlanta.c6e1dc555a6eff57c623d9877706c9a5.jpg
          ↪  With the conclusion of the FACEIT X
          ↪  Games Rocket League ...",
        "feedlabel": "Community Announcements",
        "date": 1500498351,
        "feedname": "steam_community_announcements",
        "feed_type": 1,
        "appid": 252950
      },
      {
        "gid": "2472890725411073707",
        "title": "RLCS Season 4 Kicks Off this
          ↪  August",
        "url": "http://store.steampow-
          ↪  ered.com/news/externalpost/steam_com-
          ↪  munity_announce-
          ↪  ments/2472890725411073707",
        "is_external_url": true,
        "author": "Dirkened",
        "contents": "https://rocketleague.me-
          ↪  dia.zestyio.com/rlcs_screen--1-
          ↪  .c6e1dc555a6eff57c623d9877706c9a5.png
          ↪  Just six weeks ago, we were
          ↪  celebrating the Rocket League ...",
        "feedlabel": "Community Announcements",
        "date": 1500313284,
        "feedname": "steam_community_announcements",
        "feed_type": 1,
        "appid": 252950
      }
    ],
    "count": 341
  }
}
```

Generated code

```
#[derive(Default, Debug, Clone, PartialEq,
  ↪  Serialize, Deserialize)]
struct SteamAppNews {
    appnews: Appnews,
}

#[derive(Default, Debug, Clone, PartialEq,
  ↪  Serialize, Deserialize)]
struct Appnews {
    appid: i64,
    newsitems: Vec<Newsitem>,
    count: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
  ↪  Serialize, Deserialize)]
struct Newsitem {
    gid: String,
    title: String,
    url: String,
    is_external_url: bool,
    author: String,
    contents: String,
    feedlabel: String,
    date: i64,
    feedname: String,
    feed_type: i64,
    appid: i64,
}
```

C.4 World Bank Indicator

API documentation: https://datahelpdesk.worldbank.org/knowledgebase/topics/125589
Sample source: http://api.worldbank.org/countries/no/indicators/NY.GDP.MKTP.CD?format=json

JSON Sample

```
[
  { "page": 1, "pages": 2, "per_page": "50",
   ↪   "total": 57 },
  [
    {
      "indicator": {
        "id": "NY.GDP.MKTP.CD",
        "value": "GDP (current US$)"
      },
      "country": { "id": "NO", "value": "Norway" },
      "value": "386383919342.271",
      "decimal": "0",
      "date": "2009"
    },
    {
      "indicator": {
        "id": "NY.GDP.MKTP.CD",
        "value": "GDP (current US$)"
      },
      "country": { "id": "NO", "value": "Norway" },
      "value": "461946808510.638",
      "decimal": "0",
      "date": "2008"
    },
    {
      "indicator": {
        "id": "NY.GDP.MKTP.CD",
        "value": "GDP (current US$)"
      },
      "country": { "id": "NO", "value": "Norway" },
      "value": "400883873279.083",
      "decimal": "0",
      "date": "2007"
    }
  ]
]
```

Generated code

```
#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct WorldBankIndicator {
    page: i64,
    pages: i64,
    per_page: String,
    total: i64,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct WorldBankIndicator2 {
    indicator: Indicator,
    country: Country,
    value: String,
    decimal: String,
    date: String,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Indicator {
    id: String,
    value: String,
}

#[derive(Default, Debug, Clone, PartialEq,
 ↪   Serialize, Deserialize)]
struct Country {
    id: String,
    value: String,
}
```

Appendix D

Comparison of project setup with dependencies in Rust and C++

Minimal project setup in Rust

1. Install the Rust toolchain:
   `curl https://sh.rustup.rs -sSf | sh`
2. Create a new (binary) project:
   `cargo new --bin timer && cd timer`
3. Add a dependency to the Cargo configuration file:
   `echo 'tokio-timer = "*"' >Cargo.toml`
4. Write some actual Rust code:
   `$EDITOR src/main.rs`
5. Build and run:
   `cargo run`

Minimal project setup in C++

1. Choose and install a C++ toolchain. What toolchains are available and how to install them differs from platform to platform, and some may have a toolchain installed. To keep things somewhat simple, installation is omitted here.

2. Choose and install a dependency manager. Throughout the years there have been several C++ dependency manager projects that have come and gone, so this choice is not entirely without risks. Without going into the details why, I chose *conan*[1]. The rest of these steps are based on the *conan* "Getting started"-guide[2]. Again installation depends somewhat on platform. For my part I installed it using *homebrew*[3]:
   ```
   brew install conan
   ```

3. Choose and install a build tool. Since the *conan* guide uses *CMake* this is also what we will use here. Again, installation method differs, but for my part:
   ```
   brew install cmake
   ```

4. Make a project directory:
   ```
   mkdir timer && cd timer
   ```

5. Make and write a *conan* configuration file:
   ```
   $EDITOR conanfile.txt
   ```

6. Write some actual C++ code:
   ```
   $EDITOR timer.cpp
   ```

7. Make and write a *CMake* configuration file:
   ```
   $EDITOR CMakeLists.txt
   ```

8. Make a build directory:
   ```
   mkdir build && cd build
   ```

9. Install dependencies:
   ```
   conan install ..
   ```

[1]http://conan.io

[2]http://docs.conan.io/en/latest/getting_started.html

[3]Which itself would have to been installed if it was not so already.

10. Prepare build:
    `cmake ..`
11. Build:
    `cmake --build .`
12. Run the resulting program:
    `bin/timer`

Bibliography

Referenced software projects

[1] *diesel. A safe, extensible ORM and Query Builder for Rust*. Version 0.14. Website: http://diesel.rs/. Repository: https://github.com/diesel-rs/diesel. July 5, 2017.

[2] *F# Data. Library for Data Access*. Version 2.3.3. Website: http://fsharp.github.io/FSharp.Data/. Repository: https://github.com/fsharp/FSharp.Data. July 5, 2017.

[3] *frunk. Funktional generic programming in Rust*. Version 0.1.30. Repository: https://github.com/lloydmeta/frunk. July 5, 2017.

[4] *quote. Rust quasi-quoting*. Version 0.3.15. Repository: https://github.com/dtolnay/quote. July 22, 2017.

[5] *serde. Serialization framework for Rust*. Version 1.0. Website: https://serde.rs/. Repository: https://github.com/serde-rs/serde. July 5, 2017.

[6] *serde_json. JSON Serialization for Rust*. Version 1.0. Repository: https://github.com/serde-rs/json. July 5, 2017.

[7] *structopt. Parse command line arguments by defining a Rust struct*. Version 0.1.0. Repository: https://github.com/TeXitoi/structopt. July 25, 2017.

[8] *vulkano. Safe and rich Rust wrapper around the Vulkan API*. Version 0.5. Website: http://vulkano.rs/. Repository: https://github.com/tomaka/vulkano. July 5, 2017.

Other references

[9] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. RFC Editor, Mar. 2014. URL: http://www.rfc-editor.org/rfc/rfc7159.txt.

[10] Paul Bryan and Mark Nottingham. *JavaScript Object Notation (JSON) Patch*. RFC 6902. RFC Editor, Apr. 2013. URL: http://www.rfc-editor.org/rfc/rfc6902.txt.

[11] Paul Bryan, Kris Zyp, and Mark Nottingham. *JavaScript Object Notation (JSON) Pointer*. RFC 6901. RFC Editor, Apr. 2013. URL: http://www.rfc-editor.org/rfc/rfc6901.txt.

[12] David Raymond Christiansen. "Dependent Type Providers". In: *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. WGP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 25–34. URL: http://doi.org/10.1145/2502488.2502495.

[13] The Rust Project Developers. *Rust standard library API documentation*. 2017. URL: https://doc.rust-lang.org/std/ (visited on 06/26/2017).

[14] Tomáš Flouri, Kassian Kobert, Solon P. Pissis, and Alexandros Stamatakis. "An optimal algorithm for computing all subtree repeats in trees". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 372.2016 (2014). URL: http://doi.org/10.1098/rsta.2013.0140.

[15] Graham Hutton. "A tutorial on the universality and expressiveness of fold". In: *Journal of Functional Programming* 9.4 (1999), pp. 355–372. URL: http://doi.org/10.1017/S0956796899003500.

[16] Nicholas D. Matsakis and Felix S. Klock II. “The Rust Language”. In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT ’14. Portland, Oregon, USA: ACM, 2014, pp. 103–104. URL: http://doi.org/10.1145/2663171.2663188.

[17] Tomas Petricek, Gustavo Guerra, and Don Syme. “Types from Data: Making Structured Data First-class Citizens in F#”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. Santa Barbara, CA, USA: ACM, 2016, pp. 477–490. URL: http://doi.org/10.1145/2908080.2908115.

[18] Donald Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. “Themes in Information-rich Functional Programming for Internet-scale Data Sources”. In: *Proceedings of the 2013 Workshop on Data Driven Functional Programming*. DDFP ’13. Rome, Italy: ACM, 2013, pp. 1–4. URL: http://doi.org/10.1145/2429376.2429378.

[19] *The JSON Data Interchange Format*. Tech. rep. Standard ECMA-404 1st Edition / October 2013. ECMA, Oct. 2013. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[20] Mitchell Wand. “Type Inference for Record Concatenation and Multiple Inheritance”. In: *Inf. Comput.* 93.1 (July 1991), pp. 1–15. URL: http://doi.org/10.1016/0890-5401(91)90050-C.