Synthesizing JSON Transformations from Input-Output Examples

Eduardo S. Hidalgo[1]

Purdue University hidalgov@purdue.edu

1 Introduction

JSON has become an ubiquitous data interchange and storage format. Some of its uses include web development, NoSQL databases, Data Science, and as a configuration language. Although originally developed for Javascript, it has found usage across all modern programming languages. It has a hierarchical and flexible format that allows it to express lots of common structures. Significant amount of code in clients and servers is written to transform JSON documents from one shape to another, which done improperly represents a tedious and error prone task. Most REST APIs provide endpoints that send and receive JSON; and when they are documented, they usually use a couple of examples to illustrate how it should be structured. Bearing this in mind, it seems that the task of formulating those transformations would be made easier if the user was only required to provide examples of how the data should end up looking like. This idea would further be useful in that it would also aid the development of "glue code" between otherwise incompatible services (i.e. when there's a shape mistmatch between the way they express JSON).

This paper proposes a program synthesizer for javascript programs that performs a transformation following the paradigm of Programming-by-Example [11] [9]. The resulting program is consistent with a set of JSON input-output examples given by the user. The approach consists of a typed DSL inspired in jq [7] and jl [6] in which interesting transformations can be expressed, along with a type-aware search strategy through the program space. Using type information has proven to be useful for program synthesis [2] [10].

To test such approach, I have implemented a prototype tool called jsyn [5] along with a benchmark of 10 JSON transformation tasks I collected from StackOverflow. The tool is written in Haskell and takes advantage of its laziness to concisely express the search space.

2 Preliminaries

2.1 JSON Format

JSON is a data format that has a textual representation but at it's core it represents a set of *JSON values*. We call this set *JsonValue* and is the most important

concept from JSON discussed in this paper. JSON syntax, encoding/decoding, and other subtleties will not be covered.

A JSON value can be either a structure or a single value. There are two structures: sets of key-values (*JsonObject*), and lists (*JsonArray*). Both *JsonObject* and *JsonArray* deal with finite sets and lists respectively, because of their text form origins. Formally, the (infinite) set *JsonValue* is defined as:

$$
\begin{aligned}
JsonValue &::= JsonObject \mid JsonArray \mid Number \mid String \mid Bool \mid Null \\
JsonObject &::= \{\ (k_i, v_i) \mid k_i \in String, v_i \in JsonValue\ \} \\
JsonArray &::= [\ v_i \mid v_i \in JsonValue\ ]
\end{aligned}
$$

Examples of JSON values:

- `"abc"`
- `1`
- `3.14`
- `true`
- `false`
- `{"name": "John", "age": 20, "gpa": 4.0 }`
- `["red", "black", "white", "green"]`

An important aspect of this definition that will resurface later throughout the paper is the fact that lists and objects do not impose constraints on the types of values can be inside them (i.e. they can be heterogenous). This characteristic gives JSON it's flexibility and expressivity. Another important characteristic is that JSON documents are *schemaless*[3], that is they do not carry metadata that describes it's structure. Being recursive, schemaless and flexible gives JSON an expressive way to describe values and structures.

2.2 JSON Transformations

We define a transformation as a function f whose domain is a structure; either objects or arrays (i.e. functions with signatures $JsonObject \rightarrow JsonValue$ or $JsonArray \rightarrow JsonValue$). This constraints the functions such that it cannot operate "inside" scalar values (*Strings*, *Numbers*, *Bools*, *Null*). This is a way to limit the domain on which to synthesize programs. If we didn't limit this, the scope of this synthesizer would be practically general purpose, since booleans, numbers, strings, etc. would be included in the domain. Even more, functions between domainds would be included too too (e.g. functions $String \rightarrow Number$ would be contained too). On the other hand, it's important to note that the codomain of the functions is not constrained (e.g. functions that receive an object and return a scalar are included). see Figure 1 for an example.

```
// input
{
  "name": "John",
  "age": 20,
  "gpa": 4.0
}
// output
{
  "age": 20,
  "gpa": 4.0
}
```

Fig. 1. A valid JSON transformation

3 Problem Definition

3.1 DSL

Our Synthesis Target is a minimal pure functional programming language specific to the task of transforming JSON:

$$e ::= v \mid get\ k \mid \{k_1 : e_1, ..., k_n : e_n\} \mid pipe\ e1\ e2 \mid map\ e \mid e1{+}{+}e2 \mid [e] \mid flatten\ e$$

were v is an element of $JsonValue$, k is a string, and $e1$, $e2$ are also in e. We refer to $\{k_1 : e_1, ..., k_n : e_n\}$ as the construct operator.

Every program we want to find has the shape: $\lambda \boldsymbol{x}.e$ so the synthesis algorithm only searches for a body e consistent with the specification.

Table 1. Informal semantics of operators

Operator	returns...
get k	the value at key k from an object
$\{k_1 : e_1, ..., k_n, e_n\}$	a new object where every key k_i has assigned the value e_i.
pipe $e1$ $e2$	feeds the value of e1 to e2, we can think of it as a regular *apply* function with its arguments flipped.
map e	a new list with the results of evaluating e1 against each position.
$e1$ ++ $e2$	the appending of e1 and e2.
[e]	a singleton list of containing e1.
flatten e	all all elements of econcatenated, i.e. it "flattens" the list.

A brief description of each operator is given in Table 1. For the sake of brevity, formal semantics are not given but the actual implementation in Haskell does not differ a lot from the mathematical one.

This DSL is an extremely simple PL in the sense that it doesn't have conditionals, loops, recursion, functional abstraction, or bindings. This is a key design

idea to keep the synthesis problem efficient enough for search. Expressions have strictly 1 argument that is implicit, and are evaluated with respect to the current value. Hence the importance of the pipe operator, since it allows nested computations to take place without the need of introducing new variables. To illustrate the point consider evaluating the expression: *pipe* (*get* "*hostinfo*") (*get* "*name*") against the value:

Example 1.

```
{
  "hostinfo": {
    "host": "<host1>",
    "name": "<name1>",
    "online": true
  },
  "id": 0
}
```

The first expression *get* "*hostinfo*" is evaluated against the current value and returns:

```
{
  "host": "<host1>",
  "name": "<name1>",
  "online": true
}
```

Then *get* "*name*" is evaluated against that and returns: `"<host1>"`. The same happens for map operator: in *map e*, *e* is evaluated in the context of each element of map. For example:

$$map\ (get\ "a") \tag{1}$$

against the value:

```
[{"a": 10}, {"a": 20}]
```

should return:

```
[10, 20]
```

As we will see in the discussion, this decision simplifies both the search and type inference algorithms.

3.2 Translation to JS

Circling back to our original goal of outputting *JS* JavaScript programs, we now need a way to convert our DSL expressions to *JS*. Every expression has a semantically equivalent expression in *JS* therefore translation is straightforward. The only thing that needs to be taken care of is keeping track of the

name that stands for the current value. The only operator that has to be printed differently is pipe: we first need to translate the first argument, and then substitute that as the current value in the second argument. For example, for $pipe(get\text{“}hostinfo\text{”})(get\text{“}name\text{”})$ with x as the argument `name`, first $get\text{”}hostinfo\text{”}$ is translated to `x.hostinfo` and then the e in `e.name` is replaced to get `x.hostinfo.name`. Finally, this is wrapped inside $\lambda x.e$ which in *JS* would be written:

```
function program(x) {
  return x.hostinfo.host;
}
```

This approach can be thought of as inlining the expression. Another alternative is to transform expressions to a variant of SSA by declaring variables for the intermediate transformations. e.g. for program:

$$\{\text{“}age\text{”} : (get\text{“}age\text{”}), \text{“}gpa\text{”} : (get\text{“}gpa\text{”})\}$$

the output would be:

```
function program(a) {
  const b = a.age;
  const c = a.gpa;
  const d = {"age": b, "gpa": c };
  return d;
}
```

In reality, the best approach would be a mix between this two, since the SSA approach is pretty verbose, but the inline version can get cluttered and produce long lines of code, this however is out of the scope.

It's important to explain that synthesis is not performed in *JS* itself because it is a language with a lot of quirks and subleties [8] [4]. A careful subset of it has been choosen as a pure language that fits the needs of our problem.

3.3 Synthesis Problem

A *json input-output example* is a pair (v_{in}, v_{out}) where $v_{in}, v_{out} \in JsonValue$. The input to the synthesis algorithm is a set $\mathcal{E}$ of input output examples. Let prg be a function $JsonValue \rightarrow JsonValue$, our synthesis problem is to find such a function that is *consistent* with $\mathcal{E}$, formally:

$$\exists\, prg.\ \forall (v_in, v_out) \in \mathcal{E}.\ prg(v_in) \equiv v_out$$

Since the specification is just a set of examples it is not meant to be complete.

3.4 Type System

$$T ::= T \rightarrow T \mid V$$
$$V ::= \{k_1 : V_1, ..., k_n : V_n\} \mid [V] \mid String \mid Number \mid Bool \mid Null \mid Value$$

A type T represents a subset of possible values. In a way, it is abstracting information about particular values and can be thought about as sets. There are two main Kinds of types: arrow types (that represent functions) and value types that are disjoint subsets of *JsonValue*. This distinction is important since Objects and Lists can't hold functions in JSON. Some examples of type judgements:

- `"abc" : String`
- `2 : Number`
- `{"name": "Bob", "age": 22} : {"name": String, "age": Number}`
- `[1,2,3] : [Number]`

JSON's flexibility with respect to heterogeneous values in lists poses the question of which is the type of the array `["one", 1]`. In our system only heterogeneous arrays are expressed, so if it has different types it is assigned the type `[Value]`. Value is a supertype containing all value types. The only exception to this is the case in which the array has objects. Then its type is the maximum subset of key-values shared among all elements.[1] Following is an example of type judgements for mixed arrays: `["one", 1] : [Value]`

```
  [{"name": "Alice", "age": 20, "gpa": 4.0},
  {"name": "Bob", "age": 2}]
: [{"name": String, "age": Number}]
```

4 Solution

4.1 Overview

Fig. 2. High-level System overview

The algorithm is a variant of the implementation of the algorithm proposed by [2]. It combines *inductive generalization* with enumerative search in a top-down fashion.

First, it proposes an initial hypothesis of the form λ x. e, and infers the type of this expression given the input-output examples.

```
// input
[
  {
    "key": 1,
    "host": "<host1>",
    "name": "<name1>"
```

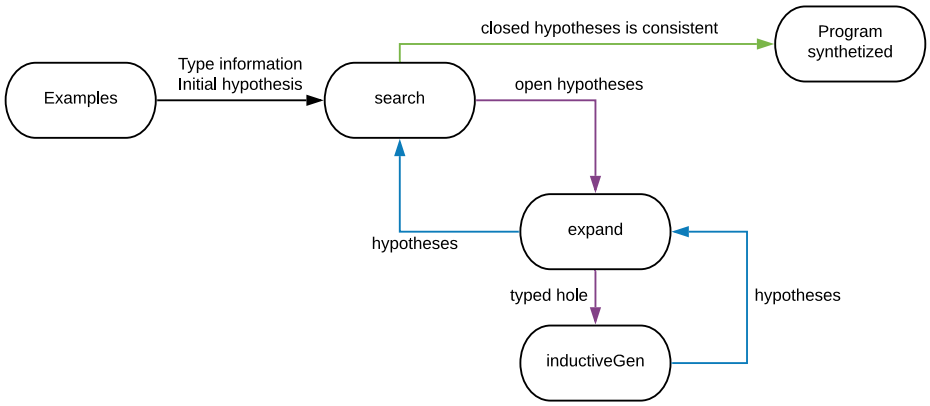

Fig. 3. Synthesis Algorithm Overview (hypotheses imply both open and closed unless noted)

```
  },
  {
    "key": 2,
    "host": "<host2>",
    "name": "<name2>"
  }
]
//output
["<name1>", "<name2>"]
```

We can infer the program to have type `[{key: Number, name: String, host : String}] -> [String]`, x should have type `{key: Number, name: String , host: String}]` and the body e should return `[String]` when evaluated. Next the algorithm proposes a stream of closed and open hypotheses that can "fill" such hole given it's type **and** the type of **x**. the **inductiveGen** procedure generates the following open hypotheses:

1. $map(h_1 : \{key : Number, name : String, host : String\} \rightarrow String)$
2. $[h_1 : String]$
3. $flatten(hole : [[String]])$
4. $hole : [String] + + hole : [String]$

With this, the algorithm searches among the closed ones to find a consistent one, and if it fails, **expands** every open hypothesis and repeats the process.

This process has 3 possible outcomes:

1. it finds a *closed, consistent* hypothesis which is returned as the correct program
2. both the closed and open hypotheses are empty, which means there isn't a program in the dsl that can meet such specification.

3. the set of open and closed hypothesis always grows. In which case we set a bound of maximum iterations.

5 Evaluation

I collected a set of 10 tasks from StackOverflow questions, and adapted them to have simpler values but tried to keep the problems essence unchanged. The results are summarized in Table 2.

Table 2. Benchmarks

Task Description	Running Time	Standar Deviation
identity function	1.342 μs	27.46 ns
get single field	365.5 ns	14.30 ns
get all but one field	2.612 μs	100.0 ns
get a nested field	1.965 μs	177.7 ns
map getting a single field	1.114 μs	81.31 ns
flatten objects in an array	75.74 μs	1.694 μs
flatten an object	14.34 μs	97.04 ns
flatten a deeply nested array	44.83 ms	1.659 ms
appending two projections	392.6 μs	23.16 μs
mapping heterogeneous	3.821 μs	372.6 ns

6 Conclusion

The resulting tool `jsyn` is able efficiently to synthesize a considerable range of json transformations. This is achieved by searching over a simple yet expressive DSL, and uses *inductive generalization* combined with top-down enumeration as it's search strategy.

There are a few different directions for further work. First, the type system is not expressive enough to capture all of json uses, for example mixed arrays, dinamic keys for objects, etc. Extending this should extend the number of possible programs that can be synthesized. Secondly, [2] interleaves another method called *deduction* to find contradictions and prune hypotheses, and to generate new examples. Work can be done to integrate this method too. Thirdly, Adding operations for Numbers, Strings and booleans can help increase the usefulness of the tool. Lastly, making the tool more accesible to users as a web tool can be a good engineering exercise.

References

1. Chaudhuri, A.: Flow: Abstract interpretation of javascript for type checking and beyond. In: Proceedings of the 2016 ACM Workshop on Programming Lan-

guages and Analysis for Security. p. 1. PLAS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2996280, https://doi.org/10.1145/2993600.2996280
2. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (Jun 2015). https://doi.org/10.1145/2813885.2737977, https://doi.org/10.1145/2813885.2737977
3. Fowler, M.: Schemaless data structures. https://martinfowler.com/articles/schemaless/, accessed: 2020-04-30
4. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. In: D'Hondt, T. (ed.) ECOOP 2010 – Object-Oriented Programming. pp. 126–150. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
5. Hidalgo, E.: jsyn. https://github.com/EduardoHi/jsyn, accessed: 2020-04-30
6. jl. https://github.com/chrisdone/jl, accessed: 2020-04-30
7. jq. https://stedolan.github.io/jq/, accessed: 2020-04-30
8. Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: Tejas: Retrofitting type systems for javascript. SIGPLAN Not. **49**(2), 1–16 (Oct 2013). https://doi.org/10.1145/2578856.2508170, https://doi.org/10.1145/2578856.2508170
9. Lieberman, H.: Your wish is my command: Programming by example (01 2001)
10. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. SIGPLAN Not. **50**(6), 619–630 (Jun 2015). https://doi.org/10.1145/2813885.2738007, https://doi.org/10.1145/2813885.2738007
11. Yaghmazadeh, N., Wang, X., Dillig, I.: Automated migration of hierarchical data to relational tables using programming-by-example (11 2017)