`jq`Express: Synthesizing `jq` expressions for faster hacking

VENKATA SAIKIRANPATNAIK BALIVADA, UW-Madison
SAMBHAV SATIJA, UW-Madison

`jq` is an essential CLI utility in most sysadmins' toolbelt. It allows transformation, selection and query of JSON streams from the command line. Effectively using it requires knowledge of its query DSL, which is unintuitive for beginners. We aim to synthesize `jq` query expressions from example input/outputs provided by the user. We build `jqExpress` to achieve this. We build a fresh solver for 2 key reasons: gaining hands-on experience building a solver and exploring enumerative strategies specific to our setting. We walk through our experience of building a solver and evaluate how well it performs in practice.

1 INTRODUCTION

JSON (JavaScript Object Notation) is the defacto encoding for message transmission on the internet. The popularity of JSON for web APIs is easy to understand: browsers natively support JSON for transfer (e.g. `HTML5.Response.json()`). Even outside web client APIs, JSON is widely deployed for systems communication. For example, distributed systems employing consensus algorithms like RAFT can use JSON for encoding consensus steps[1]. Outside message passing, JSON is also used to store data. For e.g, logs[2] are frequently persisted as JSON to be later fed into log-analysis systems. Popular databases (MongoDB[3]) exist with the sole purpose of storing JSON-like content.

The key reason for JSON's popularity is that it is expressive and flexible while being semi-structured. It supports primitives like numeric types, boolean and string. It allows structure in messages by allowing List and Map operations. JSON is flexible by allowing a mix of different types as children of List and Map objects. It is expressive by allowing List and Map objects to recursively store List and Map objects.

As JSON started being used for storing system configurations and logs, among other things, sysadmins have begun requiring tools for manipulating JSON objects on the command line. `jq`[4] is a CLI tool that allows users to transform, select and manipulate JSON streams. `jq` provides the same kind of flexibility for JSON streams that `sed`, `awk` and `grep` provide for unstructured text streams.

Sysadmins need to be a jack of multiple trades, and master of few. Having knowledge of a tool like `jq` is critical. The problem starts, as always, with `jq`'s DSL language. Using `jq` requires learning its DSL which might be an overhead for new sysadmins who need to learn N other tools as well. As a running example, we consider a sysadmin who is building a script for managing the server infrastructure. She needs to get the list of private IPs of all running VMs, per region. Azure CLI returns a huge JSON object for each running VM, and the private IP is embedded deep somewhere inside that object. Ideally, she will need to use `jq` to extract the private IP and will need to learn `jq`'s DSL for this.

[1]https://github.com/peterbourgon/raft/blob/3e45f3d150111fb39d0b962ae51d3816fb170ee5/rpc.go#L14
[2]https://www.velebit.ai/blog/nginx-json-logging/
[3]https://www.mongodb.com/json-and-bson
[4]https://stedolan.github.io/jq/

Authors' addresses: Venkata Saikiranpatnaik Balivada, UW-Madison; Sambhav Satija, UW-Madison.

2023. XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We build `jqExpress` to help such a sysadmin: She makes the query for 1 region and gives Azure's beefy response as an example input to our solver. She then manually extracts the private IPs from the response and gives this as an example output. `jqExpress`then uses this example to generate a `jq` DSL expression that she can directly plug into `jq` and use for the other regions.

We stress the fact that `jq` is just one tool in the list of things a sysadmin needs to learn. Mastery is not always required for all tools, and thus `jqExpress` can obviate the need for learning a highly specific DSL.

In this report, we present how we synthesize `jq` query expressions given example input/output pairs. We also evaluate the correctness, and usefulness, of our solver.

2 BACKGROUND

2.1 jq operators

`jq` has a multitude of operators catering to both new and advanced users. Since we gear our solver towards solving most simple expressions, we limit the number of operators we consider. Specifically, we consider *identity*('`.`'), attribute('`.foo`'), *indexing*('`.[2]`'), *any*('`any`'), *all*('`all`'), *has*('`has(some_key)`'), *sort*('`sort`'), *sortBy*('`sortBy(some_key)`'), *forEach*('`.[] | `'), *groupBy*('`groupBy`') and *select*('`select(.attr == 42)`').

2.2 Magic constants

`jq` parse synthesis, similar to Scythe[5] SQL synthesis, expects predicate constants from the user. Without user specification, range queries are nearly impossible to synthesize. Consider for example, a database administrator wants to find all logs between the January and December but the logs are only available from March to October, the synthesizer has no way to determine user intention to filter logs, let alone the magic constants for the predicates. Therefore, we take predicate constants as input from the user.

2.3 Ranking expressions

We have a limited support for rule-based ranking of `jq` parse expressions in our system. `jqExpress` prefers those expressions where the intermediate results are a superset of the output. Moreover, the system prefers attribute selection over the for each operator when the outputs are equivalent. Finally, we prefer plain transformations over object construction.

3 BUILDING JQEXPRESS

3.1 Interface

The user provides (input JSON, output JSON) pairs as examples to `jqExpress`. Since `jqExpress` eventually enumerates through the entire search space, a single example would be sufficient to find a solution. (That is, if one exists within the maximum depth of our search space). However, providing multiple examples helps the `jqExpress` in two ways: a) a stricter query can be generated in favour of a more generic one (ref §3.7), and b) if examples have differing schema types, `jqExpress` can enumerate faster by pruning the list of operators it considers (§3.3).

In our running example, the user gives Azure's JSON output as `input_1.json`. She then manually reads the JSON file and writes the list of private IP addresses into `output_1.json`.

[5]https://scythe.cs.washington.edu/media/scythe-pldi.pdf

3.2 Defining a Grammar for jq query DSL

jq query expressions vary in complexity. They can be simple, for example, select the second element of a list (.[2]). They can be reasonably complicated, for e.g. given an input list, create a new JSON object with min and max ({smallest: .min, largest: .max}). They can also be highly complex, for example, allowing user-defined functions[6].

We support the jq operators that seem to be most commonly used. Most of these have an arity of 0 or 1. Thus, the query expression often ends up looking like a pipeline instead of a typical standard RTG. The arguments for these operators are either attributes (e.g. key inside a map, or a list's index), or predicates (comparators). Concretely, we define a grammar for a jq query based on the operators we support (Appendix A.1).

3.3 Schema extraction for fun and profit

The above grammar is theoretically sufficient for an enumerative solver. However, we exploit the semi-structured nature of JSON and the typed JSON operators to prune our search space. To motivate this, assume our given input is [1,2,3]. We need not try the '.keys' operator since keys is only supported for objects and not lists.

Given an input example, we parse it to generate a tree-schema of the object. We parse the input into StrSchema, IntSchema, BoolSchema, ListSchema and DictSchema. As an example, {'a': [2,1], 'b': [2,41,12]} is converted to DictSchema(StrSchema, ListSchema(IntSchema)).

Typed production rules: The biggest advantage we gain from the knowledge of the input schema is pruning operators from our search space. Our production rules are typed. (e.g. 'all' or 'any' will only be called for ListSchema(BoolSchema). This is even more powerful because our queries are mostly pipelines. When our solver is trying a candidate operator, it knows the schema of the output, and can only apply relevant operators down the chain. Details of our typed production rules can be found in Appendix A.2. Example of typed rules generated by one schema (ListSchema) is in A.3.

Exploiting heterogenous examples: This extraction is done for all input examples. We intersect all these schemas with each other to get the most restrictive set of production rules at each depth. This is valid pruning because our resultant query needs to work for all input examples.

Attribute selection: JSON makes heavy usage of maps (key/value pairs). Many operators (*e.g.* attribute selection, sort_by) need these keys. Keys in JSON can be any primitive (numeric, bool) or even arbitrary strings. Thus, it is infeasible to brute force for all possible keys (infinite search space). Instead, our DictSchema also stores the keys for this object, and ListSchema stores the length of the list. This allows us to cleverly only use the applicable keys/indices. In the case of multiple input examples, we take the subset of attributes for each peer DictSchema, reducing the search space more if possible.

Constant extraction: While parsing, we also store the constants we find (e.g hardcoded strings and numbers). This lets us use these constants in generating predicate expressions.

3.4 Weighted enumeration

Most of the operators we support are *reductive*. When applied, they reduce the size and structure of their input. This was not intentional. It turns out that most common operators are indeed reductive: in other words, they filter the input, while potentially restructuring the output. Luckily, we can exploit this property in our solver: when we are considering the output of a chain of operators,

[6]https://stedolan.github.io/jq/manual/#Advancedfeatures

we check if the required output is a complete subset of the current intermediate output. If it is not, we reduce the search priority of this branch. It is useful to deprioritize it because if some elements are missing from the intermediate output, with a high probability, we will not get them back. However, we cannot completely prune this branch because we support `all` and `any` which are additive. Moreover, we also support `jq`'s reconstruction feature which is additive (ref. §3.5).

3.5 Support for object reconstruction

`jq` allows the creation of new objects, where each element is composed of multiple `jq` expressions. As an example, the `jq` query '`{smallest: .min, largest: .max}`' will output '`{smallest: 1, largest: 3}`' for the input '`[1,2,3]`'. We need to therefore generate such a query expression. This is different from the rest of our discussion because this does not cleanly fit into our reductive pipeline grammar. For now, we implement this by retrying our search with the additional knowledge of the schema of the example output.

3.6 Algorithm: bottom-up solver

Finally, we introduce our bottom-up enumerative solver which uses the techniques described above.

```python
input_examples = [example["input"] for example in examples]
input_schema = get_schema(input_examples)
spec = Spec(examples, constants)
return bottom_up(spec, input_schema, depth, max_results)
```

Listing 1. Input schema generation

We call `get_schema` to generate the production rules applicable for each depth.

```python
def bottom_up(
    spec: Spec, input_schema: Schema, depth: int, max_results: int
) -> list[str]:
    results = []  # All jq query expressions which match expected behaviour
    worklist = []  # Candidate query expression chains

    # begin working with identity method. score=1, length=0
    heappush(worklist, Work(1, 0, identity(), input_schema))

    while len(worklist) > 0:
        work = heappop(worklist)  # partial expression with most promising score.
        expr, expr_schema = work.expr, work.schema
        if len(expr) >= depth:
            continue
        for op, schema in expr_schema.rules(spec):  # typed rule generation
            next_expr = expr + [op]  # try new op

            # feed into jq to check parity with expected output.
            # get score depending on how many additional fields we have.
            # if we are missing a field, score < 0
            # if jq's output != example output, expr_str is None
            expr_str, score = spec.verify(next_expr)

            if expr_str is not None:
                results.append(expr_str)  # Don't stop at first result.
                if len(results) == max_results:
                    return results
            # Add this expression as a candidate partial expressions.
            heappush(worklist, Work(score, len(next_expr), next_expr, schema))
    if results:
        return results
    else:
        raise OutOfDepth(depth)
```

Listing 2. bottom-up solver

3.7 Ranking of proposed expressions

Our solver returns a list of (< 5) `jq` expressions which work for the given input/output examples. We realized that it is important to not return just the first result because it might not be generic enough for the user to use. As an example, consider the input ‘`{"foo": {"bar": 42}}`’ with the required output ‘42’. Our solver would happily return the parse expression ‘`.[].bar.`’ which is correct for the given example. However, a user might have prefered the more obvious ‘`.foo.bar`’ expression.

One way to fix it is by prioritizing the rules per schema (*e.g.* prefer attribute selection over *forEach* operator). This, however requires usage statistics of `jq` operators which we do not have. While we do make some effort to give some priority to rules based on what we believe is important, this is insufficient. Another way to solve this requires user intervention. They can feed in a counter-example to guide the solver. For example, they can add a child without the `bar` attribute to the input which prevents the forEach operator from being applied. *e.g.* ‘`{"foo": {"bar": 42}, "foo1": {"bar1": 42}}`’

We instead focus on returning multiple candidate expressions to the user and let them pick the one they require.

4 RESULTS

We evaluate `jqExpress` on 2 dimensions: correctness and utility.

Per operator correctness: For each operator we support, we look at its example usage from the `jq` manual[7]. We use that example to validate that our solver is indeed able to search and generate each operator. We have 12 such tests for our operators.

Chaining operators: From the examples in `jq`'s manual, we pick up instances where multiple operators are chained. We use them to validate if our solver can combine operators correctly. We have 6 such tests.

Utility examples: Finally, we consider 3 cases to check if `jqExpress` might be useful in the real world.

(1) We first consider our original example where a sysadmin wants to get the list of private IPs. When she feeds in the large JSON object with superfluous VM information as an input, and the manually extracted private IPs as the output, our solver recommends the following `jq` parse expression for future usage: ‘`.VMs | .[] | .virtualMachine | .network | .privateIpAddresses | .[]`’. This is precisely the `jq` expression a user familiar with `jq`'s DSL would have generated.
(2) We consider one of the top voted `jq` question/answers[8] on StackOverflow which use the operators we support. The proposed query is ‘`.example."sub-example" | .[] | .name`’.

[7]https://stedolan.github.io/jq/manual/#Builtinoperatorsandfunctions
[8]https://stackoverflow.com/a/39228718

This is one of the suggestions our solver generates. The other suggestions are also correct, but more generic.

(3) Finally, we consider the case where a user might need to extract information about a git commit[9] from a heavy commit object. This also tests our object reconstruction along with other operators. Our solver is able to successfully generate '`{ message: .commit | .message, name: .commit | .committer | .name, parents: [.parents | .[] | .html_url]}`'

5 CONCLUSION

In this project, we designed and developed a synthesizer for generating `jq` parse expressions in a tool called `jqExpress`. At a high-level, our system uses a bottom-up enumeration strategy with typed-production rules for effective pruning. We ran our synthesizer on both contrived and real-world examples to find reasonable success in generating the target expressions. Finally, the project was a learning experience to explore various various enumeration strategies.

A APPENDIX

A.1 jq query DSL grammar

jq query DSL grammar:

sep → "|" //jq operator separator/pipeline
S → **T sep S** | **T** // Chain of operators, or single operator
T → **G** | **P** | **B** | **L** // Each op can be classified as G, B, L or P, based on arg type
G → all | any | .[] | keys | sort // jq operators with 0 args
B → group_by(**P**) | sort_by(**P**) // jq operators requiring an attribute arg
L → select(**C**) // jq operators requiring constraint
C → **P** == constant //constraint between attribute and constant
P → <attribute index> //select key/value from map, or element from list

A.2 jqExpress typed rules

```
all → list bool -> bool
any → list bool -> bool
foreach → list 'a -> stream 'a
foreach → dict v -> stream v
group by → list dict -> list (list dict)
keys → dict -> list str
obj index → dict v -> v
select → 'a -> 'a?
sort → list 'a -> list 'a
sort by → list dict -> list dict
```

A.3 ListSchema typed rules

The `ListSchema` stores the Schema of the elements it contains in `self.elem_schema`. Thus, it generates the following rules:

```
def rules(self, spec) -> list[tuple[Operator, Schema]]:
```

[9]https://api.github.com/repos/stedolan/jq/commits?per_page=5

```
    ops = []

    #.[]
    ops.append((ForEach(), self.elem_schema))

    # any, all
    if isinstance(self.elem_schema, BoolSchema) or isinstance(
        self.elem_schema, AnySchema
    ):
        ops.append((All(), BoolSchema()))
        ops.append((Any(), BoolSchema()))

    # group_by, sort_by
    if isinstance(self.elem_schema, DictSchema):
        for index in self.elem_schema.kvs:
            ops.append((GroupBy(ObjectIndex(index)), ListSchema(self)))
            ops.append((SortBy(ObjectIndex(index)), self))

    # sort
    ops.append((Sort(), self))

    return ops
```

Listing 3. Rules generated for a list