json-autotype: Automatic type declaration for JSON input data

[bsd3, data, library, program, tools, type-provider] [Propose Tags]

Generates datatype declarations with Aeson's `Data.Aeson.FromJSON'` instances from a set of example `.json` files.

To get started you need to install the package,

and run `json-autotype` binary on an input `.json` file.

That will generate a new Aeson-based JSON parser.

```
$ json-autotype input.json -o JSONTypes.hs
```

Feel free to tweak the by changing types of the fields

— any field type that is instance of `Data.Aeson.FromJSON'` should work.

You may immediately test the parser by calling it as a script:

```
$ runghc JSONTypes.hs input.json
```

One can now use multiple input files to generate better type description.

Now with Elm code generation support!

(If you want your favourite programming language supported too —

name your price and mail the author.)

See introduction on https://github.com/mgajda/json-autotype for details.'
[Skip to Readme]

Documentation Available

Modules

[Index] [Quick Jump]

Data
- *Aeson*
  - *AutoType*
    - Data.Aeson.AutoType.CodeGen
      - Data.Aeson.AutoType.CodeGen.Elm
      - Data.Aeson.AutoType.CodeGen.ElmFormat
      - Data.Aeson.AutoType.CodeGen.Generic
      - Data.Aeson.AutoType.CodeGen.Haskell
      - Data.Aeson.AutoType.CodeGen.HaskellFormat
    - Data.Aeson.AutoType.Extract
    - Data.Aeson.AutoType.Format
    - Data.Aeson.AutoType.Nested
    - Data.Aeson.AutoType.Pretty
    - Data.Aeson.AutoType.Split
    - Data.Aeson.AutoType.Test
    - Data.Aeson.AutoType.Type
    - Data.Aeson.AutoType.Util

Downloads

- json-autotype-3.1.2.tar.gz [browse] (Cabal source package)
- Package description (as included in the package)

Maintainer's Corner

For package maintainers and hackage trustees

- edit package information

Candidates

- 0.2.0.0, 0.2.1.0, 0.2.1.2, 2.0.0, 3.0.1, 3.0.4, 3.0.5, 3.1.0, 3.1.1, 3.1.2

Versions [RSS] [faq]
0.2.0.0, 0.2.1.0, 0.2.1.1, 0.2.1.2, 0.2.1.3, 0.2.1.4, 0.2.2.0, 0.2.3.0, 0.2.4.0, 0.2.5.0, 0.2.5.1, 0.2.5.2, 0.2.5.3, 0.2.5.4, 0.2.5.6, 0.2.5.7, 0.2.5.8, 0.2.5.9, 0.2.5.10, 0.2.5.11, 0.2.5.12, 0.2.5.13, 0.3, 0.4, 0.5, 1.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.0.6, 1.0.7, 1.0.8, 1.0.9, 1.0.10, 1.0.11, 1.0.12, 1.0.13, 1.0.14, 1.0.15, 1.0.16, 1.0.17, 1.0.18, 1.1.0, 1.1.1, 1.1.2, 2.0.0, 3.0.0, 3.0.1, 3.0.4, 3.0.5, 3.1.0, 3.1.1, **3.1.2** (info)

Change log
changelog.md

Dependencies
aeson (>=1.2.1 && <1.5), base (>=4.9 && <5), bytestring (>=0.9 && <0.11), containers (>=0.3 && <0.7), data-default (==0.7.*), filepath (>=1.3 && <1.5), GenericPretty (==1.2.*), hashable (>=1.2 && <1.4), json-alt, json-autotype, lens (>=4.1 && <4.20), mtl (>=2.1 && <2.3), optparse-applicative (>=0.12 && <1.0), pretty (>=1.1 && <1.3), process (>=1.1 && <1.7), QuickCheck (>=2.4 && <3.0), run-haskell-module, scientific (>=0.3 && <0.5), smallcheck (>=1.0 && <1.2), template-haskell, text (>=1.1 && <1.4), unipl ate (>=1.6.*), unordered-containers (==0.2.*), vector (>=0.9 && <0.13), yaml (>=0.8 && <0.12) [details]

License
BSD-3-Clause

Copyright
Copyright by Migamake `2014-2020

Author
Michal J. Gajda

Maintainer
simons@cryp.to, mjgajda@gmail.com

Category
Data, Tools

Home page
https://github.com/mgajda/json-autotype.git#readme

Bug tracker
https://github.com/mgajda/json-autotype.git/issues

Source repo
head: git clone https://github.com/mgajda/json-autotype.git

Uploaded
by MichalGajda at 2020-04-19T19:25:32Z

Distributions
NixOS:3.1.2

Executables
json-autotype

Downloads
30888 total (71 in the last 30 days)

Rating
2.5 (votes: 3) [estimated by Bayesian average]

Your Rating

Status
Docs uploaded by user
Build status unknown [no reports yet]

Readme for json-autotype-3.1.2

[back to package description]

json-autotype

Takes a JSON format input, and generates automatic Haskell type declarations.

Parser and printer instances are derived using Aeson.

The program uses union type unification to trim output declarations. The types of same attribute tag and similar attribute set, are automatically unified using recognition by attribute set matching. (This option can be optionally turned off, or a set of unified types may be given explicitly.) `:|:` alternatives (similar to `Either`) are used to assure that all JSON inputs seen in example input file are handled correctly.

I should probably write a short paper to explain the methodology.

pipeline — passed | hackage v3.1.2 | dependencies outdated | docker build automated | microbadger no longer available

Details on official releases are on Hackage We currently support code generation to Haskell, and Elm.

Please volunteer help or provide financial support, if you want your favourite language supported too! Expression of interest in particular feature may be filed as GitHub issue.

USAGE:

After installing with `cabal install json-autotype`, you might generate stub code for the parser:

```
    json-autotype input1.json ... inputN.json -o MyFormat.hs
```

Then you might test the parser by running it on an input file:

```
    runghc MyFormat.hs input.json
```

At this point you may see data structure generated automatically for you. The more input files you give to the inference engine `json-autotype`, the more precise type description will be.

Algorithm will also suggest which types look similar, based on a set of attribute names, and unify them unless specifically instructed otherwise.

The goal of this program is to make it easy for users of big JSON APIs to generate entries from example data.

Occasionally you might find a valid JSON for which `json-autotype` doesn't generate a correct parser. You may either edit the resulting file *and* send it to the author as a test case for future release.

Patches and suggestions are welcome.

You can run with Docker:

```
docker run -it migamake/json-autotype
```

EXAMPLES:

The most simple example:

```
{
    "colorsArray":[{
            "colorName":"red",
            "hexValue":"#f00"
        },
        {
            "colorName":"green",
            "hexValue":"#0f0"
        },
        {
            "colorName":"blue",
            "hexValue":"#00f"
        }
    ]
}
```

It will produce the module with the following datatypes and TH calls for JSON parser derivations:

```
data ColorsArray = ColorsArray {
    colorsArrayHexValue    :: Text,
    colorsArrayColorName :: Text
  } deriving (Show,Eq)

data TopLevel = TopLevel {
    topLevelColorsArray :: ColorsArray
  } deriving (Show,Eq)
```

Note that attribute names match the names of JSON dictionary keys.

Another example with ambiguous types:

```
{
    "parameter":[{
            "parameterName":"apiVersion",
            "parameterValue":1
        },
        {
            "parameterName":"failOnWarnings",
            "parameterValue":false
        },
        {
            "parameterName":"caller",
            "parameterValue":"site API"
        }]
}
```

It will produce quite intuitive result (plus extra parentheses, and class derivations):

```
data Parameter = Parameter {
    parameterParameterValue :: Bool :|: Int :|: Text,
    parameterParameterName :: Text
  }

data TopLevel = TopLevel {
    topLevelParameter :: Parameter
  }
```

Real-world use case examples are provided in the package source repository.

Methodology:

1. JSON-Autotype uses its own union type system to derive types from JSON documents as the first step.
2. Then it finds all those records that have 90% of the same key names, and suggest them as similar enough to merit treating as instances of the same type. (Note that this is optional, and can be tuned manually.)
3. Last step is to derive unique-ish type names - we currently do it by concatenating the name of the container and name of the key. (Please open PR, if you want something fancy about that - initial version used just key name, when it was unique.)
4. Finally it generates Haskell or Elm code for the type.

Combination of robust *union type system*, and heuristic makes this system extremely reliable. Main test is QuickCheck-based generation of random JSON documents, and checking that they are all correctly parsed by resulting parser.

More details are described in Haskell.SG meetup presentation.

Other approaches:

- There is a TypeScript type provider, and PLDI 2016 paper on solving this problem using *preferred type shapes* instead of union types. One can think about it as a alternative theory that gives very similar results, with more complicated exposition. It also does not tackle the problem of tagged records. It also does not attempt to *guess* unification candidates in order to reduce type complexity.
- There was a json-sampler that allows to make simpler data structure from JSON examples, but doesn't seem to perform unification, nor is it suitable for big APIs.
- PADS project is another attempt to automatically infer types to treat *arbitrary* data formats (not just JSON). It mixes type declarations, with parsing/printing information in order to have a consistent view of both. It does not handle automatic type inference though.
- JSON Schema generator uses .NET types to generate JSON Schema instead (in opposite direction.) Similar schema generation is used here
- Microsoft Developer Network advocates use of Data Contracts instead to constrain possible input data.
- QuickType uses Markov chains heuristic instead of theory