# Type Interface Synthesis for Unstructured Data: json-synthesizer

Steven Yuan

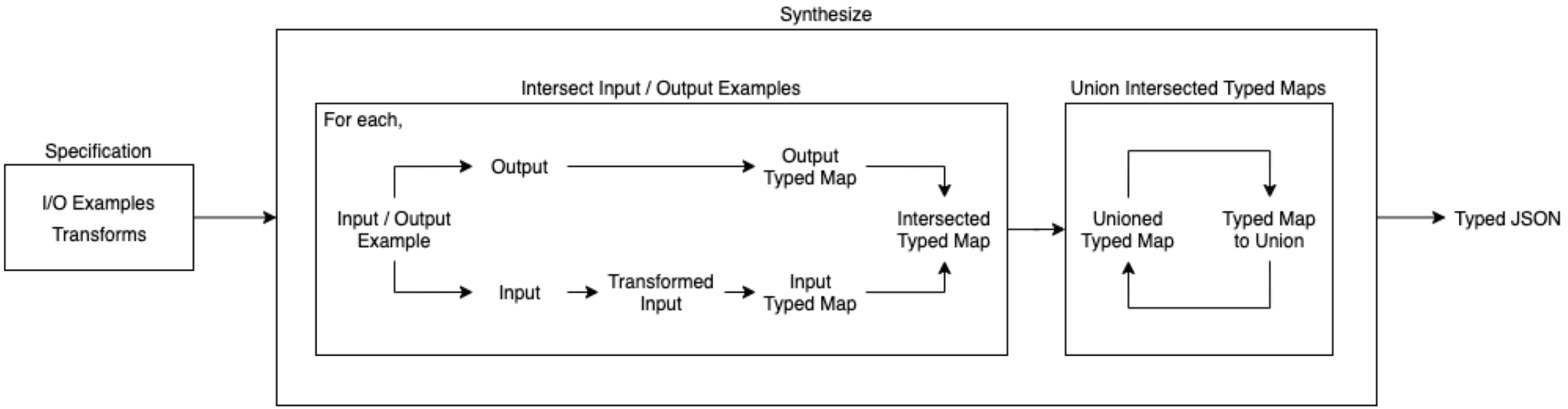
CS515 Programming Languages and Compilers I

Fall 2020

# Overview

- Modern technologies use Unstructured Data (JSON, XML, etc.) for a wide variety of use cases (Databases, APIs, etc.).
- For developers, managing and cleaning data from multiple sources manually is both error-prone and tedious.
- Even if the overall process on the data is independent from the source, custom code needs to be written for each source.
- json-synthesizer uses a Modified Version Space Algebra to synthesize a Generic Type Interface given Input/Output Examples and Transforms

# Architecture



# Process

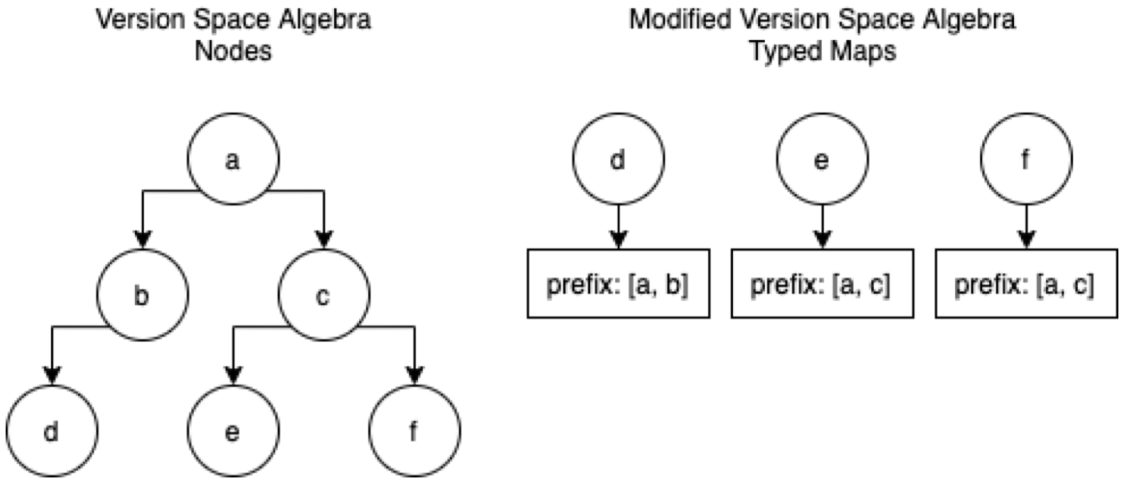
- Given a Specification with Input/Output Examples and Transforms
- For each Input/Output Example, learn a Typed Map
  - Apply relevant Transforms to the input based on Source Tags
  - Convert the transformed input to a Typed Map
  - Convert the output example to a Typed Map
  - Restructure and Intersect the Input Typed Map and Output Typed Map
- Reduce with Union the Typed Maps to a Single Typed Map
- Produce Typed JSON from the Final Typed Map

# Specification

- Specification: {
  - inputOutputExamples: List of Input/Output Examples
  - Transforms: List of Transforms
- }
- Input/Output Example: {
  - source: List of Tags to denote here the Input is from
  - input: JSON from the Source
  - output: Desired Restructured JSON considering Transforms
- }
- Transform: {
  - source: Single Tag that matches to Input/Output Example source
  - transform: JSON -> JSON function
- }

# Version Space Algebra Synthesis

| **Version Space Algebra** | **Modified Version Space Algebra** |
|---|---|
| Node | Typed Map |
| Values/Children | Types |
| Learn | Restructure |
| Intersect | Intersect |
| Union | Union |



# Types

- Each Type acts a Struct that denotes possible types for a value:
- Type {
  - array: Type | null
  - object: Typed Map | null
  - Types: Set of Single Value Types | null
- }
- Single Value Types: string, number, boolean, and null

# Typed Maps

- Typed Maps represent Unstructured Data for Keys with Concrete Values (Arrays, Single Value Types)
- Instead of mapping the recursive structure of JSON, use dot “.” paths to denote structure
- Typed Map = Map<key, {
  - prefix: List of Keys in dot “.” path
  - type: Type
- }>
- Typed Maps recover structure when serialized to Typed JSON

# Limitations

- Duplicate Key Restriction
  - Having duplicate keys that can be reached with a dot “.” path is illegal
  - No heuristic in VSA operations to determine which key to use
- Loss of Semantic Keys
  - json-synthesizer only see keys as structural units, not semantic
  - Predefined schemas will lose information if not properly transformed
- Limited Type Range
  - Semantic keys can sometimes denote complex types
  - However, only types in the JSON specification are in Type

# Example: DynamoDB JSON

input:

```
{
  'Surname': { 'S': 'Smith' },
  'Members': {
    'L': [
      {
        'M': {
          'Name': { 'S': 'John' },
          'Age': { 'N': '34' },
          'Education': { 'S': 'GED' }
        }
      }
    ]
  },
  'Address': { 'S': '123 Mulberry Lane' },
  'Count': { 'N': '1' }
}
```

output:

```
{
  Surname: 'Smith',
  Members: [
    { Name: 'John' },
  ],
  Count: 1
}
```

transform:

unmarshallDynamoJson

Unwraps Data Types, such as:

```
'Surname': {
  'S': 'Smith'
}
```

into

```
'Surname': 'Smith'
```

Synthesized Typed JSON:

```
{
    "Surname": {
        "array": null,
        "object": null,
        "types": [
            "string"
        ]
    },
    "Members": {
        "array": {
            "array": null,
            "object": {
                "Name": {
                    "array": null,
                    "object": null,
/* ... Too long to show on slide ... */
```

# Example: Generic JSON

input:

```
{
  ‘Pets’: [
    {
      ‘Name’: ‘Fido’,
      ‘Type’: ‘Dog’,
      ‘Age’: 7,
      ‘Rescued’: false
    },
    {
      ‘Name’: ‘Polly’,
      ‘Type’: ‘Parrot’,
      ‘Age’: 2,
      ‘Color’: ‘Red’
    }
  ]
}
```

output:

```
{
  Other: {
    Pets: [
      {
        ‘Name’: ‘Name’,
        ‘Type: ‘Type’,
        ‘Age’: 0,
        ‘Rescued’: true
      }
    ]
  }
}
```

transform: None

Synthesized Typed JSON:

```
{
    "Other": {
        "array": null,
        "object": {
            "Pets": {
                "array": {
                    "array": null,
                    "object": {
                        "Name": {
                            "array": null,
                            "object": null,
                            "types": [
                                "string"
                            ]
                        },
/* ... Too long to show on slide ... */
```

# Example: Multiple Input/Output Examples

- Combine the Input/Output Examples Lists and Transforms Lists from:
  - DynamoDB JSON
  - Generic JSON
- As long as the duplicate key restriction is not broken, a new Type Interface will be synthesized

```
Synthesized Typed JSON:
{
    "Surname": {
        "array": null,
        "object": null,
        "types": [
            "string"
        ]
    },
    "Members": {
        "array": {
            "array": null,
            "object": {
                "Name": {
                    "array": null,
                    "object": null,
/* ... Too long to show on slide ... */
```

# Evaluation

- Test Suite
  - Transforms
  - Restructure Typed Maps
  - Intersect Types
  - Intersect Typed Maps
  - Union Types
  - Union Typed Maps
- Evaluation of Unit Tests guarantees correctness for Generic JSON

# Related Work

- “Version Space Algebra and its Application to Programming by Demonstration” by Tessa Lau, Pedro Domingos, Daniel S. Weld
  - Canonical Paper on Version Space Algebra Synthesis
  - Explore Complex Program Spaces via Operations
- “Error-Tolerant Version Space Algebra” by Eugene R. Creswick, Aaron M. Novstrup
  - “similar” heuristic operation may be useful for duplicate key restriction
  - Learning does not improve with repeated demonstration

# Conclusion

- json-synthesizer can synthesize Generic Type Interfaces from Input/Output Examples and Transforms
- Representation-based search maps well to unstructured data
- Modified Version Space Algebra with Typed Maps removes the complexity of tree structures
- Implementation can be thoroughly tested to guarantee correctness