Rhyme: A Data-Centric Multi-Paradigm Query Language based on Functional Logic Metaprogramming

Supun Abeysinghe[0000−0001−6054−2432] and Tiark Rompf[0000−0002−2068−3238]

Purdue University, West Lafayette IN 47906, USA
{tabeysin,tiark}@purdue.com

Abstract. We present Rhyme, a declarative multi-paradigm query language designed for querying and transforming nested structures such as JSON, tensors, and beyond. Rhyme is designed to be multi-paradigm from ground-up allowing it to seamlessly accommodate typical data processing operations–ranging from aggregations and group-bys to joins–while also having the versatility to express diverse computations like tensor expressions (à la einops) and declaratively express visualizations (e.g., visualizing query outputs with tables, charts, and so on). Rhyme generates optimized JavaScript code for queries by constructing an intermediate representation that implicitly captures the program structure via dependencies. This paper presents a system description of Rhyme implementation while highlighting key design decisions and various use cases covered by Rhyme.

1 Introduction

This paper introduces the design of Rhyme, a new declarative multi-paradigm query language tailored for querying and transforming nested structures, encompassing formats such as JSON, tensors, and more. Rhyme draws inspiration from an array of existing approaches, including query languages such as GraphQL [18], JQ [3], XQuery [4], logic programming languages like Datalog [10], and recent functional logic programming languages like Verse [12]. Rhyme is implemented in JavaScript and currently available as an open source Node.js package[1]. Below are the main defining characteristics of Rhyme:

- **Data-centric query language:** Focused explicitly on extracting information and transforming data, not general-purpose programming.
- **Multi-paradigm:** Still, flexible enough to support a wide range of typical data processing operations (e.g., SQL/DataFrames), tensor expressions, visualization, etc., in a single language.
- **Functional:** Not based on relations as the core abstraction but on functions, including representing data as materialized functions (i.e., objects mapping keys to values).

[1] Available at `https://rhyme-lang.github.io/`

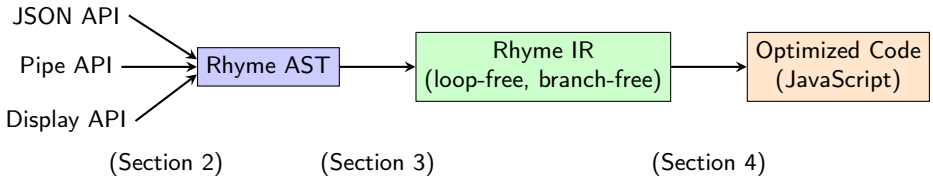


Fig. 1: The end-to-end workflow of Rhyme. Rhyme provides multiple APIs for different types of workloads that targets the common Rhyme AST which serves as an entry point. The AST gets transformed to an IR that implicitly captures program structure (i.e., no loops or branches). Optimized JavaScript code is then generated from this IR.

- **Logic:** Using logic variables and unification to express joins, aggregations, and other forms of iteration.
- **Metaprogramming:** Relying on an expressive host language to compose query fragments and run queries (using functional/Object-Oriented APIs, as well as quasiquotation syntax).

Rhyme is designed to be multi-paradigm from the ground up, seamlessly accommodating standard data processing operations like aggregations, group-bys, joins, and others, along with paradigms beyond conventional data-processing operations. These include the expression of various types of computations, such as tensor operations (akin to einops [24]), and the ability to declaratively express visualizations (e.g., a table summarizing query outputs).

This facilitates the expression of diverse workloads; for example, a complex data processing pipeline mixing tasks such as group-based aggregations intertwined with tensor computations (e.g., incorporating a pre-trained model), followed by the creation of visualizations, such as dashboards—all achieved through a unified query language. This bears significance on two fronts. Firstly, in terms of programmer productivity, end-users can leverage a single, unified query language to express the entirety of their workload logic, eliminating the need for multiple domain-specific languages. Secondly, concerning performance, the unified system seamlessly manages all workload paradigms, mitigating the performance overhead typically incurred at system boundaries when employing various domain-specific systems—a pressing problem in practical scenarios [9, 23].

Figure 1 illustrates the end-to-end workflow of Rhyme. Queries can be expressed through different frontend APIs (as detailed in Section 2), where all these APIs target a common AST representation, serving as the entry point to the system. Subsequently, this AST representation serves as the basis for deriving an intermediate representation (IR) that encapsulates the query logic. A key characteristic of this IR is that it does not explicitly capture the program structure corresponding to the query. Instead, the program structure is implicitly captured through dependencies, making it easier to perform optimizations (as detailed in Section 3). Then, the IR is transformed into optimized JavaScript code, incorporating various optimizations

Operation	Query	Result
Sample Dataset	`[ {key: A, value: 10}, {key: B, value: 20},` `  {key: B, value: 15}, {key: A, value: 30} ]`	-

Simple Indexing	`// directly indexing into a sequence of data`
	`data.0.value`
	`data.2.key`
		`10`
		`B`

Creating structures	`// creating structures from queries`
	`// (can compose arbitrarily)`
	`{ first : data.0.key }`
		`{ first : 10}`

Iterating (using ∗)	`// iterating data and collecting into an array`
	`[data.∗.value]`
	`// iterators are logic variables (named ∗A, ∗B, ...)`
	`[data.∗A.key]`
		`[10,20,15,30]`
		`[A,B,B,A]`

Aggregations	`// computing aggregates`
	`sum(data.∗.value)`
	`// creating a new object from a query`
	`{ total : sum(data.∗.value) }`
		`75`
		`{ total: 75 }`

Group-bys, Joins	`// group-by: expressed by having ∗, ∗A, ... as key`
	`// compute per-key sum of values`
	`{ data.∗.key : sum(data.∗.value) }`
	`// compute per-key relative sum`
	`// sum(data.∗A.value):`
	`//   nested within data.∗A.key -> per-group sum`
	`// sum(data.∗B.value):`
	`//   not nested within another ∗B key -> total sum`
	`{`
	`  data.∗A.key :`
	`  sum(data.∗A.value) / sum(data.∗B.value)`
	`}`
		`{ A: 40, B: 35 }`
		`{`
		`  A: 0.53,`
		`  B: 0.47`
		`}`

Table 1: Basic query operators of Rhyme JSON API. Full API documentation is available at `https://rhyme-lang.github.io/docs/frontends/json/`.

The remainder of the paper is structured into sections that delve into each component of Rhyme, as depicted in Figure 1. Section 2 outlines the various frontends provided by Rhyme, catering to the expression of diverse workload types. We offer several alternative APIs designed to express Rhyme queries, each tailored for specific use cases. Moving forward, Section 3 delves into the design of Rhyme AST, Rhyme IR, and the process of constructing the IR from the AST, how an optimized program structure is derived from this loop-free, branch-free representation and elucidate the generation of optimized JavaScript code for a given query.

The foundational concepts of Rhyme were initially introduced in [8]. This paper takes a more in-depth exploration to the design and implementation of Rhyme frontend, providing comprehensive discussions on recent advancements in some of the APIs, including surface syntax (e.g., quasiquotations, etc.), and offering extensive insights into the graphics API.

2 Rhyme Front End

As illustrated in Figure 1, Rhyme offers users various frontends to express their queries. In this section, we will explore each of these APIs, demonstrating their utility across diverse use cases.

2.1 JSON API

Given the thorough discussion of this API in previous work [8], we refrain from offering an exhaustive discussion here. Nonetheless, we add a summary of key components for the sake of completeness.

Table 1 provides a summary of the primary operators in the JSON API. Most operators are self-explanatory and intuitive in nature. It is worth noting that Rhyme also supports user-defined functions (UDFs), joins, and various typical data processing operations, other than the ones mentioned in the table. One noteworthy aspect of Rhyme lies in its approach to expressing group-bys and the contextual evaluation semantics of expressions like `sum(data.*A.value)`. Specifically, when nested within a `{ data.*A.key : ...`, the `sum(data.*A.value)` corresponds to a per-group sum; otherwise, it represents a total sum (note the use of the same `*A`).

The following code snippet illustrates the utilization of the Rhyme JSON API to query the DBLP REST API and compute various details about past FLOPS conference publications. This snippet is a fully executable JavaScript program, assuming the Rhyme library is appropriately set up and imported[2].

```
let url =
     "https://dblp.org/search/publ/api?q=stream:streams/conf/flops:&h=1000&format=json"

// fetch the data from the REST endpoint and query
fetch(url).then(p => p.json()).then(data => {
  // Query1: produces all papers by year
  let query1 = {
    "data.result.hits.hit.*.info.year": ["data.result.hits.hit.*.info.title"]
  }
  let res = api.query(query1)
  display(res({data}))
  // result: {1998: [...FLOPS papers of 1998 ...], 1999: [ ... ], ...}

  // Query2: compute number of papers per year
  let query2 = {
    "data.result.hits.hit.*.info.year": api.count("data.result.hits.hit.*.info.title")
  }
  res = api.query(query2)
  display(res({data}))
  // result: {1998: 16, 1999: 24, 2001: 25, ...}

  // Query3: For all authors, list years they published
  let query3 = {
    // need to build union of single-author and
    // multi-author papers
    "data.result.hits.hit.*.info.authors.*A.*B.text": ["data.result.hits.hit.*.info.year"],
    "data.result.hits.hit.*.info.authors.*A.text": ["data.result.hits.hit.*.info.year"],
  }
  res = api.query(query3)
  display(res({data}))
  // result: {"author A": [2022, 2012, 2010, ...], "author B": [2016, 2012, ...], ...}
})
```

As depicted in Figure 1, similar to the other APIs, the JSON API follows the process of constructing the Rhyme AST, which is subsequently transformed into Rhyme IR. Moreover, since all of these APIs are implemented in the same host

[2] As detailed in `https://github.com/rhyme-lang/rhyme?tab=readme-ov-file#using-in-the-browserfrontend`

language, JavaScript, they can be seamlessly mixed and matched as the users see fit. This IR is then further processed to generate optimized JavaScript code, as detailed in Section 3.

2.2 Pipe API

The JSON API enables expressing queries in a format resembling the expected output structure, akin to approaches like GraphQL. However, there are scenarios where it is more intuitive to formulate computations as a sequence of transformation steps on the input(s). Rhyme offers the Pipe API to accommodate such workloads.

Rhyme offers a textual API based on JavaScript's template literals for this purpose. Template literals enable string interpolation with embedded expressions and allow the addition of custom desugaring logic. Rhyme utilizes the rh`...` template to express queries through this API. To explain how this API functions, let us take a simple example that computes the sum of values expressed using the JSON API: `api.sum("data.*.value")`. This same query can be written using our surface syntax in the following ways:

```
rh`sum(data.*.value)`
rh`sum data.*.value`
rh`data | sum(.*.value)`
rh`data | .* | sum(.value)`
rh`data | .* | .value | sum`
```

All of the above queries are equivalent and are parsed into the same AST as the original query. This approach offers a more intuitive and straightforward means of expressing the query as a series of transformation steps applied to the inputs. Under the hood, each transformation step (i.e., operators appear sequentially after pipes) gets desugared to operators with holes. For a call expression like `f(a)` (e.g., `sum(.value)` above) or `a|f` (e.g., `.value|sum` above), we explicitly desugar '`f`' with the expectation that it is a function. This process is akin to bidirectional typing [16], i.e., potentially transforming code to conform to an expected type.

In this process, we examine whether any expressions contain holes—e.g., '`.value`' is incomplete, conceptually featuring a hole on the left, and is desugared to ☐`.value`. Similarly, `sum(.value)` is parsed as `sum(`☐`.value)`. Moreover, if any intermediate operations in the sequence do not contain holes, such as when UDFs are applied as part of the transformation (e.g., `splitPipe` in the example query later in this section), the expression is treated as a reference to a function and is eta-expanded by inserting a hole on the right for application (e.g., `splitPipe(`☐`)`). In our implementation, we treat these terms with holes as anonymous functions; for instance, `sum(`☐`.value)` becomes `x => sum(x.value)`.

There may be situations where a single operator involves multiple holes. In such cases, we have a choice: whether to consider all of them as references to the same argument (`x => ...`) or treat them as arguments to a multi-argument function (`(x, y)=> ...`). Presently, Rhyme exclusively handles single-argument functions (which is the situation with pipes like `a | f`), but it does support currying, allowing for repeated applications (as in `b | f(a)`).

Moreover, this surface language provides a way of using quasi-quotations via `${...}`, for example to define and directly call user-defined functions. For instance, consider a case where we want to compute the average using a UDF. The average UDF can be expressed using `sum` and `count`.

```
let avg = x => rh`sum(${x}) / count(${x})`
// computing average of data.*.value (both queries compute the avg)
rh`${avg}(data.*.value)`
rh`data.*.value | ${avg}`
```

To demonstrate the effectiveness of this API, we take an example query from a previous work [8], which is originally taken from Advent of Code 2022 [1]. The input is a sequence of numbers separated by pipes into chunks where each chunk contains multiple comma-separated numbers. The task is to compute the sum of each chunk and find the maximum sum across all chunks.

```
let input = '100,200,300|400|500,600|700,800,900|1000' // sample input
// some UDFs for parsing the data
let udf = {
  'splitPipe' : x => x.split('|'),
  'splitComma': x => x.split(','),
  'toNum'     : x => Number(x)
}
```

As shown in [8], this query can be expressed using Rhyme's JSON API, as depicted on the left. Alternatively, the same query can be expressed in a nice, intuitive manner using the pipe-based surface language, as illustrated on the right.

```
// JSON API
let query = max(get({
'*chunk': sum(
  apply('udf.toNum',
    get(apply('udf.splitComma',
      get(apply('udf.splitPipe', '.input'),
            '*chunk')),
      '*line')))
},'*'))
```

```
// Pipe API
//  Produce output by applying a sequence of
//  transformations to input
let query =
  rh`.input | udf.splitChunk | .*chunk
    | udf.splitComma | .*line | udf.toNum
    | sum | group *chunk | .* | max`
```

While both queries generate the same AST and yield the same expected result, the query on the right is better suited for these types of tasks. It offers a cleaner and more intuitive approach to expressing the query as a sequence of transformations on the input. This stands in contrast to the JSON API, which requires the query to be written in a way that mirrors the output structure.

2.3 Visualization API

Many data processing workloads demand some form of visualization as the final output. Traditionally, the implementation of these visualizations is separate from the data processing logic. However, Rhyme allows users to integrate the logic for visualization directly within the same language. Since Rhyme and all its existing APIs are implemented in JavaScript, end users can effortlessly import the corresponding JavaScript file into their browser/HTML and easily create visualizations.

Rhyme introduces the `"$display"` keyword for specifying visualizations. One type of visualization Rhyme supports is the declarative specification of SVG

drawings. To create SVG drawings, users can employ `"$display": "dom"` and `"type": "svg:<svg_shape>"`, where `<svg_shape>` represents any available SVG shape, such as ellipse, polyline, rect, polygon, etc. The properties associated with each shape can be passed using the `props` key. For example, the query below draws a rectangle, incorporating features like colors, rounded edges, and additional attributes by passing the corresponding properties to `props` (visualization is shown on the right).

```
{
  "$display": "dom", type: "svg:rect",
  props: {
    x: 100, y: -50, width: 50, height: 50, rx: 10, ry: 10,
    fill: "lightblue", stroke: "hotpink", "stroke-width": 5,
    opacity: 0.7, transform: "rotate(45)"
  }
}
```

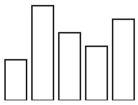
It is important to highlight that this graphical drawing capability seamlessly integrates with Rhyme's other APIs. To illustrate its utility, let us consider a scenario where we aim to visualize some sample data. While we currently use synthetic data for illustration purposes, it is crucial to note that this data could be sourced from another Rhyme query that processes and analyzes real-world data. Shown below is some sample data.

```
let data = [{x:20,y:70},{x:40,y:30},{x:60,y:50},{x:80,y:60},{x:100,y:40}]
```

Suppose we aim to represent this data through three distinct types of charts: line, bar, and points. Below, we show the queries for generating these three drawings, accompanied by their respective outputs.

```
let line = {
  "$display": "dom", type: "svg:polyline",
  props: {
    points: rh`${join}((data.*.x + ',' + data.*.y), ' ')`,
    stroke: "black", fill:"none" }
}
```

```
let bars = {
  "$display": "dom", type: "svg:rect",
  props: {
    width: "16px", height: rh`100 - data.*.y`,
    x: rh`data.*.x - 8`, y: rh`data.*.y`,
    stroke: "black", fill: "none" },
}
```



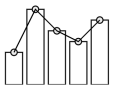
```
let points = {
  "$display": "dom", type: "svg:ellipse",
  props: {
    rx: "3px", ry: "3px",
    cx: "data.*.x", cy: "data.*.y",
    stroke: "black", fill: "#EEE" },
}
```

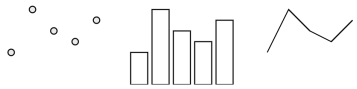
Multiple SVG graphics can be overlapped by using `"type": "svg:svg"` and providing the corresponding queries as an array to the `"children"` key. Shown below is a query that draws a data-dependent graphic by overlapping multiple `"svg:elipse"` drawings.

```
let data = []
for (let j = 0; j < 360; j += 20) data.push(j)
let query = {
  "$display": "dom", type: "svg:svg",
  props: { width: "300px", height: "200px" },
  children: [{
    "$display": "dom", type: "svg:ellipse",
    props: { rx: "40px", ry: "15px", cx: "200px", cy: "100px",
      fill: rh`'hsl(' + data.* + ' 90% 50%)'`,
      'fill-opacity': "70%",
      transform: rh`'rotate(-' + data.* + ' 150 100)'` },
  }]
}
```

Another `"$display"` option available for Rhyme is `"select"`. This option adds a set of buttons at the top to toggle between different sets of visualizations. For instance, suppose we want to visualize the above three graphics in both an overlapped and a side-by-side arrangement. This can be achieved using the select option. The corresponding query is shown below. The images on the right illustrate both scenarios, each activated by clicking the corresponding button.

```
{
  "$display": "select",
  data: {
    "All in one": {
      "$display": "dom", type: "svg:svg",
      props: { width: "300px", height: "100px" },
      children: [bars, line, points]
    },
    "Side by side": {
      "$display": "dom", type: "div",
      children: [{
        "$display": "dom", type: "svg:svg",
        props: { width: "120px", height: "100px" },
        children: [points]
      },{
        "$display": "dom", type: "svg:svg",
        props: { width: "120px", height: "100px" },
        children: [line]
      },{
        "$display": "dom", type: "svg:svg",
        props: { width: "120px", height: "100px" },
        children: [bars]
      }]
    },
  }
}
```





Rhyme offers additional display options beyond the ones discussed earlier. Two particularly useful options are `"table"` and `"bar"`, designed for creating tables and bar charts from data, respectively. While the complete query is not included here, the following (excerpt) visualization shows a table mixed with bar charts. This visualization was generated by computing summary statistics from a sample warehouse dataset using a Rhyme query and specifying all the visualization-related logic entirely within Rhyme.

	Quantity	Bar Chart	Percent Total
Total	1210		100 %
San Jose	650		53 %
iPhone	300		24 %
7	50		4 %
6s	100		8 %
X	150		12 %
Samsung	350		28 %
Galaxy S	200		16 %
Note 8	150		12 %
San Francisco	560		46 %
iPhone	260		21 %
7	10		0 %
6s	50		4 %

3 Rhyme Backend

This section explores the end-to-end process of transforming queries into optimized JavaScript code. This translation happens in three steps. As we saw in Figure 1, all the frontend API directly targets Rhyme AST, which serves as the entry point to the system. This AST representation is further transformed into Rhyme IR, which implicitly captures the program structure. Finally, this Rhyme IR is transformed into optimized JavaScript code. To enhance comprehension, we will employ the following query as a running example throughout this section. As we saw in Table 1 (second example query for group-by), this query computes per-key relative sum for all the keys.

```
{
  data.*A.key: sum(data.*A.value) / sum(data.*B.value)
}
```

3.1 AST

The Rhyme AST is represented using JSON objects and closely resembles the JSON API. The keys corresponding to implicit group-by clauses (for example `{ data.*.key: ...}`) remain unchanged in the AST representation. Other parts corresponding to query operators are transformed into nested JSON objects. For instance, reduction operators like `sum`, `max`, `min`, etc., are transformed into objects containing the aggregate name and the parameters passed to the aggregate. For example, `sum(data..value)` will be transformed into `{ agg: 'sum', param: 'data..value'}`. Similarly, for other operators that are not reductions (i.e., stateless ones), an object with the operator type (referred to as `path`) and parameters is created. The AST representation for the running example query is illustrated below.

```
{
  "data.*.key": {
    path: 'div',
    param: [
      {agg: 'sum', param: 'data.*A.val'},
      {agg: 'sum', param: 'data.*B.val'}
    ]
  }
}
```

The translation from different frontends to the AST representation is relatively straightforward and, therefore, not extensively discussed. For example, in the JSON API, we define functions for each stateless and stateful operator, such as `sum`, `max`, `min`, `plus`, and so on, to generate the corresponding AST fragment. Below, we provide the logic for `sum` and `plus`:

```
api["sum"] = (e) => ({
  agg: "sum",
  param: e
})
```

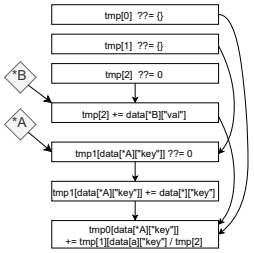
```
api["get"] = (e1, e2) => ({
  path: "get",
  param: [e1, e2]
})
```

The same set of AST fragment building functions is utilized by the rest of the frontends. For example, in the template literals-based frontend discussed in Section 2.2, the corresponding operators are desugared by invoking the relevant `api` functions.

3.2 Structure of the Rhyme IR

The Rhyme IR is comprised of two types of operators: *generators* and *assignments*. As the name implies, generators are instructions responsible for iterating values from data sources, such as an array of JSON objects. In contrast, assignments correspond to instructions that execute computations and assign (partial) results to intermediate state variables. This approach, employing multiple intermediate state variables to compute the final result, draws inspiration from previous work on generating triggers for incremental execution [7, 20].

In our ongoing example query, Rhyme generates three temporaries—`tmp[0]`, `tmp[1]`, and `tmp[2]`—which correspond to the final result object, per-key aggregate object, and total aggregate, respectively. The presence of `*A` and `*B` in the query signifies two generators. The IR for this query is presented below, where the assignment instructions on temporaries are self-explanatory.



A defining feature of the Rhyme IR lies in its avoidance of explicitly storing the program structure (inspired by prior work [13, 14]). Taking the ongoing query as an example, the query indicates that the `*B` generator should be nested within `*A`, forming a nested loop structure. However, rather than strictly enforcing this structure within the IR, Rhyme stores dependencies that implicitly capture the query's structure.

Specifically, the use of a generator within an assignment instruction gives rise to a generator-assignment dependency, while other assignment-assignment dependencies typically represent data dependencies. As we delve into Section 3.3, we will explore how this dependency-based representation facilitates certain optimizations, making tasks such as loop hoisting more straightforward.

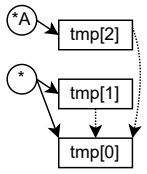
In the current Rhyme implementation, dependencies are tracked at the level of temporaries, such as `tmp[0]`, `tmp[1]`, and so on. To achieve this, each assignment is assigned a write rank, ensuring a specific order during code generation. While this systematic approach guarantees a sequential order of writes, it may introduce imprecise dependencies in certain scenarios, potentially resulting in suboptimal code. This is because automatically enforcing a serial order for writes might not accurately capture their true dependencies. An alternative approach involves treating each write rank more like a static-single assignment (SSA)-style variable, allowing for a finer-grained tracking of dependencies. This approach provides a more nuanced representation of dependencies, potentially improving the precision of the generated code.

3.3 Code Generation

Following the construction of the IR, the next step involves generating the final optimized JavaScript code for the query based on this IR. Given that our IR does not store explicit program structure, reconstructing the program's architecture requires an analysis of dependencies. To accomplish this, we perform multiple analysis passes over the IR.

The initial analysis focuses on determining the placement of intermediate temporaries with respect to loops and other temporaries. Specifically, this pass computes `tmpInsideLoop` and `tmpAfterTmp`, which track which temporaries should be scheduled inside particular loops and which temporaries should be scheduled after certain other temporaries, respectively. The following code excerpt illustrates this process, where `e.writeSym` denotes the left-hand side temporary of assignments. On the right, a visual representation is provided for our example query, showing the computed `tmpInsideLoop` (solid lines) and `tmpAfterTmp` (dotted lines).

```
// compute tmpInsideLoop and tmpAfterTmp
for (let e of assignmentStms) {
   for (let dep of e.deps) {
      // depends on a loop, it should be inside the loop
      if (isloop(dep)) tmpInsideLoop[e.writeSym][dep] = true
      // depends on a tmp, expression should be after that
      if (istmp(dep)) tmpAfterTmp[e.writeSym][dep] = true
}  }
```



In the above code, we iterate through all the assignment statements and examine all their dependencies. If a statement depends on a loop, it means the corresponding temporary variable must be scheduled inside the loop, thus updating the `tmpInsideLoop` variable. Furthermore, if an assignment depends on another temporary value, it means the corresponding temporary variable must be scheduled after that, consequently updating the `tmpAfterTmp` variable.

The next step involves determining which temporaries should be scheduled *after* a given loop has been fully scheduled. This analysis leverages the information obtained from the earlier computations of `tmpInsideLoop` and `tmpAfterTmp`. The outcomes of this analysis are then stored in the `tmpAfterLoop`. The following is an excerpt from the corresponding analysis code in Rhyme.

```
// compute tmpAfterLoop
for (let t2 in tmpAfterTmp) {
   // gather loop 'prior' tmps are in
   for (let t1 in tmpAfterTmp[t2]) {
    for (let l in tmpInsideLoop[t1])
      tmpAfterLoop[t2][l] = true
   }
   // remove own loops
   for (let l in tmpInsideLoop[t2])
    delete tmpAfterLoop[t2][l]
}
```

In essence, if our analysis indicates that a temporary variable `t2` should be scheduled after another temporary `t1` (i.e., `tmpAfterTmp[t2][t1]`), and `t1` is known to be inside a loop `l` (provided that `t2` itself is not part of loop `l`), it implies that `t2` should be scheduled after the completion of loop `l`. For example, in our sample query `tmp[0]` is determined to be scheduled after `*A`.

The final step of the analysis involves figuring out the relationships between loops. Our objective is twofold: first, to discern which loops should be scheduled strictly after others (captured in `loopAfterLoop`), and second, to identify which loops should be scheduled within the same loop nest (recorded in `loopInsideLoop`). These essentially help identify how the loops should be scheduled. Specifically, if our previous analysis revealed that a given temporary variable `t` should be scheduled within both loops `l1` and `l2`, it implies that `l1` and `l2` should form part of the same loop nest. Conversely, if we determine that for a specific temporary variable `t`, it should be scheduled inside a loop `l2`, and we also know that `t` should be scheduled after another loop `l1`, it implies that loop `l2` should be scheduled after `l1`—provided they are not part of the same loop nest. The corresponding code illustrating these analyses is presented below.

```
// compute loopInsideLoop
for (let t in tmpInsideLoop) {
   for (let l2 in tmpInsideLoop[t]) {
      for (let l1 in tmpInsideLoop[t]) {
         loopInsideLoop[l2][l1] = true
         loopInsideLoop[l1] ??= {}
         loopInsideLoop[l1][l2] = true
      }
   }
}
```

```
// compute loopAfterLoop
for (let t in tmpAfterLoop) {
   for (let l2 in tmpInsideLoop[t]) {
      for (!loopInsideLoop[l2] ||
           !loopInsideLoop[l2][l1]) {
         if (!loopInsideLoop[l2][l1])
            loopAfterLoop[l2][l1] = true
      }
   }
}
```

After completing the analysis steps, we move on to the code generation process. The core driver of the code emitting logic is encapsulated within the `emitConvergence` function. This function relies on two additional helper functions: `emitGenerators` and `emitAssignments`. The former is responsible for emitting loop structures, while the latter deals with the generation of assignment instructions. These functions are invoked repeatedly until all generators and assignments are fully emitted. We present an excerpt from `emitGenerators` below:

```
function emitGenerators() {
   let (available, remaining) = getAvailable(generators) // identify current scope loops
   generators = remaining     // remaining generators that are not available yet
   for (let g of available) {
      emitLoopHeader(g)
      emitConvergence()       // emit loop body (nested generators and assignments)
      emitLoopFooter()
   }
}
```

Here, `generators` is a global variable that keeps track of the remaining generators that is yet to be scheduled. The `emitGenerators` function is responsible for orchestrating the generation of loops corresponding to these generators. It commences by identifying generators that become available in the current scope, leveraging information such as `loopAfterLoop`, `tmpAfterTmp`, and others we saw before. The `getAvailable` function checks the dependencies of pending generators and picks the ones where all dependencies are satisfied. Consequently, this function schedules loops that are available at the present scope and invokes `emitConvergence` recursively to schedule any inner loops and assignments that must be scheduled inside these loops. `emitAssignments` function follows a similar structure to that of the `emitGenerators`. It schedules assignment instructions as soon as they become available in the current context.

As program control structures were not enforced from the front end, our code scheduling mechanism freely moves assignments and generators during the code generation phase. For example, generators without dependencies on the 'other query' are hoisted and scheduled as separate queries. This approach prevents redundant computations within nested loops, enhancing efficiency. For instance, for our example query, the computation of the total aggregate (i.e. `sum(data.*B.value)`) is hoisted out of the `*A` loop into a completely independent loop, preventing the recomputation of the aggregate for each different key.

4 Related Work

This section provides a summary of the closely related works to Rhyme, as presented in [8].

Various efforts have aimed to address the challenge of efficiently handling multi-paradigm workloads by constructing common IR, as seen in prior work like Weld [23], Delite [25], and Flern [9]. In this landscape, Rhyme takes a different approach by introducing a *query language*—as opposed to a common low-level IR—that possesses the capability to express diverse multi-paradigm workloads at a higher level.

Numerous query languages have been crafted for working with semi-structured data like JSON, each exhibiting its unique focus and strengths. Notable among them is JSONiq [5, 17], a query language explicitly tailored for JSON data and borrowing syntax from XQuery [4], featuring constructs such as FLWOR expressions. Engines like Zorba [6] and RumbleDB [21] support JSONiq, with RumbleDB utilizing Spark [26] as a backend to leverage its scalability for execution. In the realm of semi-structured data, AsterixDB adopts AQL [2] and SQL++ [22] as its query languages.

Rhyme draws inspiration from functional logic programming languages such as Verse [12], Curry [19], miniKanren [15], and Scalogno [11], adapting these ideas into a novel data-centric declarative language.

5 Conclusion

This system description paper outlined the design and implementation of Rhyme, a data-centric and multi-paradigm declarative query language rooted in functional logic and metaprogramming. Rhyme offers diverse front-end APIs tailored to various workloads, extending beyond conventional data processing to encompass applications like tensor expressions (not discussed in this paper but elaborated in [8]) and expressive visualizations. The system achieves performance efficiency by constructing a loop-free and branch-free IR, facilitating optimization before transforming it into optimized JavaScript code for the given query. Currently, Rhyme is accessible as an open-source Node.js package, adaptable for integration into other Node.js projects or as a script importable into HTML pages.

References

1. Advent of code 2022. `https://adventofcode.com/2022/day/1`, accessed: 2023-09-27
2. The asterix query language (aql). `https://asterixdb.apache.org/docs/0.9.8/aql/manual.html`, accessed: 2023-09-27
3. jq manual. `https://jqlang.github.io/jq/manual/`, accessed: 2023-09-27
4. Xquery 3.1: An xml query language. `https://www.w3.org/TR/xquery-31/` (2017), accessed: 2023-09-27
5. Jsoniq. `https://www.jsoniq.org/` (2018), accessed: 2023-09-27
6. Zorba. `https://www.zorba.io/` (2018), accessed: 2023-09-27
7. Abeysinghe, S., He, Q., Rompf, T.: Efficient incrementialization of correlated nested aggregate queries using relative partial aggregate indexes (RPAI). In: SIGMOD Conference. pp. 136–149. ACM (2022)
8. Abeysinghe, S., Rompf, T.: Rhyme: A data-centric expressive query language for nested data structures. In: PADL. Lecture Notes in Computer Science, Springer (2024)
9. Abeysinghe, S., Wang, F., Essertel, G.M., Rompf, T.: Architecting intermediate layers for efficient composition of data management and machine learning systems. CoRR **abs/2311.02781** (2023)
10. Abeysinghe, S., Xhebraj, A., Rompf, T.: Flan: An expressive and efficient datalog compiler for program analysis. Proc. ACM Program. Lang. **8**(POPL) (2024)
11. Amin, N., Byrd, W.E., Rompf, T.: Lightweight functional logic meta-programming. In: APLAS. Lecture Notes in Computer Science, vol. 11893, pp. 225–243. Springer (2019)
12. Augustsson, L., Breitner, J., Claessen, K., Jhala, R., Peyton Jones, S., Shivers, O., Steele Jr., G.L., Sweeney, T.: The verse calculus: A core calculus for deterministic functional logic programming. Proc. ACM Program. Lang. **7**(ICFP) (2023)
13. Bracevac, O., Wei, G., Jia, S., Abeysinghe, S., Jiang, Y., Bao, Y., Rompf, T.: Graph irs for impure higher-order languages (technical report). CoRR **abs/2309.08118** (2023)
14. Bračevac, O., Wei, G., Jia, S., Abeysinghe, S., Jiang, Y., Bao, Y., Rompf, T.: Graph irs for impure higher-order languages: Making aggressive optimizations affordable with precise effect dependencies. Proc. ACM Program. Lang. **7**(OOPSLA2), 236:1–236:31 (2023)

15. Byrd, W.E.: Relational programming in miniKanren: techniques, applications, and implementations. Ph.D. thesis, Indiana University (2009)
16. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5), 98:1–98:38 (2022)
17. Florescu, D., Fourny, G.: Jsoniq: The history of a query language. IEEE Internet Comput. **17**(5), 86–90 (2013)
18. GraphQL: A query language for your api. `https://graphql.org/`, accessed: 2023-09-27
19. Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics. Lecture Notes in Computer Science, vol. 7797, pp. 123–168. Springer (2013)
20. Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., Shaikhha, A.: Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. **23**(2), 253–278 (2014)
21. Müller, I., Fourny, G., Irimescu, S., Cikis, C.B., Alonso, G.: Rumble: Data independence for large messy data sets. Proc. VLDB Endow. **14**(4), 498–506 (2020)
22. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. CoRR **abs/1405.3631** (2014)
23. Palkar, S., Thomas, J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S.P., Zaharia, M.: A common runtime for high performance data analysis. In: CIDR. www.cidrdb.org (2017)
24. Rogozhnikov, A.: Einops: Clear and reliable tensor manipulations with einstein-like notation. In: ICLR. OpenReview.net (2022)
25. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. **13**(4s), 134:1–134:25 (2014)
26. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)