Integrating Relational Programming into Mainstream Programming Contexts

Joshua Horowitz*
University of Washington

1 MOTIVATION

Programmers process data using various styles, or *paradigms*, of programming. The most traditional is *imperative* programming, where you tell the computer how to take small steps, iterating through old data and performing modifications ("mutations"), to lead to a desired result. An increasingly popular alternative is *functional* programming, where you produce new data from old through functions – sometimes functions themselves produced with higher-order functions.

A third option has maintained an important presence in the wings for many decades, with SQL as its flagship example: *relational* programming. Relational programming starts with data structured as multi-way relations between entities and values (which SQL calls "tables"). It then gives programmers high-level languages which they can use to express their goal in terms of these relations, in more of a mathematically descriptive way than a mechanistic one. In fact, engines must often do a great deal of work to transform a relational query into a procedure that can run efficiently on data. This gap between language and execution is a clear sign of just how high-level relational programming can be.

As the example of SQL suggests, relational programming is (by a large margin) most commonly used for querying databases – stable stores of data that are "far away" from a running program. While programs constantly manipulate data stored in arrays and maps sitting in variables, "close by" in memory, this data is almost never transformed relationally.

But why not? The advantages of the relational paradigm go well beyond its suitability for querying large databases. A relational language gives a programmer a higher-level way of expressing themself which may more closely match how they think about their domain and their task. Meanwhile, the query engine takes on responsibility for managing inessential concerns around data structures and algorithms. Ideally, the relational paradigm could offer both expressive and computational efficiencies to programmers.

2 OVERVIEW

In this project, we prototyped a system called Relat which explores embedding relational programming inside of JavaScript, the most commonly-used language among developers today.[1] Our aim was to make it possible for programmers to ergonomically write queries against data structures that already exist in their program "in the wild" and solve problems that would be more difficult without our system, thereby unobtrusively integrating relational programming into existing programming practices.[2]

*e-mail: joho@uw.edu

[1] https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof

[2] This focus on unobtrusively complementing existing practices imposed constraints on our approach. For instance: although building our system as an extension to the JavaScript language would offer an appealingly tight integration, it would impose unacceptable obstacles to adoption in practice, so we instead chose to implement Relat as a normal library.

We structured Relat as a stack of three levels, a pattern we believe will be common to any attempt to integrate relational programming into a mainstream, conventional language.

First, we need a query engine that runs inside our programming-language host. Relational programming relies at its core on a well-engineered engine. To enable easy access to relational programming from many contexts (and without communication delays), this engine must to be collocated with running programs – not across a network or even in a separate process. For our engine, we chose Souffle [3], the most well-developed open-source Datalog engine. Fortunately, Souffle contributors have already compiled Souffle to WebAssembly using Emscripten[3], so only minimal modifications were necessary to integrate Souffle into our system. With this set-up, we can run Souffle from essentially any JavaScript context, including browsers and Node.

Second, we need a compact, expressive relational language that can compile to this engine. This means not SQL and not Datalog. Although it would be convenient to leverage one of these existing languages, their verbosity (and poor ergonomics) disqualifies them. They would not fit harmoniously into the host language or cultural practices – a JavaScript programmer accustomed to writing `advisorById[studentById[id].advisorId]` would not be eager to replace this with a join-filled SQL query or Datalog Horn clauses. To meet this need, we built a new relational language called Relat which compiles to Datalog for execution by Souffle. Relat is closely modelled on Rel [8], a commercial relational language grounded in relational algebra.[4] Relat joins Rel in supporting expressive features like relational abstractions and quantifiers. Its syntax packs down tightly in many cases to rival JavaScript's in concision. More details on Relat are found in §4.

Lastly, we need a way of reinterpreting data between the host language and the relational language. Our goal was to make a library that JavaScript programmers in a wide variety of settings could "drop in" to solve a problem. This means that, rather than assuming full control over how the programmer represents their data, our library ought to work seamlessly with existing patterns of data storage. While we had less opportunity to make progress on this level of the stack than on the Relat language itself, we did make some initial steps, as discussed in §5.

3 PRIOR WORK

A number of research projects have developed languages which tightly integrate functional and relational features. Flix [5] is a general-purpose functional language with built-in support for Datalog constraints as first-class values and ergonomic syntax for moving data in and out of embedded Datalog. Functional IncA [6] similarly embeds Datalog inside a functional language, but goes further than Flix in that even the non-Datalog parts of functional IncA are ultimately compiled to Datalog. These projects offer valuable sources of inspiration, showing different ways relational and

[3] https://emscripten.org/

[4] Before finding Rel, we were similarly inspired by the relational language at the heart of Alloy [2]. Ultimately, we found Rel's design to be more elegant and better-suited to the applications we had in mind.

functional/imperative languages can be integrated and different applications of this integration. Our project differs in that we are not looking to create an entirely new hybrid language, but are instead trying to incrementally provide relational benefits to programmers working in JavaScript.

A number of libraries already provide JavaScript with relational programming features. In particular, "Datalog UI" provides an implementation of differential Datalog as a JavaScript library [1], and DataScript is an "immutable in-memory database and Datalog query engine" which runs in the browser [7]. Our project differs from these in several ways important ways. These projects treat the "database" as a separate entity – the user deliberately puts data into the database. In contrast, we want to make a system where the user can write queries directly against JavaScript values, easing incremental adoption of relational features. Furthermore, inspired by Rel [8], we want to explore a wide gamut of notations for operating relationally on data, varying in compactness and expressivity.

A final fascinating point of reference is Riffle [4], an architecture for building JavaScript applications from reactive relational queries running on a client-side database. While we share Riffle's interest in bringing the conveniences of relational programming into conventional JavaScript contexts, we are interested here in developing a general-purpose technique for bringing relational programming "in the small" into many corners of the JavaScript world, rather than constructing a domain-specific framework that an application fits into.

4 RELAT

As mentioned previously, we believe the best chance for integrating relational programming with conventional JavaScript is through a compact, flexible, expressive notation. Our exploratory attempt at this is called Relat. Relat began with strong inspiration from the relational language at the heart of Alloy. More recently, we have been taken by many of the design decisions taken by Rel (like unification of booleans & relations) and begun to incorporate them.

Relat is a language for writing and evaluating relation-valued *expressions*, which may refer to a number of externally provided relations. (The operations and syntax of Relat are summarized in Figure 1.) For instance, given the relations `isPerson(person)`, `isHappy(person)`, and `hasChild(parent, child)`, we can evaluate:

- "`isPerson \ isHappy`": the set of unhappy people,
- "`isHappy.hasChild`": the set of children of happy parents (using a "dot join", which joins the last argument of the first relation to the first argument of the second relation),
- "`^hasChild`": the transitive closure of `hasChild` – a binary relation that relates all ancestors to their descendants, or
- "`x : isPerson | some (x.^hasChild & isHappy)`": the set of all people with a happy descendent.

As these examples show, Relat supports a variety of operations from relational algebra, as well as "set-theoretic" constructions like relational abstractions (comparable to comprehensions in a language like Python). These allow a spectrum of styles, from a highly compact but occasionally abstruse "point-free" style to an expressive and literate style centered around named variables. These styles can be mixed and matched as desired, as they all consume and produce relations.

Relat evaluates expressions by compiling them to Datalog. It does this straightforwardly, descending recursively down the Relat AST. Each expression in the tree is represented by a Datalog relation, together with a set of rules which define that relation in terms of the Datalog relations of various dependent expressions. For instance, the

Relat syntax

All expressions in Relat are relations. Relation arguments can be numbers or symbols (Souffle's name for strings).

Basics

Syntax	Description
`123` ‖ `'abc'` ‖ `"abc"`	constant literals
var	reference to variable
`let` *var* `=` *exp1* `|` *exp2*	let-binding
`(`*exp*`)`	parentheses for grouping
`` ``*str*`` ``	formula *escape hatch for abitrary constraints*

Relational algebra

Relat represents booleans as zero-argument relations, so boolean operators are specializations of relational operators.

Syntax	Description
exp1 `;` *exp2*	union *also works as boolean OR*
exp1 `,` *exp2*	product *also works as boolean AND*
exp1 `&` *exp2*	intersection *also works as boolean AND*
exp1 `\` *exp2*	difference
`some` *exp*	test if non-empty
`not` *exp*	test if empty *also works as boolean NOT*
exp1`.`*exp2*	dot join
exp1`[`*exp2*`]`	(partial) relational application
exp1`._` ‖ `_.`*exp1* ‖ *exp1*`[_]`	wildcard joins / application *projections*
`~`*exp*	transpose of *exp*
`^`*exp*	transitive closure of *exp*
exp1 `<:` *exp2* ‖ *exp1* `:>` *exp2*	prefix / suffix join

Relational abstraction

Whereas relational algebra is "pointless" (it operates on relations as a whole), relational abstraction is "pointed" (it extracts arguments of relations). It is similar to "comprehensions" in languages like Python, but more general.

Syntax	Description
var1[, var2, ...] `:` *exp1* `|` *exp2*	relational abstraction (for-style) *result includes var1[, var2, ...] and exp2*
var1[, var2, ...] `:` *exp1* `->` *exp2*	relational abstraction (from-style) *result only includes exp2*

Aggregates

Aggregates operate on the last argument of a relation, without removal of duplicate values.

Syntax	Description
`#`*exp*	count
`min` *exp* ‖ `max` *exp*	min / max *applicable to numbers or symbols*
`sum` *exp*	sum *applicable to numbers*
`concat` *exp*	concatenate *applicable to symbols; arbitrary order*
`index` *exp*	add argument with unique indices *not itself an aggregate, but useful for building them*

Scalar operators

Scalar operators operate on 1-argument relations.

Syntax	Description
exp1 `+` *exp2* ‖ *exp1* `-` *exp2* ‖ *exp1* `*` *exp2*	add / subtract / multiply *applicable to numbers*
exp1 `<` *exp2* ‖ *exp1* `>` *exp2* ‖ *exp1* `<=` *exp2* ‖ *exp1* `>=` *exp2* ‖ *exp1* `=` *exp2*	comparisons *applicable to numbers or symbols*

JavaScript objects

Relat is most fundamentally run with Souffle relations as input. With the experimental `mkJsObjDB` adapter, it can run directly on a network of JavaScript objects.

Syntax	Description
`<`*prop*`>`	property access *a relation mapping obj to obj.prop*
`<_>`	wildcard property access

Figure 1: An overview of syntax supported by the Relat language.

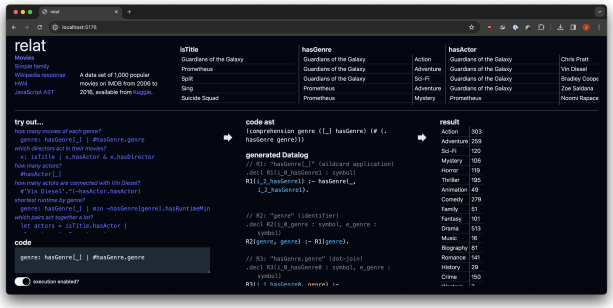

Figure 2: The interactive Relat testbed.

Relat expression "`isPerson \ isHappy`" from earlier is converted to the Datalog:

```
R1(p) :- isPerson(p), ! isHappy(p)
```

and the Relat expression "`isHappy.hasChild`" is converted to:

```
R2(b) :- isHappy(a), hasChild(a, b).
```

To compile larger expressions with sub-expressions will

An interesting subtlety arises with the use of relational abstractions, which introduce bound variables. Expressions in the body of a abstraction are parameterized by the scalar variable introduced by the abstraction. For instance, the expression "`x.^hasChild`" in the fourth example above is semantically a 1-argument relation – the set of descendants of `x`. However, `x` is not a constant, but is rather a variable which can take on many values (while being constrained by `isPerson`). To represent this in Datalog, we compile this expression to a 2-argument Datalog relation. The first argument is "internal", representing the single argument of the computed relation. The second argument is "external", representing the in-scope variable `x` that controls this relation. Naturally, this approach generalizes to any number of internal and external arguments. (If this approach is established in the literature, we would be delighted to read more about it.)

To test Relat in a few different settings and communicate its features, we built an interactive testbed, shown in Figure 2 and available online at `https://joshuahhh.com/relat/`.

5 Integration into JavaScript

Relat is up and running in JavaScript, which means it already has a JavaScript integration, of a sort. The current API is centered around a function `runRelat`, which takes two parameters: 1. a string of Relat code, and 2. an object mapping relation names to relations. (Here, a "relation" is just a set of tuples together with type annotations.) So, for instance, we can run:

```
const hasHappyDescendant = runRelat(
  "{x : isPerson | some (x.^hasChild & isHappy)}",
  {isPerson, hasChild, isHappy}
);
```

(provided that `isPerson`, `hasChild`, and `isHappy` are already in scope as relations). This will return a new relation `hasHappyDescendant`.

A slight infelicity here is the need to redundantly pass input relations into `runRelat`'s second argument, even though we are already referring to them in its first argument. This could be improved with a simple (but as-yet unimplemented) use of JavaScript's "tagged template literal" syntax, which allows arbitrary values to be embedded into string-like syntax:

```
const hasHappyDescendant = runRelat`
  {x : ${isPerson}
    | some (x.^${hasChild} & ${isHappy})}
`;
```

With `runRelat`, Relat is already useful. However, it does not yet meet our vision of relational programming fluidly embedded into JavaScript, because all uses of Relat code must occur through manual conversion of JavaScript data into and out of relations.

One interesting approach would be to provide tools to aid this manual conversion. For instance, suppose our `hasChild` data is stored in a JavaScript object like:

```
const people = {
  id48348: {
    name: "Abe",
    children: ["id24018", "id05830"],
    ...
  },
  id24018: {
    name: "Homer",
    children: ["id08414", "id80531", "id58023"],
    ...
  },
  ...
}
```

To construct the `hasChild` relation from this, perhaps we could run a command like:

```
const hasChild = toRel(
  people,
  "{<1>: {children: [<2>...]}}"
);
```

Here, the `toRel` tool uses a special pattern-matching DSL to ergonomically extract a relation's tuples from JavaScript data. An analogous `fromRel` could convert back. We have not further explored the design or implementation of converters like this, though we think it would be worthwhile.

5.1 Directly representing JS data

More ambitiously, we wondered how arbitrary JavaScript data could be represented immediately in a relational language, without the need for manual conversion to and fro.

Consider a scenario where a programmer is managing a server that builds documents into LaTeX. Their data can be represented in TypeScript as:

```
type BuildsDoc = {
  builds: { [buildId: string]: Build },
}

type Build = {
  id: string,
  startTime: Date,
  status: 'pending' | 'ok' | 'error',
  ...
}
```

This is hierarchical data. If `bd` is a `BuildsDoc`, we might navigate to `bd.builds.id12345.startTime` to get the start time of a specific build. More generally, the JavaScript object graph might instead be a directed acyclic graph or even a graph with cycles.

The lowest-level way to represent this relationally is via object-key-value triples. We define a relation `okv`, where `okv(o, k, v)`

means that `o[k]` is `v`. This single (admittedly not particularly well-typed) relation captures everything there is to know about compound data in JavaScript. For instance, if `bd` is a scalar representing a `BuildsDoc`, the Relat expression `bd.okv["builds"].okv` produces a relation mapping a `buildId` to its corresponding `Build`.

1. `bd.okv` joins `bd` to the "O" slot of `okv`, thus producing a 2-argument relation mapping the property keys of `bd` to property values.
2. `bd.okv["builds"]` is "relational application", equivalent (in this case) to `"builds".(bd.okv)`, so this results in a 1-argument relation (scalar) representing the JS object `bd.builds`.
3. Finally, `bd.okv["builds"].okv` does the same kind of join we did in step 1, turning the JS object `bd.builds` into a 2-argument relation mapping keys of `bd.builds` to their values.

`okv` allows us to move between two representations of a JavaScript object – first, as a scalar node in a graph of relations, and second, as a mapping of property keys to property values.

We implemented a scheme built around `okv` called `mkJsObjDB`. A user can call `mkJsObjDB` with an arbitrary JavaScript object, called the root. `mkJsObjDB` then "crawls" the JavaScript object graph starting from the root. To each object it encounters, it assigns a unique ID (a simple string like `"#7"`). `mkJsObjDB` ultimately returns a set of relations ready to be fed into `runRelat`:

- `root(obj)`: a singleton set consisting of the root object's ID,
- `any(obj)`: a set consisting of all objects' IDs,
- `okv(obj, string, obj)`: the triple relation discussed above, mapping object IDs to object IDs through string keys,
- `str(obj, string)`: a map from objects that are actually strings to their string contents,
- `num(obj, number)`: a map from objects that are actually numbers to their numeric contents,
- `special(obj, string)`: a map from objects that are actually "special values" (true, false, undefined, null) to a string marking this special value.

By feeding an object to `mkJsObjDB`, a user gets a miniature database that can be queried freely with Relat.

Of course, there are significant issues of ergonomics to face. No JavaScript programmer wants to replace `student.advisor.department` with `student.okv["advisor"].okv["department"]`. We have just begun to grapple with some of these issues using a simple syntactic sugar for property access. With this sugar, `<key>` converts to the binary relation (computed from `okv`) mapping `obj` to `obj.key`. This way, the programmer can write `student.<advisor>.<department>`, which more comfortably matches JavaScript syntax. And Relat brings powerful new possibilities from the relational paradigm: a programmer can write `<advisor>.<department>."Earth Sciences"` to find all students advised by faculty in the Earth Sciences department, effortlessly inverting relations which would ordinarily require contortions in JavaScript code.

This syntactic sugar is a good first step, but more work (not yet done) using Relat in actual JavaScript applications would likely reveal more pain points and more opportunities to iterate the language's design, perhaps in more fundamental ways than shallow syntax. We leave this to future work.

6 EXAMPLE USAGE

Here, we list a few scenarios we have used to experiment with Relat & `mkJsObjDB`. They are all explorable in our testbed at `https://joshuahhh.com/relat/`.

6.1 IMDB movies

A data set of 1,000 popular movies on IMDB from 2006 to 2016 available from Kaggle.[5]

- How many movies are released in each genre?

```
genre: hasGenre[_] | #hasGenre.genre
```

- How many actors are connected to Vin Diesel through co-starring in films?

```
#'Vin Diesel'.^(~hasActor.hasActor)
```

(The expression `~hasActor.hasActor` joins `hasActor` to its transpose to produce the "co-starring" relation from actors to actors, which is then transitively closed with `^`.)

- Which pairs of actors act together in at least three films?

```
let actors = isTitle.hasActor |
a1: actors | a2: actors | a2 > a1,
let hasBothActors = hasActor.a1 & hasActor.a2 |
#hasBothActors >= 3,
#hasBothActors, concat hasBothActors
```

6.2 JavaScript AST

We parsed a short JavaScript program with Acorn[6] and fed the resulting AST object into `mkJsObjDB`.

- Which functions call themselves?

```
let idName = <type>.str.'"Identifier"'
<: <name>.str |
let fnRef = (fnDecl : any ->
fnDecl.<type>.str = '"FunctionDeclaration"',
fnDecl.<id>.<name>.str,
fnDecl.<body>.^<_>.idName
) |
x, y : fnRef -> x = y, x
```

6.3 HTML tree

We downloaded the Wikipedia page on Datalog and parsed it into a JSON object to feed to `mkJsObjDB`.

- What are URLs of images on the page?

```
x : any ->
  x.<tagName>.str = '"IMG"',
  x.<attrs>.<src>.str
```

[5] https://www.kaggle.com/datasets/PromptCloudHQ/imdb-data
[6] https://github.com/acornjs/acorn

7 Limitations & future work

Relat was built as an exploratory prototype, and as such, it has several notable limitations that would serve as excellent starting points for further inquiry. We have mentioned some already, such as the need for more work on Relat's integration with JavaScript structures. Here we list several more.

Foremost is performance. Performance was deliberately not a focus of this project, and we see four ways that it could likely be improved.

1. In an effort to be simple and easy to implement, our Relat-to-Datalog compiler produces very verbose, redundant code. We made no effort to limit the number of intermediate relations we use, or to optimize the definitions of these relations. If Souffle is particularly good at optimizing, then we don't need to worry about this. But query optimizers often need some help, and our current implementation gives them none.

2. Our use of Souffle does not maximize its potential. We are running Souffle in its interpreter mode. If we instead ran Souffle as a compiler, it would generate C++ code which could be further optimized by a C++ compiler. WebAssembly is getting good enough these days that this entire process could probably be done in the browser.

3. Moving data into and out of Souffle brings many inefficiencies. We must serialize and deserialize relations, and transfer them into and out of a virtual file system accessed by the Souffle WebAssembly binary. This suggests that we might be able to achieve better performance by using Souffle's library bindings.

4. Many JavaScript applications re-perform computations in response to user interaction, although the data these computations take as input may change very little. This is an ideal setting for incremental computation, which relational models like Datalog are well-suited for. However, this is a more ambitious direction for this project to pursue, as, to our knowledge, Souffle does not have incremental features. A system like Differential Datalog [9] might be more appropriate here.

Now we move to limitations beyond performance. One is integration with TypeScript, a popular static type system for JavaScript. Like many programmers, we have found TypeScript very useful in our work. Not only does it diligently check our code for type errors at edit time, but it speeds our programming tasks with intelligent auto-complete and related editor features. Ideally, Relat would fit into this typing scheme – Relat could take in types from the relations it is provided with, these types could flow through Relat expressions, and they could come out in Relat's outputs. However, implementing this is not only outside the scope of our project, but may in fact be impossible. This is essentially because TypeScript has no macro system (or other compile-time evaluation) by which Relat code could be processed and provided to the TypeScript type system.

Finally, we note that we do not provide a way for Relat code to make use of arbitrary JavaScript functions, although the need for this will probably arise when examining our scenarios of use. Souffle has hooks to load "user-defined functors" from dynamically loaded library – perhaps these could be adapted in our WebAssembly setting.

8 Future work

References

[1] Datalog UI. @datalogui/datalog. https://github.com/datalogui/datalog, 2024.

[2] D. Jackson. Alloy. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 4 2002.

[3] H. Jordan, B. Scholz, and P. Subotić. *Soufflé: On Synthesis of Program Analyzers*, pp. 422–430. Springer International Publishing, 2016.

[4] G. Litt, N. Schiefer, J. Schickling, and D. Jackson. Riffle: Reactive Relational State for Local-First Applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, oct 29 2023.

[5] M. Madsen, M.-H. Yee, and O. Lhoták. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, jun 2 2016.

[6] A. Pacak and S. Erdweg. Functional Programming with Datalog, 2022.

[7] N. Prokopov. Datascript. https://github.com/tonsky/datascript, 2024.

[8] RelationalAI. The Rel Language. https://docs.relational.ai/rel, 2024.

[9] L. Ryzhyk and M. Budiu. Differential Datalog. In *Datalog*, 2019.