Josh Horowitz

CSE 544, Winter 2024

Integrating Relational Programming into a General-Purpose Language

Relational programming is typically only used when querying large databases external to code, such as with SQL. **We believe the relational paradigm could be useful in many different contexts if it were more tightly integrated into a conventional, general-purpose language, operating on runtime data.**

Proper integration requires:

Level 1: A query engine that runs inside the programming-language host.
Level 2: A compact, expressive relational language.
Level 3: A way of reinterpreting data between between the host language and the relational language without tedious boilerplate.

We have built an integration of relational programming into JavaScript satisfying these requirements, centered around a new relational language called **Relat**.

How it works

Level 1: Souffle in JavaScript

We use Emscripten to build the Souffle Datalog engine into WebAssembly, which can be run in any JavaScript environment – the browser and client-side with Node.

Level 2: Relat

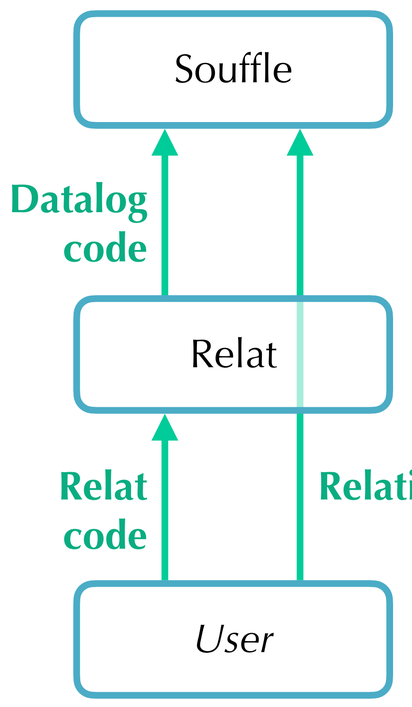
Relat is a compact, expressive relational language strongly inspired by Alloy and Rel. Relat supports a variety of operations from relational algebra, as well as "relational abstractions" which support a more "pointed" style.

Our implementation compiles Relat's syntax into Datalog which can be run by Souffle. For example, consider the Relat code:

```
isHappy.hasChild
```

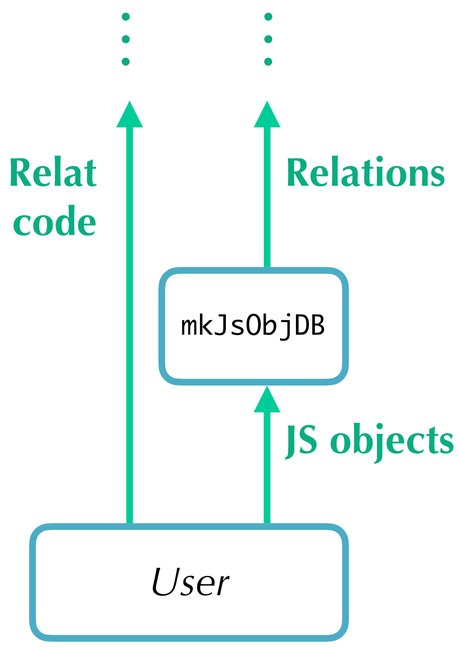
This uses a "dot join" to find children of happy parents. It compiles to:

```
R1(b) :- isHappy(a), hasChild(a, b).
```



Level 2: Relations from JavaScript objects

Relat can be called directly from JavaScript, passing in relations with number and string arguments. But work in JavaScript typically involves navigating complex hierarchies (or networks) of objects. We explore immediate use of Relat in this context with an adapter called `mkJsObjDB`. This adapter crawls a JavaScript object and its references, representing links between objects in a triple relation called `okv` (object-key-value).



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

Example code

Scenario: IMDB movies

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

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)
```

What 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
```

Scenario: CSE 544 Homework 4

A database of family relations.

Which woman and which man have the most children?

```
let num_children =
  (x : person._ | #parent_child[x]) |
let most_mothered = max num_children[female] |
let most_fathered = max num_children[male] |
x, c: num_children &
  (female, most_mothered; male, most_fathered) |
person[x]
```

Scenario: JavaScript AST

JavaScript code is parsed with Acorn and the AST is fed directly 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
```