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 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. Selectors arguments can be numbers or symbols (Souffle’s names for strings).

<table>
<thead>
<tr>
<th>Basics</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>$arity$</td>
<td>number of input arguments</td>
</tr>
<tr>
<td>$name$</td>
<td></td>
<td>reference to variable</td>
</tr>
<tr>
<td>let $var$ = $expr$</td>
<td></td>
<td>binding</td>
</tr>
<tr>
<td>$cond$</td>
<td></td>
<td>pattern for grouping</td>
</tr>
<tr>
<td>iftrue</td>
<td></td>
<td>escape back for legacy comments</td>
</tr>
</tbody>
</table>

**Relational algebra**

Relat represents relations as argument relations, as boolean operations are specializations of relational operators.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| $expr1 op expr2$ | algebraic operator
| $cond ? expr1 : expr2$ | conditional operator
| $expr1$ & $expr2$ | conjunction
| $expr1$ | | expression
| $expr1 ? op $expr2$ | conditional operator
| $cond ? expr1 : expr2$ | conditional operator
| $expr1$ | | expression
| $cond$ | | pattern for grouping |

**Relational abstraction**

Whereas relational algebra is “first-order” it operates on relations as a whole, relational abstraction is “second-order” in various arguments of relations. It is similar to “comprehensions” in languages like Python, but more general.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x : type).expr$</td>
<td>map over arguments</td>
</tr>
<tr>
<td>$let expr in expr$</td>
<td></td>
</tr>
<tr>
<td>$let expr = expr$</td>
<td></td>
</tr>
</tbody>
</table>

**Aggregates**

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$min$</td>
<td></td>
</tr>
<tr>
<td>$max$</td>
<td></td>
</tr>
<tr>
<td>$count$</td>
<td></td>
</tr>
<tr>
<td>$distinct$</td>
<td></td>
</tr>
<tr>
<td>$index$</td>
<td></td>
</tr>
</tbody>
</table>

**Scalar operators**

Scalar operators operate on 1-argument relations.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y$</td>
<td></td>
</tr>
<tr>
<td>$x - y$</td>
<td></td>
</tr>
<tr>
<td>$x * y$</td>
<td></td>
</tr>
<tr>
<td>$x / y$</td>
<td></td>
</tr>
</tbody>
</table>

**Relational algebra**

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.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$getprop$</td>
<td></td>
</tr>
<tr>
<td>$setprop$</td>
<td></td>
</tr>
</tbody>
</table>

**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?

```javascript
let 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?

```javascript
let actors = isLittle hasActor | al: actors | a2: actors | a2 > al,
let hasBothActors = hasActor.al & 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?

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

**Scenario: JavaScript AST**

JavaScript code is parsed with Acorn and the AST is fed directly into mkJsObjDB.

- Which functions call themselves?

```javascript
let idName = <type>.str."Identifier" |
let fnRef = fnDecl | fnDecl.id().str. "FunctionDeclaration",
let Decl.id().body. "<", .idName |
let x, y: fnRef -> x = y, x
```

Try our LIVE DEMO!