Rel: A Programming Language for Relational Data

Molham Aref
RelationalAI
Woodside, USA
molham.aref@relational.ai

Paolo Guagliardo
University of Edinburgh
Edinburgh, UK
paolo.guagliardo@ed.ac.uk

George Kastrinis
RelationalAI
Athens, Greece
george.kastrinis@relational.ai

Leonid Libkin
RelationalAI & Univ of Edinburgh
Paris & Edinburgh, France & UK
leonid.libkin@relational.ai

Victor Marsault
LIGM, Univ. Gustave Eiffel, CNRS
Marne-la-Vallée, France
victor.marsault@univ-eiffel.fr

Wim Martens
RelationalAI & University of Bayreuth
Bayreuth, Germany
wim.martens@relational.ai

Mary McGrath
RelationalAI
Little Compton, USA
mary.mcgrath@skylight.digital

Filip Murlak
University of Warsaw
Warsaw, Poland
f.murlak@uw.edu.pl

Nathaniel Nystrom
RelationalAI
Lugano, Switzerland
nate.nystrom@relational.ai

Liat Peterfreund
Hebrew University
Jerusalem, Israel
liat.peterfreund@mail.huji.ac.il

Allison Rogers
RelationalAI
Seattle, USA
allison.rogers@relational.ai

Cristina Sirangelo
Université Paris Cité, CNRS, IRIF
Paris, France
cristina@irif.fr

Domagoj Vrgoč
PUC Chile
Santiago, Chile
vrdomagoj@uc.cl

David Zhao
RelationalAI
Sydney, Australia
david.zhao@relational.ai

Abdul Zreika
RelationalAI
Melbourne, Australia
abdul.zreika@relational.ai

Abstract

From the moment of their inception, languages for relational data have been described as *sublanguages* embedded in a host programming language. Rel is a new relational language whose key design goal is to go beyond this paradigm with features that allow for programming in the large, making it possible to fully describe end to end application semantics. With the new approach we can model the semantics of entire enterprise applications relationally, which helps significantly reduce architecture complexity and avoid the well-known *impedance mismatch* problem. This paradigm shift is enabled by 50 years of database research, making it possible to revisit the sublanguage/host language paradigm, starting from the fundamental principles. We present the main features of Rel: those that give it the power to express traditional query language operations and those that are designed to grow the language and allow programming in the large.

CCS Concepts

• **Information systems → Relational database query languages**; *Relational database model*; • **Software and its engineering** → *General programming languages.*

Keywords

Relational data model, programming in the large, relational programming, query language design, impedance mismatch, relational knowledge graph, graph normal form

ACM Reference Format:
Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoč, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724450

1 Introduction

From the moment of their inception, languages for relational data have been described as *sublanguages* [16, 17]. This means that such languages are not designed to describe the entire end-to-end application logic of programs that involve data, but rather focus on specific operations that concern the storage, retrieval, or manipulation of data within such programs. This view of database languages made a lot of sense fifty years ago, when database query languages were being introduced in an environment of procedural imperative programming, when declarative languages were a revolutionary idea, and when it was unclear if this idea could be extended beyond its domain-specific use. Today, SQL is the prime success story of a declarative and domain-specific language. It remains in high demand by tech employers [12], which is a remarkable achievement for a 40-year-old programming (sub)language.

The sublanguage paradigm however has important limitations. To start with, it entails having to communicate with a host programming language. This causes the well-known *impedance mismatch*: the two languages have different data types, different data structures, different memory models, etc. Often they are even based on *different programming paradigms*, one being fundamentally declarative and the other fundamentally imperative. Furthermore, the two languages typically have separate runtime environments, which limits the optimizations that can be applied to programs that use both languages. The host language runtime environment may not support features of the database such as automatic out of core computation, automatic parallelism, incremental computation. In other words, the impedance mismatch

(1) causes extra complexity for developers, unrelated to solving the problem that they are actually trying to solve; and
(2) limits the capabilities of automatic support that we are used to in databases.

Another effect of the sublanguage paradigm is that query languages are not equipped with important features needed for programming in the large, as these are delegated to the host language. Most notably, query languages lack support for building libraries. Indeed, SQL does not have a standard library, and when new features are needed they are added by means of expanding the language itself. Over the years, this led to an 11-part ISO standard comprising well over 4000 pages which is, for comparison, an order of magnitude larger than the C standard. This breaks a fundamental principle of programming language design formulated by Steele [45]: *define a small core and provide the functionality to build libraries.*

Rel aims to go beyond the sublanguage paradigm. The database industry and research communities have accumulated a vast array of insights that reshuffle the cards on which the sublanguage design decision was based. We ask ourselves, can we *design and implement* a language that natively handles data and semantics (or "meaning", as Codd called it in his Turing Award lecture [19]) in a database, preserving all the bedrock principles of databases (such as the data independence, communicability, and set-at-a-time processing objectives [19]), providing the programmer with the necessary constructs to factor semantics out of application programs? Such a full-featured language for data and semantics would allow for new powerful simplifications and optimizations owing to a single runtime environment, increasing the productivity of users and application developers.

Towards this goal, Rel has been designed and implemented as a programming language for relational data that does away with the sublanguage paradigm. Its key features are:

(1) manipulation of both logical formulas and entire relations;
(2) powerful *recursion* built on the foundations of Datalog;
(3) *abstraction* and *application* as key constructs;
(4) variables that can *range over tuples and relations.*

We will touch upon all these points in the course of the paper.

A relational language beyond the sublanguage paradigm that supports programming in the large makes it possible to build database engines that can automate much more of the work being done by applications programmers working in the "two language paradigm". Building an engine for such a language is an ambitious goal that will take time to achieve. We believe, however, that it will bring significant gains over the sublanguage approach and that the time is ripe to embark on this journey.

A Rel Teaser

The starting point of Rel is Datalog with first-order logic formulas and aggregation in the bodies, which allows it to naturally express (recursive) database queries. We offer a few teasers to provide a glimpse of how the language goes beyond classical database querying and refer to Sections 3–4 for deeper explanation.

First, we define matrix multiplication as a general operation on relations. Since relations can easily model vectors, matrices, and tensors, Rel can naturally deal with analytics and ML workloads. Given an $n \times m$ matrix $\mathbf{A}$ and an $m \times p$ matrix $\mathbf{B}$, their product is an $n \times p$ matrix $\mathbf{M}$ whose element m_{ij} is defined as $\sum_{k=1}^{m} a_{ik} b_{kj}$. If we represent matrices as relations with triples (row number, column number, value), then the Rel definition of matrix multiplication mimics its mathematical definition:

```
def MatrixMult[{A},{B},i,j] : sum[ [k] : A[i,k]*B[k,j] ]
```

Given two matrices `M1` and `M2`, `MatrixMult[M1,M2]` evaluates to the relation that represents their product. So, not only is this definition similar to the mathematical definition and easy to program, it is also suitable as a library definition, since it takes relations as parameters. It perfectly fits the paradigm of growing a language from a small core, making it easy for users to expand that core using library functions. Furthermore, the relational model's data independence principle makes it possible for a Rel engine to automatically choose the right data structures for `M1` and `M2` depending on whether they are dense or sparse, in-memory or not, etc.

As a second example, consider the following definition of all pairs shortest paths (APSP), given sets of nodes `V` and edges `E`:

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,i) :
i = min[{(j): exists((z) | E(x,z) and APSP(V,E,z,y,j-1))}]
```

We can read the code as follows. The shortest path from x to y has length 0 if x and y are nodes and $x = y$. Otherwise, the shortest path length i is the minimum j such that an out-neighbor z of x has a shortest path of length $j-1$ to y. Again, `APSP` can serve as a library definition in the sense that, if we have a directed graph with nodes `N` and edges `NN`, and two nodes u and v, we can call `APSP[N,NN,u,v]` to give us the length of the shortest path from u to v.

Rel is a Relational Programming Language

Functional programming languages use functions as the main building block. Similarly, imperative programming languages use procedures. In Rel, this role is fulfilled by relations. In principle, the relational approach subsumes the functional approach because every function is a relation.

We illustrate this by showing how Rel's notion of *relational application* generalizes function application. For simplicity, consider a function $f : V \times V \longrightarrow V$, where V is a domain of values. Such a function can be represented as a ternary relation F consisting of the triples $(a, b, f(a, b))$ in which the first two columns determine the third. Rel's syntax `F[a,b]` corresponds to the case where we provide f with two parameters a and b and obtain $f(a, b)$ as a result. But Rel also allows writing `F[a]` to return all pairs $(b, f(a, b))$ or `F[a,b,c]` to return true if and only if $(a, b, c) \in F$ (in other words, if and only if $c = f(a, b)$). Likewise, it is possible to write `F` to return the entire relation F.

Paper Overview. In Section 2, we explain the principles of modeling in graph normal form (GNF). Then, we move to the fundamental ingredients of Rel in Section 3 and move to the features that give Rel the capability to do programming in the large in Section 4. In Section 5 we show how relational algebra, linear algebra, and graph algorithms can be implemented as libraries. We explain the ideas behind building relational knowledge graphs using Rel in Section 6. Section 7 discusses Rel influences, its use, and future plans. We present a formal semantics of (a core of) Rel in Addendum A.

2 Data Modeling and Graph Normal Form

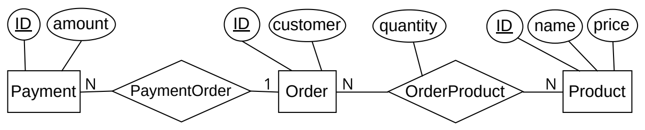
Codd's original papers [15, 16, 18, 19] introduced a model based on n-ary relations, a normal form for database relations, and the concept of a universal sublanguage. The n-tuples in each relation represent facts about the domain being modeled. Since a fact can hold neither multiple times, nor partially, the pure relational model [15] has neither multiplicities (bags) nor nulls, i.e., it is based on set semantics and two-valued predicate logic. Rel takes this one step further: tuples represent facts that are *indivisible.* A complex fact like *"Edgar was born on 19 August 1923 in Underhill"* is better thought of as two indivisible facts, *"Edgar was born on 19 August 1923"* and *"Edgar was born in Underhill".* Rel shares this perspective with fact-based modeling [29], inspired by the same work referenced by Codd [18]. As facts involve real-world concepts rather than database constants (*things, not strings* [44]), tuples should store unique, context-independent representations of these concepts. In the above example, *Underhill* is the area on the Isle of Portland in Dorset (UK), not the town in Wisconsin nor the travel name of Frodo Baggins; the database should be able to distinguish between them all. The combination of indivisibility of facts and the *things, not strings* paradigm gives rise to Rel's *graph normal form (GNF)*, as we explain below.

Indivisibility of facts. Traditional modeling methodologies take a record-based perspective, in which a tuple may represent an entity with multiple attributes, such as a person with a date and place of birth. One can reconcile this with Rel's fact-based perspective, and ensure the indivisibility of facts principle, by assuming a higher degree of normalization, by enforcing *sixth normal form (6NF)* [21]. Indeed, GNF requires each relation to be in 6NF, which means that:

- the set of all its columns is its unique key, or
- the set of all its columns except one is its unique key.

We can view such a relation as either a set of distinct composite keys $\bar{k}$, or a set of key-value pairs $(\bar{k}, v)$ representing a function that maps keys $\bar{k}$ to atomic values v. (Indeed, in Rel we assume that if there is a non-key column, it is the last one.)

For instance, consider a simple conceptual model shown below as an Entity-Relationship (ER) diagram.



Orders may involve multiple products ordered in some quantities. Multiple payments can be made for each order. This conceptual schema leads to the following GNF database schema (key attributes are underlined):

$\texttt{ProductPrice(}\underline{\texttt{product}}\texttt{, price)}$
$\texttt{ProductName(}\underline{\texttt{product}}\texttt{, name)}$
$\texttt{OrderCustomer(}\underline{\texttt{order}}\texttt{, customer)}$
$\texttt{OrderProductQuantity(}\underline{\texttt{order}}\texttt{,}\underline{\texttt{product}}\texttt{,quantity)}$
$\texttt{PaymentAmount(}\underline{\texttt{payment}}\texttt{,amount)}$
$\texttt{PaymentOrder(}\underline{\texttt{payment}}\texttt{,order)}$

Note that if we had a relation $\texttt{Product(}\underline{\texttt{product}}\texttt{, name, price)}$, it would not be in GNF, as neither `name` nor `price` are key attributes. This is why we have two relations: `ProductName` and `ProductPrice`, which store atomic facts about products.

Things, not strings. GNF relies on a conceptual model that distinguishes between *entities* (products, orders, etc.) and *values* (integers, dates, etc.). In relations, values are represented by themselves as usual, whereas entities are represented by internal *identifiers.* In GNF, identifiers are disjoint from values and every entity in the database is represented by an identifier that is unique within the entire database. So, GNF does not allow disjoint concepts, such as product and order, to have the same identifier in the database. We call this the *unique identifier property.*

In summary, *graph normal form (GNF)* comprises the following conditions:

(1) for each k-ary relation R,
  - all k columns of R are the key, or
  - the first $k - 1$ columns of R are the key for R;

(2) the unique identifier property holds.

Condition (1) captures indivisibility of facts, and Condition (2) the *things, not strings* paradigm. Relational databases in GNF can be thought of as *Relational Knowledge Graphs.* We explain differences between them and other models of knowledge graphs in Section 6.

Under GNF, each relation is a set. Moreover, there is no need for nulls: rather than using a null in the non-key column, we simply omit the whole tuple (possibly using a separate relation for keys alone). Other benefits of GNF such as semantic stability and support for temporal features are discussed in [3]. This level of normalization also allows dropping column names, because relation names alone (under the naming scheme we use) are sufficiently informative; for instance, compare `Product(product,name,price)` with `ProductName` and `ProductPrice`.

3 Basics

To introduce basic features of Rel, we use the database in Figure 1, which comprises a subset of relations from the example in Section 2. We will use the identifiers "Pmt1", "Pmt2", . . . for payments, "P1", "P2", . . . for products, and "O1", "O2", . . . for orders. These need to be disjoint to meet the *unique identifier property* (condition (2) of GNF), which is illustrated by the identifier naming scheme. Recall that the Rel data model does not associate names with the attributes of relations, and refers to them by their position only. As explained in Section 2, under GNF, SQL columns essentially give rise to separate relations, which means that SQL column names are naturally represented in relation names in Rel (under the adopted naming scheme).

PaymentOrder	
"Pmt1"	"O1"
"Pmt2"	"O2"
"Pmt3"	"O1"
"Pmt4"	"O3"

PaymentAmount	
"Pmt1"	20
"Pmt2"	10
"Pmt3"	10
"Pmt4"	90

OrderProductQuantity		
"O1"	"P1"	2
"O1"	"P2"	1
"O2"	"P1"	1
"O3"	"P3"	4

ProductPrice	
"P1"	10
"P2"	20
"P3"	30
"P4"	40

Figure 1: Example database containing orders, products included in orders (with their amount), and product prices.

3.1 Datalog as a Starting Point

The starting point of Rel is Datalog rules with first-order formulas in their bodies. We assume a very basic level of familiarity with the relational data model (relational algebra and calculus) [4, 41]. A Rel program is a set of rules, the most basic of which has the form

```
def RName( VariableList ) : RelExpression        (1)
```

The rule is structured like a Datalog rule: `RName` is a relation name, *VariableList* is a list of variables, and *RelExpression* is an expression that evaluates to a result that gives meaning to the variables in *VariableList*. For example, consider the rule

```
def OrderWithPayment(y) : exists ((x) | PaymentOrder(x,y))
```

which adds to *OrderWithPayment* the tuples ⟨"O1"⟩, ⟨"O2"⟩, ⟨"O3"⟩, that is, the orders that have received at least one payment. Throughout, we write tuples using angular brackets (for example, ⟨"O1"⟩ is a unary tuple, and ⟨1, 2, 3⟩ is a ternary tuple), and fonts such as `R` for variables and relation names in Rel, while italics are used to refer to an extent *R* of `R`. Rel uses **set semantics**: in the above example, "O1" only occurs once in the result, even though this order received two payments.

Notice that the rule uses an existentially quantified variable `x` that is not used elsewhere. Rel allows anonymous variables, denoted "_", to simplify the syntax. It is equivalent to write

```
def OrderWithPayment(y) : PaymentOrder(_,y)
```

We note that different occurrences of _ can bind to different values. For example,

```
def OrderedProducts(y) : OrderProductQuantity(_,y,_)
```

computes the products that were ordered. In particular, we get ⟨"P1"⟩, ⟨"P2"⟩, ⟨"P3"⟩ as the result.

As usual, repeated variables express join conditions: the rule

```
def OrderedProductPrice(x,y) :
            OrderProductQuantity(_,x,_) and ProductPrice(x,y)
```

takes the ordered products and adds them with their price into *OrderedProductPrice*. This is the set {⟨"P1", 10⟩, ⟨"P2", 20⟩, ⟨"P3", 30⟩}.

Finally, *RelExpression* allows both existential and universal quantification, as well as all Boolean operations. The rule

```
def NotOrdered(x) : ProductPrice(x,_) and
    not exists ((y1,y2) | OrderProductQuantity(y1,x,y2))
```

adds the products that were not ordered to *NotOrdered*. The rule

```
def NotOrdered(x) : ProductPrice(x,_) and
    forall ((y1,y2) | not OrderProductQuantity(y1,x,y2))
```

is equivalent: both add "P4" to *NotOrdered*.

Using wildcards, we can equivalently define *NotOrdered* as

```
def NotOrdered(x) :
    ProductPrice(x,_) and not OrderProductQuantity(_,x,_)
```

This means that _ is equivalent to an anonymous variable that is existentially quantified immediately outside of the atom where it is used. It is possible to restrict the range of quantifiers. For example, if we have a (database or pre-computed) relation V = {"O1, "O2"}, we can write

```
def AlwaysOrdered(x) : ProductPrice(x,_) and
    forall ((o in V) | OrderProductQuantity(o,x,_))
```

to obtain the set of products that were in every order in V. Rel allows Boolean connectives **`implies`**, **`iff`**, and **`xor`** as syntactic sugar, with their usual meanings.

Safety. Notice that negation can lead to *safety* issues. These occur, for example, when we cannot limit the number of results to a query such as

```
def NotP1Price(x) : not ProductPrice("P1",x)
```

It computes the set of prices that product "P1" does not have, which is (conceptually) infinite. Relational calculus and SQL use *range restriction* to circumvent this kind of problem. Specifically, variables can only range over elements in the database or those constructed from database entries by means of expressions, thereby rendering ranges of variables finite. Rel takes a more flexible approach, allowing unsafe subexpressions as long as the whole expression is safe, and reasons about safety of expressions using a set of rules, based on [28]. Since safety is an undecidable condition [4], it is impossible to distinguish all safe and unsafe expressions. Rel takes a conservative approach, ensuring that the engine never attempts to evaluate an expression that could be unsafe.

3.2 Infinite Relations

Rel makes it possible to use (conceptually) infinite relations. For example, `Int(x)` tests if `x` is an integer. Another example is the ternary relation `add`, which contains all triples $\langle x, y, z\rangle$ of all data types such that $x + y = z$. Using these, one can write rules such as

```
def DiscountedProductPrice(x,y) :
    exists ((z) | ProductPrice(x,z) and add(y,5,z))
```

which computes the *DiscountedProductPrice* relation in which every product received a discount of 5, that is, {⟨"P1", 5⟩, ⟨"P2", 15⟩, ⟨"P3", 25⟩, ⟨"P4", 35⟩}. Again, using such infinite relations requires some care, because queries may be unsafe, i.e., return conceptually infinite results. One such unsafe example is

```
def AdditiveInverse(x,y) : Int(x) and Int(y) and add(x,y,0)
```

Indeed, it asks for the pairs of integers (x, y) whose sum is zero, which is an infinite set. Rel's aforementioned set of safety rules [28] will detect that this expression is potentially infinite. Still such expressions can be written and used in other queries; for example, an expression that intersects `AdditiveInverse` with a finite set will be seen as safe and thus evaluated to produce a finite result.

Arithmetic. Rel supports all standard arithmetic operators such as addition, multiplication, division, modulo, etc., using the standard infix notation. For example, one can write

```
def PsychologicallyPriced(x) :
    exists ((y) | ProductPrice(x,y) and y % 100 = 99)
```

to find products whose prices are 99 modulo 100. Each arithmetic operator has an equivalent "relational" notation. For instance, `add` is the relational notation for +, `multiply` is the relational notation for *, and `modulo` is the relational notation for %.

3.3 Code Flow and Recursion

Analogously to Datalog programs, rules can be written in any order. The ordering of rules in Rel programs has no effect on their semantics. The following small program computes products that are ordered together with some expensive product. The rules are easiest to understand from top to bottom, but the program would compute the same result if the rules were ordered differently.

```
def SameOrder(p1, p2) :
    exists((order) | OrderProductQuantity(order, p1, _)
                 and OrderProductQuantity(order, p2, _))
def SameOrderDiffProduct(p1, p2) :
    SameOrder(p1, p2) and p1 != p2
def Expensive(p) :
    exists ((price) | ProductPrice(p,price) and price > 15)
def BoughtWithExpensiveProduct(p) :
    exists((x in Expensive) | SameOrderDiffProduct(x, p))
```

Here, `SameOrder` evaluates to the set of pairs of products bought together in the same order; `SameOrderDiffProduct` limits those to pairs of distinct products and evaluates to $\{\langle"P1","P2"\rangle, \langle"P2","P1"\rangle\}$. Then, `Expensive` evaluates to all products whose price is more than 15, and `BoughtWithExpensiveProduct` evaluates to the set of products that were bought together with an expensive product ("P1").

Recursion. Being squarely rooted in Datalog, Rel allows recursion. For example, assume that we have a binary relation E of edges in some graph. Then the program

```
def TC_E(x,y) : E(x,y)
def TC_E(x,y) : exists((z) | E(x,z) and TC_E(z,y))
```

computes the transitive closure of `E`, that is, the node pairs x, y such that y is reachable from x using the edges in E. We note that recursion in Rel does not need to be linear, that is, `TC_E` is allowed to occur multiple times on the same right-hand side of a rule.

Rules Defining the Same Relation Name. Rel allows multiple rules with the same relation name on the left-hand side, such as our example for `TC_E`. The semantics of multiple such rules is similar to Datalog. Indeed, having two rules such as

```
def ID ( VariableList ) : RelExpression1
def ID ( VariableList ) : RelExpression2
```

is equivalent to

```
def ID ( VariableList ) : RelExpression1 or RelExpression2
```

that is, the union of the results of the two rules.

Giving Meaning to Recursive Rules. Although the program for transitive closure is fairly easy to understand, this is much less so for general recursive programs that involve negation. In general, the semantics is defined based on a *dependency graph* of the program, which is divided into so-called *strata* by non-monotonic operators, such as negation. The semantics of Rel is consistent with the *stratified semantics of Datalog* [1, Chapter 15], but Rel also allows non-stratified programs, see Addendum A.

3.4 Output and Updates

A Rel *query* (or program) is a sequence of Rel rules defining some relations, typically referring to *base relations* stored in the database (e.g., `ProductPrice`, `OrderProductQuantity`, `PaymentOrder`, and `PaymentAmount`). The execution of a query against a database is called a *transaction.* A transaction performs computation using *derived relations* and interacts with the environment using *control relations.* The latter are just reserved relation names, such as **`insert`**, **`delete`**, and **`output`**. They are defined using rules and computed just like any other relation, but they have a special purpose.

The control relation **`output`** specifies query answers: when a query is issued, what gets returned is the computed content of the relation **`output`**. For instance, the query

```
def output(x) : exists( (y) | ProductPrice(x,y) and y > 30)
```

outputs all products whose price exceeds 30. Note that **`output`** is defined as any other relation; its side effect is that its contents are displayed to the user.

A transaction can modify the content of the database using control relations **`insert`** and **`delete`**. Assume we have binary relations `OrderTotal` and `OrderPaid` containing total prices and total payments of orders (these will be defined with the help of aggregation in Section 5.2). The following deletes information about fully paid orders by adding the rule

```
def delete(:OrderProductQuantity,x,y,z) :
     OrderProductQuantity(x,y,z) and
     exists( (u) | OrderPaid(x,u) and OrderTotal(x,u) )
```

and inserts the fully paid orders to the base relation `ClosedOrders` by adding the rule

```
def insert(:ClosedOrders,x) :
    exists( (u) | OrderPaid(x,u) and OrderTotal(x,u))
```

(here the use of : in front of a relation name indicates passing the name of a relation as a parameter). There is no need to declare a new base relation: if `ClosedOrders` does not exist, it will be created on the spot.

When a transaction terminates, changes to the database are persisted, unless the transaction is aborted (for instance, when integrity constraints are violated; more on this in Section 3.5).

3.5 Integrity Constraints

Integrity constraints check whether relations comply with specified requirements. They are specified using **`ic`** and **`requires`** keywords. Common types of constraints are type constraints which are used to ensure that values conform to a type. For instance, to ensure that quantities of ordered products are integers we write

```
ic integer_quantities() requires
    forall((x) | OrderProductQuantity(_,_,x) implies Int(x))
```

where `Int` is the predicate that tests if its argument is an integer To retrieve the entries that violate this constraint, we add a parameter:

```
ic integer_quantities(x) requires
    OrderProductQuantity(_,_,x) implies Int(x)
```

Then `integer_quantities` will be populated with the values x that violate the constraint.

Integrity constraints can be used to express functional dependencies, foreign keys, etc. The following verifies that each product in the `OrderProductQuantity` appears in the `ProductPrice` relation:

```
ic valid_products(x) requires
    OrderProductQuantity(_,x,_) implies ProductPrice(x,_)
```

If a transaction violates a constraint, it is aborted.

4 Enabling Programming in the Large

We now discuss language features that, together with the basics from Section 3, enable Rel to do programming in the large, giving it the power to define libraries. For this, we need mechanisms for passing parameters that are more complex than individual values.

4.1 Tuple Variables

Assume that we have binary relations $R = \{\langle 1,2\rangle, \langle 3,4\rangle\}$ and $S = \{\langle 5,6\rangle\}$, and we want to compute their Cartesian product, defined as the set of tuples $\langle a,b,c,d\rangle$ such that $\langle a,b\rangle \in R$ and $\langle c,d\rangle \in S$. We could write this as

```
def ProductRS(a,b,c,d) : R(a,b) and S(c,d)
```

This works fine and will add two tuples to *ProductRS*. But what if R were ternary and S binary? We would have to write

```
def ProductRS(a,b,c,d,e) : R(a,b,c) and S(d,e)
```

Writing such code for all different arities is not only repetitive and error-prone, but also incomplete, because there is a maximal arity for which the code works. This is why Rel allows *tuple variables*, which are syntactically distinguished from ordinary variables by trailing dots. Using them, we can compute the Cartesian product of `R` and `S` without knowing the arities of the tuples that are in them:

```
def ProductRS(x...,y...) : R(x...) and S(x...)
```

Here, the variables `x...` and `y...` can bind to arbitrary-length parts of tuples (including length zero).

Using tuple variables, one can produce relations of tuples of different arities; for example

```
def Prefix(x...) : R(x...,_...)
```

computes all prefixes of the tuples in `R`. Here, `_...` plays the role of a wildcard for tuples: it matches an arbitrary tuple of arbitrary arity, again including arity zero.

Tuple variables are a powerful tool, especially if combined with recursion; for example,

```
def Perm(x...) : R(x...)
def Perm(x...,a,y...,b,z...) : Perm(x...,b,y...,a,z...)
```

computes all permutations of tuples in `R`, using the fact that each permutation is a product of transpositions.

4.2 Relation Variables

Being able to define `ProductRS` independently of the arities of `R` and `S` is nice, but it would be even better if we could pass `R` and `S` as parameters to a more general operator that computes the Cartesian product. For example, we would like `Product[R,S]` to return the Cartesian product of `R` and `S`. We will achieve this in two steps: the first is *relation variables*, the second is *relational application*.

In Rel syntax, relation variables in the head of rules are indicated by enclosing them in curly brackets:

```
def Product({A},{B},x...,y...) : A(x...) and B(y...)
```

Using this definition, we can write `Product(R,S,a,b,c,d)` to test if $\langle a,b,c,d\rangle$ is in the Cartesian product of `R` and `S`.

Until now, every rule of the form (1) defined a *relation*, that is, a set of tuples. But what about `Product`? In fact, this is also a relation, but it is conceptually second-order. Its first two columns contain (first-order) relations instead of values:

Product

$\{0\}$	$\{0\}$	0	0		
$\{0\}$	$\{1\}$	0	1		
$\{1\}$	$\{0\}$	1	0		
...					
$\{\langle 1,2\rangle, \langle 3,4\rangle\}$	$\{\langle 5,6\rangle\}$	1	2	5	6
$\{\langle 1,2\rangle, \langle 3,4\rangle\}$	$\{\langle 5,6\rangle\}$	3	4	5	6
...					

Notice that *Product* is infinite. It has infinitely many rows, because it contains infinitely many relations in its first column alone. It also has infinitely many columns, because the relations in the first two columns do not have an upper bound on their arity. As such, the tuples in their Cartesian product become arbitrarily long.

4.3 Relational Application

We can already write `OrderProductQuantity("O1","P1",2)` to test if $\langle$"O1", "P1", 2$\rangle \in$ *OrderProductQuantity*. This is the standard notation in our field for *relational atoms* [4, 41]. Another, more general way of understanding this is seeing `OrderProductQuantity` as a Boolean function that, given three input parameters, a, b, c, returns whether $\langle a,b,c\rangle \in$ *OrderProductQuantity*. This principle works for all kinds of relations that we have seen, including second-order relations. For example, using R and S from Section 4.1,

```
Product(R, S, 1, 2, 5, 6)
```

evaluates to true, since the second-order tuple $\langle R,S,1,2,5,6\rangle$ is in the relation *Product*. We just described one form of *relational application* in Rel. We call it *full (relational) application*, because the syntax with `( )` requires that all arguments to the Boolean function are produced in order for it to evaluate.

Rel also supports *partial (relational) application*, which uses `[ ]` instead of `( )`. Partial application returns all suffixes in a relation that are consistent with a given prefix. For example, `OrderProductQuantity["O1"]` evaluates to $\{\langle$"P1", 2$\rangle, \langle$"P2", 1$\rangle\}$, because $\langle$"O1", "P1", 2$\rangle$ and $\langle$"O1", "P2", 1$\rangle$ are the tuples in *OrderProductQuantity* that start with "O1".

The same principle holds for second-order relations. Hence,

```
Product[R, S]
```

evaluates to the Cartesian product of the relations R and S. In fact, Rel has a special notation for this ubiquitous operation: `(R,S)`. For example, `(PaymentOrder,ProductPrice)` is the Cartesian product of `PaymentOrder` and `ProductPrice`, while `("P4",40)` is the relation containing a single tuple $\langle$`"P4", 40`$\rangle$.

Partial application does not need to provide all arguments, and thus evaluates to a relation. If this relation has arity zero, the result is either $\{\langle\rangle\}$ (the relation with the empty tuple) or $\{\}$ (the empty relation). In Rel, these encode Boolean values true and false: true is encoded as $\{\langle\rangle\}$ and false as $\{\}$, respectively (as usual in the relational data model [4]). Therefore, partial application is identical to full application if all arguments are provided.

4.4 Abstraction

Recall, Rel rules are of the form (1): `def RName(`*VariableList*`):` *RelExpression*. Actually, Rel rules have a more general form, which is

`def RName` *Abstraction* (2)

where *Abstraction* can have one of the following forms:

`{ (` *VariableList* `) :` *RelExpression* `}` (3a)

`{ [` *VariableList* `] :` *RelExpression* `}` (3b)

and the outer curly braces can be omitted to allow the form (1). In the general form (2), the result of *Abstraction* is evaluated and added to *RName*. We say *added to*, because there can be multiple rules with the same `RName` on their left hand side. We now explain how *Abstraction* works.

In abstractions of the form (3a), the *RelExpression* on the right hand side should evaluate to a Boolean. In fact, Rel only allows a syntactic subset of *RelExpression* on the right hand side, namely *Formula*, which guarantees evaluation to a Boolean. The form (3b) allows general expressions instead of Boolean conditions right of the colon.

Abstractions of the form (3a) are inspired by *set comprehension* from mathematics where we write, e.g., $\{n \in \mathbb{N} : \exists m \in \mathbb{N} \text{ such that } n = 2m\}$ for the set of even numbers. For example,

```
{(x,y) : OrderProductQuantity(x,"P1",y)}
```

evaluates to the set of orders and quantities of orders of product "P1". Set comprehensions can be used to perform selection and projection, see Section 5.3.1.

The difference with (3b) is that we now use square brackets left of the colon. For example,[1]

```
{[x,y] : (OrderProductQuantity[x], PaymentOrder(y,x))}   (4)
```

works as follows. For each possible pair $\langle v_x, v_y\rangle$ of values for `x` and `y`, we evaluate the expression `(OrderProductQuantity[x], PaymentOrder(y,x))` which gives products and their quantities in the order v_x for which a payment v_y was made. Whenever the result of the evaluation of this expression is a nonempty set S of tuples, we include $\{\langle v_x, v_y\rangle\} \times S$ in the answer. For example, for values v_x = "O1" and v_y = "Pmt1", this set S contains the result of `OrderProductQuantity["O1"]`, i.e., $\{\langle\text{"P1"}, 2\rangle, \langle\text{"P2"}, 1\rangle\}$. Thus, tuples $\langle\text{"O1"}, \text{"Pmt1}, \text{"P1"}, 2\rangle$ and $\langle\text{"O1"}, \text{"Pmt1}, \text{"P2"}, 1\rangle$ will be included in the result. Similar to quantification, we can restrict the range of variables on the left hand side to a finite set. If `V`$= \{\langle\text{"Pmt2"}\rangle, \langle\text{"Pmt4"}\rangle\}$,

```
{[x, y in V] : (OrderProductQuantity[x], PaymentOrder(y,x))}
```

returns only results of the previous query pertaining to payments "Pmt2" and "Pmt4"; i.e., $\langle\text{"O2"}, \text{"Pmt2}, \text{"P1"}, 1\rangle$ and $\langle\text{"O3"}, \text{"Pmt4"}, \text{"P3"}, 4\rangle$.

We call this operation *(relational) abstraction* to draw a parallel with functional abstraction in functional programming languages.

[1]Rel uses the comma as an infix operator to denote the Cartesian product, see Sections 4.3 and 5.3.1.

Indeed, lambda-abstraction $\lambda x.e$ denotes a function that for argument x computes $e(x)$. This function can be viewed as a relation, namely the set of pairs $(x, e(x))$. In relational abstraction, the difference is that e can be an expression that produces a relation instead of a function. The semantics of the abstraction then is the set of tuples $(x, y_1, \ldots, y_k)$ with $(y_1, \ldots, y_k) \in e(x)$.

5 Growing the Language

In this section we discuss how the language constructs in Section 4 enable Rel to grow. We discuss different libraries of Rel, starting with its standard library, and then explaining how to implement relational algebra, linear algebra, and graph analytics operations. The code examples will also illustrate several prominent features of Rel programming. In Section 5.2 we show how Rel does aggregation under set semantics, and explain that bag semantics is not necessary for correct computation of aggregate functions. In Section 5.3.2 we explain how to guard variables by limiting them to a finite domain, and in Section 5.4 we show how to recurse until a stop condition is met. These examples illustrate how features that, under the sublanguage paradigm (e.g., in SQL), would require language extensions can simply be defined as libraries.

5.1 Standard Library

Rel's standard library provides definitions of dozens of commonly used relations, ranging from trigonometric functions, through type and format conversions, to regex matching. Some of these relations are directly implemented in Rel, like dot-join

```
def dot_join({A},{B},x...,y...) :
    exists((t) | A(x...,t) and B(t,y...))
```

which makes the join on the last position of `A` and the first position of `B` (dropping the join position in the result). Another example is left-override

```
def left_override({A},{B},x...) : A(x...)
def left_override({A},{B},x...,v) :
    B(x...,v) and not A(x...,_)
```

which adds to `A` all tuples from `B` whose prefix obtained by dropping the last position does not appear as a prefix of a tuple in `A`.

Others are just wrappers for external implementations, such as

```
def log[x, y] : rel_primitive_log[x, y]
```

These could be treated as language primitives, but in Rel we prefer to think about them as library functions. Note that `add` and `multiply`, mentioned in Section 3 are also library relations defined likewise using primitives `rel_primitive_add` and `rel_primitive_multiply`.

Recall that `add` and `multiply` have corresponding infix operators. These are defined as follows in the library:

```
def (+)(x,y,z) : add(x,y,z)
def (*)(x,y,z) : multiply(x,y,z)
```

Other library relations have infix versions as well; for instance, we can use `.` for `dot_join` and `<++` for `left_override`.

5.2 Aggregation and Reduce

In Rel, aggregation is implemented as part of the standard library relying on a single additional primitive, exposed as a second-order relation `reduce`. It is a ternary relation whose tuples are of the form $\langle F, R, v\rangle$ where F is a binary operator, R is a non-empty relation, and v is a value. The meaning of `reduce(F,R,v)` is that v is the value obtained by "aggregating" the values in the last column of R using the operation F (in the same way the `reduce` or `fold` operation works in many languages). The operations are performed in an arbitrary order, which means that F should be associative and commutative. Otherwise, the result could change from execution to execution. For example, relational aggregates can be defined as

```
def sum[{A}] : reduce[add,A]
def count[{A}] : reduce[add,(A,1)]
def min[{A}] : reduce[minimum,A]
def max[{A}] : reduce[maximum,A]
def avg[{A}] : sum[A] / count[A]
```

where `minimum[x,y]` and `maximum[x,y]` return minimum, resp. maximum of two numbers. Note that `(A,1)` includes 1 at the end of each tuple in A; summing these ones gives the cardinality of A.

Combining relational operators with aggregation we can define further operations such as, for example, `Argmin` that on a set $A = \{(\bar{a}_1, v_1), \ldots, (\bar{a}_n, v_n)\}$ returns tuples $\bar{a}_i$ such that $v_i = \min_j v_j$:

```
def Argmin[{A}] : A.{min[A]}
```

Aggregation is naturally combined with grouping. Suppose we want to sum up all payments for each order. We can then write:

```
def Ord(x) : OrderProductQuantity(x,_,_)
def OrderPaymentAmount(x,y,z) :
     PaymentOrder(y,x) and PaymentAmount(y,z)
def OrderPaid[x in Ord] : sum[OrderPaymentAmount[x]]
```

Note, however, that orders without payments will not be included in the result, as for them `OrderPaymentAmount[x]` will evaluate to the empty set and, consequently, so will `sum[OrderPaymentAmount[x]]`. To include them, one can use left override and write instead

```
def OrderPaid[x in Ord] : sum[OrderPaymentAmount[x]] <++ 0
```

which replaces the empty set by 0.

5.3 Relational and Linear Algebra: Point-Free Extensions

Notations for linear algebra (LA) and relational algebra (RA) as well as languages such as APL [30] and FP [6] allow writing programs using a set of generally useful primitives and avoiding named variables, a style called point-free or tacit programming. The point-free style is also quite popular in business intelligence tools. Rel supports point-free style via libraries rather than language extensions, which we now demonstrate with libraries for RA and LA.

5.3.1 *Relational Algebra.*

A simple example of relations defined in the library are familiar relational algebra operators. Cartesian product is defined as the `Product` relation that we already discussed in Sections 4.2 and 4.3. Furthermore, as we already mentioned in Section 4.2, Rel uses the infix notation `(A,B)` to denote the Cartesian product of `A` and `B`.

The union of two relations can also be defined in the library:

```
def Union({A},{B},x...) : A(x...) or B(x...)
```

Similarly to product, union has a special shorthand for it: `{A; B}`. This way we can build arbitrary relations from constants; e.g.,

```
{(1,2,3) ; (4,5,6) ; (7,8,9)}
```

evaluates to $\{\langle 1,2,3\rangle, \langle 4,5,6\rangle, \langle 7,8,9\rangle\}$. The remaining set operator of relational algebra — difference — is similarly defined:

```
def Minus({A},{B},x...) : A(x...) and not B(x...)
```

A selection operator simply takes a relation and a condition — which could be an infinite set — and returns their intersection:

```
def Select({A},{Cond},x...) : A(x...) and Cond(x...)
```

Consider, for example, a relational algebra expression $\sigma_{A_1=A_2}(R \times S) \cup B$, where relations R and S have attributes A_1 and A_2, respectively. To express this in Rel in point-free style, we first need an infinite set of tuples whose first and second components are equal:

```
def Cond12(x1,x2,x...) : {x1=x2}
```

and then define the entire expression as

```
Union[Select[Product[R,S],Cond12],B]
```

Projection can be easily expressed in Rel using *abstraction*. For instance, the projection of a relation R on the first and third attribute may be expressed as:

```
(x,y) : R(x,_,y,_...)
```

Expressions vs Formulas. We already know that some expressions always evaluate to Boolean values, which are $\{\langle\rangle\}$ (true) and $\{\}$ (false), the only two sets of tuples of arity zero. In Rel, *Formula* is a subclass of *RelExpression* for which we can statically infer that only Boolean values are produced. These expressions can be combined with **and**, **or**, and **not**. Notice that for formulas, **and** is equivalent to the Cartesian product, and **or** to union: F_1 **and** F_2 is the same as (F_1, F_2), while F_1 **or** F_2 is the same as $\{F_1; F_2\}$.

In Rel, if we want to condition the evaluation of a *RelExpression* on the truth of a formula, we can simply write (*RelExpression*, *Formula*). This expression returns the result of *RelExpression* when *Formula* evaluates to true, and { } otherwise. Indeed, taking the Cartesian product of any relation R with $\{\langle\rangle\}$ is R itself, and taking the Cartesian product of R with the empty relation is empty.

Given the importance of such conditioning in queries (essentially corresponding to the WHERE clause in SQL), Rel has syntactic sugar

RelExpression **where** *Formula*

which is equivalent to (*RelExpression*, *Formula*). Expression (4) can then be rewritten as `{[x,y] : OrderProductQuantity[x] where PaymentOrder(y,x)}`.

5.3.2 *Linear Algebra.*

Vectors, matrices, and tensors can be modeled with relations. Vectors are encoded as binary relations: the first column holds a position, and the second holds the value at that position. Matrices are encoded as ternary relations: the first two columns encode a position (row and column), and the third the value. Two such examples of encoding a vector of length 4 and a 2×2-matrix are shown below:

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} \Rightarrow \begin{array}{c|c} \multicolumn{2}{c}{V} \\ \hline 1 & v_1 \\ 2 & v_2 \\ 3 & v_3 \\ 4 & v_4 \end{array} \qquad \mathbf{M} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \Rightarrow \begin{array}{cc|c} \multicolumn{3}{c}{M} \\ \hline 1 & 1 & m_{11} \\ 1 & 2 & m_{12} \\ 2 & 1 & m_{21} \\ 2 & 2 & m_{22} \end{array}$$

The same principle works for tensors: k-dimensional tensors are encoded as $(k+1)$-ary relations, where the first k columns encode the tensor coordinates, and the last one the value.

Given two vectors $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{v} = (v_1, \ldots, v_n)$, their scalar product is $\mathbf{u} \cdot \mathbf{v} = \sum_k u_i v_i$. Rel's definition of this mimics the mathematical definition:

```
    def ScalarProd[{U},{V}] : { sum[[k] : U[k]*V[k]] }
```

Notice the mechanics of this expression: the range of k is guarded by the first columns of U and V. Subexpression `[k] : U[k]*V[k]` evaluates to $\{\langle i, u_i v_i \rangle \mid i \in \{1, \ldots, n\}\}$. The `sum` aggregate, which computes the sum of the values in the last column of the subexpression, therefore indeed computes $\mathbf{u} \cdot \mathbf{v}$. Crucially, by the definition of `sum`, it is applied to the *entire relation* $\{\langle i, u_i v_i \rangle \mid i \in \{1, \ldots, n\}\}$, not its projection on the last column.

Indeed, assume that $\mathbf{u} = (4, 2)$ and $\mathbf{v} = (3, 6)$, which are encoded by $U = \{\langle 1, 4 \rangle, \langle 2, 2 \rangle\}$ and $V = \{\langle 1, 3 \rangle, \langle 2, 6 \rangle\}$. Then the values of k in `U[k]` and `V[k]` are limited to 1 and 2. So, `[k] : U[k]*V[k]` evaluates to $\{\langle 1, 12 \rangle, \langle 2, 12 \rangle\}$ and the sum correctly results in 24.

The same approach makes it easy to define matrix multiplication. Recall that if $\mathbf{M} = \mathbf{A} \cdot \mathbf{B}$, then its entry m_{ij} in the ith row and jth column is $\sum_k a_{ik} \cdot b_{kj}$, as reflected in this Rel definition:

```
 def MatrixMult[{A},{B},i,j] : { sum[[k] : A[i,k]*B[k,j]] }
```

Similarly, $\mathbf{A} \cdot \mathbf{v}$ for a matrix $\mathbf{A}$ and a vector $\mathbf{v}$ is defined in Rel by

```
  def MatrixVector[{A},{V},i] : { sum[[k] : A[i,k]*V[k]] }
```

An advantage of point-free code is that it is more robust under changes of the underlying data. For instance, `Union[R,S]` and `MatrixMult[A,B]`, work independently of the arities of the relations `R` and `S` or the dimensions of the matrices `A` and `B`.

5.4 Graph Library

All Pairs Shortest Paths. Next, we demonstrate code for the all pairs shortest path problem. The following code expects two relation variables `V` and `E`, representing the finite set of vertices and edges (which are pairs of vertices) of some graph.

```
  def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
  def APSP({V},{E},x,y,i) :
    exists ((z in V) | E(x,z) and APSP[V,E](z,y,i-1)) and
    not exists ((j in Int) | j < i and APSP[V,E](x,y,j))
```

The first rule states that the shortest path from a node to itself has length 0. The second rule states that the shortest paths from x to y have length i if there exists an out-neighbor z of x such that the shortest paths from z to y have length $i - 1$ and we have not already discovered paths shorter than i from x to y. We use relational application in `APSP[V,E]`, which returns the set of triples $\langle u, v, k \rangle$ such that the shortest paths from u to v has length k. Although we use quantification over the entire relation `Int`, the Rel engine will not loop over all integers to compute the query.

We can also define `APSP` using aggregation and abstraction:

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,i) :
  i = min[(j) : exists((z) | E(x,z) and APSP[V,E](z,y,j-1))]
```

PageRank. Using the vector and matrix encoding from Section 5.3.2, we can write (a simplified version of) the PageRank algorithm in Rel. The code also illustrates how a Rel program can perform a number of steps until a stopping condition is met.

```
def dimension[{Matrix}] : max[(k) : Matrix(k,_,_)]
def vector[d,i]  : 1.0/d where range(1,d,1,i)
```

```
def abs(x,y) : (x >= 0 and y = x) or (x < 0 and y = -1 * x)
def delta[{Vec1},{Vec2}] : max[[k] : abs[Vec1[k] - Vec2[k]]]

def next[{G},{P}]: {MatrixVector[G,P]}
def stop({G},{P}): {delta[next[G,P],P] < 0.005}

def PageRank[{G}] :
    {vector[dimension[G]] where empty(PageRank[G])}
def PageRank[{G}] : {next[G,PageRank[G]]
    where not empty(PageRank[G]) and not stop(G,PageRank[G])
     }
def PageRank[{G}] : {PageRank[G] where
    not empty(PageRank[G]) and stop(G,PageRank[G])}
```

Here, `empty` is the emptiness test given by `def empty(R) : not exists( (x...) | R(x...))`, and the predicate `range(1,d,1,i)` is true for i= 1, 2, . . . , d. By calling `PageRank[M]`, we now do PageRank until the delta between two consecutive iterations is at most 0.005.

Rel comes with an extensive graph library that provides many common graph algorithms, as well as a library for path finding.

6 Building Relational Knowledge Graphs in Rel

Intelligent applications (IAs) are applications that leverage a variety of AI techniques and reasoners to improve and automate decision making. Building such applications is complex because it requires a combination of disparate technology stacks (e.g., stacks for transactional, analytical, planning, graph, predictive, and prescriptive reasoning), each with its own data management methods and programming paradigm. Combining all these stacks together makes it very difficult and expensive to develop these types of applications. It is in fact so difficult that there exists an industry of companies whose sole purpose is to build and maintain such applications.

The emergence of cloud-native relational databases and Data Clouds built with them (e.g., Snowflake [20]) provides us with an opportunity to rethink the way we build applications. These data clouds are sufficiently scalable to hold all of the data from a given enterprise. There is a growing appreciation for the value of *semantic layers* as the basis for bringing data and semantics together in these data clouds. Semantic layers come in two flavors: those based on dimensional models (e.g., Looker, DBT, AtScale) and those based on knowledge graphs (e.g., Palantir, ServiceNow, Blue Yonder). The former supports analytical tasks and the latter is used for application logic. Semantic layers based on knowledge graphs make it possible to take a *data-centric* approach to application development.

Relational databases in GNF can be thought of as Relational Knowledge Graphs. Throughout the paper, we argued why Rel is ideally suited as a language for a new kind of *relational* knowledge graph. Semantics based on relational knowledge graphs are critical for leveraging LLMs as they are used to provide a description or model of the users' domain of interest. This enables LLMs to answer questions by reasoning over the knowledge graph, by generating queries that leverage a variety of symbolic and neural reasoners.

Indeed, Rel can be used as the modeling language that expresses database queries, the entire business logic for intelligent applications and, as such, all the concepts needed for a semantic layer, using a single programming paradigm. The following components allow us to define the concept of a *relational knowledge graph (RKG)*, which provides both the data description and adds semantics to it:

(1) The *relational data model*;
(2) The *normal form*: graph normal form (GNF) allowing us to define higher-arity relations in simple terms; and
(3) The *language*: Rel, which enables expressing complex reasoning tasks and application logic in a declarative manner using a single programming paradigm.

In a relational knowledge graph (see https://docs.relational.ai/rel/concepts/relational-knowledge-graphs) we model the domain as a set of concepts and relationships between them using graph normal form. In addition, Rel can define derived concepts and relationships that model the application semantics. These can be computed using a mixture of reasoners: rule-based, prescriptive (e.g., SAT solvers, integer or linear programming solvers), and predictive (e.g., GNNs), all of which can be expressed directly in Rel.

Compared to traditional approaches to knowledge graph management such as RDF coupled with SPARQL and OWL [40], or property graphs with Cypher/GQL [23, 26, 27], relational knowledge graphs offer several benefits. Firstly, RKGs naturally capture higher-arity relations via GNF, which are more suited to enterprise applications using relational databases, as opposed to the binary relations of the Semantic Web and property graphs. Secondly, RKG support for view definitions using Rel facilitates accumulating and structuring knowledge — a feature missing in GQL or the Semantic Web stack. Finally, Rel uniquely allows us to integrate a variety of symbolic and neural reasoners beyond traditional database querying. Overall, we believe that Rel makes it possible to define enterprise knowledge graphs based on the relational paradigm, using a single (declarative) paradigm and a relational technology stack.

7 Rel: Past, Present, and Future

Before Rel: Influences. To achieve its key design goal of going beyond the sublanguage paradigm, Rel's design borrows from decades of research in databases and programming languages. It follows the paradigm of a small core expanded by user-written libraries [45]. Key ideas for Rel's core come from Boute's functional language based on predicate calculus [9] and built around four main constructs: identifier, tupling, abstraction, and application.

Much of the computational capabilities of Rel are based on Datalog-style recursion with fewer restrictions than traditional textbook Datalog and even SQL's recursive CTE. Rel's design was influenced by commercial Datalog implementations such as LogiQL of LogicBlox [3], Soufflé [31], .QL of Semmle (now GitHub) [5], and Dyna [24], with the latter two reflected in Rel's ability to handle infinite sets and to combine recursion with numerical computations. Updating databases via rules applied to control relations is influenced by Dedalus [2] and Statelog [34].

The use of relation variables in Rel borrows some ideas from Data HiLog [42, 50], a function-free fragment of HiLog [14]. While Rel does not at this point offer the functionality of storing relation names in databases and using them in queries as in [42], its parameter-passing is similar to the mechanism described in [50].

Rel influences are not limited to databases. It incorporates works on building bridges between databases and programming languages including the design of type systems for database queries [11, 48] and using comprehensions as the basis of query languages [10, 49]. Among pure programming language research influences is the ability to overload parameters in abstractions and to apply different definitions depending on types of those parameters, similarly to languages with multiple dispatch [37] such as Julia [7]. In such languages a method can be dynamically dispatched based on runtime types of its arguments. The use of sets and tuples and first-order logic notation in queries are also influenced by SETL, a language for set manipulation that predates relational databases [43].

Rel is also heavily influenced by the fact-based modeling paradigm, particularly by Object-Role Modeling (ORM) [29]. Unlike traditional relational query languages, Rel is geared towards working with facts about abstract ORM-style attribute-free entities rather than records representing ER-style entities with attributes. Unnamed columns, partial application, last-column aggregation, and the null-free set semantics are largely consequences of this. Also the rich language of integrity constraints—in place of a more classical database schema—reflects the ORM philosophy.

The ORM-inspired approach to data modeling entails splitting data into many relations and performing many joins. This can be done without sacrificing performance by embracing factorized representations [39] and worst-case optimal joins [38, 47]; the existence of this toolbox enabled many of Rel's design decisions.

Rel's approach to solving classical mathematical problems shares similarities with AMPL [25], Alloy [46], and SolverBlox [8]. AMPL's integration with solvers for linear, nonlinear, integer, and constraint programming, aligns with Rel's support for mixed-integer programming and satisfiability [22, 35]. Alloy's declarative and relational modeling language and its use of SAT solvers resonate with Rel's focus on modeling business processes and solving predictive and prescriptive analytics problems. SolverBlox incorporated solver abilities into a declarative language by integrating mixed-integer linear programming with LogiQL [3].

Rel Today. Rel is implemented today as a part of RelationalAI's relational knowledge graph management system. This system is available as a co-processor (or native extension) to Snowflake and is available via the Snowflake marketplace. Following the philosophy of *meeting users where they are* and to facilitate adoption of the language, the first access to Rel via Snowflake is provided in the form of a Python library that gives users access to many of Rel's core features described here, as well as Rel-written libraries for tasks such as graph analytics. This reflects the fact that users' adoption does not happen overnight. As advocated by studies on users' adoption of programming languages [13, 36], we endeavor to offer them a rewarding initial experience in a familiar environment (e.g., Python and SQL in Snowflake) that encourages them to explore further and gradually get accustomed to all language features.

Many large enterprises are using Rel to build applications that include fraud detection, taxation, and supply chain management. The entire business logic for these applications is modeled in Rel, leveraging its support for programming in the large. Applications developed in Rel have run faster, scaled to larger data sets, with drastically smaller (up to 95%) code bases, when compared to the legacy applications that they replaced. We see this as a clear sign that the foundational concepts behind Rel deliver.

The Future of Rel. Rel is an evolving language. Its foundations presented here meet its key design goals of allowing programming

```
FOBinding  ::=  Literal | ID | ID... | ID in ID           Formula  ::=  {} | {()}
  Binding  ::=  FOBinding | {ID}                                     |   {Expr}(Argument, ..., Argument)
     Expr  ::=  Literal | ID | ID...                                 |   reduce(&{Expr},&{Expr},?{Expr})
           |    (Expr, ..., Expr)                                    |   Formula and Formula | Formula or Formula
           |    Expr where Formula                                   |   not Formula
           |    {Expr; ...; Expr}                                    |   exists((FOBinding, ..., FOBinding) | Formula)
           |    Formula                                              |   forall((FOBinding, ..., FOBinding) | Formula)
           |    [Binding, ..., Binding] : Expr                       |   (Formula)
           |    (Binding, ..., Binding) : Formula          Argument  ::=  _ | _... | ID... | ?{Expr} | &{Expr}
           |    {Expr}[Argument, ..., Argument]              RelDef  ::=  def ID {Expr}
           |    reduce[&{Expr},&{Expr}]                  RelProgram  ::=  RelDef | RelDef RelProgram
```

Figure 2: Syntax of Rel. Braces around `{Expr}` can often be omitted.

in the large and unlocking the full power of the pure relational model. These foundations are stable, but several new features are under development, with partial support for them already in place.

One direction is modeling dynamic behavior via active transition rules that modify the database state. Essentially, this means adopting ideas from Statelog [34] and Dedalus [2] that reconcile active and deductive databases by means of incorporating state into Datalog rules. Another extension is to fully unlock the potential of syntactic higher-order queries in Data HiLog [42]. In this approach, relations store both data and schema information as relation names indicating where relevant data comes from. Rel has already taken steps in that direction by allowing relation variables, and passing relation names as parameters. We also plan to enhance the type system of Rel, providing it with more complete support for ADTs.

A more speculative direction is to revisit an old idea from 1990s: constraint databases and query languages [32, 33]. Their clean declarative formalism for programming with infinite sets — more general than what Rel currently provides — is especially well-suited for spatio-temporal applications. Back then constraint solvers, which are the backbone of query evaluation for such languages, were not yet sufficiently mature. With their remarkable progress in the past two decades, and Rel embracing constraint solving, it is possible that these ideas will achieve broader adoption.

The development of Rel will also lead to a host of new research problems. One of them is developing operational semantics of query languages, that takes us one step closer to a precise mathematical formalization. Such a formalization can be encoded and verified in a proof assistant. It is also more algorithmic than denotational semantics, and thus closer to the style in which query language standards are written. Another research problem is extending classical database concepts (relational languages, their equivalence, static analyses) to languages with tuple variables. Handling infinite sets in Rel goes far beyond the realm of traditional query safety in relational languages, and understanding what is possible requires much additional research. Translations between Rel and other languages, first and foremost SQL, which must account for Rel's expressivity (e.g., the just mentioned safety issue) are also on the radar.

A Addendum: Formal Semantics of Rel

Rel comes equipped with a formal semantics of its language constructs, described in this section. Figure 2 gives a (slightly simplified) grammar of the logical core of Rel. Two most important syntactic constructs are `Expr`, which evaluate to any relation, and `Formula` which evaluate to Boolean values. `Argument` defines expressions that may be used in application. Keywords and reserved symbols of the language are written in a bold blue font (like **`{}`** or **`def`**) and we use `...` between separating characters for a non-empty list.

Note that Rel allows some flexibility in the syntax from Fig. 2. In particular braces around a rule's body can be omitted if the body is an abstraction. It is also allowed to write *ID* instead of **{***ID***}**, unless it is a rule body or a binding. In all cases braces can be omitted around expressions which are already in braces.

Data Model We assume a set **Values** of constant values, **IDs** of `ID`s from which variables and relation names come, and **IDs**$\cdots$ of `ID...`s for names of tuple variables. Since we deal with first-order and second-order relations, we need to cleanly differentiate between single values and sets that contain a single element. For example, $\langle 1\rangle$ denotes a unary tuple and $\{\langle 1\rangle\}$ denotes a relation containing a single unary tuple. Formally, we define the following.

- The set $\mathbf{Tuples}_1$ of *first-order tuples* contains all elements $\langle r_1,\ldots,r_n\rangle$, where $r_i \in \mathbf{Values}$ for all $i \in [n]$. In particular the empty tuple $\langle\rangle$ belongs to $\mathbf{Tuples}_1$.
- The set $\mathbf{Rels}_1$ of *first-order relations* contains all sets of first-order tuples, that is $\mathbf{Rels}_1 = \mathcal{P}(\mathbf{Tuples}_1)$. Note that $\mathbf{Rels}_1$ contains finite and infinite sets.
- The set $\mathbf{Tuples}_2$ of *second-order tuples* contains all tuples of the form $\langle t_1,\ldots,t_n\rangle$, where $t_i \in \mathbf{Values} \cup \mathbf{Rels}_1$ for every $i \in [n]$. In particular, $\mathbf{Tuples}_1 \subseteq \mathbf{Tuples}_2$.
- The set $\mathbf{Rels}_2$ of *second-order relations* contains all sets of second-order tuples, i.e., $\mathbf{Rels}_2 = \mathcal{P}(\mathbf{Tuples}_2) \supseteq \mathbf{Rels}_1$.

For example, $\langle\{\langle 1,2\rangle,\langle 3,4\rangle\},5\rangle$ is a tuple from $\mathbf{Tuples}_2$ whose first element is $\{\langle 1,2\rangle,\langle 3,4\rangle\} \in \mathbf{Rels}_1$ and the second is $5 \in \mathbf{Values}$.

While Rel programs can work with higher-order relations (in particular with $\mathbf{Rels}_2$), they can only output first-order relations (that is, elements of $\mathbf{Rels}_1$). Also recall that a relation (either in $\mathbf{Rels}_1$ or $\mathbf{Rels}_2$) can contain tuples of different arity.

Semantics of Expressions, Formulas, and Programs. The semantics of Rel expressions (and formulas) is defined with respect to an *environment* μ, which is a partial mapping that maps identifiers from **IDs** to relations from $\mathbf{Rels}_2$ and tuple variables from **IDs**$\cdots$ to singletons in $\mathbf{Rels}_1$ (i.e. relations containing a single element of $\mathbf{Tuples}_1$). Recall here that $\mathbf{Rels}_2$ includes $\mathbf{Rels}_1$, so we can work with both second-order and first-order objects.

In order to handle variable scoping, we define the operation $\mu \oplus \nu$ which extends the environment μ with the variable assignments in ν. If μ and ν consider a common variable, ν takes precedence.

Below, we assume $c \in$ Const and $x, r \in$ **IDs**.

- $\llbracket c \rrbracket_\mu = \{\langle c \rangle\}$
- $\llbracket x \rrbracket_\mu = \mu(x)$
- $\llbracket x\cdots \rrbracket_\mu = \mu(x\cdots)$
- $\llbracket \mathtt{\_} \rrbracket_\mu = \{\langle v \rangle \mid v \in \textbf{Values}\}$
- $\llbracket \mathtt{\_...} \rrbracket_\mu = \textbf{Tuples}_1$
- $\llbracket \mathtt{\{}\mathit{Expr}_1\,\mathtt{;}\,\mathit{Expr}_2\mathtt{\}} \rrbracket_\mu = \llbracket \mathit{Expr}_1 \rrbracket_\mu \cup \llbracket \mathit{Expr}_2 \rrbracket_\mu$
- $\llbracket \mathtt{(}\mathit{Expr}_1\,\mathtt{,}\,\mathit{Expr}_2\mathtt{)} \rrbracket_\mu = \llbracket \mathit{Expr}_1 \rrbracket_\mu \times \llbracket \mathit{Expr}_2 \rrbracket_\mu$
- $\llbracket \mathit{Expr}\ \mathtt{where}\ \mathit{Formula} \rrbracket_\mu = \llbracket \mathit{Expr} \rrbracket_\mu \times \llbracket \mathit{Formula} \rrbracket_\mu$
- $\llbracket \mathtt{[\{}x\mathtt{\}]:}\mathit{Expr} \rrbracket_\mu = \{\langle R \rangle \cdot t \mid R \in \textbf{Rels}_1,\ \ t \in \llbracket \mathit{Expr} \rrbracket_{\mu \oplus \{x \mapsto R\}}\}$
- $\llbracket \mathtt{[}c\mathtt{]:}\mathit{Expr} \rrbracket_\mu = \{\langle c \rangle \cdot t \mid t \in \llbracket \mathit{Expr} \rrbracket_\mu\}$
- $\llbracket \mathtt{[}x\mathtt{]:}\mathit{Expr} \rrbracket_\mu = \{\langle v \rangle \cdot t \mid v \in \textbf{Values},\ \ t \in \llbracket \mathit{Expr} \rrbracket_{\mu \oplus \{x \mapsto \{\langle v \rangle\}\}}\}$
- $\llbracket \mathtt{[}x\ \mathtt{in}\ r\mathtt{]:}\mathit{Expr} \rrbracket_\mu = \{\langle v \rangle \cdot t \mid v \in \textbf{Values},\ \langle v \rangle \in \llbracket r \rrbracket_\mu,\ t \in \llbracket \mathit{Expr} \rrbracket_{\mu \oplus \{x \mapsto \{\langle v \rangle\}\}}\}$
- $\llbracket \mathtt{[}x\cdots\mathtt{]:}\mathit{Expr} \rrbracket_\mu = \{t \cdot t' \mid t \in \textbf{Tuples}_1,\ \ t' \in \llbracket \mathit{Expr} \rrbracket_{\mu \oplus \{x\cdots \mapsto \{t\}\}}\}$
- $\llbracket \mathtt{(}\mathit{Bindings}\mathtt{)}:\mathit{Formula} \rrbracket_\mu = \llbracket \mathtt{[}\mathit{Bindings}\mathtt{]}:\mathit{Formula} \rrbracket_\mu$
- $\llbracket \mathtt{\{}\mathit{Expr}\mathtt{\}}\ \mathtt{[\_]} \rrbracket_\mu = \left\{t \,\middle|\, \langle v \rangle \cdot t \in \llbracket \mathit{Expr} \rrbracket_\mu,\ \ v \in \textbf{Values}\right\}$
- $\llbracket \mathtt{\{}\mathit{Expr}\mathtt{\}}\ \mathtt{[\_...]} \rrbracket_\mu = \left\{t \,\middle|\, s \cdot t \in \llbracket \mathit{Expr} \rrbracket_\mu,\ \ s \in \textbf{Tuples}_1\right\}$
- $\llbracket \mathtt{\{}\mathit{Expr}\mathtt{\}}\ \mathtt{[}x\cdots\mathtt{]} \rrbracket_\mu = \left\{t \,\middle|\, s \cdot t \in \llbracket \mathit{Expr} \rrbracket_\mu,\ \{s\} = \llbracket x\cdots \rrbracket_\mu\right\}$
- $\llbracket \mathtt{\{}\mathit{Expr}_1\mathtt{\}}\ \mathtt{[?\{}\mathit{Expr}_2\mathtt{\}]} \rrbracket_\mu = \left\{t \,\middle|\, \langle v \rangle \cdot t \in \llbracket \mathit{Expr}_1 \rrbracket_\mu,\ v \in \textbf{Values},\ \langle v \rangle \in \llbracket \mathit{Expr}_2 \rrbracket_\mu\right\}$
- $\llbracket \mathtt{\{}\mathit{Expr}_1\mathtt{\}}\ \mathtt{[\&\{}\mathit{Expr}_2\mathtt{\}]} \rrbracket_\mu = \left\{t \,\middle|\, \left\langle \llbracket \mathit{Expr}_2 \rrbracket_\mu \right\rangle \cdot t \in \llbracket \mathit{Expr}_1 \rrbracket_\mu\right\}$
- $\llbracket \mathtt{reduce[\&\{}\mathit{Expr}_1\mathtt{\},\&\{}\mathit{Expr}_2\mathtt{\}]} \rrbracket_\mu = v_1 \otimes \cdots \otimes v_n$ where $\llbracket \mathit{Expr}_1 \rrbracket_\mu$ defines an associative binary operation $\otimes$ and $\llbracket \mathit{Expr}_2 \rrbracket_\mu = \{t_1 \cdot \langle v_1 \rangle, \ldots, t_n \cdot \langle v_n \rangle\}$.

Figure 3: Semantics of Rel expressions.

- $\llbracket \mathtt{\{()\}} \rrbracket_\mu = \{\langle\rangle\}$
- $\llbracket \mathtt{\{\}} \rrbracket_\mu = \varnothing$
- $\llbracket \mathtt{\{}\mathit{Expr}\mathtt{\}}\ \mathtt{(}\mathit{Arg}\mathtt{,}\ \ldots\ \mathtt{,}\mathit{Arg}\mathtt{)} \rrbracket_\mu = \llbracket \mathtt{\{}\mathit{Expr}\mathtt{\}}\ \mathtt{[}\mathit{Arg}\mathtt{,}\ \ldots\ \mathtt{,}\mathit{Arg}\mathtt{]} \rrbracket_\mu \cap \{\langle\rangle\}$
- $\llbracket \mathtt{\{}\mathit{Expr}\mathtt{\}}\ \mathtt{()} \rrbracket_\mu = \llbracket \mathit{Expr} \rrbracket_\mu \cap \{\langle\rangle\}$
- $\llbracket \mathit{Formula}_1\ \mathtt{or}\ \mathit{Formula}_2 \rrbracket_\mu = \llbracket \mathit{Formula}_1 \rrbracket_\mu \cup \llbracket \mathit{Formula}_2 \rrbracket_\mu$
- $\llbracket \mathit{Formula}_1\ \mathtt{and}\ \mathit{Formula}_2 \rrbracket_\mu = \llbracket \mathit{Formula}_1 \rrbracket_\mu \cap \llbracket \mathit{Formula}_2 \rrbracket_\mu$
- $\llbracket \mathtt{not}\ \mathit{Formula} \rrbracket_\mu = \{\langle\rangle\} - \llbracket \mathit{Formula} \rrbracket_\mu$
- $\llbracket \mathtt{(}\mathit{Formula}\mathtt{)} \rrbracket_\mu = \llbracket \mathit{Formula} \rrbracket_\mu$
- $\llbracket \mathtt{exists(\ (}x\mathtt{)\ |\ }\mathit{Formula}\mathtt{\ )} \rrbracket_\mu = \langle\!\langle \text{there is } v \in \textbf{Values} \text{ such that } \llbracket \mathit{Formula} \rrbracket_{\mu \oplus \{x \mapsto \{\langle v \rangle\}\}} = \{\langle\rangle\} \rangle\!\rangle$
- $\llbracket \mathtt{exists(\ (}x\ \mathtt{in}\ r\mathtt{)\ |\ }\mathit{Formula}\mathtt{\ )} \rrbracket_\mu = \langle\!\langle \text{there is } v \in \textbf{Values} \text{ such that } \langle v \rangle \in \llbracket r \rrbracket_\mu \text{ and } \llbracket \mathit{Formula} \rrbracket_{\mu \oplus \{x \mapsto \{\langle v \rangle\}\}} = \{\langle\rangle\} \rangle\!\rangle$
- $\llbracket \mathtt{exists(\ (}x\cdots\mathtt{)\ |\ }\mathit{Formula}\mathtt{\ )} \rrbracket_\mu = \langle\!\langle \text{there is } t \in \textbf{Tuples}_1 \text{ such that } \llbracket \mathit{Formula} \rrbracket_{\mu \oplus \{x\cdots \mapsto \{t\}\}} = \{\langle\rangle\} \rangle\!\rangle$
- $\llbracket \mathtt{forall(\ (}x\mathtt{)\ |\ }\mathit{Formula}\mathtt{\ )} \rrbracket_\mu = \langle\!\langle \text{for all } v \in \textbf{Values} \text{ it holds that } \llbracket \mathit{Formula} \rrbracket_{\mu \oplus \{x \mapsto \{\langle v \rangle\}\}} = \{\langle\rangle\} \rangle\!\rangle$
- $\llbracket \mathtt{forall(\ (}x\ \mathtt{in}\ r\mathtt{)\ |\ }\mathit{Formula}\mathtt{\ )} \rrbracket_\mu = \langle\!\langle \text{for all } v \in \textbf{Values} \text{ such that } \langle v \rangle \in \llbracket r \rrbracket_\mu \text{ it holds that } \llbracket \mathit{Formula} \rrbracket_{\mu \oplus \{x \mapsto \{\langle v \rangle\}\}} = \{\langle\rangle\} \rangle\!\rangle$
- $\llbracket \mathtt{forall(\ (}x\cdots\mathtt{)\ |\ }\mathit{Formula}\mathtt{\ )} \rrbracket_\mu = \langle\!\langle \text{for all } t \in \textbf{Tuples}_1 \text{ it holds that } \llbracket \mathit{Formula} \rrbracket_{\mu \oplus \{x\cdots \mapsto \{t\}\}} = \{\langle\rangle\} \rangle\!\rangle$
- $\llbracket \mathtt{reduce(\&\{}\mathit{Expr}_1\mathtt{\},\&\{}\mathit{Expr}_2\mathtt{\},}\mathit{Expr}_3\mathtt{)} \rrbracket_\mu = \langle\!\langle \llbracket \mathit{Expr}_3 \rrbracket_\mu = \llbracket \mathtt{reduce[\&\{}\mathit{Expr}_1\mathtt{\},\&\{}\mathit{Expr}_2\mathtt{\}]} \rrbracket_\mu \rangle\!\rangle$

where $\langle\!\langle \mathit{Condition} \rangle\!\rangle = \{\langle\rangle\}$ if *Condition* holds, and $\langle\!\langle \mathit{Condition} \rangle\!\rangle = \varnothing$ if *Condition* does not hold.

Figure 4: Semantics of Rel Formulas.

When defining the semantics, we assume quantification with a single binding, application with a single argument, and binary `;` and `,`. The semantics for general expressions can be obtained by straightforward syntactic transformations. We use x for identifiers from **IDs**, $x\cdots$ for elements of **IDs**$\cdots$ and c for constant values from **Values**. Additionally, we use *Expr* to refer to the instances of `Expr` tokens. The semantics of a Rel expression e with respect to an environment μ, denoted $\llbracket e \rrbracket_\mu$, is in Figure 3. The semantics of formulas is in Figure 4. Note that $\llbracket e \rrbracket_\mu$ can be in $\textbf{Rels}_2$ and possibly infinite.

The semantics of programs is defined much like in recursive Datalog programs [4], with an added complication of handling higher-order relations. In broad terms, this is based on the idea of building a *dependency graph* which allows to track the flow of information, as explained in Section 3.3. The information is then propagated in an iterative fashion until no new facts can be inferred.

Disambiguating First- and Second-Order Arguments. The reader may have noticed annotations `?` and `&` in front of arguments that have not been used in any of our examples. These annotations formally disambiguate first- and higher-order arguments. In most production code and real-life examples, such ambiguities never arise and annotations are dropped. As an example of an ambiguous application, consider two rules for the same relation `addUp`.

```
def addUp[{A}] : sum[A]
def addUp[x in Int] : x%10 + addUp[(x-x%10)/10] where x >= 0
```

The first rule sums up the last column of a relation, and the second sums up the digits of a non-negative integer. What should `addUp[{11;22}]` evaluate to? The first rule gives tuple $\langle 33 \rangle$; the second rule gives tuples $\langle 2 \rangle$ and $\langle 4 \rangle$. In such cases, we do not apply both rules to return $\{\langle 2 \rangle, \langle 4 \rangle, \langle 33 \rangle\}$, but rather require disambiguation by indicating explicitly how the argument should be treated. We use `&` to indicate that we are passing a second-order argument, and `?` to indicate a *first-order* (that is, ordinary) argument; tuple variables are passed as first-order arguments by default. That is, `addUp[?{11;22}]` evaluates to $\{\langle 2 \rangle, \langle 4 \rangle\}$ and `addUp[&{11;22}]` evaluates to $\{\langle 33 \rangle\}$. We can drop `&` and `?` if the engine can figure out whether the argument should be passed as first-order or as second-order by examining the definition of the called relation—which is the case in most real-life programs. For the expression `addUp[{11;22}]` the engine would raise an error, as both first-order and second-order arguments can be passed to `addUp`, forcing the programmer to disambiguate by using either `&` or `?`.

Acknowledgments

We are deeply indebted to Martin Bravenboer, who is one of the original designers of Rel and who provided many helpful comments on an early draft of the paper, and to the RelationalAI product and field engineering teams: the compiler team for implementing Rel, the engine team for implementing its runtime, the infrastructure team for cloud deployment, the user experience team for providing a Python based developer experience, and to the knowledge engineering, data science, and other field engineering teams for being early users of Rel and providing invaluable feedback that helped improve it.

References

[1] Serge Abiteboul, Rick Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1371–1382. https://doi.org/10.1145/2723372.2742796

[4] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2022. *Database Theory.* Open source at https://github.com/pdm-book/community.

[5] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPICS.ECOOP.2016.2

[6] John W. Backus. 1978. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *Commun. ACM* 21, 8 (1978), 613–641. https://doi.org/10.1145/359576.359579

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM Review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[8] Conrado Borraz-Sánchez, Diego Klabjan, Emir Pasalic, and Molham Aref. 2018. SolverBlox: algebraic modeling in datalog. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM Books, Vol. 20. ACM / Morgan & Claypool, 331–354. https://doi.org/10.1145/3191315.3191322

[9] Raymond T. Boute. 2005. Functional declarative language design and predicate calculus: a practical approach. *ACM Trans. Program. Lang. Syst.* 27, 5 (2005), 988–1047. https://doi.org/10.1145/1086642.1086647

[10] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Rec.* 23, 1 (1994), 87–96. https://doi.org/10.1145/181550.181564

[11] Peter Buneman and Atsushi Ohori. 1996. Polymorphism and Type Inference in Database Programming. *ACM Trans. Database Syst.* 21, 1 (1996), 30–76. https://doi.org/10.1145/227604.227609

[12] Stephen Cass. 2024. The Top Programming Languages 2024. https://spectrum.ieee.org/top-programming-languages-2024.

[13] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (2021), 98–106. https://doi.org/10.1145/3469279

[14] Weidong Chen, Michael Kifer, and David Scott Warren. 1993. HILOG: A Foundation for Higher-Order Logic Programming. *J. Log. Program.* 15, 3 (1993), 187–230. https://doi.org/10.1016/0743-1066(93)90039-J

[15] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[16] Edgar F. Codd. 1971. A Database Sublanguage Founded on the Relational Calculus. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, USA, November 11-12, 1971*, Edgar F. Codd and Albert L. Dean Jr. (Eds.). ACM, 35–68. https://doi.org/10.1145/1734714.1734718

[17] Edgar F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *Research Report / RJ / IBM / San Jose, California* RJ987 (1972).

[18] Edgar F. Codd. 1979. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.* 4, 4 (1979), 397–434. https://doi.org/10.1145/320107.320109

[19] Edgar F. Codd. 1982. Relational Database: A Practical Foundation for Productivity. *Commun. ACM* 25, 2 (1982), 109–117. https://doi.org/10.1145/358396.358400

[20] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[21] C. J. Date, Hugh Darwen, and Nikos A. Lorentzos. 2002. *Temporal data and the relational model.* Elsevier.

[22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[23] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258. https://doi.org/10.1145/3514221.3526057

[24] Jason Eisner and Nathaniel W. Filardo. 2011. Dyna: Extending Datalog For Modern AI. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Lecture Notes in Computer Science, Vol. 6702. Springer, 181–220. http://cs.jhu.edu/~jason/papers/#eisner-filardo-2011 Longer version available as tech report.

[25] Robert Fourer, David M Gay, and Brian W Kernighan. 1990. AMPL: A mathematical programming language. *Management Science* 36, 5 (1990), 519–554.

[26] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researcher's Digest of GQL. In *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece (LIPIcs, Vol. 255)*, Floris Geerts and Brecht Vandevoort (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:22. https://doi.org/10.4230/LIPICS.ICDT.2023.1

[27] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[28] Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, and Cristina Sirangelo. 2025. Queries with External Predicates. In *28th International Conference on Database Theory, ICDT 2025, March 25-28, 2025, Barcelona, Spain (LIPIcs, Vol. 328)*, Sudeepa Roy and Ahmet Kara (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:20. https://doi.org/10.4230/LIPICS.ICDT.2025.22

[29] Terry A. Halpin and Tony Morgan. 2008. *Information modeling and relational databases (2. ed.).* Morgan Kaufmann.

[30] Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc., USA.

[31] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[32] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. 1995. Constraint Query Languages. *J. Comput. Syst. Sci.* 51, 1 (1995), 26–52. https://doi.org/10.1006/JCSS.1995.1051

[33] Gabriel Kuper, Leonid Libkin, and Jan Paredaens. 2000. *Constraint Databases.* Springer.

[34] Georg Lausen, Bertram Ludäscher, and Wolfgang May. 1998. On Active Deductive Databases: The Statelog Approach. In *Transactions and Change in Logic Databases (Lecture Notes in Computer Science, Vol. 1472)*, Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov (Eds.). Springer, 69–106. https://doi.org/10.1007/BFB0055496

[35] Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52, 8 (2009), 76–82. https://doi.org/10.1145/1536616.1536637

[36] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 1–18. https://doi.org/10.1145/2509136.2509515

[37] Radu Muschevici, Alex Potanin, Ewan D. Tempero, and James Noble. 2008. Multiple dispatch in practice. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 563–582. https://doi.org/10.1145/1449764.1449808

[38] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40. https://doi.org/10.1145/3180143

[39] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[40] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

https://doi.org/10.1145/1567274.1567278

[41] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems (3. ed.)*. McGraw-Hill.

[42] Kenneth A. Ross. 1992. Relations with Relation Names as Arguments: Algebra and Calculus. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, Moshe Y. Vardi and Paris C. Kanellakis (Eds.). ACM Press, 346–353. https://doi.org/10.1145/137097.137905

[43] Jacob T. Schwartz, Robert B. K. Dewar, Ed Dubinsky, and Edith Schonberg. 1986. *Programming with Sets - An Introduction to SETL*. Springer. https://doi.org/10.1007/978-1-4613-9575-1

[44] Amit Singhal. 2012. Things Not Strings. https://blog.google/products/search/introducing-knowledge-graph-things-not/.

[45] Guy L. Steele. 1999. Growing a Language. *High. Order Symb. Comput.* 12, 3 (1999), 221–236. https://doi.org/10.1023/A:1010085415024

[46] Alloy Team. 2005. The Alloy Analyzer 3.0 Beta. http://alloy.mit.edu/index.php. Accessed: November 2024.

[47] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/ICDT.2014.13

[48] Limsoon Wong. 1995. Polymorphic Queries Across Sets, Bags, and Lists. *ACM SIGPLAN Notices* 30, 4 (1995), 39–44. https://doi.org/10.1145/202176.202181

[49] Limsoon Wong. 2000. Kleisli, a functional query system. *J. Funct. Program.* 10, 1 (2000), 19–56. https://doi.org/10.1017/S0956796899003585

[50] Peter T. Wood. 1993. Bottom-Up Evaluation of DataHiLog. In *Rules in Database Systems. Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh, Scotland, 30 August - 1 September 1993 (Workshops in Computing)*, Norman W. Paton and M. Howard Williams (Eds.). Springer, 401–415. https://doi.org/10.1007/978-1-4471-3225-7_24