TypeQL: A Type-Theoretic & Polymorphic Query Language

CHRISTOPH DORN, TypeDB, United Kingdom and University of Oxford, United Kingdom
HAIKAL PRIBADI, TypeDB, United Kingdom

Relational data modeling can often be restrictive as it provides no direct facility for modeling polymorphic types, reified relations, multi-valued attributes, and other common high-level structures in data. This creates many challenges in data modeling and engineering tasks, and has led to the rise of more flexible NoSQL databases, such as graph and document databases. In the absence of structured schemas, however, we can neither express nor validate the intention of data models, making long-term maintenance of databases substantially more difficult. To resolve this dilemma, we argue that, parallel to the role of classical *predicate logic* for relational algebra, contemporary foundations of mathematics rooted in *type theory* can guide us in the development of powerful new high-level data models and query languages. To this end, we introduce a new polymorphic entity-relation-attribute (PERA) data model, grounded in type-theoretic principles and accessible through classical conceptual modeling, with a near-natural query language: TypeQL. We illustrate the syntax of TypeQL as well as its denotation in the Pera model, formalize our model as an algebraic theory with dependent types, and describe its stratified semantics.

CCS Concepts: • **Theory of computation** → **Database query languages (principles)**; **Type theory**; • **Information systems** → *Database design and models*; • **Computing methodologies** → *Knowledge representation and reasoning*.

Additional Key Words and Phrases: query languages, type theory, type-theoretic databases, data models, ER modeling, logic programming, polymorphism, formal semantics

ACM Reference Format:
Christoph Dorn and Haikal Pribadi. 2024. TypeQL: A Type-Theoretic & Polymorphic Query Language. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 110 (May 2024), 27 pages. https://doi.org/10.1145/3651611

INTRODUCTION

Classical foundations of mathematics rely on truth-valued *propositions* to express relationships between objects. This perspective influenced many early developments in computer science including logic programming [12, 37] and relational databases [13], which encode data in "true facts", such as the proposition Marriage(p_1, p_2) that 'persons p_1 and p_2 are married'.

Modern practical mathematical foundations [14, 53] emphasize proof construction over truth values, captured by the pivotal *propositions-as-types* paradigm [55]. From this perspective, all propositions are types with terms representing how a proposition is satisfied: for example, a term m in the *type* Marriage(p_1, p_2) may represent a past marriage event m between p_1 and p_2. Note that we may now have zero, one, or multiple such marriage events between the same p_1, p_2. This seemingly minor change has drastic implications, as it allows for terms to be referenced by other terms or types, thus leading to a compositional theory of structured data and proof: for example, we may now consider the type WitnessName(m) of 'names of witnesses at the marriage m', making explicit reference to a specific marriage m. Note, replacing m with a placeholder variable x, we also obtain the type-theoretic analogue of a *predicate* (i.e., a "proposition with variables"): the resulting type WitnessName(x) with variable x is an example of a so-called *dependent* type.

Authors' addresses: Christoph Dorn, TypeDB, London, United Kingdom, christoph@typedb.com and University of Oxford, Mathematical Institute, Radcliffe Observatory Quarter (550), Oxford, United Kingdom, dorn@maths.ox.ac.uk; Haikal Pribadi, TypeDB, London, United Kingdom, haikal@typedb.com.

Table 1. Type-theoretic view on conceptual modeling

Conceptual modeling	Type Theory
entity type (e.g., type of "persons p")	type *without* dependencies, containing *objects* (e.g., p : Person)
relation type (e.g., type of "marriages m between p_1 and p_2")	type *with* dependencies, containing *objects* (e.g., m : Marriage(p_1, p_2))
attribute type (e.g., type of "witness names n of marriages m")	type *with* dependencies, containing *values* (e.g., n : WitnessName(m))

Mathematical foundations based on *dependent type theory* (DTT) have been highly successfully applied [15, 29, 45, 50], with recent applications also to database theory [9, 10, 20, 51]. However, DTT has arguably found little application in practice, with its perceived abstractness remaining a continued obstruction to its adoption. A core innovation of the present work is to marry type-theoretic ideas with well-established intuitive notions of conceptual data models. The underlying correspondence is illustrated in Table 1, which compares notions from entity-relation-attribute (ERA) modeling [18, §3] to their type-theoretic counterparts. Note that the table makes a practice-oriented distinction between *objects*, i.e., terms that are stored using unique but arbitrary object identifiers (OIDs), and *values*, i.e., terms that are stored in a meaningful pre-defined format representing elements from a specific mathematical domain (such as booleans, integers, strings).

Based on the correspondence in Table 1, our work introduces a new query language, TypeQL, which builds on types in lieu of propositions. Namely, we understand (composite, dependent) types themselves as data retrieval queries: *types declaratively describe the collection of data that is to be retrieved.* This type-theoretic approach enables several important features of TypeQL:

- Queries in TypeQL can be parametric in that they can contain type variables, i.e., variables in types of types. This allows queries to adapt to changes in the user-specified type schema.
- Type schemas can express type polymorphism: this includes, firstly, *type inheritance* polymorphism (types may inherit the specification of a single parent type) and, secondly, *type interface* polymorphism (types may implement multiple interface types, on which other types depend).
- The type system of TypeQL can further be extended with user-specified inference rules that enable rule-based reasoning in analogy to classical logic programming.[1]

Contributions and goals. In this paper we introduce a simple, but high-level, new data model, called the *polymorphic entity-relation-attribute* (**Pera**) model, and formalize it as a type system. In parallel, we introduce a practical type-theoretic query language, **TypeQL**. We describe the denotational semantics of TypeQL in the Pera model and exhibit a computable class of queries for the model following the 'queries as types' paradigm. Importantly, TypeQL's intuitive near-natural syntax makes our model a practical and accessible choice for real-world applications. Concretely, with the development of TypeQL, we aim to address three key needs of modern database engineering:

[1] While we adopt the classical perspective of logic programming here, there is also purely type-theoretic perspective on such rules: namely, one can envision them themselves as dependent types or, rather, as dependent *sub*types (just as proposition ϕ are subtypes of the unit type $\mathbf{1}$). We refer to [17] for a more detailed discussion of this generalization of rules.

(1) *Direct and intuitive modeling of diverse data.* Our polymorphic conceptual approach seamlessly subsumes existing database paradigms, including relational, graph, and document databases (see Remark 2.2). We posit that this makes TypeQL a powerful tool for addressing several common integration tasks and interoperability issues [31, 32, 41, 42].
(2) *Schema extensibility and composability.* Type-theoretic querying and type inference allow queries to be written in generic form using type variables, which makes them robust to changes in the underlying schema by dynamically inferring types from the query's context.
(3) *Semantic integrity of complex data.* An expressive type system provides effective protection against unintentional mistakes, improving long-term maintainability of database applications. Type-theoretic rule-based reasoning provides a further powerful way to keep data in-sync and consistent.

A central goal of our paper is to describe the basic theoretical foundations for the Pera model, enabling future investigations into the complexity and optimization of TypeQL queries. Note, the query language TypeQL is being actively developed and implemented as part of the open source DBMS TypeDB [1]. The intuitiveness and expressivity of TypeQL has resonated with a world-wide userbase, but no formal theory for its approach has so far been proposed—in this paper, we will describe such a theory. We remark that our paper will work with a mathematically idealized core fragment of TypeQL, which we hope can guide future development of the language.

Related work. *Conceptual* (or *semantic*) *data models* [19, 43], such as Chen's ER model [7] and its extensions [54], provide intuitive and expressive representations of domains, but are often less practical to directly formalize and implement, cf. [27]. *Object-oriented databases* are practice-oriented with native support for inheritance [4, 33, 38, 40] but add substantial complexity, which has hampered their scalability and adoption [2, 34–36]. Our approach differs from this by instead building on a set of simple type-theoretic primitives, including type dependency and subtyping. *Deductive databases* [48] combine logic programs and databases, with Datalog [2, §12] [30] being the prominent example. We will adapt several ideas from logic programming to define interpretations of queries, turning type-theoretic derivations into logic program rules, and using negation-as-failure [11] and stratifications [3, 23, 46]. *Formal semantics* are generally recognized as a crucial step in database language design [28], as they provide correctness guarantees for query planning and optimization algorithms. Recent work includes formalizations of, e.g., graph models, both semi-structured [21, 49] and structured [52], RDF models [47], and document models [5]. Our formalization follows a type-theoretic approach, using 'algebraic theories with dependent types' [6] in place of the more traditional multi-sorted algebraic theories.

Preliminaries. We generally focus on outlining key intuitions rather than providing a comprehensive formal discussion. For most technical details, we refer the reader to [17].

ACKNOWLEDGMENTS

We thank Chloé Colson, Swaraj Dash, Sean Moss, and Matthijs Vákár for proof-reading earlier drafts of our manuscript and for providing many insightful comments on its mathematical exposition and type-theoretic soundness. We are deeply grateful to the team at TypeDB including, in particular, Krishnan Govindraj, Vladimir Izmalkov, Joshua Send, Dmitrii Ubskii, and James Whiteside, for in-depth discussions on all practical aspects of TypeQL and many related engineering questions.

1 TYPE THEORY FOR THE WORKING DATABASE THEORIST

Type-theoretic thinking has inspired the design of many practical high-level programming languages. To quote from Harper [29]:

Types are the central organizing principle of the theory of programming languages. Language features are manifestations of type structure. The syntax of a language is governed by the constructs that define its types, and its semantics is determined by the interactions among those constructs. The soundness of a language design—the absence of ill-defined programs—follows naturally.

Type-theoretic syntax is often a close reflection of the semantics of a given application domain. In its purest theoretical form, this is the domain of mathematical foundations, which led to the inception of *dependent type theory*. For our purpose of designing a simple, and yet expressive, type-theoretic data model, we will borrow several ideas from DTT—this includes, in particular, dependent types themselves. In this section, we provide an introduction to these ideas and the design choices for the PERA type system. For details on DTT we refer the reader to existing excellent introductions [8, 26, 39, 45].

1.1 Type-theoretic systems. The notion of a 'type system' is often used in a rather broad sense. In this paper, we will use it in the sense of *type-theoretic systems*, meaning systems of inference rules that allow us to construct a (typed) language. We begin by giving a brief overview of the key syntactic components of such languages.

A *type*, like a set, can be thought of as a formal collection of elements, also called *terms*. If a term t is contained in a type T, we express this using the *typing judgment* $t : T$. In fact, in DTT, types, too, are terms of a type called a *type universe* (or a *type of types*). We denote this type by the special symbol **Type**: the typing judgment $T : \mathbf{Type}$ should thus be read as 'T is a type, i.e., a term in the type of types'. While universes are often themselves terms in yet larger universes, we will not require this: to emphasize the resulting two-level hierarchy, we call **Type** a *meta-type*.

EXAMPLE 1.1 (TYPES AND TERMS). Type systems commonly include types such as the unit type $\mathbf{1}$, the type of booleans $\mathbb{B}$, and a type of natural numbers $\mathbb{N}$. Each type T will come with a notion of well-formed terms t (i.e., the typings $t : T$ constructible in the type system). For example, $\mathbb{N}$ could have terms that are number constants such as $11 : \mathbb{N}$, or arithmetic expressions such as $4 + 7 : \mathbb{N}$.

Type systems not only record types and their terms but also describe *constructs* of types and terms. Such constructs allow us to construct new terms or types from old ones. For example, given two terms 4 and 7 of $\mathbb{N}$, our type system may allow us to construct the term $4 + 7 : \mathbb{N}$.

A common construct is the *tuple* construct which operates on both types and terms. On types, it constructs, given two types T and S, a new type $T \times S$. On terms, we can input existing terms $t : T$ and $s : S$ and construct a term $(t, s) : T \times S$—this directly models how we expect tuples to work.

The PERA type system will comprise two closely related constructs: *ordered T-sets*, i.e., non-repeating T-lists $l = [t_1, ..., t_k]$ whose elements are terms t_i all belonging to the *same* type T, and *typed bags*, i.e., multisets $u = \{t_1 › T_1, t_2 › T_2, ...\}$ whose elements are terms t_i annotated with their *individual* types T_i (note: we use the symbol '›' to avoid confusion when typing the bag $u : P$ itself). Such annotations become necessary since the same term may live in multiple types and since we work with bags up to reordering: e.g., we consider $\{t›T, t›T', s›S\}$ identical to $\{t›T', s›S, t›T\}$.

Besides tuples, a second common construct concerns *functions* between types. Roughly, given a function $f : T \to S$ between types T and S and a term $t : T$, we can construct a term $f(t) : S$. While functions play a key role in programming languages, their role for database languages is naturally more limited. This is because, functions are *expensive to search*. The PERA type system, therefore, limits the usage of functions to three simple, but fundamental, cases: projections, subtypes, and (first-order) dependent types. We detail these below.

(1) *Projections.* Given an ordered set $l = [t_1, ..., t_k]$, our type system will allow projecting to its ith element, written $l(i) = t_i$. We remark that bags, in contrast to ordered sets, are unordered, which makes defining projections trickier. Nonetheless, we will later see how so-called 'canonical projections' are applicable to bags in special cases.
(2) *Subtypes.* Subtypes are usually captured simply as special functions, namely, the injective ones. In the absence of a general notion of functions $T \longrightarrow S$, the TypeQL type system resorts to a direct axiomatization of *subsumptive subtyping* [44, §15.1]: this means that, when T is a subtype of S, written $T \leq: S$, then, for any typed term $t : T$, we may construct the typed term $t : S$ (in functional terms, we may think of this as the image $\mathsf{inj}(t) : S$ of t under a function $\mathsf{inj} : T \longrightarrow S$).
(3) *Dependent types.* Consider a type T and a function F from T to our meta-type **Type**. As before, we can now construct, for each term $t : T$, a term $F(t) : \mathbf{Type}$. But **Type** is the type of types, so $F(t)$ is, in fact, a type which may have terms itself. In the Pera type system, we work with dependent types not via functions $F : T \longrightarrow \mathbf{Type}$ but (as common in DTT) using so-called hypothetical typing judgments $x : T \vdash F(x) : \mathbf{Type}$, which we discuss in the next section.

Example 1.2 (Dependent types). For any number n in the natural number type $\mathbb{N}$, we could define the type $\mathsf{Factor}(n) : \mathbf{Type}$ to contain all factors of n—this makes Factor an example of a dependent type. Of course, for our practical database, real-world modeling examples are more relevant. Consider the following: for any person p : Person, we could define a type $\mathsf{Name}(p)$ collecting all names of p (of which there could be zero or more)—the Pera type system will precisely allow us to make such definitions.

Intuitively, while types may be thought of as sets, dependent types may be thought of as indexed *families* of sets. In fact, we sometimes speak of a type family to mean a dependent type with a single 'index' dependency (such as $\mathsf{Factor}(n)$, which depends on the index n).

Remark 1.1. When working with general types T, note that the symbol T may represent a dependent type (e.g., $\mathsf{Factor}(x)$). The result of substituting all occurrences of a term x by another term s in T will be abstractly denoted by $T[s/x]$ (e.g., when T is $\mathsf{Factor}(x)$, then $T[6/x]$ is $\mathsf{Factor}(6)$).

1.2 Inference rules. So far, we have described types, terms and their constructs in natural language. Formally, a type system is captured by a system of so-called *inference rules* (IRs). Inference rules allow us to iteratively generate new statements, also called *judgments*, for the language defined by our type system. Generally, an IR will be of the following form

$$\frac{\mathcal{J}_1 \quad \mathcal{J}_2 \quad \dots \quad \mathcal{J}_k}{\mathcal{J}}\textsc{RuleName}$$

This IR states that the judgment $\mathcal{J}$ can be *derived* from the judgments $\mathcal{J}_1, \mathcal{J}_2, ..., \mathcal{J}_k$ by the rule RuleName. In our later formal presentation of the Pera model, we use in-line notation for IRs, writing the above as $\langle \mathcal{J}_1, \mathcal{J}_2, ..., \mathcal{J}_k \Rightarrow \mathcal{J} \rangle$, and we usually leave IRs unnamed. We remark that IRs are often given schematically, containing variabilized expressions ('meta-variables') that can be substituted. The following example illustrates this.

Example 1.3 (Inference rules). Consider the IR $\langle T : \mathbf{Type}, S : \mathbf{Type} \Rightarrow T \times S : \mathbf{Type} \rangle$. This entails that the type $T \times S$ is *well-formed* (i.e., $T \times S : \mathbf{Type}$ is derivable) as long as, individually, T and S are well-formed. For example, assume our type system guarantees that the type $\mathbb{N} : \mathbf{Type}$ is well-formed. Substituting $\mathbb{N}$ for both S and T in our IR, we derive that $\mathbb{N} \times \mathbb{N}$, too, is well-formed.

Importantly, an IR can also have no hypotheses, in which case we speak of an *axiom* and write $\langle \mathcal{J} \rangle$. We say a conclusion $\mathcal{J}$ is *specified* to mean it is derivable from an axiom.

1.3 Hypothetical judgments and contexts. Most type systems incorporate variables directly into their syntax in order to express functional dependencies. This is achieved by specialized judgments $\mathcal{J}$, called *hypothetical judgments*, which are of the form $\Gamma \vdash \mathcal{I}$, comprising two ingredients:

(1) Γ is a *context*, i.e., a list of typed variables $x_1 : T_1, ..., x_n : T_n$.
(2) $\mathcal{I}$ is some (non-hypothetical) judgment which may reference the variables specified in Γ.

A judgment $\mathcal{J}$ of the form $\Gamma \vdash \mathcal{I}$ should be read as 'assuming Γ, we know $\mathcal{I}$'.

Example 1.4 (Hypothetical judgments). The judgment $x : \mathbb{N} \vdash \mathsf{Factor}(x) : \mathbf{Type}$ expresses a dependency of the RHS of '$\vdash$' (the judgement that $\mathsf{Factor}(x)$ is a type) on the variable x specified to be of type $\mathbb{N}$ in the context shown on the LHS of '$\vdash$'.

In order to describe well-formed typings and contexts in parallel, two basic kinds of judgments are commonly found in type systems. We adopt these in hypothetical form for our system as follows:

(1) *Typing judgment.* The judgment $\Gamma \vdash t : T$ states that '$t : T$ is a valid typing assuming the typed variables from Γ' (note: in general, *both* t and T may contain variables from Γ).
(2) *Context judgment.* The judgment $\Gamma \vdash \Delta\ \mathsf{Ctx}$ states that 'Δ is a valid context assuming the typed variables from Γ' (note: types in Δ may contain variables from Γ).

Example 1.5 (Contexts). The context $\Delta = \big(x : \mathbb{N}, y : \mathbb{N}, n : \mathsf{Factor}(x), m : \mathsf{Factor}(y), p : (n =_{\mathbb{N}} m)\big)$ assumes, in words, two number variables x and y that share a common factor (note: p is a term of the equality proposition $n =_{\mathbb{N}} m$—we discuss how to interpret *propositions as types* in the next section). In reasonable type systems, this context will be *well-formed* (i.e., the context judgment $\bullet \vdash \Delta\ \mathsf{Ctx}$ is derivable, where $\bullet$ denotes the empty context). Observe that variables may appear in types only *after* being introduced (e.g., x appears in $\mathsf{Factor}(x)$ after being typed as $x : \mathbb{N}$).

We emphasize that, in the Pera type system, *all* judgments in IRs will fall into one of the above two cases. Moreover, in most IRs, the context Γ in such judgments will be a meta-variable that is to be replaced by an actual well-formed context. Since IRs of the form $\langle \Gamma \vdash \mathcal{I}_1, ..., \Gamma \vdash \mathcal{I}_k \Rightarrow \Gamma \vdash \mathcal{I}_0 \rangle$ with all judgments using the *same* variabilized context Γ occur frequently, we simply write them as $\langle \mathcal{I}_1, ..., \mathcal{I}_k \Rightarrow \mathcal{I}_0 \rangle$ instead. We remark that the notation is unambiguous: it is the only case of IRs in which non-hypothetical judgments are used.

Notation 1.1. We use vector notation $\vec{a}$ to denote finite, potentially empty, lists $a_1, ..., a_n$. The notation extends component-wise to composite expressions. For example, by writing $\vec{x} : \vec{T}$, we will mean the list of typings $x_1 : T_1, ..., x_n : T_n$.

Given a context $\Gamma = \vec{x} : \vec{T}$, a *context substitution* replaces the variables $\vec{x}$ by a choice of terms $\vec{t}$. We write $\Gamma[\vec{t}/\vec{x}]$ to denote the resulting list of typings $t_1 : T_1[\vec{t}/\vec{x}], ..., t_n : T_n[\vec{t}/\vec{x}]$ (here, Remark 1.1 applies). Note that, a priori, this construction is purely syntactic, meaning these typings need not be well-formed (i.e., $\bullet \vdash t_i : T_i[\vec{t}/\vec{x}]$ need not be derivable in our type system).

Example 1.6 (Context substitutions). A reasonable substitution $[\vec{t}/\vec{x}]$ for the context Δ from the previous Example 1.5 could be $[(6, 4, 2, 2, \mathsf{refl})/(x, y, n, m, p)]$ which would yield the list of judgements $\Delta[\vec{t}/\vec{x}] = \big(6 : \mathbb{N}, 4 : \mathbb{N}, 2 : \mathsf{Factor}(6), 2 : \mathsf{Factor}(4), \mathsf{refl} : (2 =_{\mathbb{N}} 2)\big)$. Here, refl is a constant that represents a 'proof by reflexivity' [45, §A.2.10]: importantly, starting in the next section, we will ignore proofs in our syntax and, thus, the mechanics of such terms can safely be ignored. The substitution is 'reasonable' in that all of the resulting typings should be well-formed in a reasonable type system: for example, the term 6 is indeed of type $\mathbb{N}$.

1.4 Propositions as Types. A core innovation underlying DTT is the 'Proposition as Types' paradigm, providing a proof-centric view on classical predicate-based logic. To quote Wadler [55]:

Such a synthesis is offered by the principle of Propositions as Types, which links logic to computation. At first sight it appears to be a simple coincidence—almost a pun—but it turns out to be remarkably robust, inspiring the design of automated proof assistants and programming languages, and continuing to influence the forefronts of computing.

When treating propositions as types, *proofs* become the terms of propositions. Moreover, type constructs immediately apply to propositions. For example, the product construct allows us to construct, given proofs $p : \phi$ and $q : \psi$, a combined proof $(p, q) : \phi \times \psi$ (this is rightfully reminiscent of proving the ordinary conjunction $\phi \wedge \psi$). In this way, type theory uniformly captures both set-like types (say, $\mathsf{Factor}(x)$) and propositional types (say, $2 + x =_{\mathbb{N}} 4$).

While in DTT we usually *equate* all proof terms of a proposition, in the Pera type system, for simplicity, we *ignore* proof terms altogether. More precisely, the Pera type system includes a meta-type **Prop** whose terms are propositions $\phi : \mathbf{Prop}$. We ignore proof terms $p : \phi$ in all our syntax, effectively leading to a mixed context approach, in which contexts are lists that mix typings $x : T$ and propositions ϕ: for example, $x_1 : T_1, \phi_2, x_3 : T_3, \phi_4, \phi_5, \ldots$. Our previous discussion of contexts and notation for context substitutions still applies, simply by ignoring all terms of propositions.

Example 1.7 (A taste of TypeQL). Consider the following context, which omits proof terms:

$$\Delta = \big(x : \mathsf{Person}, y : \mathsf{Person}, n : \mathsf{Name}(x), m : \mathsf{Name}(y), (n =_{\mathbb{N}} m)\big)$$

In words, Δ assumes person variables x and y that share a common name. In fact, TypeQL will allow us to express such contexts of variables in a straight-forward manner: assuming Person and Name are valid types specified in the type schema of our database, the following will be a valid TypeQL pattern:

```
$x isa person; $y isa person; $x has name $n; $y has name $m; $n == $m;
```

We will discuss TypeQL patterns in more detail in Section 2.

The example hints at a relation between contexts and database queries—this should be understood as an extension of the ‘Propositions as Types’ paradigm to ‘Queries as Types’, as we now explain.

1.5 Queries as Types. In the relational model, queries are propositions as manifested by relational calculus [2, §5.3]. Using the ‘Propositions as Types’ paradigm, we will instead consider *Queries as Types*. As an example, the type of tuples (x, y) where $x : \mathsf{Person}$ and $y : \mathsf{Name}(x)$ is a query: it queries for tuples of persons x and their (potentially many) names y. Note that, in DTT, this type would be written using the Σ-type construct:

$$\Sigma_{x:\mathsf{Person}} \mathsf{Name}(x)$$

However, for simplicity, the Pera type system avoids explicit usage of Σ-types. Instead, we express dependent tuples by *contexts*[2]. (We also avoid dependent function types, i.e., Π-types, altogether in our queries since querying functions is expensive, as remarked earlier.) Concretely, this means that the above query/type will be expressed by the context

$$x : \mathsf{Person}, y : \mathsf{Name}(x)$$

While we will later formally define ‘Queries as Contexts’ in this way (see Definition 3.1), the reader should keep in mind that this is just a convenient way of realizing the ‘Queries as Types’ approach.

[2]For experts: this is based on the observation that in DTT, dependent pairs $(t, s) : \Sigma_T S$ are equiderivable with substitutions $[t/x, s/y]$ of contexts $x : T, y : S$. We also remark that, in a similar vein, first-order functions $f : \Pi_T S$ are equiderivable with hypothetical typings $x : T \vdash f : S$.

1.6 Data: values vs. objects. Types, like Person and Name(x) in the above query, collect terms (i.e., 'data'). There are two different kinds of such terms in the Pera model, as we now describe.

- *Values.* Values are terms from a pre-defined mathematical domain, like booleans, integers, or strings. The Pera *value system* comprises both value types (e.g., integers '$\mathbb{Z}$') which capture these domains, and operations between them (e.g., addition '+'). The value system, as a subsystem of the Pera type system, will not be of essence to our exposition of the Pera model, and so we will only give a minimal example of it later on. Types in the database's type schema can be defined to contain terms of a specific value type (e.g., the type Name(x) may be defined to contain strings)—such types will be called *attribute types.*
- *Objects.* Objects, unlike values, are not sourced from a pre-defined domain but instead, are abstractly represented by unique *object identifiers* (OIDs). The representation is 'arbitrary' in that the choice of OIDs can be implementation-dependent. Types in the type schema containing objects will be referred to as *object types.* Object types are called *relation types* if they are dependent types, and *entity types* if their are independent types.

1.7 Polymorphism: inheritance vs. interfaces. Subtypes allow terms to be cast between types, see Sec. **1.1**. There are two different usages of subtyping in the Pera model.

- *Inheritance.* Frequently, one wants to capture that one type is a specialization of another type, e.g., Adult may be a specialization of Person. This is a classical case of inheritance polymorphism which, in the Pera type system, can be modeled by specifying the corresponding subtyping Adult $\leq$: Person in our type schema. We remark that the Pera model enforces *single inheritance*, meaning that types have at most one parent type which they inherit from.
- *Interface implementation.* For any user-defined dependent type, say Name(x), the Pera type system abstracts the type of terms t which can be substituted for x by creating a new type for these terms, called an *interface.* For example, in the case of Name(x), we would introduce a new interface, say, the type NameOwner, together with a dependency:

$$x : \mathsf{NameOwner} \vdash \mathsf{Name}(x) : \mathbf{Type}$$

  Now, in order to speak of the names Name(p) of a person p : Person, the interface must be *implemented*, meaning we specify the subtyping Person $\leq$: NameOwner in our type schema. This allows us to cast p : Person as p : NameOwner, which entails that Name(p) is a valid type.

The usage of interfaces is unique to the Pera type system and motivated by real-world modeling challenges: since a single interface may be implemented by multiple types, it enables expressing polymorphic type dependencies. For example, we may additionally specify City $\leq$: NameOwner, which allows us to also consider the names Name(c) of cities c : City. In the Pera model, object types can implement interfaces (in fact, any number thereof), but attribute types cannot. Moreover, an attribute type will depend on only a *single* interface, called its *ownership*, while a relation type (i.e., dependent object type) may depend on *multiple* interfaces, called its *roles.*[3]

Remark 1.2 (Polymorphic foreign keys). While the previous example of the NameOwner interface concerned the *attribute* type Name(x), interfaces for *relation* types are also highly useful in practice. For example, they allow us to model what, in relational modeling, could be reasonably described as 'join tables with foreign keys from multiple tables'.

[3]These choices avoid ambiguity for practical modeling tasks: e.g., the ambiguity between n-ary attributes (say, 'distance of cities x and y') and unary attributes of n-relations (say, 'distance of a path p between city x and y'). A detailed discussion of the motivations behind these choices is beyond our current scope.

2 TYPEQL BY EXAMPLE

Having learned about the individual type-theoretic ingredients of the PERA type system, in this section, we will see an example of how they work in concert by describing a complete database in TYPEQL. Since the PERA syntax aims to produce high-level, natural-language-like code, code examples usually need minimal textual explanation. We will highlight the correspondence to earlier type-theoretic ideas in selected cases. Key conventions of TYPEQL code include: all variables are prefixed with '$', comments start with '#', and completed statements end with ';'. As a general syntactic construction, TYPEQL allows the contraction of statements `subj phrase1; subj phrase2;` into a single statement `subj phrase1, phrase2;`. Value terms will be restricted to strings (`"abc"`), integers (`12`), and booleans (`true`). In this paper, object terms will be represented by OIDs that are written using human-readable labels prefixed with '&'.

A. Type schema

```
define
  user sub entity,
    owns username,
    plays post:author,
    plays collab_post:contributor;
  moderator sub user,
    plays review:reviewer,
    plays collab_post:lead_author;
  thread sub entity,
    owns title,
    plays post:parent;
  post sub relation,
    relates author,
    relates parent,
    owns visibility,
    owns title,
    owns content,
    plays review:submission;
  collab_post sub post,
    relates lead_author as author,
    relates contributor @card(10);
  review sub relation,
    relates reviewer,
    relates submission,
    owns score;
  username sub attribute,
    value string;
  title sub attribute,
    value string;
  content sub attribute,
    value string;
  score sub attribute,
    value long;
  visibility sub attribute,
    value boolean;
```

B. Data and rules

```
insert
  $bob isa user,
    has username "Bob23";
  $ana isa moderator,
    has username "Ana_ACM";
  $rules isa thread,
    has title "Forum Rules";
  $rule isa collab_post,
    with (lead_author: $ana,
          contributor: $bob,
          contributor: $ana,
          parent: $rules),
    has title "Rule 1",
    has content "you must say hi";
  $bobs_message isa post,
    with (author: $bob,
          parent: $rules),
    has content "hi ana! u r cool";
  $anas_reaction isa review,
    with (reviewer: $ana,
          submission: $bobs_message),
    has score 50;

define rule post_visibility:
  when {
    $post isa post,
      with (author: $user);
    { $user isa moderator; }
    or
    { $r isa review,
        with (submission: $post);
      $r has score >= 10; };
  } then {
    $post has visibility true;
  }
```

C. Queries

```
# Query 1
match
  $user isa user, has username $name;
# Query 2
match
  $user isa user, has $attribute;
  $attribute isa $Type_of_Attr;
# Query 3
match
  { $text isa title,
      contains "Forum"; }
  or
  { $text isa content,
      contains "hi"; };
# Query 4
match
  $post isa post, with (author: $author);
  $post has visibility true;
  try { $post with (contributor: $user);
        not { $user is $author; }; };
# Query 5
match
  $r isa review,
    with (reviewer: $a,
          submission: $post);
  $s isa review,
    with (reviewer: $a,
          submission: $post);
  $r has score $num;
  $s has score != $num;
# Query 6
match
  $r with (reviewer: $a, submission: $post);
  not { $s with (reviewer: $b, submission: $post);
        not { $a is $b; }; };
```

Fig. 1. A TYPEQL database for a simplified 'pre-moderated forum' and selected TYPEQL queries

2.1 Type schema. TYPEQL's schema specifications comprise the following general ingredients. Firstly, schemas specify *entity* types (e.g., `ent_typ sub entity`), *relation* types (e.g., `rel_typ sub relation`) depending on one or more *role* interface (e.g., `rel_typ relates rol_typ`), and *attribute* types (e.g., `att_typ sub attribute`) whose single ownership interface is kept tacit. Secondly, they specify the respective *inheritance* hierarchies of entity, relation, and attribute types (e.g., `typ1 sub typ2`), as well as interface implementations of both role interfaces (e.g., `typ plays rol_typ`) and ownership interfaces (e.g., `typ owns att_typ`).

Panel **A** of Figure 1 illustrates the *type schema* of a toy database for a pre-moderated forum. The first line begins by specifying a new entity type, by stating `user sub entity`. Under the hood, the Pera type system will translate this specification into an axiom of the form $\langle \texttt{user} : \mathbf{Ent} \rangle$. The type **Ent** is, in fact, a short-hand for the type $\mathbf{Obj}(\{\})$ where, more generally, $\mathbf{Obj}(\{I_1, ..., I_k\})$, for $k \geq 0$, represents the 'type of all object types whose interfaces are $\{I_1, ..., I_k\}$'. Of course, since $\{\}$ is the empty bag, **Ent** precisely collects the object types without any interfaces and, thus, without dependency on other types—in other words, it collects entity types (see Sec. **1.6**). From the given axiom, our type system will then be able to derive, for example:

$$\bullet \vdash \texttt{user} : \mathbf{Type} \quad .$$

In words, without any required assumptions, `user` is a well-formed type in our type system.

Next, consider the specification `moderator sub user`. Under the hood, this now corresponds to two axioms: first, as before, we introduce an entity type $\langle \texttt{moderator} : \mathbf{Ent} \rangle$. Secondly, we also introduce the subtyping axiom $\langle \texttt{moderator} \leq: \texttt{user} \rangle$. From these axioms, our type system will allow us to then derive, for example, the hypothetical judgment:

$$x : \texttt{moderator} \vdash x : \texttt{user} \quad .$$

In words, assuming x is a `moderator`, then x is also a `user`.

Progressing through panel **A**, we next consider the specification `post sub relation, relates author, relates parent`. This translates, firstly, to the axiom $\langle \texttt{post} : \mathbf{Obj}(\{\texttt{author}, \texttt{parent}\}) \rangle$ stating `post` is an object type with dependency on the role interfaces `author` and `parent`. Secondly, we need to specify said interfaces, which is achieved by axioms $\langle \texttt{author} : \mathbf{Itf} \rangle$ and $\langle \texttt{parent} : \mathbf{Itf} \rangle$, where the special type **Itf** represents the 'type of all interfaces' in our system. From these axioms, our type system will be able to infer, in particular, the following more familiar judgment:

$$x : \texttt{author}, y : \texttt{parent} \vdash \texttt{post}(\{x{\rangle}\texttt{author}, y{\rangle}\texttt{parent}\}) : \mathbf{Type} \quad .$$

In words, for each post `author` x and each `parent` section y we have a type of 'posts by x in y' in our type system. Observe that $\texttt{post}(u)$ depends on the bag term $u = \{x{\rangle}\texttt{author}, y{\rangle}\texttt{parent}\}$: recall from **1.1**, this is an unordered bag of terms annotated with their types, and, thus, identical to, say, $\{y{\rangle}\texttt{parent}, x{\rangle}\texttt{author}\}$. As an aside, note that we model `post` as a relation type mainly for illustrative purposes. One could alternatively conceive it as an entity type. Importantly, by choosing `post` to be a relation type, objects in `post` can be instantiated with direct references to an `author` object and `parent` object, as we will see later in our discussion of data insertion.

We can also specify subtypes of relation types: consider the specification `collab_post sub post, relates lead_author as author, relates contributor @card(10);` which specifies a new relation type $\langle \texttt{collab\_post} : \mathbf{Obj}(\{\texttt{lead\_author}, \texttt{parent}, \texttt{contributor}, ..., \texttt{contributor}\}) \rangle$ together with appropriate specifications of new interfaces. Note, the existing `parent` role interface is tacitly inherited from `post` even though `collab_post relates parent` is not explicitly stated. Note further, the `contributor` role is repeated 10 times, corresponding to the annotation `@card(10)`, meaning `collab_post` objects can be instantiated with up to 10 contributors.[4] We also specify the subtypings $\langle \texttt{collab\_post} \leq: \texttt{post} \rangle$ ('`sub`' statement) as well as $\langle \texttt{lead\_author} \leq: \texttt{author} \rangle$ ('`as`' statement). From these axioms, the IRs in the Pera type system will allow us to derive, for example, that:

$$\begin{aligned} x : \texttt{lead\_author}, a : \texttt{contributor}, b : \texttt{contributor}, y : \texttt{parent}, \\ z : \texttt{collab\_post}(\{x{\rangle}\texttt{lead\_author}, a{\rangle}\texttt{contributor}, b{\rangle}\texttt{contributor}, y{\rangle}\texttt{parent}\}) \\ \vdash z : \texttt{post}(\{x{\rangle}\texttt{author}, y{\rangle}\texttt{parent}\}) \end{aligned} \quad .$$

[4]Of course, the number 10 was arbitrarily chosen. We remark that infinite cardinalities are possible in TypeQL (using `@card(*)`) but we omit them here for simplicity.

In words, for any collaborative post z with lead author x, contributors a and b, and parent section y, z is also a post with author x and parent section y. Here, the RHS bag $u = \{x›...y›...\}$ is called the *canonical projection* of the LHS bag $v = \{x›..., a›..., b›..., y›...\}$. We write canonical projections as $u = \pi(v)$: these projections either upcast or remove individual terms in a bag in a *unique* (thus *canonical*) way, but we forego a detailed definition here; see [17] for details.

Further down in panel **A**, we specify the attribute types of our schema. For example, `title sub attribute, value string` translates, firstly, to an axiom $\langle \mathtt{title} : \mathbf{Att}(\{\mathtt{title\_owner}\})\rangle$, which should be read as '`title` is a term in the type of attribute types with ownership interface `title_owner`'. Secondly, we create the required ownership interface by an axiom $\langle \mathtt{title\_owner} : \mathbf{Itf}\rangle$. Thirdly, we add the subtyping $\langle \mathtt{title} \leq: \mathbb{S}\rangle$, where $\mathbb{S}$ is the value type of strings (we remark that, technically, this is a generalized kind of subtyping *of families*, since $\mathtt{title}(x)$ is really a type family with index x; see Remark 3.2). These axioms will allow our type system to derive, e.g., the judgement:

$$x : \mathtt{title\_owner}, t : \mathtt{title}(\{x›\mathtt{title\_owner}\}) \vdash t : \mathbb{S} \quad .$$

In words, when some object x has `title` t, then we know that t is a string.

We remark that, like objects types, attribute types can be organized by inheritance, but we omitted this case from our example for brevity.

Finally, we briefly address implementations of interfaces. In TypeQL, these are specified using the keywords `plays` (used for role interfaces) and `owns` (used for ownership interfaces). For example, the specification `user plays post:author` (which appears, in contracted form, in lines 1-2 of panel **A**) should be read as '`user`s play the role of `author`s for `post`s'.[5] Under the hood, the specification will translate simply to the axiom $\langle \mathtt{user} \leq: \mathtt{author}\rangle$. Similarly, the statement `post owns title` should be read as '`post`s can be owners of `title`s', and translates to the axiom $\langle \mathtt{post} \leq: \mathtt{title\_owner}\rangle$. The latter axiom, for example, will allow our type system to infer

$$x : \mathtt{author}, y : \mathtt{parent}, z : \mathtt{post}(\{x›\mathtt{author}, y›\mathtt{parent}\}) \vdash z : \mathtt{title\_owner} \quad .$$

In words, any `post` z (with author x and parent section y) may own a `title`, i.e., may be cast into a `title_owner` z.

2.2 Data and rules.

Panel **B** of Figure 1 comprises two parts: the insertion of *data* using `insert` and the definition of a new *rule* using `define rule`. We preface our discussion with the remark that the presentation of rules here is directly inspired by classical logic programming principles; however, as mentioned in footnote 1, there is also a 'natively type-theoretic' perspective on rule-based reasoning, as further explained in [17]. Semantically, both approaches are closely related, and, thus, we will stick to the more familiar case of classical rule-based reasoning here.

In a TypeQL `insert` clause, new objects are instantiated using `isa`, references to role-playing objects are made using `with`[6], while attributes are associated to owner objects using `has`. In general, each object instantiation will create a *unique new* OID as a new term in the corresponding object type, while, for attribute types, we store value terms. In our example, we work with the simplification that an insertion of a variable `$var` will create a same-named OID `&var` in the appropriate object type. For example, in Panel **B**, `$bob isa user` corresponds to the axiom $\langle \mathtt{\&bob} : \mathtt{user}\rangle$, specifying that `&bob` is a term in `user`. Similarly, the statement `$bobs_message isa post, with (author: $bob, parent: $rules)` corresponds to the axiom $\langle \mathtt{\&bobs\_message} : \mathtt{post}(\{\mathtt{\&bob}›\mathtt{author}, \mathtt{\&rules}›\mathtt{parent}\})\rangle$. Finally, the statement `$rules has title "Forum Rules"` (which, in contracted form, appears in line 3 of the `insert` clause) corresponds to the axiom $\langle \mathtt{"Forum\ Rules"} : \mathtt{title}(\{\mathtt{\&rules}›\mathtt{title\_owner}\})\rangle$.

[5] Note, we write `post:author` instead of just `author` since, in practical TypeQL, role type identifiers are required to be unique only within their respective relation type hierarchy. However, we will not bother with this detail here. Instead, we will assume all role type identifiers to be globally unique (and thus, '`post:`' may be omitted).

[6] The naming of `with` is still tentative at the time of writing.

Next, consider the `define rule` clause in Panel **B**. In analogy to Datalog, rules in TypeQL allow us to infer new data, specified in a `then {...}` statement, whenever all statements in specified in a `when {...}` block hold for an appropriate substitution of variables. The statements inside the `when` block are collectively called a *pattern*. Substitutions of *patterns* can be formalized using context substitutions (see Sec. **1.3**) after translating patterns into contexts—we'll examine this translation process in more detail when discussing queries below. Note, the rule in panel **B** concretely states the following: for any post `$p` which, either has author `$u` who is a `moderator`, or has been reviewed with a score >10, we infer that `$p` has a `visibility` attribute `true`. Importantly, rules fit well into the framework of type-theoretic inference systems: they can be formalized simply as extensions of our type system by corresponding IRs. However, care needs to be taken when working with *negated* propositions: the precise semantics of rules can then be understood using classical logic program theory; see Section 3.2. For simplicity, we ignore negations in rules in this section.

2.3 Queries.

Finally, panel **C** of Figure 1 illustrates six simple queries that can be run on our database. Queries are indicated using the keyword `match`. What follows after `match` is again a pattern. Every pattern can be formally understood as a (list of) contexts by a process of *denotation* that is detailed in [17]—the process is mostly straight-forward and we illustrate it here by example. For instance, the pattern of **`Query 1`** will denote the following context:

$$\Delta \quad = \quad u : \mathtt{user}, n : \mathtt{username}(\{u \rangle \mathtt{username\_owner}\}) \quad .$$

(Note: variables correspond by their first letter; e.g., `$name` from panel **C** corresponds to n in the context Δ above, and `$user` corresponds to u.) In words, the context Δ assumes a user u and a username n with owner u. The query will retrieve all pairs (s,t) of users with their username.

Let us briefly sketch how this works in type-theoretic terms. Query results of Δ are formally defined to be *well-formed* context substitutions $[(s,t)/(u,n)]$ for Δ, meaning each typing in the resulting list $\Delta[(s,t)/(u,n)]$ is derivable in our type system. Given the specifications made in Panels **A** and **B**, and their corresponding type-theoretic interpretation, it is not too hard to see that there are two well-formed substitutions for our query Δ (and thus, **`Query 1`** has two results):

(1) For the first substitution, we set $(s,t) = (\mathtt{\&ana}, \mathtt{"AnaACM"})$. In this case $\Delta[(s,t)/(u,n)]$ comprises the typings $\mathtt{\&ana} : \mathtt{user}$ and $\mathtt{"AnaACM"} : \mathtt{username}(\{\mathtt{\&ana} \rangle \mathtt{username\_owner}\})$. The former is well-formed since we defined `&ana` to be a term in the type `moderator` which is a subtype of `user`. The latter is well-formed since we defined `&ana` to have `username` `"AnaACM"`.
(2) For the second substitution, set $(s,t) = (\mathtt{\&bob}, \mathtt{"Bob23"})$. As in the first case, one verifies the well-formedness of the resulting typings.

More interestingly, the denotation of **`Query 2`** yields the following context:

$$\Delta \quad = \quad u : \mathtt{user}, I : \mathbf{Itf}, \mathtt{user} \leq: I, T : \mathbf{Att}(\{I\}), a : T(\{u \rangle I\})$$

In words, this context assumes a user u, and some interface I that `user` implements, as well as an attribute type T with ownership interface I, and a value term a in T owned by u. In other words, **`Query 2`** queries for any attribute a that u may have. One verifies that this has the following well-formed substitutions $\Delta[(s,u,v,t)/(u,I,T,a)]$:

(1) $(s,u,v,t) = (\mathtt{\&ana}, \mathtt{username\_owner}, \mathtt{username}, \mathtt{"AnaACM"})$;
(2) $(s,u,v,t) = (\mathtt{\&bob}, \mathtt{username\_owner}, \mathtt{username}, \mathtt{"Bob23"})$.

Two interesting observations apply to **`Query 2`** and its denotation Δ. First, the type-theoretic approach elegantly allows us to variabilize types themselves in queries: indeed, `$Type_of_Attr` in the query (corresponding to T in Δ) is a type variable that is returned as part of the query result. Second, formal and practical queries are not *exactly* the same—indeed, our formal type-theoretic query Δ needed to introduce an *auxiliary* variable I to express its practical counterpart (note:

variables in blue are those which are *not* auxiliary). To describe the results of practical TypeQL queries, we can simply project away auxiliary variables, i.e., we would only return the terms (s, v, t) for **`Query 2`**.

`Query 3` introduces a new keyword, `or`, akin to a traditional sum type. Instead of introducing a new sum type construct $(\Delta_1 + \Delta_2 + \ldots)$, we model such sum types by *lists* of contexts $(\Delta_1, \Delta_2, \ldots)$. Concretely, the denotation of **`Query 3`** will yield two contexts as follows:

$$\begin{aligned}\Delta_1 &= o : \texttt{title\_owner}, t : \texttt{title}(\{o \rangle \texttt{title\_owner}\}), \texttt{"Forum"} \subset t \\ \Delta_2 &= o : \texttt{content\_owner}, t : \texttt{content}(\{o \rangle \texttt{content\_owner}\}), \texttt{"hi"} \subset t\end{aligned}$$

where '$s \subset t$' means 't contains s as strings'. In words, Δ_1 assumes some object o which has `title` t containing `"Forum"`, while Δ_2 assumes some object o which has `content` t containing `"hi"`. Note the variable o is auxiliary in both cases.

A result for the combined query $\vec{\Delta} = (\Delta_1, \Delta_2)$ will precisely be a result for one of its list members Δ_1 or Δ_2. It is not hard to convince oneself that the following are the results for $\vec{\Delta}$:

(1) The substitution $[(\texttt{\&rules}, \texttt{"Forum rules"})/(o, t)]$ for Δ_1;
(2) The substitution $[(\texttt{\&rule}, \texttt{"you must say hi"})/(o, t)]$ for Δ_2;
(3) The substitution $[(\texttt{\&bobs\_message}, \texttt{"hi ana! u r cool"})/(o, t)]$ for Δ_2.

Recall that in order to compute the results for the original **`Query 3`**, we will project away substitutions for auxiliary variables, thus only returning substitutions for t in this case.

Similarly, **`Query 4`** will denote a list of two contexts, but, in order to see this, a bit more work is required. First, any pattern of the form `P; try {Q;}` may be desugared to the more elementary pattern `{P; Q;} or {P; not {Q;};}`. This introduces a new keyword, `not`, called *pattern negation*. The negation `not {Q;}` of a pattern `Q` states that the pattern `Q` *cannot be satisfied*: formally, this means no well-formed substitution of $\Delta_{\texttt{Q}}$ (the denotation of `Q`) exists. In the Pera type system, negated patterns are described by a combination of two more fundamental constructs:

- each proposition $\phi : \mathbf{Prop}$ has a *negation* $\neg\phi : \mathbf{Prop}$, which states 'ϕ does not hold';
- each context Λ has a *satisfaction proposition* $\|\Lambda\| : \mathbf{Prop}$, which states that 'a well-formed substitution of Λ exists'. For experts, recalling from Sec. **1.5** that we think of contexts as composite types, we remark that satisfaction propositions precisely model the usual *propositional truncation* of types; see [45, §3.7].

Having introduced these additional constructs, let us now describe the denotation of **`Query 4`**. Since the denotation of patterns is compositional (though care needs to be taken when re-using variables across patterns, as we will see), we first break up the query pattern into three subpatterns as follows.

```
A = $post isa post, with (author: $author); $post has visibility true;
B = $post with (contributor: $user);
C = not { $user is $author; };
```

A good first guess for the denotation of pattern `A` is:

$$\begin{gathered}\Delta_{\texttt{A}} = a : \texttt{author}, r : \texttt{parent}, p : \texttt{post}\big(\{a \rangle \texttt{author}, r \rangle \texttt{parent}\}\big), \\ v : \texttt{visibility}\big(\{p \rangle \texttt{visibility\_owner}\}\big), v =_{\mathbb{B}} \top\end{gathered}$$

In words, this context assumes an `author` a of a `post` p with (auxiliary) `parent` section r such that p has `visibility` attribute v that equals `true`. However, this denotation misses a subtlety about the Pera model: when specifying data in dependent types some 'interfaces may be left unspecified'. For example, when specifying a `collab_post` object, *not all 10* `contributor` interfaces need to be specified; instead, the number of objects playing the role of `contributor` for that `collab_post` object may range from 0 and 10. Similarly, without imposing further constraints on our type schema in

Panel **A**, the number of `parent` sections for each `post` object may range from 0 to 1. This subtlety is not captured by our context Δ_{A} above: it *only* considers `post`s with exactly one specified parent section r. In order to correct this mismatch, we can use the *typed bag* construct of the Pera type system. Namely, we can rewrite the first line of Δ_{A} to be:

$$\Delta_{\mathsf{A}} = a : \texttt{author}, q : \{\texttt{parent}\}, p : \texttt{post}\big(\{a \rangle \texttt{author}\} \cup q\big), \ldots$$

Now, q is a *typed bag* of bag type $\{\texttt{parent}\}$ and this entails it can either be the empty bag $\{\}$ or a singleton bag $\{r \rangle \texttt{parent}\}$. The operation '$\cup$' represents the union of two typed bags. The precise workings of bags in the Pera type system are spelled out in the next section.

We next consider pattern `B`, and the issue of 'unifying variables' across different patterns. Note, the variable `$post` appears in both patterns `A` and `B`. To resolve this, first, let `B` denote the following context with p renamed to $\tilde{p}$:

$$\Delta_{\mathsf{B}} = u : \texttt{contributor}, w : \{\texttt{parent}, 9 \cdot \texttt{contributor}\}, \tilde{p} : \texttt{collab\_post}\big(\{u \rangle \texttt{contributor}\} \cup w\big)$$

In words, Δ_{B} assumes a `collab_post` $\tilde{p}$ with a `contributor` u and an (auxiliary) bag w comprising a potential parent section and potentially up to 9 contributors. Now, since $\tilde{p}$ and p correspond to the *same* variable `$post`, we would like to equate them: however, since equality is a typed operation, care needs to be taken in which type this comparison can take place. As a generic way to achieve comparability of some given terms $t : T$ and $s : S$, we introduce the following context snippet:

$$\mathsf{is}(t : T, s : S) \quad := \quad X : \mathbf{Type}, T \leq: X, S \leq: X, t =_X s \quad .$$

In words, this assumes a joint supertype X in which $t : T$ and $s : S$ become equal.

In fact, the same snippet can be used to deal with pattern `C`. Abbreviating by A and U the types of a and u previously specified in Δ_{A} and Δ_{B}, respectively, we let `C` denote the context $\Delta_{\mathsf{C}} = \neg\|\mathsf{is}(a : A, u : U)\|$. In words, Δ_{C} assumes that it is not the case that we can find a type T in which a and u become equal. We remark that Δ_{C} is not well-formed on its own (since it doesn't specify the variables a and u), but it will be well-formed as long as we also assume Δ_{A} and Δ_{B}.

Putting the above pieces together, we finally construct the denotation of **`Query 4`** as a list of type-theoretic contexts. First, following our initial remarks, we rewrite the query to be of the form:

```
match {A; B; C;} or {A; not {B; C;};};
```

where `A`, `B`, and `C` are patterns as defined earlier. Abbreviating by C the type of $\tilde{p}$ in Δ_{B} and by P the type of p in Δ_{A}, **`Query 4`** will correspond to the list of contexts:

$$\Delta_1 = \Delta_{\mathsf{A}}, \Delta_{\mathsf{B}}, \mathsf{is}(\tilde{p} : C, p : P), \Delta_{\mathsf{C}}$$
$$\Delta_2 = \Delta_{\mathsf{A}}, \neg\|\Delta_{\mathsf{B}}, \mathsf{is}(\tilde{p} : C, p : P), \Delta_{\mathsf{C}}\|$$

The translation of **`Query 4`** into type-theoretic language is instructive in many ways. Of course, it still falls short of providing a detailed and rigorous procedure for the translation of TypeQL code into a formal type-theoretic system. Such a procedure goes hand-in-hand with the formal specification of TypeQL's syntax and is spelled out in [17]. The formal type-theoretic system of the Pera model, which forms the backend of this translation, will be outlined in the next section.

Exercise 2.1 (More queries). We omit a detailed discussion of the remaining **`Query 5`** and **`6`**, which do not introduce new keywords, but encourage the reader to ponder their semantics. How many results can we expect for **`Queries 4`**, **`5`**, and **`6`**, respectively?[7]

Remark 2.1 (Ordered sets). The example in Figure 1 does not exhaustively cover all aspects of the Pera model. A central omission is the construct of ordered sets. Ordered sets provide a powerful counterpart to bags. Namely, while bags are used to formalize objects with an unordered multiset

[7]Hint: the rule defined in Panel **B** affects results of Query 4. Answers: the queries yield, respectively, 2, 0, and 1 result(s).

Contexts

A. Empty context

1. $\langle \bullet \vdash \bullet \text{ Ctx} \rangle$

B. Types

1. $\langle \Gamma \vdash T : \mathbf{Type} \Rightarrow \bullet \vdash \Gamma, x : T \text{ Ctx} \rangle$
   for fresh $x \in \mathbf{Var}$
2. $\langle \bullet \vdash \Gamma \text{ Ctx} \Rightarrow \bullet \vdash \Gamma, X : \mathbf{Type} \text{ Ctx} \rangle$
   for fresh $X \in \mathbf{Var}$
3. $\langle \bullet \vdash (\Gamma, x{:}T, \Delta) \text{ Ctx} \Rightarrow \Gamma, x{:}T, \Delta \vdash x{:}T \rangle$

C. Propositions

1. $\langle \Gamma \vdash \phi : \mathbf{Prop} \Rightarrow \bullet \vdash \Gamma, \phi \text{ Ctx} \rangle$
2. $\langle \bullet \vdash (\Gamma, \phi, \Delta) \text{ Ctx} \Rightarrow \Gamma, \phi, \Delta \vdash \phi \rangle$

D. Hypothetical Ctx judgments

1. $\langle \Gamma \vdash \Delta \text{ Ctx} \Rightarrow \bullet \vdash \Gamma, \Delta \text{ Ctx} \rangle$
2. $\langle \bullet \vdash (\Gamma, \Delta) \text{ Ctx} \Rightarrow \Gamma \vdash \Delta \text{ Ctx} \rangle$

Propositions

E. Negation

1. $\langle \phi : \mathbf{Prop} \Rightarrow \neg\phi : \mathbf{Prop} \rangle$

F. Satisfaction

1. $\langle \vec{\Delta} \text{ Ctx} \Rightarrow \|\vec{\Delta}\| : \mathbf{Prop} \rangle$
   for $\vec{\Delta} = \Delta_1, ..., \Delta_n$
2. $\langle \vec{\Delta} \text{ Ctx}, \Delta_i[\vec{t}/\vec{x}] \Rightarrow \|\vec{\Delta}\| \rangle$
   for any i, terms $\vec{t}$

G. Equality

1. $\langle t : T, s : T \Rightarrow (t =_T s) : \mathbf{Prop} \rangle$

... IRs for reflexivity, symmetry, transitivity of $=_T$

H. Structural equality

1. $\langle t : T, T =_{\mathbf{Type}} S \Rightarrow t : S \rangle$
2. $\langle \phi, \phi =_{\mathbf{Prop}} \psi \Rightarrow \psi \rangle$

Subtyping

I. Subtypes and subfamilies

1. $\langle T_1 : \mathbf{T}_1, T_2 : \mathbf{T}_2 \Rightarrow T_1 <: T_2 : \mathbf{Prop} \rangle$
   for appropriate $(\mathbf{T}_1, \mathbf{T}_2)$ (see Rmk. 3.2)
2. $\langle T_1 : \mathbf{T}_1, T_2 : \mathbf{T}_2 \Rightarrow T_1 \leq: T_2 : \mathbf{Prop} \rangle$
   for $\mathbf{T}_i$ as before
3. $\langle T_1 <: T_2 \Rightarrow T_1 \leq: T_2 \rangle$

... IRs for reflexivity, transitivity of $\leq:$

J. Subsumption

1. $\langle T, S : \mathbf{Type}, t : T, T \leq: S \Rightarrow t : S \rangle$
2. $\langle T, S{:}\mathbf{Type}, t{:}T, T{\leq:}S, t =_S s \Rightarrow s{:}T \rangle$
3. $\langle T, S : \mathbf{Type}, t : T, T \leq: S, t =_S s \Rightarrow t =_T s \rangle$

Fig. 2. Logical part of the Pera type system

of references to other objects, say, `$path with (edge3: $e, edge1: $f...)`, ordered sets are used to formalize ordered sets of references, say, `$path with (edges[]: $e)`, which can then be accessed sequentially as `$e[0]`, `$e[1]`, etc. We include ordered sets in our formalization in Section 3, but omit an in-depth illustration of their usage for brevity. An example of a type schema comprising ordered interface sets is given in Appendix A.

Remark 2.2 (Comparison to other data models). Data in the Pera model is structured based on two simple ingredients: dependent types for expressing dependencies between data, and subtypes for expressing polymorphism. Nonetheless, the resulting model is powerful enough to, in a natural manner, capture several existing data models—in Appendix A, we substantiate this claim with examples that illustrate how the Pera model compares to other common data models.

3 TYPE SYSTEM AND SEMANTICS

In the previous two sections we discussed, at an intuitive level, the basic type-theoretic ingredients of the Pera type system, how these can be packaged into a practical high-level query language, and how queries can be evaluated. Taken together, these sections cover essentially all key ideas of the Pera model. However, for the purpose of developing a formal theory of the Pera model, in this section we provide a full presentation of its type system. As explained in Section 1, this is defined as a system of inference rules (IRs). In the second part of the section, we then discuss the formal semantics of queries, addressing, in particular, the case of rules containing negated patterns.

Remark 3.1 (System vs. model). Analogous to the ‘relational calculus’ being a specific formal incarnation of the ‘relational model’ (describing both a data model for structuring data and queries for querying the data), we consider the ‘Pera type system’ to be a specific formal incarnation of the ‘Pera model’ (comprising, similarly, both a data model and a definition of queries).

3.1 Type system and algebras

The complete **Pera type system** is shown across Figures 2, 3, and 4. Figure 2 captures the logical constructs (e.g., *contexts, subtyping, negation,* etc.), Figure 3 captures the conceptual constructs (e.g., *object types, attribute types, interfaces, values,* etc.), and Figure 4 handles term collections in the type system. **Pera algebras** are extensions of the Pera type system, comprising additional axioms

Interfaces and values

K. Types of value/interface types

- $\langle \mathbf{T} : \mathbf{Type} \rangle$
  for $\mathbf{T} \in \{\mathbf{Val}, \mathbf{Itf}\}$

L. Type casting

- $\langle T : \mathbf{T} \Rightarrow T : \mathbf{Type} \rangle$
  (for $\mathbf{T}$ as before)

Objects and attributes

M. Type families of obj./attr. types

- $\langle P : \mathbf{IBag} \Rightarrow \mathbf{T}(P) : \mathbf{Type} \rangle$
  for $\mathbf{T} \in \{\mathbf{Obj}, \mathbf{Att}\}$

N. Dependent type casting

- $\langle T : \mathbf{T}(P), u : P \Rightarrow T(u) : \mathbf{Type} \rangle$
  for $\mathbf{T}$ as before

Value system

O. Booleans

1. $\langle \mathbb{B} : \mathbf{Val} \rangle$
2. $\langle \top : \mathbb{B} \rangle$
3. $\langle \bot : \mathbb{B} \rangle$

P. Integers

1. $\langle \mathbb{Z} : \mathbf{Val} \rangle$
2. $\langle n : \mathbb{Z} \rangle$ for $n \in \mathbb{Z}$

Q. Operators

1. $\langle n : \mathbb{Z}, m : \mathbb{Z} \Rightarrow (n \operatorname{op} m) : \mathbb{Z} \rangle$
   for $\operatorname{op} \in \{+, -, *\}$
2. $\langle n : \mathbb{Z}, m : \mathbb{Z} \Rightarrow (n \operatorname{op} m) : \mathbb{B} \rangle$
   for $\operatorname{op} \in \{\equiv, \leqq, \lneq\}$

... IRs for evaluation: $a \operatorname{op} b =_T c$

Other types/operators: strings $\mathbb{S}$, ...

Type abstraction system

R. Object-to-Object and Attribute-to-Attribute casting ("*inheritance*")

- $\langle T : \mathbf{T}(P), S : \mathbf{T}(Q), T \leq: S, r : T(\{\vec{t}\}) \Rightarrow r : S(\pi(\{\vec{t}\})) \rangle$
  for closed $\{\vec{t}\}$, P, Q, and for $\pi(\{\vec{t}\})$ the canonical projection of $\{\vec{t}\}$ to Q, with $\mathbf{T} \in \{\mathbf{Obj}, \mathbf{Att}\}$

S. Object-to-Interface abstraction ("*interface implementation*")

- $\langle P : \mathbf{IBag}, u : P, T : \mathbf{Obj}(P), I : \mathbf{Itf}, T \leq: I \Rightarrow T(u) \leq: I \rangle$

T. Attribute-to-Value abstraction

- $\langle P : \mathbf{IBag}, u : P, T : \mathbf{Att}(P), V : \mathbf{Val}, T \leq: V \Rightarrow T(u) \leq: V \rangle$

Fig. 3. Conceptual part of the Pera type system

Ordered sets of objects

U. Ordered I-set types
("ordered sets of objects from a *single* I")

1. $\langle I : \mathbf{Itf} \Rightarrow I[\,] : \mathbf{Type} \rangle$

V. Terms of ordered I-set types

1. $\langle \vec{t} : I, I : \mathbf{Itf} \Rightarrow [\vec{t}] : I[\,] \rangle$
   for distinct closed terms $\vec{t} = t_1, ..., t_n$, $n \geq 1$
2. $\langle I : \mathbf{Itf}, f : I[\,], i : \mathbb{N}, i \leq_{\mathbb{Z}} |f| \Rightarrow f(i) : I \rangle$
3. $\langle I : \mathbf{Itf}, \vec{t} : I, i : \mathbb{N}, i \leq_{\mathbb{Z}} n \Rightarrow [\vec{t}](i) =_I t_i \rangle$
   for $\vec{t}$ as before

W. Ordered set size function

1. $\langle I : \mathbf{Itf}, f : I[\,] \Rightarrow |f| : \mathbb{Z} \rangle$
2. $\langle \vec{t} : I, I : \mathbf{Itf} \Rightarrow |\,[\vec{t}]\,| =_{\mathbb{Z}} n \rangle$
   for $\vec{t}$ as before

Bags of objects

X. Type of interface bag types

1. $\langle \mathbf{IBag} : \mathbf{Type} \rangle$
2. $\langle P : \mathbf{IBag} \Rightarrow P : \mathbf{Type} \rangle$

Y. Interface bag types
("bags of objects from *multiple* interfaces")

1. $\langle \{\} : \mathbf{IBag} \rangle$
2. $\langle I : \mathbf{Itf}, P : \mathbf{IBag} \Rightarrow \{I\} \cup P : \mathbf{IBag} \rangle$

... analogous rules for ordered I-sets $I[\,]$

Z. Terms of interface bag types

1. $\langle P : \mathbf{IBag} \Rightarrow \{\} : P \rangle$
2. $\langle I : \mathbf{Itf}, P : \mathbf{IBag}, u : P \Rightarrow u : \{I\} \cup P \rangle$
3. $\langle I : \mathbf{Itf}, t : I, P : \mathbf{IBag}, u : P \Rightarrow \{t \rangle I\} \cup u : \{I\} \cup P \rangle$

... analogous rules for ordered I-sets $I[\,]$

Fig. 4. Ordered interface sets and unordered interface bags in the Pera type system

describing a *type schema* and *data* of types, and additional IRs describing *rules*, as illustrated in Section 2. In other words, Pera algebras formally present complete databases. Below, we discuss these components individually, providing brief and selected comments in each case. A more detailed discussion can be found in [17]. For our presentation, we fix disjoint, countably infinite sets **oid** of object identifiers (OIDs), **Tid** of type identifiers (TIDs), and **Var** of variable names.

Logical type system. The IRs for constructing well-formed contexts, together with reflexive hypothetical judgements, are fairly standard. We also introduce propositions for negation and context satisfaction, as well as propositional equality, which were illustrated previously in Sec. **2.3**. We remark that equality is decidable, and thus the usual distinction between definitional and propositional equality need not be made. Note, our subtyping rules are slightly more verbose than usual, firstly, since we introduce $\leq:$ as the transitive closure of another proposition, $<:$, and secondly, since we allow $T_1 \leq: T_2$ for various constellations of terms T_1, T_2 as described in Remark 3.2 below.

Database type system. In Figure 3, we first introduce **Val** and **Itf** for collecting value and interface types, respectively. We also provide a rather minimal example of a value system where, for simplicity, terms are introduced as constants from the meta-theory, and functions are evaluated by equations from the meta-theory. We also introduce a type of object types $\mathbf{Obj}(P)$ (resp. attribute types $\mathbf{Att}(P)$) with interfaces P. Here, P is a *bag of interface types*. This is relevant for object types, for which we may specify multiple interfaces (with repetitions), while for attribute types the bag P must always have cardinality 1 (as discussed in Sec. **1.7**). We also introduce various ways of abstracting types, as further detailed in the next remark.

Remark 3.2 (Families and subtypes). A term $T : \mathbf{Obj}(P)$ (or $T : \mathbf{Att}(P)$) should be thought of as a type *family*: for each bag of objects $p : P$, there is a type $T(p) : \mathbf{Type}$. In order to effectively work with subtype polymorphism, we, therefore, extend the classical notion of subtyping $T_1 \leq: T_2$ from bare types $T_i : \mathbf{Type}$ to more general type families $T_i : \mathbf{T}_i$. In the Pera type system, the following constellations of 'types of type families' $(\mathbf{T}_1, \mathbf{T}_2)$ may be used for subtyping (see IRs **(I)**):

- $(\mathbf{T}_1, \mathbf{T}_2) = (\mathbf{Obj}(P), \mathbf{Obj}(Q))$ (or $(\mathbf{Att}(P), \mathbf{Att}(Q))$). This is the case of *inheritance*, in which we allow objects $t : T_1(p)$ in an object type $T_1 : \mathbf{Obj}(P)$ to be cast into objects $t : T_2(q)$ of another type $T_2 : \mathbf{Obj}(Q)$, for appropriate bags $p : P$ and $q : Q$ (namely, p must canonically project to q).
- $(\mathbf{T}_1, \mathbf{T}_2) = (\mathbf{Obj}(P), \mathbf{Itf})$. This is the case of *interface implementation*, in which an object $t : T(p)$ may be cast into the 'implementer' object $t : T_2$ of an interface $T_2 : \mathbf{Itf}$.
- $(\mathbf{T}_1, \mathbf{T}_2) = (\mathbf{Att}(P), \mathbf{Val})$. This is the case of *value abstraction*, which takes a term $t : T_1(p)$ in an attribute type $T_1 : \mathbf{Att}(P)$, and casts it to its underlying value $t : T_2$ of a value type $T_2 : \mathbf{Val}$.

Type schema, data, rules. In addition to the Pera type system outlined above, each individual Pera algebra features axioms and IRs that capture its type schema, its data, and its rules.

(1) The *type schema* comprises axioms for types, interfaces, inheritance, interface implementations, and value abstractions. For example, given identifiers $E, I \in \mathbf{Tid}$, the axiom $\langle E : \mathbf{Ent} \rangle$ specifies an entity type E, $\langle I : \mathbf{Itf} \rangle$ specifies an interface type I, and $\langle E <: I \rangle$ specifies that E implements I.
(2) *Data* comprises axioms defining objects in object types or values in attribute types. For example, given $\&o \in \mathbf{oid}$, an axiom $\langle \&o : E \rangle$ specifies that $\&o$ is an object in E.[8]
(3) *Rules* are IRs which, given a well-formed substitution $[\vec{t}/x]$ for some fixed context Δ (the *rule body*), derive that $t[\vec{t}/\vec{x}] : T[\vec{t}/\vec{x}]$ is well-formed, for some fixed typing $t : T$ (the *rule head*).

Sec. **2.1** and **2.2** illustrate these specifications. Some mild constraints must necessarily be placed on type and data axioms, e.g., to ensure canonical projections are well-defined. Rules must adhere to conditions that avoid creation of infinite data, which go hand-in-hand with the computability of query results discussed below. We refer the reader to [17] for these technical details.

3.2 Query results and computability

We now fix a Pera algebra $\mathfrak{P}$ as described in the previous section. In analogy with classical database theory, we will say that a context is *generic* if no type or proposition in it contains OIDs.

Definition 3.1. A *query* in $\mathfrak{P}$ is a list of generic contexts that are well-formed in $\mathfrak{P}$.

The semantics of queries relies, in essence, on the traditional semantics of logic programs [22–25], with negation as failure [11]. The basic idea is to cast type-theoretic derivations as logic program rules. We now outline this correspondence of type-theoretic systems and logic programs.

[8]We remark that, technically, this axiom should be written $\langle \&o \leq: E(\{\}) \rangle$. Indeed, $E(\{\}) : \mathbf{Type}$ is the *type* obtained from the *term* $E : \mathbf{Obj}(\{\})$ applied to the empty bag $\{\} : \{\}$ using IR (**N**.2). However, abusing notation, we usually simply denote this type by E itself. We secretly used this convention in previous sections: e.g., we wrote x : `user` in place of x : `user`$(\{\})$.

An *atom* $\mathcal{A}$ is a hypothetical judgment of the form $\bullet \vdash \mathcal{J}$, where $\mathcal{J}$ is either a typing $t : T$ or proposition ϕ and $\mathcal{J}$ does not contain free variables (note, variables in satisfaction propositions $\|...\|$ are considered bound). A *model* $\chi(-)$ for our PERA algebra $\mathfrak{P}$ is a predicate on atoms, subject to the following conditions:

(1) χ is closed under IRs, meaning $\chi(\mathcal{A})$ holds if $\mathcal{A}$ can be derived from $\mathcal{A}_1, ..., \mathcal{A}_k$ via an IR in $\mathfrak{P}$ and each $\chi(\mathcal{A}_i)$ holds,
(2) exactly one of $\chi(\bullet \vdash \phi)$ and $\chi(\bullet \vdash \neg\phi)$ holds.

Note, we can think of χ as a model of a ground normal logic program in the usual sense, with atoms $\mathcal{A}$, and rules as in (1), *up to* identifying the negated atoms $\neg(\bullet \vdash \phi)$ with the atoms $(\bullet \vdash \neg\phi)$.

A simple way to ensure a unique minimal model exists is to apply the standard notion of *stratifications* [3, 23, 46]. The novelty here, is to port this notion to our type-theoretic setting. Moreover, to ensure queries yield *finitely* many results, care needs to be taken since our type system does contain types with infinitely many terms: variables in these types must be *bounded* to avoid infinitely many possible substitutions. Both stratifications and boundedness are discussed in detail in [17]. Assuming stratifiability and boundedness are enforced on the rules of $\mathfrak{P}$ (in which case we call $\mathfrak{P}$ *reasonable*), then a unique minimal model, denoted $\chi_{\mathfrak{P}}(-)$, exists.

DEFINITION 3.2 (QUERY RESULTS). Given a query $\Delta_1, ..., \Delta_k$ for a reasonable PERA algebra $\mathfrak{P}$, a *result* is a context substitution $[\vec{t}/\vec{x}]$ for some Δ_i such that, for each judgment $\mathcal{J}$ in the list $\Delta[\vec{t}/\vec{x}]$, $\chi_{\mathfrak{P}}(\bullet \vdash \mathcal{J})$ holds.

Based on this definition, in [17], we prove the following computability result.

THEOREM 1 (COMPUTABILITY). *Given a query $\Delta_1, ..., \Delta_k$ for some reasonable PERA algebra such that all Δ_i are bounded, then the set of results for that query can be computed in finite time.*

The proof of the theorem computes query results by a familiar iterative ('stratum by stratum') fixpoint algorithm, though care needs to be taken to restrict our attention only to a bounded set of relevant facts. In light of the theorem, it makes sense to revise our earlier definition of queries slightly and require boundedness of their contexts by default. This carves out a natural class of *computable queries* in the PERA model. In fact, one can show that TYPEQL queries are, in an appropriate sense, *complete* for this class; i.e., any PERA query can be expressed by an equivalent TYPEQL query. However, a detailed exposition of this result requires the formal specification of TYPEQL's syntax and goes beyond our present scope.

CONCLUSION AND FUTURE WORK

In this paper, we described a succinct formalization of a powerful novel data model for polymorphic ERA (PERA) modeling, querying, and reasoning. We used type-theoretic ideas to formalize the *data model* and *queries* of the PERA model, and logic programming to describe its *semantics*. The data model is built on subtypes and dependent types, featuring type *inheritance* and polymorphic type dependencies via *interfaces*, and queries are defined using common type-theoretic constructs. Most importantly, the approach comes with a high-level and intuitive query language, TYPEQL. We described TYPEQL's syntax through a comprehensive example, and illustrated how its queries translate to type-theoretic queries in the PERA model.

Our work yields practical and intuitive modeling techniques for diverse structures of data. It also establishes a novel type-theoretic pattern-based querying paradigm, which robustly adapts query interpretations to changes in the underlying type schema. Based on the theoretical foundations laid in our work, in future work, we intend to investigate important related questions, including query planning and query optimization for the PERA model. We also hope that the fruitful interaction of type theory and database language demonstrated here will inspire further research in the area.

REFERENCES

[1] 2023. TypeDB. (2023). https://www.typedb.com

[2] S. Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Addison-Wesley, Reading, Mass. http://webdam.inria.fr/Alice/

[3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. 1988. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming.* Elsevier, 89–148. https://doi.org/10.1016/B978-0-934613-40-8.50006-3

[4] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. 1990. The Object-Oriented Database System Manifesto. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 395.

[5] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. 2016. A Formal Presentation of MongoDB (Extended Version). *CoRR* abs/1603.09291 (2016). arXiv:1603.09291 http://arxiv.org/abs/1603.09291

[6] John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Log.* 32 (1986), 209–243. https://doi.org/10.1016/0168-0072(86)90053-9

[7] Peter P. Chen. 1976. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.* 1, 1 (1976), 9–36. https://doi.org/10.1145/320434.320440

[8] Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. (2017). http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[10] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, Barcelona Spain, 510–524. https://doi.org/10.1145/3062341.3062348

[11] Keith L. Clark. 1977. Negation as Failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Hervé Gallaire and Jack Minker (Eds.). Plemum Press, New York, 293–322. https://doi.org/10.1007/978-1-4684-3384-5_11

[12] William F. Clocksin and Christopher S. Mellish. 1994. *Programming in Prolog (4. ed.).* Springer.

[13] E. 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

[14] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

[15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388.

[16] Christoph Dorn and Haikal Pribadi. 2023. Type Theory as a Unifying Paradigm for Modern Databases. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23).* Association for Computing Machinery, New York, NY, USA, 5238–5239. https://doi.org/10.1145/3583780.3615999

[17] Christoph Dorn and Haikal Pribadi. 2024. Formal Semantics of the Type-Theoretic Language TypeQL. *In preparation* (2024).

[18] R Elmasri and SB Navathe. 2016. *Fundamentals of Database Systems.* Pearson, Boston.

[19] David W Embley. 2009. Semantic Data Model. In *Encyclopedia of Database Systems*, M. Tamer Özsu Ling Liu (Ed.). Springer New York, NY, 3391–3393. https://doi.org/10.1007/978-0-387-39940-9

[20] Henrik Forssell, Hakon Robbestad Gylterud, and David I Spivak. 2020. Type theoretical databases. *Journal of Logic and Computation* 30, 1 (Jan. 2020), 217–238. https://doi.org/10.1093/logcom/exaa009 arXiv:1406.6268 [cs, math].

[21] 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.* ACM, Houston TX USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[22] Allen Van Gelder. 1993. The Alternating Fixpoint of Logic Programs with Negation. *J. Comput. Syst. Sci.* 47, 1 (1993), 185–221. https://doi.org/10.1016/0022-0000(93)90024-Q

[23] Michael Gelfond. 1987. On Stratified Autoepistemic Theories. In *Proceedings of the 6th National Conference on Artificial Intelligence. Seattle, WA, USA, July 1987*, Kenneth D. Forbus and Howard E. Shrobe (Eds.). Morgan Kaufmann, 207–211. http://www.aaai.org/Library/AAAI/1987/aaai87-037.php

[24] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, Robert A. Kowalski and Kenneth A. Bowen (Eds.). MIT Press, 1070–1080.

[25] Michael Gelfond and Vladimir Lifschitz. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Gener. Comput.* 9, 3/4 (1991), 365–386. https://doi.org/10.1007/BF03037169

[26] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge university press Cambridge.

[27] Martin Gogolla and Uwe Hohenstein. 1991. Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Trans. Database Syst.* 16, 3 (1991), 369–416. https://doi.org/10.1145/111197.111200

[28] Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (2017), 27–39. https://doi.org/10.14778/3151113.3151116

[29] Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

[30] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011 (SIGMOD '11)*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, NY, USA, 1213–1216. https://doi.org/10.1145/1989323.1989456

[31] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. 2009. A Classification of Object-Relational Impedance Mismatch. In *The First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDS 2009, Gosier, Guadeloupe, France, 1-6 March 2009*, Qiming Chen, Alfredo Cuzzocrea, Takahiro Hara, Ela Hunt, and Manuela Popescu (Eds.). IEEE Computer Society, 36–43. https://doi.org/10.1109/DBKDA.2009.11

[32] Anirudh Kadadi, Rajeev Agrawal, Christopher Nyamful, and Rahman Atiq. 2014. Challenges of data integration and interoperability in big data. In *2014 IEEE International Conference on Big Data (IEEE BigData 2014), Washington, DC, USA, October 27-30, 2014*, Jimmy Lin, Jian Pei, Xiaohua Hu, Wo Chang, Raghunath Nambiar, Charu C. Aggarwal, Nick Cercone, Vasant G. Honavar, Jun Huan, Bamshad Mobasher, and Saumyadipta Pyne (Eds.). IEEE Computer Society, 38–40. https://doi.org/10.1109/BIGDATA.2014.7004486

[33] Alfons Kemper, Guido Moerkotte, Hans-Dirk Walter, and Andreas Zachmann. 1991. GOM: A Strongly Typed Persistent Object Model With Polymorphism. In *Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Kaiserslautern, 6.-8. März 1991, Proceedings (Informatik-Fachberichte, Vol. 270)*, Hans-Jürgen Appelrath (Ed.). Springer, 198–217. https://doi.org/10.1007/978-3-642-76530-8_11

[34] Won Kim. 1990. Object-Oriented Databases: Definition and Research Directions. *IEEE Trans. Knowl. Data Eng.* 2, 3 (1990), 327–341. https://doi.org/10.1109/69.60796

[35] Neal Leavitt. 2000. Whatever Happened to Object-Oriented Databases? *Computer* 33, 8 (2000), 16–19. https://doi.org/10.1109/MC.2000.10067

[36] Mengchi Liu. 1999. Deductive Database Languages: Problems and Solutions. *ACM Comput. Surv.* 31, 1 (1999), 27–62. https://doi.org/10.1145/311531.311533

[37] John Wylie Lloyd. 1984. *Foundations of Logic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-96826-6

[38] Mary E. S. Loomis and Akmal B. Chaudhri (Eds.). 1997. *Object Databases in Practice*. Prentice-Hall.

[39] Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in proof theory, Vol. 1. Bibliopolis, Naples.

[40] Atsushi Ohori, Peter Buneman, and Val Tannen. 1989. Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. (1989), 46–57. https://doi.org/10.1145/67544.66931

[41] Pasquale Pagano, Leonardo Candela, and Donatella Castelli. 2013. Data Interoperability. *Data Sci. J.* 12 (2013), GRDI19–GRDI25. https://doi.org/10.2481/DSJ.GRDI-004

[42] Christine Parent and Stefano Spaccapietra. 2000. Database integration: The key to data interoperability. In *Advances in Object-Oriented Data Modeling*, Z. Tari M. P. Papazoglou, S. Spaccapietra (Ed.). MIT Press.

[43] Joan Peckham and Fred J. Maryanski. 1988. Semantic Data Models. *ACM Comput. Surv.* 20, 3 (1988), 153–189. https://doi.org/10.1145/62061.62062

[44] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[45] The Univalent Foundations Program. 2013. Homotopy Type Theory: Univalent Foundations of Mathematics. (2013). https://homotopytypetheory.org/book/

[46] Teodor C. Przymusinski. 1988. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, Jack Minker (Ed.). Morgan Kaufmann, 193–216. https://doi.org/10.1016/B978-0-934613-40-8.50009-9

[47] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (Aug. 2009), 1–45. https://doi.org/10.1145/1567274.1567278

[48] Raghu Ramakrishnan and Jeffrey D. Ullman. 1995. A survey of deductive database systems. *J. Log. Program.* 23, 2 (1995), 125–149. https://doi.org/10.1016/0743-1066(94)00039-9

[49] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, James Cheney and Thomas Neumann (Eds.). ACM, 1–10. https://doi.org/10.1145/2815072.2815073

[50] Peter Scholze. 2022. Liquid Tensor Experiment. *Exp. Math.* 31, 2 (2022), 349–354. https://doi.org/10.1080/10586458.2021.1926016

[51] Patrick Schultz and Ryan Wisnesky. 2017. Algebraic data integration. *J. Funct. Program.* 27 (2017), e24. https://doi.org/10.1017/S0956796817000168

[52] Chandan Sharma and Roopak Sinha. 2022. FLASc: a formal algebra for labeled property graph schema. *Autom. Softw. Eng.* 29, 1 (2022), 37. https://doi.org/10.1007/S10515-022-00336-Y

[53] Paul Taylor. 1999. *Practical Foundations of Mathematics.* Cambridge studies in advanced mathematics, Vol. 59. Cambridge University Press.

[54] Bernhard Thalheim. 2018. Extended Entity-Relationship Model. (2018). https://doi.org/10.1007/978-1-4614-8265-9_157

[55] Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84. https://doi.org/10.1145/2699407

A COMPARISON OF DATA MODELS

The PERA model is based on two simple and fundamental ingredients: *dependent types* for expressing dependencies between data, and *subtypes* for expressing polymorphism. In this appendix, we will illustrate how these two ingredients provide a unifying perspective on existing data models, and use this to compare them with the PERA model. We will focus on three prominent (data) models: the relational model [13], the labeled property graph model [21], and the document model [5].

REMARK A.1 (ON POLYMORPHISM IN DATA MODELS). While polymorphism is a common feature of many data domains, most contemporary data models provide no direct support for it. Historically, object-oriented databases (OODBs) were considered a promising approach to infusing data models with polymorphism: however, while OODBs promoted polymorphism as a first-class construct in their language, data dependencies now had to be described in a less structured way, using combinations of object constructors and methods. This additional complexity impeded scalability and the emergence of a unified query language for OODBs, which put other, conceptually simpler, data models at an advantage.

In fact, none of the three models discussed in this section provide first-class support for structured polymorphisms (and, therefore, polymorphism will play a secondary role in the comparisons below). Instead, in each case, we will focus on the following two basic aspects of each model:

(1) In any model, data is sorted into named categories (e.g., using labels, keys, tables, columns, etc.)—we usually loosely refer to these 'categorical containers of data' as *types.*
(2) Types may logically *depend* on other types, meaning that in order to instantiate one type with data, one or more existing data instances in other types must be referenced.

Exhibiting the types and type dependencies of a given data model will make the translation into the PERA model essentially immediate: types in the respective models will translate into either entity, relation, or attribute types in the PERA model, depending on the kind of data that the type collects (i.e., objects or values) and whether they depend on other types or not. The interesting observation that types and type dependency provide a simple and unifying way of studying and comparing how common data models structure their data was previously made in [16].

Visualizing PERA algebras. In Figure 5, we provide a visual guide to key PERA modeling concepts, which were introduced in the main text. Note that the three primary *type kinds* (i.e., entity, relation, and attribute types) are distinguished by color and shape.

- Entity types, i.e., types in our type schema containing objects and independent from other types, are shaded in **purple** and with a rounded rectangular shape;
- Relation types, i.e., types in our type schema containing objects and dependent on other types, are shaded in **yellow** and with a diamond shape;
- Attribute types, i.e., types in our type schema containing values and dependent on other types, are shaded in **blue** and with an oval shape;
- Interface types, on which other types may depend, are shaded in the color corresponding to the type they belong to (either a relation or attribute type), and are shown with a dotted boundary.

We remark that the distinction of types by shapes mirrors the usual conventions used in entity-relationship diagrams [7]. Note also that Figure 5 distinguishes between *data-storing* types, i.e., those for which terms can be created directly through a TYPEQL `insert` clause (or, in formal terms, through the data axioms of a PERA algebra), and *data-structuring* types, i.e., those for which terms cannot be created directly by the user, but are instead derived through the rules of the PERA type system.

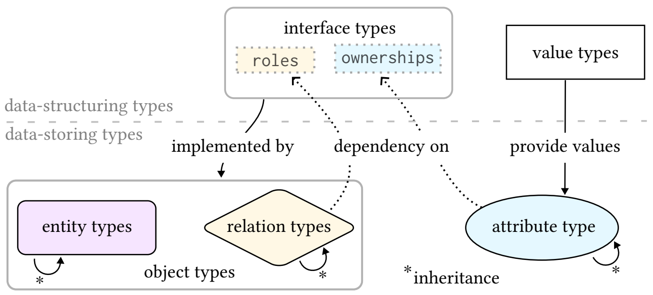

Fig. 5. Visualizing Pera model terminology

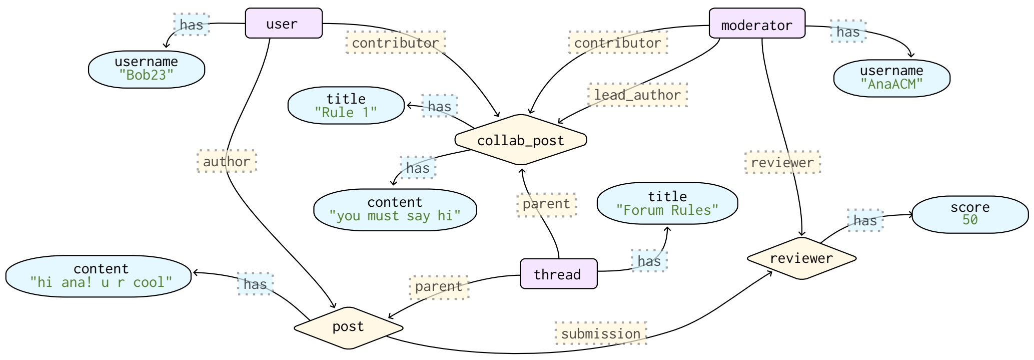

Fig. 6. Visualizing our TypeQL example database

Based on this visualization of types, we may similarly visualize the *data* in Pera algebras. (Recall, Pera algebras are extensions of the Pera type systems with a concrete type schema, data, and rules.) Namely, we will depict the *terms* in types by accordingly labeled and color-coded shapes: e.g., terms in entity types are shown with a rectangular shape, shaded in purple. In general, when a term `t` is specified with type `t:T(s,...)`, then we add the type identifier `T` as a **black** label on `t`'s shape (e.g., the label '`user`' in Figure 6). The dependency of `T(s,...)` on `s` is shown as an arrow `s` ⟶ `t` labeled with `I` where `I` is the appropriate interface of `T` that `s` is cast into as `s:I` (e.g., `author` in Figure 6). As each attribute type depend on a *single* interface, we use the label `has` instead of a type identifier for that interface. We indicate literal values of attribute terms in **green**.

Example A.1 (Visualizing the Forum database). To illustrate our convention for depicting data in a Pera algebra, we revisit our earlier example in Figure 1 showing a database that models a 'pre-moderated forum'. The data of that database is visualized in Figure 6. Note that we do not include the OIDs of object terms in our visualization, but the correspondence to OIDs (and, thus, to variables in the `insert` clause of Figure 1) should be evident from objects owning attributes and playing roles: for example, the `user` object in Figure 6 represents the term `&bob : user`.

A.1 Relational model

We begin our comparison to other data models with the relational data model. Due to its simplicity, expressivity, and scalability, the model has remained a leading database paradigm in industry

```
define
  table_1 sub entity,
    owns val_column_1,
    owns val_column_2,
    plays FK_column:owner @count(1);
  table_2 sub entity,
    owns val_column_3,
    owns val_column_4,
    owns val_column_5,
    plays FK_column:reference;
  FK_column sub relation:
    relates owner,
    relates reference;
  val_column_1 sub attribute,
    value string;
  val_column_2 sub attribute,
    value datetime;
  val_column_3 sub attribute,
    value string;
  val_column_4 sub attribute,
    value long;
  val_column_5 sub attribute,
    value boolean;
```

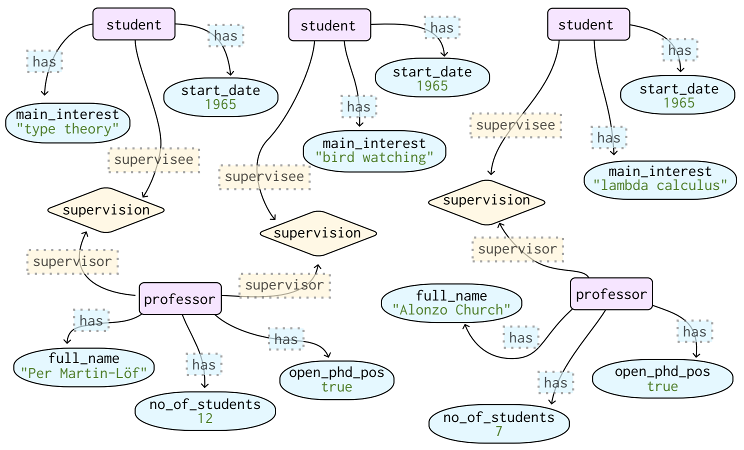

Fig. 7. Abstract relational schema in TypeQL with concrete visualized database

for more than four decades. Types and type dependencies of the relational model can be easily described, as summarized below.

(1) Data is sorted using named *tables* and named *columns*. The data collected by tables are called *row* objects. The data collected in columns are either *values* (in which case we speak of a *value column*), or objects holding *references* to rows in other tables (in which case we speak of an *foreign key (FK) column*)—note that, while foreign keys are usually encoded as values as well, we here presuppose referential integrity and work directly with references to other objects.
(2) The dependencies between these three kinds of types can be seen as follows. In order to create (or update) a value in a value column, we must reference an existing row in a table for the value to live in; thus, value columns depend on tables, and correspond to attribute types. Similarly, in order to create (or update) a reference in a FK column, we must reference both an existing row in the column's table as well as a row object in the referenced table; thus, FK columns correspond to relation types. Finally, since rows in tables can be freely created (assuming no further constraints are imposed, see below), tables themselves correspond to entity types.

We remark that real-world relational databases are often equipped with various ways of constraining the structure of data (e.g., using `KEY` or `NOT NULL` constraints). This may affect type dependencies: e.g., a 'join' table with `NOT NULL` foreign keys to other tables will now correspond to a *relation* type, since its row objects can only be instantiated with reference to other row objects. Since we here focus only on 'core fragments' of the PERA model and other models, we will forego discussing these more advanced aspects of relational data modeling.

Example A.2 (Relational schema). In Figure 7, on the left, we first give an *abstract* relational schema in TypeQL, in which type names correspond to general terminology in the relational model, as just described, thereby explaining the intended function of these types. Then, replacing abstract type names with practical ones (but without spelling out the resulting 'practical' schema), we depict a concrete database comprising tables of students and professors. We remark that students have *at most one* `supervision` (note the `@count(1)` annotation, see [17]) by a `professor`. Unlike relational database schemas, this PERA schema can now be effortlessly extended, e.g., to include student supervisions by two or more professors, or to record multiple interests for individual students.

```
define
  node_label sub entity,
    owns key_to_property_1,
    plays edge_label:start,
    plays edge_label:end;
  edge_label sub relation,
    owns key_to_property_2,
    relates start,
    relates end,
    plays path:edge;
  key_to_property_1 sub attribute,
    value string;
  key_to_property_2 sub attribute,
    value string;
  path sub relation,
    relates edge @ordered;
  rule inferring_paths: when {
    $path with (edge[]: $edges);
    $edges[last] with (end: $node);
    $edge with (start: $node);
  } then {
    (edges[]: $edges + [$edge]) isa path;
  };
```

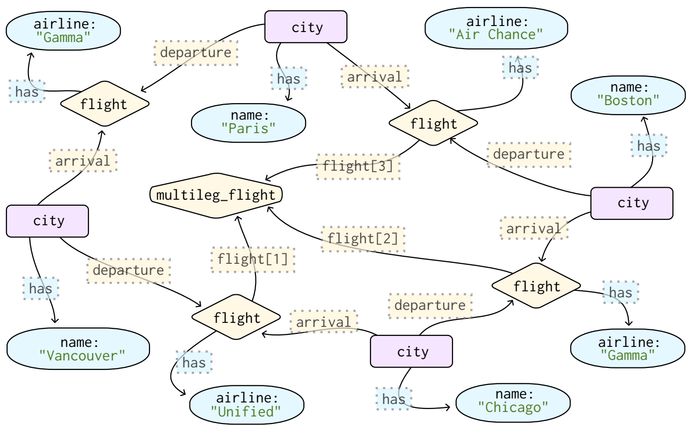

Fig. 8. Abstract property graph schema in TypeQL with concrete visualized database

A.2 Graph model

The labeled property graph model emphasizes binary connections between data and thereby facilitates traversal of data in ‘highly connected’ data domains. The types and type dependencies of the labeled property graph model are summarized below.

(1) Data is organized using *labels* and *property keys*. Labels fall into two categories: *node labels* which categorize *node* objects, and *edge labels* which categorize *edge* objects. In contrast, property keys collect values.
(2) The dependencies between these three kinds of types can be seen as follows. Node objects may be freely instantiated and, thus, *node labels* correspond to entity types. Edge objects must be instantiated with reference to their start and end nodes and, thus, *edge labels* correspond to relation types. Finally, property keys must be associated to (and thus are instantiated with reference to) either a node or an edge, and so depend on either node or edge types. This means property keys correspond to attribute types.

In addition to nodes and edges, an important feature of the graph model query languages is the ability to directly query *paths*. Paths (analogous to, say, views in the relational model) are data that are *inferred* from the existing ‘physical’ data. TypeQL can natively replicate this behaviour using rules (or dependent subtypes, see footnote 1 and [17]). We remark that the labeled property graph model usually also supports a form of ‘unstructured inheritance polymorphism’, manifested in the ability to freely combine labels together (this is ‘unstructured’ in that a node may be labeled both as a `car` and a `person`). We will forego a detailed comparison with TypeQL’s (structured) inheritance polymorphism.

Example A.3 (Graph schema). An abstract property graph schema is shown in Figure 8 on the left, together with a concrete database on the right. The latter comprises `city`-labeled nodes and `flight`-labeled edges between them. Importantly, rules give us fine-grained control over which paths are to be inferred: for example, we could choose to consider only those paths of flights with at least 30 minutes overlay between legs (which is difficult to express in existing graph databases).

```
define
  collection sub entity,
    owns key_to_val_1;
    plays key_to_subdoc_1:parent;
    plays key_to_subdoc_2:parent;
  key_to_subdoc_1 sub relation,
    relates parent;
    owns key_to_val_2;
  key_to_subdoc_2 sub relation,
    relates parent;
    plays key_to_subdoc_3:parent;
    owns key_to_val_1;
  key_to_subdoc_3 sub relation,
    relates parent;
    plays key_to_subdoc_4:parent;
    owns key_to_val_1;
    owns key_to_val_3;
  key_to_subdoc_4 sub relation,
    relates parent;
  key_to_val_1 sub attribute,
    value string;
  key_to_val_2 sub attribute
    value long;
  key_to_val_3 sub attribute
    value datetime;
```

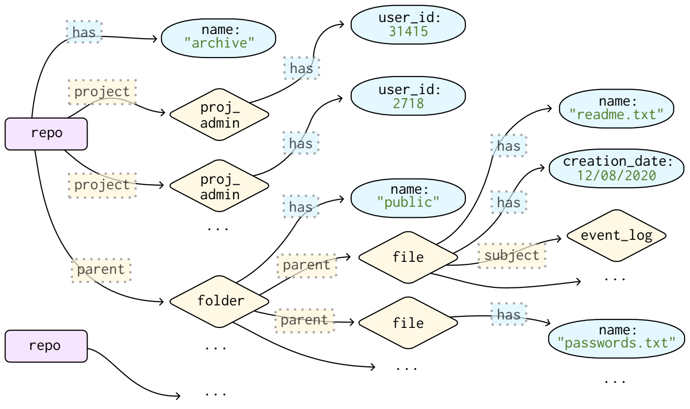

Fig. 9. Abstract document schema in TypeQL with concrete visualized database

A.3 Document data

Document databases store data in tree-like structures that can be navigated with keys. They provide great flexibility in representing data dependencies, and allow recording dependent data 'locally' to enable fast data access. Document databases work primarily with unstructured data. This means (re)structuring trees can quickly become a complex task, and so can maintaining duplicate data or checking referential integrity. Nonetheless, even in their unstructured form, we may still distill notions of types and type dependencies as summarized below.

(1) Data is primarily categorized by *collections* which collect *document* objects. Data is also structured by *keys*: keys may categorize either nested *subdocument* objects (yielding *keys to subdocuments*), or values (yielding *keys to values*), or objects holding references to other documents (yielding *keys to references*). We remark that the latter case is usually implemented using literal object IDs—here, in analogy to our earlier discussion of the relation model, we will directly work with references to other objects instead.
(2) The dependencies between these three kinds of types can be seen as follows. Document objects may be freely instantiated and so collections correspond to entity types. Keys to subdocuments can only be instantiated inside (and thus with reference to) an existing document or subdocument objects and, thus, correspond to relation types. Keys to values are also instantiated inside (sub)document objects and so correspond to attribute types. Finally, keys to references are instantiated inside (sub)document objects and, in addition, will reference an existing document object; they, therefore, also correspond to relation types.

Example A.4 (Document schema). An abstract document schema is given in Figure 9 on the left, together with a concrete database on the right. The latter comprises a collection of 'repo' documents, which have keys to subdocuments listing the 'project admins' and the 'folders' of each repo. The latter further have 'file' subdocuments associated to them. Documents and subdocuments have keys to values, such as 'names', 'user IDs', and 'creation dates'. Note that we did not include keys to references in our example (as we only show a single document object).

The example illustrates the importance of TypeQL's *interface polymorphism*. Indeed, interface polymorphism allows us to have multiple types of 'implementers' of keys across different locations in a document tree: for example, the key `name` to a `string` value is owned by `repo` documents, `folder` subdocuments, and `file` subdocuments simultaneously. Similarly, if other types would implement the `parent` role of the `file` type then these types, too, could contain `file` subdocuments.

It is instructive to revisit the preceding example in a format more common to the document paradigm: this is shown in Figure 10, which formats the database shown in Figure 9 in a familiar JSON structure. Note that we pluralized type names, e.g., writing `proj_admins` in place of `proj_admin`, in order to conform with JSON conventions.

We remark that, throughout this section, we implicitly made the assumption that *lists* present *sets*; in other words, the orders of lists are irrelevant to our data domain (e.g., the list of `proj_admins` is ordered, but the order will not be relevant to our data model and application). Of course, in general, list orders may be important, in which case we can work with ordered sets in the Pera model to capture this. We omitted the case from our example above for simplicity.

```
{
    name: "archive",
    proj_admins: [
        {
            user_id: 31415
        },
        {
            user_id: 2718
        },
        { ... } # further proj_admins
    ],
    folders: [
        {
            name: "public",
            files: [
                {
                    name: "readme.txt",
                    creation_date: 12/08/2020,
                    event_logs: [
                        { ... } # event_log subdocuments
                    ]
                },
                {
                    name: "passwords.txt",
                    ...
                },
                { ... } # further files
            ]
        },
        { ... } # further folders
    ]
},
{ ... } # further repos
```

Fig. 10. Our document database in JSON format

A.4 Summary

Like other data models, the Pera model is conceptually simple: its basic building blocks are type dependencies and subtypes. Types may collect either objects or values, and types may depend on either unordered bags or ordered sets of terms. The examples in the preceding sections illustrate how prominent existing data model can be distilled into specific subsets of these basic ideas. This makes the Pera model an interesting 'unifying' framework to translate and compare between these models.

Received December 2023; revised February 2024; accepted March 2024