AnonQL: A Query Language for the Graph-Relational Data Model

ANONYMOUS AUTHOR(S)**

We present the *graph-relational* data model and the AnonQL query language, designed for expressive and efficient manipulation of complex domain data in a type-safe manner. Traditional relational data models and SQL are increasingly inappropriate for use in the development of applications due to the object-relational impedance mismatch. The graph-relational model can be viewed as an extension of the relational data model that adds object references (links) and set-valued attributes. This eliminates the mismatch, resulting in a more efficient and safe application/database interaction. AnonQL is a general-purpose SQL-style query language built on top of the graph-relational model, supporting nested typed fetching of structured object data in a flexible and composable way. We present a core calculus for AnonQL, with static and dynamic semantics and safety theorems.

1 INTRODUCTION

In databases with relational data models [8], SQL [5–7] is overwhelmingly the most common query language used. This combination, however, in which data is stored as—and queries return—rows of tabular data, is a poor match for how data is stored and manipulated in applications. Applications typically represent data as objects that are connected via references. Because of this mismatch, applications often rely on Object-Relational Mapper (ORM) libraries to query the database and present the data in a more application-friendly manner. Unfortunately, this approach sacrifices much of the power of SQL and the relational model. Often the ORM needs to perform multiple queries when one would be possible. Expressing a complex query typically requires abandoning the ORM and writing SQL directly.

Various forms of NoSQL databases, such as document-oriented databases like MongoDB [11], key-value stores like Redis [14], and graph databases [2, 3] have achieved popularity by sidestepping the object-relational mismatch entirely. Unfortunately, they typically sacrifice the flexibility of a powerful query language, often lack strong typing of structured data, and commonly have weaker consistency properties.

The graph-relational data model and the AnonQL query language is an attempt to bridge the object-relational divide while preserving the core advantages of the relational model. The graph-relational model is a data model that extends the relational model [6] with the following features:

(1) Every object (row) has a unique identifier. In a database, two objects are identified if and only if they have the same identifier.
(2) Objects can have link attributes, which are references to other objects. Links may be augmented with link properties containing primitive (non-object) data.
(3) Each attribute of an object is set-valued and has a specified cardinality. (Attributes are links as well as properties, which store primitive data.)

These may all be seen as reifications of traditional patterns of use of relational databases: synthetic primary keys, foreign keys, and many-to-many link tables.

The query language then has direct constructs for traversing links between objects, abstracting away the operation of joining on primary and foreign keys. Instead of representing missing values with a special NULL value, they are represented with empty sets.

We briefly demonstrate the features of AnonQL using a toy database schema, written in the concrete syntax that is used by the AnonDB implementation of AnonQL.

```
    type Movie {
        required title: str;
        required year: int64;
        multi actors: Actor {
            character: str;
        };
    }
    type Actor {
        required name: str;
    }
```

Fetching all movies can be done simply:

```
select Movie
```

The above query returns a set of objects that contain only the object IDs. We can fetch more information by specifying a “shape” on the object.

```
select Movie { title, year }
```

Shapes may be nested, to fetch the contents of objects that are linked to.

```
select Movie { title, year, actors: {name} }
```

The `actors` attribute of each `Movie` is actually a set of links. We may use the link property projections (`@label`) to query the attributes of links. For example, we may query the character that each actor plays:

```
select Movie { title, year, actors: {name, @character} }
```

The results are returned to clients in a nested form, along with type information. Here is how the results of the last query are rendered by AnonDB’s command-line interface. [1]

```
{
  Movie {
    title: 'Dune',
    year: 1984,
    actors: {
      Person {name: 'Patrick Stewart', @character_name: 'Gurney Halleck'},
      Person {name: 'Kyle MacLachlan', @character_name: 'Paul Atreides'},
      ...
    },
  },
  Movie {
    title: 'Dune',
    year: 2021,
    actors: {
      Person {name: 'Josh Brolin', @character_name: 'Gurney Halleck'},
      Person {name: 'Timothée Chalamet', @character_name: 'Paul Atreides'},
      ...
    },
  },
  ...
}
```

[1]Even though all expressions in AnonQL are set-valued, attributes such as `title` that are inferred to have at most one element are displayed as scalars. This is enabled by our type and cardinality inference.

Fetching all this data using SQL would involve explicitly joining together three tables by their primary keys. Additionally retrieving the data in a nested and typed way ranges from slightly tedious (in PostgreSQL) to essentially impossible (in MySQL [2]).

Shapes on objects are not restricted to simply selecting properties from the object but can define new "computed" properties:

```
select Movie { title, cast_size := count(.actors) }
```

The dot notation syntax `expr.label` represents projecting a property or link out of an object. If the leading `expr` is omitted, it refers to the expression being operated on by the shape.

The simple shape syntax can be viewed as syntactic sugar, expanding to queries such as:

```
select Movie { title := .title,
               actors := .actors { name := .name } }
```

AnonQL supports ordering and filtering as well, again using the leading dot notation to represent the object being operated on:

```
select Movie { title, year, actors: {name, @character} }
filter .name = 'Dune'
order by .year
```

Links may also be traversed in reverse, using the "backlink" syntax. This finds all films that a given actor appeared in:

```
select Person { films := .<actors[is Movie] { title, @character_name }}
filter .name = 'Patrick Stewart'
```

In the rest of this paper, we present an abstract syntax based formal semantics. We summarize the main contributions of this paper.

(1) The definition of the graph-relational data model, and an introduction to value sets, query expressions, and query composition. (Section 2)
(2) An overview of AnonQL language constructs. (Section 3)
(3) A type system that incorporates cardinality checking. (Section 4)
(4) A formalization of the execution semantics. (Section 5)
(5) A discussion of the metatheoretic properties. (Section 6)
(6) A description of the path factorization algorithm used in the AnonDB implementation of AnonQL. (Section 7)
(7) A general discussion of the AnonDB implementation and its features. (Section 8)

2 KEY CONCEPTS

The Graph-Relational Data Model. A graph-relational data model is a graph that consists of a set of objects with unique identifiers and a set of directed links between those objects. Objects may have properties that map attribute names to either a multi-set of primitive values or a set of links. Links may have link properties that map names to primitive values.

Value Sets and Broadcasting. In AnonQL, every *expression* e computes a *value set* U, which is a multiset of *values* $\{V_1, \ldots, V_n\}$. In order to support the *order by* operation, these value sets are optionally ordered. Most operators have broadcasting semantics, acting element-wise on the input arguments. For example, the expression $1 + \{5, 6\}$ evaluates to $\{6, 7\}$. The computation of AnonQL expressions may interact with the database μ by reading to it or writing from it. For example, the expression `select Movie.title` ($Movie \cdot title$ in abstract syntax) reads the title of all movies in the

[2]MySQL lacks an array type, which is the obvious way to represent nested set data. It does have a JSON type, which can work but is essentially untyped.

database, and the expression `insert Movie {title := "The Matrix", actors := {}}` creates a fresh movie with the title "The Matrix" in the database.

Compositionality. Queries can be built by freely composing subexpressions, even when the subexpressions are set-valued or are non-primitive data types. This is in contrast to SQL, where table-valued and scalar-valued expressions are distinct and cannot be mixed.

Static Typing. AnonQL expressions are statically typed. The result type of an expression includes information about the type and the cardinality of the result value set. The result type is written τ^m, where τ is the *type* and m is the *cardinality mode*. The result type and result size information guide the rendering of the result value set in the implementation.

3 LANGUAGE OVERVIEW

Figure 1 shows the abstract syntax of AnonQL.

3.1 Primitive Types and Primitive Values

We choose to have at least three primitive types t in AnonQL, integers *int*, booleans *bool*, and strings *str*. Primitive values v include constants for those primitive types, such as 3, *true*, *false*, "hello". Implementations may include additional primitive types as they see fit. AnonDB, for example, includes the type of floating point numbers, *datetime*, and *json*.

3.2 Expressions

The core AnonQL has the following expression forms.

(1) Primitive values v.
(2) Variables x are introduced by binders.
(3) A shaped expression $e\ s$ runs e through the shape s (defined below in Section 3.4).
(4) A union operation $e_1 \cup e_2$ combines the result value sets of e_1 and e_2 into a single value set.
(5) A multiset expression $\{e_1, \ldots, e_n\}$ is syntactic sugar for the iterative union, $e_1 \cup \cdots \cup e_n$, associated to the left. It combines multiple value sets into one value set. We have a separate empty set expression $\{\}_\tau$ that evaluates to the empty value set of type τ.
(6) Function calls $f(e_1, \ldots, e_n)$ use call-by-value semantics.
(7) Link projections $e \cdot l$ project the property l from e. Backlinks $e \cdot_{\leftarrow} l\ [is\ N]$ find all objects of type N whose link l has target e. Link property projections $e\ @\ l$ project the link property l from the reference values e. In the AnonDB implementation of AnonQL, the concrete syntaxes are `e.l`, `e.<l[is N]`, and `e@l` respectively.
(8) A conditional expression *if* e_1 *then* e_2 *else* e_3 evaluates to e_2 if e_1 evaluates to *true*. Otherwise, it evaluates to e_3.
(9) An alias expression $\text{with}(x := e_1; e_2)$ evaluates e_1, binds the result to x, and then evaluates e_2.
(10) A for expression $\text{for}(x \leftarrow e_1; e_2)$ evaluates e_1, binds x to each value in the value set of e_1 (x will be bound to a singleton value set), evaluates e_2, and then aggregates the result. As a consequence, when e_1 evaluates to an empty value set, the for expression would evaluate to an empty value set.
(11) An optional for expression $\text{optional\_for}(x \leftarrow e_1; e_2)$ acts similarly to a for expression, except when e_1 is empty, x will be bound to the empty set when evaluating e_2.
(12) Detached expressions *detached* e and subquery expressions *select* e are used during path factorization (see section 7).
(13) A select expression with filter and order e_1 *filter* $(x.e_2)$ *order* $(x.e_3)$ binds x to each value in the result value set of e_1. The filter $x.e_2$ evaluates to a value set of booleans aggregated using

i	$::=$	$0 \mid 1 \mid \infty$	cardinals
m	$::=$	$[i_1, i_2] \mid (\leq 1) \mid (= 1) \mid (\geq 1) \mid (*)$	cardinality modes
N	$::=$	(string)	type names
p	$::=$	$1 \mid ? \mid *$	parameter modifiers
l	$::=$	(string)	string labels
L	$::=$	$l \mid @l$	plain and link property labels
v	$::=$	$3 \mid \textit{true} \mid \ldots$ (see section 3.1)	primitive values
t	$::=$	$\textit{int} \mid \textit{bool} \mid \textit{str}$	primitive types
T	$::=$	$\{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}$	object types
S	$::=$	$\{l_1 : t_1^{m_1}, \ldots, l_n : t_n^{m_n}\}$	link property types
R	$::=$	$T @ S$	link types
τ, σ	$::=$	t	
(types)	$\mid$	$N @ S \mid T_N @ S$	nominal link types
s	$::=$	$\{L_1 := x.e_1, \ldots, L_n := x.e_n\}$	shapes
e	$::=$	$v \mid x \mid N$	
(expre-	$\mid$	$e\ s$	Shaped expressions
ssions)	$\mid$	$\{e_1, \ldots, e_n\} \mid e_1 \cup e_2 \mid \{\}_\tau$	multisets and unions
	$\mid$	$f(e_1, \ldots, e_n)$	function applications
	$\mid$	$e \cdot l \mid e \cdot_{\leftarrow} l\ [\textit{is}\ N]$	links and backlinks
	$\mid$	$e @ l$	links properties
	$\mid$	$\textit{if}\ e_1\ \textit{then}\ e_2\ \textit{else}\ e_3$	conditional expressions
	$\mid$	$\text{with}(x := e_1; e_2)$	alias expressions
	$\mid$	$\text{for}(x \leftarrow e_1; e_2)$	for iteration
	$\mid$	$\text{optional\_for}(x \leftarrow e_1; e_2)$	optional for iteration
	$\mid$	$\textit{detached}\ e \mid \textit{select}\ e$	scope operators
	$\mid$	$e_1\ \textit{filter}\ (x.e_2)\ \ \textit{order}\ (x.e_3)$	filters and orders
	$\mid$	$\textit{insert}\ N\ \{l_1 := e_1, \ldots, l_n := e_n\}$	insertions
	$\mid$	$\textit{update}\ e\ s \mid \textit{delete}\ e$	updates and deletions
U	$::=$	$\{V_1, \ldots, V_n\}$	value sets
V	$::=$	v	
(values)	$\mid$	$\text{ref}(\textit{id}) \diamond W$	reference values
u	$::=$	$\textit{visible} \mid \textit{invisible}$	property markers
W	$::=$	$\{L_1^{u_1} := U_1, \ldots, L_n^{u_n} := U_n\}$	object values
μ	$::=$	$\{(id_1, N_1, W_1), \ldots, (id_n, N_n, W_n)\}$	database store
Δ	$::=$	$\cdot \mid \Delta, N := \tau$	schemas
Γ	$::=$	$\cdot \mid \Gamma, x : \tau^m$	static contexts
Ω	$::=$	$\cdot \mid \Omega, x \mapsto U$	evaluation environments

Fig. 1. Abstract Syntax of AnonQL Core Calculus.

the *any* operator. The order $x.e_3$ is an optional single expression that can be ordered in the ascending direction with empty sets in the beginning. A query with only filters but not orders may specify the empty multiset (with any primitive type) as the order; for brevity, we will sometimes omit the order clause in these cases.

(14) An insert expression $\textit{insert}\ N\ \{l_1 := e_1, \ldots, l_n := e_n\}$ inserts into the database a new object of type N whose attributes are given by $e_1, \ldots, e_n$.

(15) An update expression *update* $e\,s$ updates the object references in the result of e using the shape s.

3.3 Cardinality Modes.

The *cardinality mode* m is defined to be $[i_1, i_2]$, where $i_1 \in \{0, 1\}$ is the lower bound and $i_2 \in \{1, \infty\}$ is the upper bound. A result set of cardinality mode $[i_1, i_2]$ has at least i_1 elements and at most i_2 elements (both inclusive).

There are in total four cardinality modes, and they have abbreviations.

(1) $(= 1)$ is the *required single* mode $[1, 1]$.
(2) (≤ 1) is the *optional single* mode $[0, 1]$.
(3) (≥ 1) is the *required multi* mode $[1, \infty]$.
(4) $(*)$ is the *optional multi* mode $[0, \infty]$.

The cardinality modes admit element-wise addition and multiplication operations defined by the following clauses, where the cardinality arithmetic is the usual one except that $1 + 1$ is truncated up to ∞ if they are in the upper bound and that $1 + 1$ is truncated down to 1 if they appear in the lower bound.

$$([i_1, i_2]) \times ([i_3, i_4]) = [(i_1 \times i_3), (i_2 \times i_4)]$$
$$([i_1, i_2]) + ([i_3, i_4]) = [(i_1 + i_3), (i_2 + i_4)]$$

The cardinality modes of the union and the Cartesian product of two value sets with modes m_1 and m_2 will be $m_1 + m_2$ and $m_1 \times m_2$, respectively.

The cardinality modes also admit partial orderings $m_1 \leq m_2$ defined to be contravariant in the lower bound and covariant in the upper bound.

$$[i_1, i_2] \leq [i_3, i_4] \text{ if and only if } i_3 \leq i_1 \text{ and } i_2 \leq i_4.$$

Cardinality ordering ensures that a value set of mode m_1 can always be viewed as a value set of mode m_2 if $m_1 \leq m_2$.

3.4 Shapes

A shape serves two purposes. First, it may add or override properties of an object value. Second, it changes the property marker on object properties, which ultimately affects how the object value is serialized to the end user. In a shaped expression $e\,s$, a shape $s = \{L_1 := x.e_1, \ldots, L_n := x.e_n\}$ adds or updates attributes L_1 through L_n to be the result e_1 through e_n respectively, where x is a binder referring to the current object e. The shape s will also set $L_1, \ldots, L_n$ to be visible and every other label to be invisible.

In the concrete syntax of the AnonDB implementation of AnonQL, components inside the shape s have shorthand. The following shows the grammar of concrete syntax shapes s^c and shape components c^c.

$s^c ::= \{c^c_1, \ldots, c^c_n\}$
$c^c ::= L \mid l : s^c \mid L := x.e$

We have the following syntactic sugar:

(1) The shape component L where L is a plain label l is shorthand for $L := x.(x \cdot l)$.
(2) The shape component L where L is a link label $@l$ is shorthand for $L := x.(x\,@\,l)$.
(3) If a concrete syntax shape s_c desugars into the core AnonQL shape s, then the shape component $l : s_c$ desugars into the core AnonQL shape component $l := x.(x \cdot l)\,s$, that applies shape s to the projection $x \cdot l$.

3.5 Types and Schemas

We define object types T, link property types S, and types τ simultaneously.

An object type T is a mapping from plain labels to types and their cardinality modes, written $\{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}$. It may also be viewed as an unordered set of object type components of the form $l : \tau_n^{m_n}$.

A link property type S is an object type where all the target types are primitive. That is, S is of the form $\{l_1 : t_1^{m_1}, \ldots, l_n : t_n^{m_n}\}$.

A link type R is an object type together with a link property type, i.e. R is of the for $T @ S$.

A type τ or σ is either a primitive type or a nominal link type. A nominal link type is either $N @ S$, meaning a link with target type name N and link properties S, or $T_N @ S$, where T is the definition of N plus additional computed and overridden properties.

A database schema Δ is an unordered mapping from type names to object types. The typing system for a given schema is equirecursive. Thus, if $N := T \in \Delta$, then we may use $N @ S$ and $T_N @ S$ interchangeably. In other words, type equality of types is the least reflexive, symmetric, transitive, and congruence closure over $T_N @ S = N @ S$ given $N := T \in \Delta$. It is worth noting that $T_N @ S \neq T_{N'} @ S$ if $N \neq N'$.

In the graph-relational model, we sometimes say that l_i is a property of an object if τ_i is a primitive type and say that l_i is a link between two objects if τ_i is a nominal link type. The labels in S are called link properties.

3.6 Values, Value Sets, and Database Store

A *value set* U is an optionally ordered multi-set of values V_i's. In general, only certain AnonQL expressions dictate ordering (see Section 5.3). A label l may be a plain label or a link property label (prefixed with the @ sign). An *object value* W is a mapping from labels L_i's to value sets U_i's, where each label carries a property marker u_i. A property marker u marks the visibility of a specific label and the mapped value set during the printing or serialization of this object value. Object values W may be viewed as unordered sets of object value components of the form $L_i^{u_i} := U_i$.

There are two kinds of values V.

(1) Primitive values v. They have been introduced in section 3.1.
(2) Reference values $\text{ref}(id) \diamond W$. A reference value $\text{ref}(id) \diamond W$ refers to the object in the database with the identifier id, where W supplements and overrides the object value stored in the database.

A database store μ is an unordered set of triples (id, N, W), where id is the identifier, N is the type name and W is an object value. In a well-formed database store μ, for every entry $(id, N, W) \in \mu$, W is of the form $\{l_1^{u_1} := U_1, \ldots, l_n^{u_n} := U_n\}$. Furthermore, if l_i is a property, then each element of U_i is a primitive value v. If l_i is a link, then each element of U_i is a reference value $\text{ref}(id') \diamond \{@l_1'^{(u_1')} := U_1', \ldots, @l_k'^{(u_k')} := U_k'\}$, where $l_1', \ldots l_k'$ are link properties on the link. The reference markers $u_1, \ldots, u_n, u_1', \ldots, u_k'$ are irrelevant in a database store and may be arbitrary.

3.7 Parameter Modifiers and Broadcasting

Parameter modifiers p control the broadcasting behavior of arguments during function calls. That is, they control how the set arguments to a function are interpreted before they are passed to an underlying primitive evaluation function. There are three parameter modifiers:

(1) the *singleton* modifier 1 specifies that the argument should be broken down into singleton sets before being passed to the underlying evaluation function,

(2) the *optional* modifier ? specifies that the argument should be broken down into singleton sets only if the argument is not an empty set. Otherwise, the empty set is passed as the argument.
(3) the *set of* modifier $*$ specifies that the argument should be passed as is.

Parameter modifiers are specified in the type signatures of functions, where each argument is assigned a parameter modifier:

$$f : [\tau_1^{p_1}, \ldots, \tau_k^{p_k}] \to \tau^m$$

For example, we have the following built-in functions.

(1) $+ : [int^1, int^1] \to int^{(=1)}$ — the integer addition function acts element-wise on the argument value sets.
(2) $count : [\tau^*] \to int^{(=1)}$ — the aggregate function *count* is performed on the input value set as a whole
(3) $?? : [\tau^?, \tau^*] \to \tau^{(*)}$ — the coalescing operator always returns the first argument unless it is empty, in which case it returns the second argument

The *count* and ?? functions above are polymorphic functions, in that the τ in their signature may be instantiated freely.

Formally, in a function call to f, for each argument value set $U_i = \{V_1, \ldots V_n\}$ appearing in the ith position with parameter modifier p_i, the candidate set of value sets is a set of value sets $C_i = \{U_{i,1}, \ldots, U_{i,k}\}$, where each $U_{i,j}$ is a value set. We compute the candidate sets C_i as follows.

(1) when p_i is 1, the candidate set of value sets has n value sets, $\{V_i\}$ for each i where $1 \le i \le n$. That is, $C_i = \{\{V_1\}, \ldots, \{V_n\}\}$. Note that the candidate set of value sets is empty if $n = 0$.
(2) when p_i is $*$, the candidate set of value sets has 1 value set, $\{V_1, \ldots, V_n\}$. That is, $C_i = \{\{V_1, \ldots, V_n\}\}$.
(3) when p_i is ?, the candidate set of value sets is the same as the case for $p_i = 1$, except that if $n = 0$, the candidate set of value sets has exactly one element $\{\}$. That is, $C_i = \{\{V_1\}, \ldots, \{V_n\}\}$ if $n \neq 0$ or $C_i = \{\{\}\}$ if $n = 0$.

Then, the function will be called once per element in the Cartesian product of the candidate sets of value sets, and the results will be aggregated using the union operator. We define f^* to be the lifting of f over value sets as defined by its parameter modifiers:

$$f^*(U_1, \ldots, U_n) = \cup\{f(U_1', \ldots, U_n') \mid \forall (U_1', \ldots, U_n') \in C_1 \times \cdots \times C_n\}$$

The parameter modifier p may *match* a mode and may produce a *broadcasting factor* m' as a result, written $p \rhd m \rightsquigarrow m'$, defined below. When a function $f : [\tau_1^{p_1}, \ldots, \tau_n^{p_n}] \to \tau^m$ is applied to value sets $U_1, \ldots, U_n$ of type $\tau_1^{m_1}, \ldots, \tau_n^{m_n}$, if each $p_i \rhd m_i \rightsquigarrow m_i'$, then the candidate set of value sets C_i will satisfy the cardinality mode m_i'.

(1) $1 \rhd [i_1, i_2] \rightsquigarrow [i_1, i_2]$
(2) $? \rhd [i_1, i_2] \rightsquigarrow [max(1, i_1), i_2]$
(3) $* \rhd [i_1, i_2] \rightsquigarrow (= 1)$

3.8 Concrete Syntax and Examples

It should be straightforward to see the correspondence between the abstract syntax and the concrete syntax. As an example, we show how queries from Section 1 are represented in the abstract syntax.

AnonDB Concrete Syntax	AnonQL Abstract Syntax
`select Movie;`	Movie
`select Movie { title, actors };`	$\mathit{Movie}\{\, \mathit{title} := x.x \cdot \mathit{title}, \mathit{actors} := x.x \cdot \mathit{actors}\}$
`select Movie { title, actors: { name }};`	$\mathit{Movie}\{\, \mathit{title} := x.x \cdot \mathit{title}, \mathit{actors} := x.x \cdot \mathit{actors}\{\, \mathit{name} := y.y \cdot \mathit{name}\,\}\,\}$
`select Movie { title, actors: { name, also_acted_in := .<actors [is Movie] { title } } };`	$\mathit{Movie}\{\, \mathit{title} := x.x \cdot \mathit{title}, \mathit{actors} := x.x \cdot \mathit{actors}\{ \mathit{name} := y.y \cdot \mathit{name}, \mathit{also\_acted\_in} := y. y \cdot_{\leftarrow} \mathit{actors}\ [\mathit{is\ Movie}]\{ \mathit{title} := z.z \cdot \mathit{title}\,\}\,\}\,\}$

We note that the top-level `select` keyword in the concrete syntax does not have semantic meaning but serves primarily for path factorization. Thus, we uniformly omit them in the corresponding abstract syntax.

4 STATIC SEMANTICS

We explain the typing rules for values and expressions in this section.

4.1 Typing Rules for Expressions

For a well-formed schema Δ, the typing judgment $\Delta; \Gamma \vdash e : \tau^m$ says that expression e with free variables in Γ will compute a value set of type τ and cardinality mode m, with database store μ. The context Γ is a list of the form $x : (\tau')^{m'}$ is a mapping from variables to their types and cardinalities. We also assume that no two variables in Γ are of the same name.

The typing rules are shown in Figure 2. The typing rules ensure that the types and the cardinality modes of the result are consistent. Most typing rules are self-explanatory and correspond to the semantics of the expressions described in Section 3.2. The judgment $\Delta; \Gamma \vdash s : \tau \Rightarrow \tau'$ will be explained in Section 4.3. We use the judgment $v : t$ for the typing of primitive values. For example, we have $1 : \mathit{int}$, and $\mathit{true} : \mathit{bool}$. As stated previously, we adopt the convention that when $N := T \in \Delta$, $N @ S$ and $T_N @ S$ may be used interchangeably.

4.2 Typing rules for Values

We use the judgments $\Delta \vdash_\mu U : \tau^m$, $\Delta \vdash_\mu V : \tau$, $\Delta \vdash_\mu W : R$, to mean that the value set U has type τ and the cardinality mode m, that the value V has type τ, and that the value W has link type R respectively. They are defined in Figure 3. The rule (WT-OBJ) computes the types T and S by selecting the object type components according to whether each attribute label of W is a plain label or a link property label. The typing rules use the union operation on types defined below.

4.2.1 Union of Object Values. We use the notation $\uplus$ to denote the (right-biased) union of two object values (i.e. sets of object components) defined as follows, where the properties of the right-hand side operand takes priority.

$$\begin{aligned} &\{L_1^{u_1} := U_1, \dots, L_n^{u_n} := U_n\} \uplus \{(L_1')^{u_1'} := U_1', \dots, (L_k')^{u_k'} := U_k'\} \\ &= \{(L_i^{u_i} := U_i \mid L_i \notin \{L_j'\}_{1 \le j \le k})_{1 \le i \le n}, (L_1')^{u_1'} := U_1', \dots, (L_k')^{u_k'} := U_k'\} \end{aligned}$$

4.2.2 Union of Object Types and Link Types. We overload the notation $\uplus$ to denote the (right-biased) union of two non-primitive types defined as follows, where the labels of the right-hand side take

(T-CARD-SUB)
$$\frac{\Delta;\Gamma \vdash e : \tau^{m} \qquad m \leq m'}{\Delta;\Gamma \vdash e : \tau^{m'}}$$

(T-VAR)
$$\frac{x : \tau^{m} \in \Gamma}{\Delta;\Gamma \vdash x : \tau^{m}}$$

(T-OBJ)
$$\frac{N := T \in \Delta}{\Delta;\Gamma \vdash N : (T_N \mathbin{@} \{\})^{(*)}}$$

(T-VAL)
$$\frac{v : t}{\Delta;\Gamma \vdash v : t^{(=1)}}$$

(T-SELECT)
$$\frac{\Delta;\Gamma \vdash e : \tau^{m}}{\Delta;\Gamma \vdash \mathit{select}\ e : \tau^{m}}$$

(T-DETACHED)
$$\frac{\Delta;\Gamma \vdash e : \tau^{m}}{\Delta;\Gamma \vdash \mathit{detached}\ e : \tau^{m}}$$

(T-SHAPE)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : \tau^{m}\\ \Delta;\Gamma \vdash s : \tau \Rightarrow \tau'\end{array}}{\Delta;\Gamma \vdash e\, s : (\tau')^{m}}$$

(T-UNION)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \tau^{m}\\ \Delta;\Gamma \vdash e_2 : \tau^{m'}\end{array}}{\Delta;\Gamma \vdash e_1 \cup e_2 : \tau^{m+m'}}$$

(T-FUNC)
$$\frac{\begin{array}{c}f : [\tau_1^{p_1}, \ldots, \tau_n^{p_n}] \longrightarrow \tau^{m}\\ \forall_{i,1\leq i\leq n}.(\Delta;\Gamma \vdash e_i : (\tau_i)^{m_i} \wedge p_i \triangleright m_i \rightsquigarrow m_i')\end{array}}{\Delta;\Gamma \vdash f(e_1, \ldots, e_n) : \tau^{m_1' \times \ldots m_n' \times m}}$$

(T-LPROP)
$$\frac{\Delta;\Gamma \vdash e_1 : (T_N \mathbin{@} \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\})^{m}}{\Delta;\Gamma \vdash e_1 \mathbin{@} l_i : (\tau_i)^{m \times m_i}}$$

(T-BACKLINK)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : \tau^{m}\\ N := \{\ldots, l : \tau^{m'}, \ldots\} \in \Delta\end{array}}{\Delta;\Gamma \vdash e \cdot_{\leftarrow} l\ [\mathit{is}\ N] : (N \mathbin{@} \{\})^{(*)}}$$

(T-PROJ)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : (T_N \mathbin{@} S)^{m}\\ T = \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}\end{array}}{\Delta;\Gamma \vdash (e \cdot l_i) : (\tau_i)^{m \times m_i}}$$

(T-EMPTY-SET)
$$\frac{}{\Delta;\Gamma \vdash \{\}_{\tau} : \tau^{(\leq 1)}}$$

(T-FOR)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : (\tau_1)^{m_1}\\ \Delta;\Gamma, x : (\tau_1)^{(=1)} \vdash e_2 : (\tau_2)^{m_2}\end{array}}{\Delta;\Gamma \vdash \mathsf{for}(x \leftarrow e_1; e_2) : (\tau_2)^{m_1 \times m_2}}$$

(T-OFOR-1)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : (\tau_1)^{[0,i_2]}\\ \Delta;\Gamma, x : (\tau_1)^{(\leq 1)} \vdash e_2 : (\tau_2)^{m_2}\end{array}}{\Delta;\Gamma \vdash \mathsf{optional\_for}(x \leftarrow e_1; e_2) : (\tau_2)^{[1,i_2] \times m_2}}$$

(T-OFOR-2)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : (\tau_1)^{[1,i_2]}\\ \Delta;\Gamma, x : (\tau_1)^{(=1)} \vdash e_2 : (\tau_2)^{m_2}\end{array}}{\Delta;\Gamma \vdash \mathsf{optional\_for}(x \leftarrow e_1; e_2) : (\tau_2)^{[1,i_2] \times m_2}}$$

(T-WITH)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : (\tau_2)^{m_2}\\ \Delta;\Gamma, x : (\tau_2)^{m_2} \vdash e_2 : \tau^{m}\end{array}}{\Delta;\Gamma \vdash \mathsf{with}(x := e_1; e_2) : \tau^{m}}$$

(T-FILT)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \tau^{[i_1,i_2]}\\ \Delta;\Gamma, x : \tau^{(=1)} \vdash e_2 : \mathit{bool}^{m'}\\ \Delta;\Gamma, x : \tau^{(=1)} \vdash e_3 : \tau_2^{(\leq 1)}\end{array}}{\Delta;\Gamma \vdash e_1\ \mathit{filter}\,(x.e_2)\ \mathit{order}\,(x.e_3) : \tau^{[0,i_2]}}$$

(T-UPDATE)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : (T_N \mathbin{@} S)^{m}\\ \Delta;\Gamma \vdash s : (T_N \mathbin{@} \{\}) \Rightarrow (T_N \mathbin{@} \{\})\\ N := T \in \Delta\end{array}}{\Delta;\Gamma \vdash \mathit{update}\ e\, s : (T_N \mathbin{@} \{\})^{m}}$$

(T-IF)
$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \mathit{bool}^{m_1}\\ \Delta;\Gamma \vdash e_2 : \tau^{[i_1,i_2]} \qquad \Delta;\Gamma \vdash e_3 : \tau^{[i_3,i_4]}\\ m = m_1 \times ([\mathit{min}(i_1, i_3), \mathit{max}(i_2, i_4)])\end{array}}{\Delta;\Gamma \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : \tau^{m}}$$

(T-INSERT)
$$\frac{\begin{array}{c}N := \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\} \in \Delta\\ \forall_{i,1\leq i\leq n}.\ \Delta;\Gamma \vdash e_i : \tau_i^{m_i}\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash \mathit{insert}\ N\,\{l_1 := e_1,\\ \ldots, l_n := e_n\} : (N \mathbin{@} \{\})^{(=1)}\end{array}}$$

Fig. 2. Typing Rules for Expressions

$$\textsc{(UT-vset)}\quad \frac{\forall_{i,1\leq i\leq n}.\Delta \vdash_\mu V_i : \tau \qquad i_1 \leq n \leq i_2}{\Delta \vdash_\mu \{V_1, \ldots, V_n\} : \tau^{[i_1,i_2]}}
\qquad \textsc{(VT-prim)}\quad \frac{v : t}{\Delta \vdash_\mu v : t}
\qquad \textsc{(VT-ref)}\quad \frac{(id, N, W') \in \mu \qquad \Delta \vdash_\mu W : R}{\Delta \vdash_\mu \mathsf{ref}(id) \diamond W : (N\,@\{\}) \uplus R}$$

$$\textsc{(WT-obj)}\quad \frac{\forall_{i,1\leq i\leq n}.(\Delta \vdash_\mu U_i : \tau_i^{m_i}) \qquad T = \{l_i : \tau_i^{m_i} \mid L_i = l_i\}_{1\leq i\leq n} \qquad S = \{l_i : \tau_i^{m_i} \mid L_i = @l_i\}_{1\leq i\leq n}}{\Delta \vdash_\mu \{L_1^{u_1} := U_1, \ldots, L_n^{u_n} := U_n\} : T\,@\,S}$$

Fig. 3. Typing Rules for Value Sets, Values, and Object Values

$$\frac{\forall_{i,1\leq i\leq n}.\Delta;\Gamma, x : \tau^{(=1)} \vdash e_i : \tau_i^{m_i} \qquad T = \{l_i : \tau_i^{m_i} \mid L_i := l_i\}_{1\leq i\leq n} \qquad S = \{l_i : t_i^{m_i} \mid L_i := @l_i, \tau_i = t_i\}_{1\leq i\leq n} \qquad \tau = T'_N\,@\,S'}{\Delta;\Gamma \vdash \{L_i := x.e_1, \ldots, L_n := x.e_n\} : \tau \Rightarrow (\tau \uplus (T\,@\,S))}$$

Fig. 4. Typing Rules for Shapes

priority. Recall that since any link property type S is automatically an object type T, the operation $\uplus$ also applies to link property types. Essentially, we would like $W_1 \uplus W_2$ to have type $R_1 \uplus R_2$ if W_1 has type R_1 and W_2 has type R_2.

(1) **Union of object types.** ($T_1 \uplus T_2 = T'$) and ($S_1 \uplus S_2 = S'$)
If $T_1 = \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}$, and $T_2 = \{l'_1 : (\tau'_1)^{m'_1}, \ldots, l'_k : (\tau'_k)^{m'_k}\}$,
then $T_1 \uplus T_2 = \{(l_i : (\tau_i)^{m_i} \mid l_j \neq \{l'_i\}_{1\leq i\leq n})_{1\leq j\leq n}, l'_1 : (\tau'_1)^{m'_1}, \ldots, l'_k : (\tau'_k)^{m'_k}\}$.
That is, if we view an object type as the union of its object type components, then
$T_1 \uplus T_2 = (\cup_{i,1\leq i\leq n}\{l_i : (\tau_i)^{m_i} \mid \forall j.l_i \neq l'_j\}) \cup (\cup_{i,1\leq i\leq k}\{l'_i : (\tau'_i)^{m'_i}\})$.

(2) **Union of link types.** ($R_1 \uplus R_2 = R'$)
If $R_1 = T_l\,@\,S_l$ and $R_2 = T_r\,@\,S_r$, then $R_1 \uplus R_2 = (T_l \uplus T_r)\,@(S_l \uplus S_r)$.

(3) **Union of a nominal link type with a link type.** ($(T_N\,@\,S) \uplus R = T'_N\,@\,S'$)
If $R = T_2\,@\,S_2$, then $(T_N\,@\,S) \uplus (T_2\,@\,S_2) = (T \uplus T_2)_N\,@(S \uplus S_2)$.

4.3 Typing Rules for Shapes

The shape transformation judgment $\Delta;\Gamma \vdash s : \tau \Rightarrow \tau'$ synthesizes τ' given shape s and input expression type τ. This judgment is used in the rules (T-shape), (T-insert), (T-update). The input type τ is required to be a nominal link type and the judgment does not hold if τ is a primitive type. Properties not mentioned in the shape are kept instead of discarded, and they will be set to invisible in the operational semantics. The judgment is defined in Figure 4. We compute the type for each element in the shape (the τ_i's), and merge the resulting type ($T\,@\,S$) with the input expression type (τ).

5 EXECUTION SEMANTICS

We describe the execution semantics of AnonQL in this section.

5.1 Delayed Mutation

While an AnonQL program consists of a list of AnonQL expressions, each expression is evaluated as a whole, where the mutations to the database are queued and written once the evaluation of the

(E-val)
$$\frac{}{\Omega \vdash \mu \parallel v \searrow \mu \parallel \{v\}}$$

(E-var)
$$\frac{}{\Omega, x \mapsto U \vdash \mu \parallel x \searrow \mu \parallel U}$$

(E-empty-set)
$$\frac{}{\Omega \vdash \mu \parallel \{\}_\tau \searrow \mu \parallel \{\}}$$

(E-select)
$$\frac{\Omega \vdash \mu \parallel e \searrow \mu \parallel U}{\Omega \vdash \mu \parallel \mathit{select}\ e \searrow \mu \parallel U}$$

(E-lprop)
$$\frac{\Omega \vdash \mu \parallel e \searrow \mu' \parallel \{V_1, \ldots, V_n\}}{\Omega \vdash \mu \parallel e\,@\,l \searrow \mu' \parallel^{S} \cup_{1 \leq i \leq n} \{\mathit{proj}_{\mu'}(@l, V_i)\}}$$

(E-obj)
$$\frac{U = \{\mathsf{ref}(id) \diamond \{\} \mid (id, N, W) \in \mu'_0\}}{\Omega \vdash^{\mu'_0}_{\Delta} \mu \parallel N \searrow \mu \parallel U}$$

(E-proj)
$$\frac{\Omega \vdash \mu \parallel e \searrow \mu' \parallel \{V_1, \ldots, V_n\}}{\Omega \vdash \mu \parallel e \cdot l \searrow \mu' \parallel \cup_{1 \leq j \leq n}\ \mathit{proj}_{\mu'}(l, V_j)}$$

(E-detached)
$$\frac{\Omega \vdash \mu \parallel e \searrow \mu \parallel U}{\Omega \vdash \mu \parallel \mathit{detached}\ e \searrow \mu \parallel U}$$

(E-shape)
$$\frac{\Omega \vdash \mu \parallel e \searrow \mu_0 \parallel \{V_1, \ldots V_n\} \qquad \forall_{i, 1 \leq i \leq n}. V'_i = \mathit{view}^{\Omega}(s, V_i) : \mu_{i-1} \mapsto \mu_i}{\Omega \vdash \mu \parallel e\, s \searrow \mu_n \parallel \{V'_1, \ldots, V'_n\}}$$

(E-union)
$$\frac{\Omega \vdash \mu \parallel e_1 \searrow \mu' \parallel U_1 \qquad \Omega \vdash \mu' \parallel e_2 \searrow \mu'' \parallel U_2}{\Omega \vdash \mu \parallel e_1 \cup e_2 \searrow \mu'' \parallel U_1 \cup U_2}$$

(E-with)
$$\frac{\Omega \vdash \mu \parallel e_1 \searrow \mu' \parallel U_1 \qquad \Omega, x \mapsto U_1 \vdash \mu' \parallel e_2 \searrow \mu'' \parallel U_2}{\Omega \vdash \mu \parallel \mathsf{with}(x := e_1; e_2) \searrow \mu'' \parallel U_2}$$

(E-insert)
$$\frac{\forall_{i, 1 \leq i \leq n}.\ \Omega \vdash \mu_{i-1} \parallel e_2 \searrow \mu_i \parallel U_i \qquad N := T \in \Delta \qquad (id\ \mathsf{fresh}) \qquad u = \mathit{visible} \qquad \{l^u_1 := U_1, \ldots, l^u_n := U_n\} \lhd T \Rightarrow W' : \Delta}{\Omega \vdash^{\mu'_0}_{\Delta} \mu_0 \parallel_\Delta \mathit{insert}\ N\, s \searrow \mu_n \cup \{(id, N, W')\} \parallel_\Delta \{\mathsf{ref}(id) \diamond \{\}\}}$$

(E-if)
$$\frac{\Omega \vdash \mu \parallel e_1 \searrow \mu_0 \parallel \{V_1, \ldots V_n\} \qquad \begin{cases} \Omega \vdash \mu_{i-1} \parallel e_2 \searrow \mu_i \parallel U'_i & \text{if } V_i = \mathit{true} \\ \Omega \vdash \mu_{i-1} \parallel e_3 \searrow \mu_i \parallel U'_i & \text{if } V_i = \mathit{false} \end{cases}}{\Omega \vdash \mu \parallel \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \searrow \mu_n \parallel U'_1 \cup \cdots \cup U'_n}$$

(E-for)
$$\frac{\Omega \vdash \mu \parallel e_1 \searrow \mu_0 \parallel \{V_1, \ldots V_n\} \qquad \forall_{i, 1 \leq i \leq n}.\ \Omega, x \mapsto \{V_i\} \vdash \mu_{i-1} \parallel e_2 \searrow \mu_i \parallel U'_i}{\Omega \vdash \mu \parallel \mathsf{for}(x \leftarrow e_1; e_2) \searrow \mu_n \parallel U'_1 \cup \cdots \cup U'_n}$$

(E-ofor-1)
$$\frac{\Omega \vdash \mu \parallel e_1 \searrow \mu' \parallel \{\} \qquad \Omega, x \mapsto \{\} \vdash \mu' \parallel e_2 \searrow \mu'' \parallel U}{\Omega \vdash \mu \parallel \mathsf{optional\_for}(x \leftarrow e_1; e_2) \searrow \mu'' \parallel \{x \mid x \in U\}}$$

(E-ofor-2)
$$\frac{\Omega \vdash \mu \parallel e_1 \searrow \mu_0 \parallel \{V_1, \ldots V_n\}\ (n > 0) \qquad \forall_{i, 1 \leq i \leq n}.\ \Omega, x \mapsto \{V_i\} \vdash \mu_{i-1} \parallel e_2 \searrow \mu_i \parallel U'_i}{\Omega \vdash \mu \parallel \mathsf{optional\_for}(x \leftarrow e_1; e_2) \searrow \mu_n \parallel U'_1 \cup \cdots \cup U'_n}$$

(E-backlink)
$$\frac{\Omega \vdash \mu \parallel e \searrow \mu' \parallel \{\mathsf{ref}(id_1) \diamond W_1, \ldots, \mathsf{ref}(id_n) \diamond W_n\} \qquad U = \{\mathsf{ref}(id) \diamond \{\} \mid (id, N, W) \in \mu_0, \exists i. W = \{\ldots, l^u := \{\ldots, \mathsf{ref}(id_i)\,@\,W', \ldots\}, \ldots\}\}}{\Omega \vdash \mu \parallel e \cdot_{\leftarrow} l\ [is\ N] \searrow \mu' \parallel U}$$

(E-func)
$$\frac{\forall_{i, 1 \leq i \leq n}. \Omega \vdash \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U_i \qquad U' = f^*(U_1, \ldots, U_n)}{\Omega \vdash \mu_0 \parallel f(e_1, \ldots, e_n) \searrow \mu_n \parallel U'}$$

Fig. 5. Execution Semantics (Part 1)

(E-FILT)

$$\frac{\begin{array}{c}\Omega \vdash \mu \parallel e_1 \searrow \mu_0 \parallel \{V_1^{selected}, \dots, V_n^{selected}\} \\ \forall_{i,1\le i\le n}.\Omega, x \mapsto \{V_i^{selected}\} \vdash \mu_{i-1} \parallel e_2 \searrow \mu_i \parallel U_i^{cond} \\ \{V_1^{cond\text{-}ed},\dots,V_m^{cond\text{-}ed}\} = \cup_{i,1\le i\le n}.\{V_i^{selected} \mid true \in U_i^{cond}\} \\ \forall_{j,1\le j\le m}.\Omega, x\mapsto \{V_j^{cond\text{-}ed}\} \vdash \mu_{n+j-1} \parallel e_3 \searrow \mu_{n+j} \parallel U_j^{order} \\ U^{ordered} = orderby(\{(V_1^{cond\text{-}ed}, U_1^{order}),\dots,(V_m^{cond\text{-}ed},U_m^{order})\})\end{array}}{\Omega \vdash \mu \parallel e_1\ filter\ (x.e_2)\ order\ (x.e_3) \searrow \mu_{n+m} \parallel U^{ordered}}$$

(E-UPDATE)

$$\frac{\begin{array}{c}\Omega \vdash \mu \parallel e \searrow \mu_0 \parallel \{\mathrm{ref}(id_1) \diamond W_1, \dots, \mathrm{ref}(id_n) \diamond W_n\} \\ \forall_{i,1\le i\le n}.\ \Omega \vdash V_i = view^{\Omega}(s, \mathrm{ref}(id_i) \diamond W_i) : \mu_{i-1} \mapsto \mu_i \wedge V_i = \mathrm{ref}(id_i) \diamond W_i' \\ \mu' = \{(id, N, W) \mid \forall (id, N, W) \in \mu_n \wedge id \neq id_i\} \\ \cup\{(id_i, N, W_i'') \mid \forall (id, N, W) \in \mu_n . \exists i. id = id_i \wedge N := T \in \Delta \wedge (W \uplus W_i') \lhd T \Rightarrow W_i''\}\end{array}}{\Omega \vdash \mu \parallel_\Delta update\ e\ s \searrow \mu' \parallel_\Delta \{V_1, \dots, V_n\}}$$

Fig. 6. Execution Semantics (Part 2)

entire expression finishes. The effects of an insert or update in an expression will not be seen by an immediate read in the same expression, even if the insert is executed before the read. Before the evaluation of every expression, we take a snapshot μ_0 of the database, and the snapshot μ_0 will remain unchanged throughout the evaluation of a single expression. The current database μ will be modified as we evaluate expressions as expressions may insert or update data. From a technical perspective, this is a consequence of wishing to support implementation via compilation to SQL, which has similar behavior.

5.2 Evaluation Rules

An evaluation environment Ω is an unordered mapping from variable names to value sets. The evaluation judgment is written using the big-step evaluation relation

$$\Omega \vdash_{\Delta}^{\mu_0'} \mu \parallel e \searrow \mu' \parallel U$$

This judgment says that in an evaluation environment Ω, with the current database schema Δ and read snapshot μ_0', executing query e with respect to the database μ, will produce a new database μ' with the result U. To evaluate a query e on a database μ, we find the result value set U with the updated database μ' such that the judgment $\cdot \vdash_{\Delta}^{\mu} \mu \parallel e \searrow \mu' \parallel U$ holds.

The execution semantics is shown in Figure 5 and Figure 6. The read snapshot μ_0 and the database schema Δ always stay unchanged throughout the execution of a single expression. We will omit them in the rules and just write $\Omega \vdash \mu \parallel e \searrow \mu' \parallel U$ when neither μ_0' nor Δ is referenced in the rules.

5.3 Ordering.

As mentioned earlier, the result value set is an optionally ordered multiset. In our semantics, we take value set literals, set-builder notation, and unions to produce *unordered* value sets, while the *orderby* meta-operation produces an ordered value set containing the first elements of its input tuples ordered by the second elements of its input tuples.

$view^{\Omega}(\{L'_1 = x.e_1, \dots, L'_k = x.e_k\}, V) : \mu_0 \mapsto \mu_k = V'$ where
Let $\forall_{i,1 \le i \le k}.\Omega, x \mapsto \{V\} \vdash^{\mu'_0}_{\Delta} \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U'_i$,
and $W' = \{(L'_1)^{visible} := U'_1, \dots, (L'_k)^{visible} := U'_k\}$,
If $V = \mathrm{ref}(id) \diamond \{L_1^{u_1} := U_1, \dots, L_n^{u_n} := U_n\}$,
then $V' = \mathrm{ref}(id) \diamond ((\{L_1^{invisible} := U_1, \dots, L_n^{invisible} := U_n\}) \uplus W')$.
(Note: $view^{\Omega}(s, V) : \mu_0 \mapsto \mu'$ is implicitly parameterized by μ'_0 and Δ)

Fig. 7. The Definition of the $view^{\Omega}$ Operation

(1) $proj_{\mu}(l, \mathrm{ref}(id) \diamond \{L_1^{u_1} := U_1, \dots, L_n^{u_n} := U_n\}) =$

$$\begin{cases} U_i & \text{if } l = L_i \\ U'_j & \text{otherwise, if } (id, N, \{\dots, l^{u'_j} := U'_j, \dots\}) \in \mu'_0 \\ U'_j & \text{otherwise, if } (id, N, \{\dots, l^{u'_j} := U'_j, \dots\}) \in \mu \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

(2) $proj_{\mu}(@l, \mathrm{ref}(id) \diamond \{\dots, @l^{u} := U, \dots\}) = U$

(Note: $proj_{\mu}(L, V)$ is implicitly parameterized by μ'_0)

Fig. 8. The Definition of the $proj_{\mu}$ Operation

Thus, in the rule (E-FILT), $orderby(U_1, C_2)$ orders the set U_1 by the corresponding value set in the set of value sets C_2. Additionally, (E-SELECT) and (E-DETACHED) preserve any ordering on their sub-expression, while essentially every other operation produces unordered sets. (This mostly follows from the rule's natural definitions using set operations, though in the rule (E-OFOR-1) we use set-builder notation specifically to ensure an unordered result.)

These rules stem principally from implementation concerns: SQL also provides very limited guarantees about ordering, because implementations want to be free to choose the optimal join order between tables.

5.4 The View of a Shape on an Object.

A view on an object (a) computes the properties that are in the shapes and adds or overrides the properties in the object, and (b) sets the properties of the object that are not in the shapes to be invisible. Formally, we define the meta-level operation $view^{\Omega}(s, V) : \mu \mapsto \mu'$ to denote the result of applying a shape s on V, while in the process transforming the database store from μ to μ'. The meta-level $view^{\Omega}$ operations are implicitly parameterized by the runtime configuration μ_0 and Δ. They are inductively defined in Figure 7.

5.5 Object Projections

In Figure 8, we define the singular projection function $proj_{\mu}(L, V)$ by induction on (L, V). The function $proj_{\mu}$ implicitly parameter by the read snapshot μ'_0.

5.6 Storage Coercion

During an insert or update expression, we would like to make sure the objects we stored in the database are of a well-defined format, by dropping all computed properties and retain only link properties. We formally define the process of coercion by the judgment $W \lhd T \Rightarrow W' : \Delta$, which

We have

$$\{l_1^{u_1} := U_1, \ldots, l_n^{u_n} := U_n\} \lhd \{l_1 : \tau_1^{m_1}, \ldots l_n : \tau_n^{m_n}\} \Rightarrow \{l_1^{u_1} := U_1', \ldots, l_n^{u_n} := U_n'\} : \Delta$$

where

$$U_i' = \begin{cases} U_i & \text{if } \tau_i \text{ is a primitive type} \\ \{\mathsf{ref}(id) \diamond W_j' \mid \mathsf{ref}(id_j) \diamond W_j \in U_i\} & \text{if } \tau_i \text{ is a link type } N_i \mathbin{@} S_i \end{cases}$$

where W_j' is W_j with only link props, $W_j' = \{@l_j'^{(u_j)} := U_j'' \mid @l_j'^{(u_j)} := U_j'' \in W_j\}$

Fig. 9. The Definition of Storage Coercion

says that when the object value W is coerced to object type T, the result is W'. The definition is shown in Figure 9.

6 METATHEORETIC PROPERTIES

We state several theorems regarding the formalism defined previously. Detailed proofs can be found in Appendix B.

Theorem 6.1 (Decidability of Type Checking). *Given a well-formed schema Δ, it is decidable whether $\Delta; \Gamma \vdash e : \tau$.*

Proof. Directly by induction on the type derivation. □

Theorem 6.2 (Canonical Forms). *We have a canonical forms lemma for value sets, values, and value objects. For reasons of space, we elide all but one case here.*

If $\Delta \vdash_\mu W : R$, then $W = \{l_1^{u_1} := U_1, \ldots, l_n^{u_n} := U_n, (@l_1')^{u_1'} := U_1', \ldots, (@l_k')^{u_k'} := U_k'\}$, where $R = T \mathbin{@} S$, $T = \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}$, $S = \{l_1' : t_1^{m_1'}, \ldots, l_n' : t_n^{m_n'}\}$, $\forall_{i, 1 \le i \le n}. \Delta \vdash_\mu U_i : \tau_i^{m_i}$, and $\forall_{i, 1 \le j \le k}. \Delta \vdash_\mu U_j' : t_j^{m_j'}$.

Proof. By induction on the typing derivations. □

Given a database schema Δ, and a well-formed database μ, we say that an evaluation environment Ω is well-formed with respect to a static context Γ, written $\Omega : \Gamma$, if Ω and Γ have the same set of variables and for all x in the domain of Ω and Δ, $\Delta \vdash_\mu \Omega(x) : \Gamma(x)$.

Similarly, we define $\mu : \Delta$ to mean that the database store μ is well-formed with respect to Δ. That is, for all $(id, N, W) \in \mu$ and $N := T \in \Delta$, $\Delta \vdash_\mu W : T \mathbin{@} \{\}$.

Theorem 6.3 (Preservation). *If $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e \searrow \mu' \parallel U$, and $\Delta; \Gamma \vdash e : \tau^m$, where $\Omega : \Gamma$, $\mu : \Delta$, and $\mu_0' : \Delta$, then $\Delta \vdash_\mu U : \tau^m$.*

Proof. By induction on the big step evaluation relation. □

Theorem 6.4 (Progress/Normalization). *If $\Delta; \Gamma \vdash e : \tau^m$, $\Omega : \Gamma$, $\mu : \Delta$, and $\mu_0' : \Delta$, then there exists U such that $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e \searrow \mu' \parallel U$.*

Proof. By induction on the typing derivation. □

7 PATH FACTORIZATION

In the AnonDB implementation of AnonQL, path factoring is a transformation on expressions that happens before type checking and query execution. The primary goal is to ensure that names in a query are behaving correctly, by factoring all the names in a query to top-level optional_for loops.

7.1 The Rationale of Path Factoring

Consider a database for storing social network information. Developers are used to writing the following SQL query that finds the first name and last name of any `Person` whose first name begins with the letter 'A'.

```
select Person.first_name, Person.last_name
from Person
where Person.first_name LIKE 'A%'
```

If coming directly from a SQL background, one may be tempted to try something like:

```
select (Person.first_name, Person.last_name)
filter Person.first_name LIKE 'A%'
```

where the operator `(_, _)` is a tuple construction function of type $[\tau_1^1, \tau_2^1] \rightarrow \tau_{1\times 2}^1$ [3] and `LIKE` is an infix string matching function of type $[str^1, str^1] \rightarrow bool^1$. The above query in the abstract syntax would be: $(\texttt{Person} \cdot \texttt{first\_name}, \texttt{Person} \cdot \texttt{last\_name})\ \mathit{filter}\ (y.(\texttt{Person} \cdot \texttt{first\_name}\ \mathit{LIKE}\ \text{“}\mathit{A\%}\text{”}))$.

Let us assume that there are two people in total two people “Alice Johnson” and “Bob Martinez” in the database. The intended result set should consist of a single tuple `("Alice", "Johnson")`. However, according to the execution semantics, the query will first evaluate $\texttt{Person} \cdot \texttt{first\_name}$ and $\texttt{Person} \cdot \texttt{last\_name}$ separately, aggregating the result using the broadcasting semantics of functional calls. That is, `(Person.first_name, Person.last_name)` will evaluate to the result set consisting of four tuples. Then the filter expression will evaluate to the result set $\{true, false\}$ according to the broadcasting effect of `LIKE`, and will be treated as *true* because filters aggregate the result using the *any* operator. In the end, the actual result set is
`{("Alice", "Johnson"), ("Alice", "Martinez"),`
`("Bob", "Johnson"), ("Bob", "Martinez")}`.
The actual result is dramatically different from the intended result.

Path factorization is a systematic procedure that transforms a query into a form that is more aligned with the SQL convention. In this particular case, we want the name `Person` to be factored out, resulting in this query that aligns with the intended behavior:

```
for optional X in Person union
select (X.first_name, X.last_name)
filter X.first_name LIKE 'A%'
```

Or, in our abstract syntax:

$$\begin{array}{l}\text{optional\_for}(X \leftarrow \texttt{Person};\\ \quad (X \cdot \texttt{first\_name}, X \cdot \texttt{last\_name})\ \mathit{filter}\ (y.(X \cdot \texttt{first\_name}\ \mathit{LIKE}\ \text{“}\mathit{A\%}\text{”})))\end{array}$$

For the full path factoring algorithm, we explicitly bind all paths, so we will produce this equivalent but more explicit query:

```
for optional X in Person union
for optional Y in X.first_name union
for optional Z in X.last_name union
select (Y, Z)
filter Y LIKE 'A%'
```

The rest of this section describes the detailed rules for path factorization.

[3] $\tau_{1\times 2}$ is actually a product type of τ_1 and τ_2 in the AnonDB implementation. However, due to space constraints, we did not include product types in the core AnonQL calculus.

7.2 Fences and Binding Points

Any expression is a query. A subexpression may be a subquery, a semisubquery, or neither. In general, arguments with parameter modifiers $*$ are subqueries, and arguments with parameter modifiers ? are semisubqueries. Arguments with parameter modifiers 1 are neither subqueries nor semisubqueries. The detailed definition is shown in Figure 10, where we write $e_1 \prec e_2$ to indicate that e_1 is a subquery of e_2, write $e_1 \preceq e_2$ to indicate that e_1 is a semisubquery of e_2, and write $e_1 \sim e_2$ to indicate that e_1 is neither a subquery nor a semisubquery of e_2. Expressions inside a shape always appear as a subquery.

A binding point is a location where a name may be factored by inserting optional_for. A fence acts as a barrier to factoring. A subquery introduces both a fence and a binding point. A semisubquery introduces only a binding point but not a fence. The root of an expression has an implicit binding point.

A path is an expression that begins with a type name N' or variable x and is followed by a consecutive list of $\cdot l$, $\cdot_{\leftarrow} l$ $[\mathit{is}\ N]$, or $@\, l$. We say that a path *occurs at a binding point* if there exists an occurrence of the exact same path head and path components such that a fence is not blocking such occurrence from the binding point. A path is maximal (longest) if there are no more $\cdot l$, $\cdot_{\leftarrow} l$ $[\mathit{is}\ N]$, or $@\, l$ following the path.

A path P is factored at a binding point B if
1. The path P occurs at the binding point B.
2. If the path P is not maximal, then P extended with the next path component does not occur at the binding point B.
3. There does not exist another binding point B' that is further away from P than B that contains different set of occurrences of P than B.
4. There does not exist a *detached* expression node between B and P.

We remark that prefixes of a single path may be factored at different binding points, but there will be at most one binding point at which each prefix can be factored. Not all prefixes of a path need to be factored (see examples below).

At each binding point, the optional_for's are inserted as follows. Compute the proper set (extra copies of identical paths are removed) of paths factored at this binding point, together with their common longest prefixes, and all paths in the set are factored iteratively (factored prefixes of later paths are replaced by the binders introduced by *optional_for*) and lexicographically (shorter paths come first). To recover the link deduplication feature (Section 8) of the AnonDB implementation, a *distinct* function call may be inserted for projections that compute a set of references.

We illustrate this definition through a series of examples. We use the tuple construction $(\_,\_) : [\tau_1^1, \tau_2^1] \rightarrow \tau_{1\times 2}^1$ defined earlier, and the coalescing operator $?? : [\tau^?, \tau^*] \rightarrow \tau^{(*)}$ and $\mathit{count} : [\tau^*] \rightarrow \mathit{int}^{(=1)}$ function discussed in Section 3.7.

1. $(\texttt{Person} \cdot \texttt{first\_name}, \texttt{Person} \cdot \texttt{last\_name})$ — Both arguments to the function $(\_,\_)$ have parameter modifier 1, and thus there is only one binding point, i.e. the root binding point. There are four paths, $\texttt{Person} \cdot \texttt{first\_name}$, $\texttt{Person} \cdot \texttt{last\_name}$, `Person` (in $\texttt{Person} \cdot \texttt{first\_name}$), `Person` (in $\texttt{Person} \cdot \texttt{last\_name}$). All four paths occur at the root binding point. However, only $\texttt{Person} \cdot \texttt{first\_name}$ and $\texttt{Person} \cdot \texttt{last\_name}$ are factored at the root because the other two paths fail condition (2). Then, the common longest path prefixes between the two factored paths are calculated and factored. The final factored paths at the root are `Person`, $\texttt{Person} \cdot \texttt{first\_name}$, $\texttt{Person} \cdot \texttt{last\_name}$.
2. $(\texttt{Person} \cdot \texttt{first\_name}, \texttt{Person} \cdot \texttt{last\_name})\ \mathit{filter}\ (y.(\texttt{Person} \cdot \texttt{first\_name}\ \mathit{LIKE}\ \text{“}A\%\text{”}))$ —

The *filter* clause introduces a binding point and a fence. There are two additional paths to consider: `Person` (in the *filter* clause) and `Person · first_name` (in the filter clause). We see that `Person · first_name` cannot be factored at the binding point within the *filter* clause because of condition (3), and instead should be factored at the root. The `Person` cannot be factored at either binding point because of the condition (2). In the end, we factor `Person`, `Person. first_name`, and `Person. last_name` at the root, resulting in the query presented at the end of Section 7.1.

(3) ((*select* `Person · first_name`), (*select* `Person · last_name`)) — *select* introduces a subquery, which introduces both a fence and a binding point. No path will be factored at the root point. We get a set cross product of first names and last names of all persons.

(4) *count*(`Person`) — Since the argument to *count* has parameter modifier $*$, it introduces a subquery. `Person` is not factored at the root point, and the query returns the total number of Person objects as it should.

(5) (`Person · first_name`, (*select* `Person · last_name`)) — The path `Person · last_name` will be factored at the inner binding point introduced by *select*. However, the path `Person` in the select clause will be factored at the root: condition (2) now holds! The end result of this query would be the same as (`Person · first_name`, `Person · last_name`).

(6) (`Person · first_name`, *count*(`Person. friends`)) — *count* again introduces a subquery, but like the above example, `Person` is factored at the root. This means that *count* will be called on the set of friends for each user, so the query will return tuples of names and the number of friends the person has.

(7) (`Person · first_name`, (*detached* `Person · last_name`)) — No paths in the `detached` clause will be factored at the root because of condition (4). The end result of this query would be the same as ((*select* `Person · first_name`), (*select* `Person · last_name`)).

(8) (`Person · first_name`, `Person · last_name`) ??(“*NA*”, “*NA*”) — The first argument to the function ?? has parameter modifier ?, which introduces only a binding point but no fences. We see that both paths `Person. first_name` and `Person. last_name` will be factored at this introduced binding point. The inner binding point will also factor `Person` because it is a common longest prefix. No paths will be factored at the root binding point because of condition (3): the root binding point contains the same set of occurrences as the inner binding point.

(9) ((`Person · first_name` ?? “*NA*”), (`Person · last_name` ?? “*NA*”)) — Because there is no fence in the query, both paths `Person. first_name` and `Person. last_name` will be factored at the root binding point. The root binding point will also factor `Person` because it is a common longest prefix.

We also have an account of path factoring in terms of computing a set of paths for factoring at each binding point in a top-down manner. Due to space constraints, we are unable to present it in the main body of the paper. Interested readers may refer to Appendix A for a detailed presentation.

8 THE ANONDB IMPLEMENTATION

AnonDB [Anonymous Citation] is a production implementation of AnonQL that uses PostgreSQL [13] as a backend. AnonDB schemas are converted to PostgreSQL schemas in which each object type has a table and single properties and links are stored as columns in that table; multi properties and links are stored in separate link tables. The target of a link is represented as the primary key of the target object, and link properties are stored as columns in the link table.

An AnonQL expression is compiled into a single SQL query that is sent to the PostgreSQL server for execution. Records are used to return shapes from the query, with arrays used for attributes

e	Top Level or Subquery
$f(e_1, \ldots, e_n)$	$f : [\tau_1^{p_1}, \ldots, \tau_n^{p_n}] \rightarrow \tau^m$ $\forall_{i, 1 \leq i \leq n}. \begin{cases} e_i \sim e & \text{if } p_i = 1 \\ e_i \preceq e & \text{if } p_i = ? \\ e_i \prec e & \text{if } p_i = * \end{cases}$
$\langle\tau\rangle(e_1)$, $e_1\ s$, $e_1 \cdot l$, $e_1 \mathbin{@} l$, $e_1 \cdot_{\leftarrow} l$, $e_1\ [\mathit{is}\ N]$, $\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3$, $e_1\ \mathit{filter}\ (x.e_2)\ \mathit{order}\ (x.e_3)$, $\mathit{insert}\ N\ s$, $\mathit{update}\ e_1\ s$, $\mathit{delete}\ e_1$	$e_1 \sim e, e_2 \prec e,$ $e_3 \prec e, s \prec e$
$\mathit{select}\ e_1$, $\mathit{detached}\ e_1$, $e_1 \cup e_2$, $\mathrm{for}(x \leftarrow e_1; e_2)$, $\mathrm{optional\_for}(x \leftarrow e_1; e_2)$, $\mathrm{with}(x := e_1; e_2)$	$e_1 \prec e, e_2 \prec e$

s	Top Level or Subquery
$\{L_1 := x.e_1, \ldots, L_n := x.e_n\}$	$\forall_{i, 1 \leq i \leq n}. e_i \prec s$

Fig. 10. Subquery and Semi-Subquery Relations

that can contain more than one element. The types are returned to client libraries and are used to efficiently decode the response (queries can also be configured to return JSON directly). AnonDB stores the database schema in itself, using a bootstrapped AnonQL query to load it on server startup; the schema database can be queried by users, as well.

Beyond the core AnonQL calculus, AnonDB supports more developer-friendly features such as

(1) **Delete Queries and Exceptions.** The core calculus omits any command for deleting an object from the database. This is because it is necessary to account for the case where a deleted object is still referenced by a link. By default, in AnonDB, this raises an error, and we did not wish to clutter the presentation of the core calculus with error propagation.
(2) **Default Values in Inserts.** Property and link declarations in the schema may have "default values" specified. Those values will be inserted if no other value is specified. Despite the traditional name, they may be arbitrary expressions. For optional properties and links, a value may be omitted from inserts and an empty set will be used.
(3) **Free Objects.** Queries may write bare shapes, unconnected to an object type in the schema. These ad-hoc anonymous objects are called "free objects" and are useful
(4) **Inheritance.** Object types may be declared to inherit from other types. Selecting a parent type will also return all objects of its descendant types.
(5) **Constraints.** Object types may have constraints on them that restrict the values of properties. An *exclusive* constraint requires that a property's value be unique among all objects of a type. Cardinality inference understands exclusive constraints, such that queries such as `select User filter .name = 'Alice'` will be inferred to return at most one element, if `name` is exclusive.
(6) **Access Policies.** Object types may specify conditions that must hold in order for them to be accessed. Separate conditions may be specified for selecting objects and for modifying them, and these conditions may refer to "global variables" that are configured by the connection to the database.
(7) **Triggers and Rewrites.** Object types may specify code that is run after an insert, update, or delete on that type, as well as expressions that modify the data passed to insert or update.

However, the AnonDB implementation also has restrictions not imposed by the AnonQL core calculus, including:

(1) **Restricted Mutation.** AnonDB currently bans mutation in several places, such as the filter clause of a select statement.

(2) **Altering Shape Component Types.** AnonDB currently does not allow altering shape component types, e.g. `select Person { name := “Alice” } { name := 2 }`.
(3) **Link Deduplication.** In AnonDB, links are required to be proper sets (without duplicates), and dereferencing a link always returns a proper set, even in complex expressions such as `User.friends.friends`. This link deduplication behavior can be recovered by inserting a *distinct* function call (that removes duplicate references) in appropriate places during the path factorization algorithm.

9 RELATED WORK

Inspiration. Many of the individual components of AnonQL and the graph relational model have clear inspirations or origins in other work.

Using dot notation to represent implicit joins has been done many times, including GEM (General Entity Manipulator) [18], “a query language for complex objects” [10], and Lorel [1]. The path factoring system is also largely inspired by Lorel. While these systems using dot notation for implicit joins typically also support some link-like notion of references to other objects, they do not have an equivalent to link properties (graph databases, however, do, as edges can be labeled).

Shapes take some inspiration from UnQL [4], a query language for specifying structurally recursive transformations on semistructured data such as XML, and from “a query language for complex objects“. They also have clear similarities to GraphQL [9, 15] queries, in that they return nested data; AnonQL shapes, however, support extended projections (i.e. arbitrary computations), whereas GraphQL selection sets do not and are basic projections.

The idea of allowing set-valued attributes can be viewed as simply rejecting the first normal form assumption, which has been frequently explored including in [18] and [10]. These approaches tend to treat these attributes as having a different *type*, in contrast to the AnonQL approach of treating them uniformly with singletons and distinguishing them by a cardinality checking system.

Other Approaches. SQL [8] is, of course, the dominant query language for relational databases, to the point that non-relational databases are frequently called “NoSQL”. [4]

As previously discussed, object-relational mappers [12, 16, 17] are libraries that attempt to map relational databases into object definitions in the programming language. They are typically tied to a single language, generally perform worse, and are less flexible than a full query language.

10 CONCLUSION

The graph-relational model can be seen as the elevation of several common relational database patterns to the level of first-class features of the model. This enables the use of a query language that enables powerful and convenient querying of structured data.

[4]On one occasion, when the second author told a newly-met acquaintance about working on a database, the acquaintance asked “SQL or NoSQL?” We would say, “Post-SQL!”

REFERENCES

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The lorel query language for semistructured data. *Int. J. Digit. Libr.*, 1(1):68–88, 1997.

[2] Renzo Angles. A comparison of current graph database models. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, pages 171–177. IEEE Computer Society, 2012.

[3] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[4] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[5] Donald D. Chamberlin. Early history of SQL. *IEEE Ann. Hist. Comput.*, 34(4):78–82, 2012.

[6] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev.*, 20(6):560–575, 1976.

[7] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In Gene Altshuler, Randall Rustin, and Bernard D. Plagman, editors, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes*, pages 249–264. ACM, 1974.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[9] Olaf Hartig and Jorge Pérez. Semantics and complexity of GraphQL. In Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1155–1164. ACM, 2018.

[10] Seong-Gi Kim and Sukho Lee. A formalization of a query language for complex objects. In C. Jinshong Hwang and Richard S. Brice, editors, *Proceedings of the 19th annual conference on Computer Science, CSC '91, San Antonio, Texas, USA, March 4-7, 1991*, pages 136–145. ACM, 1991.

[11] MongoDB, Inc. MongoDB: The Developer Data Platform. https://mongodb.com/. Accessed: 2023-09-01.

[12] Elizabeth J. O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1351–1356. ACM, 2008.

[13] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/. Accessed: 2023-09-01.

[14] Redis. Redis | The Real-time Data Platform — redis.com. https://redis.com. Accessed 07-11-2023.

[15] The GraphQL Foundation. GraphQL | A query language for your API. https://graphql.org/. Accessed: 2023-09-01.

[16] Alexandre Torres, Renata Galante, Marcelo Soares Pimenta, and Alexandre Jonatan B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Inf. Softw. Technol.*, 82:1–18, 2017.

[17] Catalin Tudose and Carmen Odubasteanu. Object-relational mapping using jpa, hibernate and spring data JPA. In *23rd International Conference on Control Systems and Computer Science, CSCS 2021, Bucharest, Romania, May 26-28, 2021*, pages 424–431. IEEE, 2021.

[18] Carlo Zaniolo. The database language GEM. In David J. DeWitt and Georges Gardarin, editors, *SIGMOD'83, Proceedings of Annual Meeting, San Jose, California, USA, May 23-26, 1983*, pages 207–218. ACM Press, 1983.

A PATH FACTORIZATION ALGORITHMS

We present an elaboration-based path factorization algorithm.

A.1 Paths and Path Substitutions

Let the symbol $\hookrightarrow l$ denote either $\cdot l$, $\cdot_{\leftarrow} l$ $[is\ N]$, or $@\,l$, and let R denote a type name N' or variable x. A *path* P in expression e is a subexpression of the form $R \hookrightarrow_1 l_1 \cdots \hookrightarrow_n l_n$. A path is maximal if there are no more trailing $\cdot l$ or $\cdot_{\leftarrow} l$ $[is\ N]$ or $@\,l$ following a path.

We may need to replace all particular paths in a query by another expression, for instance, when translating from the query $(X \cdot \mathit{first}, X \cdot \mathit{last})$, we may replace $X.\mathit{first}$ by Y, and getting $(Y, X \cdot \mathit{last})$. We denote such substitution by $\langle e_1/P \rangle e_2$, which replaces path P in e_2 by e_1 (except those appearing in a detached subexpression).

A.2 Pre-Top-Levels and Proper-Top-Levels of a Query

In the description below, we assume the top-level relation $\sim$, the subquery relation $\prec$, and the semi-subquery relation $\preceq$ are transitive. That is, if $e_1 \sim e_2$ and $e_2 \sim e_3$, then $e_1 \sim e_3$, and similarly for $\prec$ and $\preceq$.

The *pre-top-level* of e are positions in e that do not belong to any of its subqueries. A path P *appears in the pre-top-level* of e if it appears in e and such occurrence is not in any $e' \prec e$ (but may occur in semi-subqueries, i.e. $e'' \preceq e$), and it is maximal. A path P *appears in the proper-top-level* if it appears in the pre-top-level, and the path P and all its prefixes do not solely appear in the same semi-subquery (and not appear elsewhere in the pre-top-level and subqueries) of e. As an example, in the query

`(Person.first_name ++ Person.last_name) ?? "<name>"`

where ?? is the builtin-function of type $[str^?, str^1] \to str^{(=1)}$ has pre-top-level paths `Person.first_name` and `Person.last_name`, but has no proper-top-level path because both names appear solely in the same subquery and all the paths together with their prefixes only occur in this semi-subquery. On the other hand, in the query

`(Person.first_name ?? "<fst>") ++ (Person.last_name ?? "<lst>")`

both paths `Person.first_name` and `Person.last_name` occur in the proper-top-level because each path has a prefix `Person` that appears elsewhere in the pre-top-level.

A *factoring prefix* P_h of a proper-top-level path P of e is the longest prefix of the path that appears elsewhere (not considering detached subqueries) in e. The factoring prefix P_h of a proper-top-level path P is the longest in the sense that it is the maximal prefix that satisfies the condition: the occurrences of P_h in e do not coincide with the occurrences of P_h extended with the next path component in P. In the second example above, `Person` is a factoring prefix of `Person.first_name` as it appears in `Person.last_name`.

A.3 A Method of Computing Factoring Prefixes

Let $\mathit{toppath}(e)$ denote the list of all paths appearing in the proper-top-level of e and their factoring prefixes, without duplicates, ordered lexicographically, and let n denote the length of the list, $\mathit{toppath}(e)$ may be computed as follows:

The common longest path prefix (function clpp) between path $R \hookrightarrow_1 l_1 \cdots \hookrightarrow_n l_n$ and $R' \hookrightarrow'_1 l'_1 \cdots \hookrightarrow'_m l'_m$ is the longest path that is the prefix of both paths and is undefined/empty otherwise.

Let the common longest path prefix set (function clpps) of a set of paths S be the set of nonempty prefix paths any two paths of set S $\mathit{clpps}(S) = \{\mathit{clpp}(s,t) : \forall s,t \in S\}$. It will be the case that $S \subseteq \mathit{clpps}(S)$. We also define separate common longest path prefix set $\mathit{sclpps}(A,B) = \{\mathit{clpp}(s,t) : \forall s \in A, \forall t \in B\}$.

(1) Let A be the set of all paths appearing in the proper-top-level of e.
(2) Let B be the set of all paths appearing in e and all its subqueries but not in a detached subexpression.
(3) For each $b_i \in B$, let C_i be the set of common longest path prefixes among b_i and A, i.e., $C_i = sclpps(A, \{b_i\})$
(4) Let D be the set of all paths that are followed by a link projection and then a link property projection. If $P \cdot l \,@\, l_2 \in C_i$, $P \cdot l \,@\, l_2 \in A$, , $P \cdot_{\leftarrow} l\ [is\ N] \,@\, l_2 \in A$, or $P \cdot_{\leftarrow} l\ [is\ N] \,@\, l_2 \in C_i$, then $P \in D$.
(5) $toppath(E)$ will be A union the all the C_i's union D, deduplicated and sorted, i.e.,

$$toppath(E) = sorted(distinct(clpps(A) \cup (\cup_i C_i) \cup D))$$

.

A.4 Deduplication on References

As part of path factorization, deduplication is performed on all the paths that compute a set of references. As an implementation technique, define function $cond\_dedup : [\tau^1] \longrightarrow \tau^{(=1)}$ be a function that deduplicates a set of references (by concatenating the computed object value components) but does nothing on a set of primitive values. In practice, this function will not use the usual evaluation semantics but rather be evaluated on the input value set as a whole. That is, during evaluation, the function $cond\_dedup$ would have type $[\tau^*] \longrightarrow \tau^{(*)}$. However, the signature $[\tau^1] \longrightarrow \tau^{(=1)}$ should be used during type checking to preserve the cardinality of input paths. As a summary, we should have

$$cond\_dedup(U) = \begin{cases} U & \text{if } U = \{v_1, \ldots, v_n\} \\ \cup_{id}\{\mathsf{ref}(id) \diamond (\uplus_{W'}\{W' \mid \mathsf{ref}(id) \diamond W' \in U\}) \mid \exists W.\, \mathsf{ref}(id) \diamond W \in U\} & \text{otherwise} \end{cases}$$

Because of the function type $[\tau^*] \longrightarrow \tau^{(*)}$, the lifted version $cond\_dedup^*$ behaves the same as the primitive version $cond\_dedup$.

A.4.1 Alternative Definition. It might be better to think of $cond\_dedup(e)$ as a separate expression form that has its own type-checking rule and evaluation rule as follows.

$$\frac{\Delta; \Gamma \vdash e : \tau^m}{\Delta; \Gamma \vdash cond\_dedup(e) : \tau^m} \qquad \frac{\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e \searrow \mu' \parallel U}{\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel cond\_dedup(e) \searrow \mu' \parallel cond\_dedup(U)}$$

A.4.2 Insertion of Deduplication Operation. We define the *dedup_insert* operation that inserts the *cond_dedup* function call on all projections and reverse projections. This operation is applied to all paths that are factored.

$dedup\_insert(R \looparrowright_1 l_1 \cdots \looparrowright_n l_n) = cond\_dedup\,(\ldots cond\_dedup(R \looparrowright_1 l_1) \cdots \looparrowright_n l_n)$

Actually, we do not need to insert *cond_dedup* for link property projections (i.e. $e \,@\, l$), but since the target of link properties is always primitive, inserting them is fine.

A.5 The Factoring Process

Given an expression e, we compute $toppath(e)$, the list of all paths appearing in the proper-top-level of e and their factoring prefixes, without duplicates, ordered lexicographically, as defined in the previous subsection. Let n denote the length of the $toppath(e)$, and let P_i denote the ith path in $toppath(e)$,

$$
\begin{aligned}
\mathit{select\_factor}(e) = {} & \mathsf{optional\_for}(Y_1 \leftarrow \mathit{dedup\_insert}(P_1); \\
& \quad \mathsf{optional\_for}(Y_2 \leftarrow \mathit{dedup\_insert}(\langle Y_1/P_1\rangle P_2); \\
& \quad \mathsf{optional\_for}(Y_3 \leftarrow \mathit{dedup\_insert}(\langle Y_1/P_1\rangle\langle Y_2/P_2\rangle P_3); \\
& \quad \ldots \\
& \quad \mathsf{optional\_for}(Y_n \leftarrow \mathit{dedup\_insert}(\langle Y_1/P_1\rangle \cdots \langle Y_{n-1}/P_{n-1}\rangle P_n); \\
& \quad \mathit{sub\_select\_factor}(\langle Y_1/P_1\rangle \cdots \langle Y_n/P_n\rangle e)) \ldots)))
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{select\_factor}(e = e_1 \ \mathit{filter}\ (x.e_2)\ \mathit{order}\ (x.e_3)) = \\
& \quad ((\mathsf{optional\_for}(Y_1 \leftarrow \mathit{dedup\_insert}(P_1); \\
& \quad \mathsf{optional\_for}(Y_2 \leftarrow \mathit{dedup\_insert}(\langle Y_1/P_1\rangle P_2); \\
& \quad \mathsf{optional\_for}(Y_3 \leftarrow \mathit{dedup\_insert}(\langle Y_1/P_1\rangle\langle Y_2/P_2\rangle P_3); \\
& \quad \ldots \\
& \quad \mathsf{optional\_for}(Y_n \leftarrow \mathit{dedup\_insert}(\langle Y_1/P_1\rangle \cdots \langle Y_{n-1}/P_{n-1}\rangle P_n); \\
& \quad \mathsf{for}(\mathit{subject} := \mathit{sub\_select\_factor}(\langle Y_i/P_i\rangle_{1\le i\le n} e_1) \\
& \qquad \mathit{filter}\ (\mathit{select\_factor}(\langle Y_i/P_i\rangle_{1\le i\le n}(x.e_2)))\ \mathit{order}\ \{\}); \\
& \quad F\ \{\mathit{subject} := \_.\ \mathit{subject}, \\
& \qquad \mathit{order} := \_.\ (\mathit{select\_factor}(\langle Y_i/P_i\rangle_{1\le i\le n}([\mathit{subject}/x]e_3)))\} \\
& \quad) \ldots)))) \ \mathit{filter}\ (x.\ \mathit{True})\ \mathit{order}\ (x.x \cdot \mathit{order})) \cdot \mathit{subject}
\end{aligned}
$$

Fig. 11. Path Factoring Algorithms

let *select_factor* be the function that does path factor on a query, and let *sub_select_factor* be the function that calls *select_factor* on all direct subqueries and semi-subqueries. The definition is in the top part of Figure 11, where Y_i's are fresh.

The above query doesn't quite work when $e = e_1\ \mathit{filter}\ (x.e_2)\ \mathit{order}\ (x.e_3)$ because the factored for on the outside will break the aggregate nature of the *order* function. Thus, we have the following workaround: we pack the ordering clause into an object when executing the content of select and filter. Then we wrap the entire for clauses into an outside ordering and finally project out the subject. It is formally defined in the bottom part of Figure 11, where $\langle Y_i/P_i\rangle_{1\le i\le n}$ is a shorthand for $\langle Y_1/P_1\rangle \cdots \langle Y_n/P_n\rangle$, and F stands for a built-in type defined to be $\{\}$ and always has exactly one object in μ_0. We use F to construct a temporary object to temporarily project out its properties.

A.6 Detached Queries

Neither path factoring nor path substitution ever applies across *detached* queries. However, path factoring still occurs inside *detached* queries.

B DETAILED PROOFS OF THE METATHEOREMS

We state several theorems regarding the formalism defined previously, they are the decidability of type checking (Theorem B.3), the canonical-forms theorem (Theorem B.4), the preservation theorem (Theorem B.16), and the progress/normalization theorem (Theorem B.22).

The proofs of the preservation theorem (Theorem B.16) and the view/shape typing lemma (Lemma B.15) are mutually recursive. The proof of the progress/normalization theorem (Theorem B.22) appeals to the preservation theorem and is mutually recursive with the proof of the view/shape progress lemma (Lemma B.21).

We recall the following definitions of well-formed evaluation environments and well-formed database store.

(1) $\Omega : \Gamma$ iff and for all x in the domain of Ω and Δ, $\Delta \vdash_\mu \Omega(x) : \Gamma(x)$.
(2) $\mu : \Delta$ iff for all $(id, N, W) \in \mu$ and $N := T \in \Delta$, $\Delta \vdash_\mu W : T\,@\{\}$.

B.1 Decidability of Type Checking

Theorem B.1 (Decidability of Type Equality). *Given well-formed schema Δ, type equality is decidable.*

Proof. The equality algorithm will be a direct structural comparison except when we compare $N\,@\,S$ and $T'_{N'}\,@\,S'$, where we name expand $N\,@\,S$ to $T_N\,@\,S$ and T is the type of N. Informally, given two types τ and σ, define the metric to be the maximum depth of the two terms. All structure comparisons will break the equality problems into problems of lesser complexity, except the name expansion $N\,@\,S$ to $T_N\,@\,S$. However, because schemas are shallow, such breakdown will directly be followed by a structural comparison that decreases the metric.

Formally, define a measure $d(\tau)$ as

$$d(\tau) = \begin{cases} 0 & \text{if } \tau = t \\ 0 & \text{if } \tau = N\,@\,S \\ d(T) + 1 & \text{if } \tau = T_N\,@\,S \end{cases}$$

and $d(T)$ is defined as $d(\{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}) = max_i(d(\tau_i)) + 1$. It is clear that since the target type for any type definition in the schema is either t or $N\,@\,S$, $d(T) = 1$ for any $N := T \in \Delta$.

Given an equality relation $\tau = \sigma$, define the metric to be $max(d(\tau), d(\sigma))$. Name expansion is only performed when one side is $T_N\,@\,S$, which has a depth of at least 1. Thus, name expansion will not increase the metric, and all other structural equality will strictly replace this equation with a finite number of equations with a lesser metric. Thus, the equality algorithm terminates and type equality is decidable.

□

Theorem B.2 (Decidability of Type Synthesis). *Given a well-formed schema Δ, a context covering all free variables of e, it is decidable whether there exists some τ, and m such that $\Delta; \Gamma \vdash e : \tau^m$. Moreover, τ and m can be determined if they exist.*

Proof. By induction on e. All cases are direct. We use the following auxiliary lemma for shapes: Given a well-formed schema Δ, a context covering all free variables of s, and τ, it is decidable there exists some τ', such that $\Delta; \Gamma \vdash s : \tau \Rightarrow \tau'$. Moreover, τ' and m can be determined if they exist.

□

Theorem B.3 (Decidability of Type Checking). *Given a well-formed schema Δ, it is decidable whether $\Delta; \Gamma \vdash e : \tau^m$.*

Proof. By induction on e. All cases are direct, and some cases use Theorem B.2. We use the following auxiliary lemma for shapes: Given a well-formed schema Δ, it is decidable whether $\Delta; \Gamma \vdash s : \tau \Rightarrow \tau'$.

□

B.2 Canonical Forms

Theorem B.4 (Canonical Forms for Values). *Recall that when we fix Δ, and if $N := \tau \in \Delta$, N can be used interchangeably with τ.*

(1) If $\Delta \vdash_\mu U : \tau^{[i_1,i_2]}$, then $U = \{V_1, \ldots, V_n\}$ where $i_1 \leq n \leq i_2$ and each $\Delta \vdash_\mu V : \tau$

(2) If $\Delta \vdash_\mu V : \tau$, and

(a) if τ is a primitive type t, then V is a primitive value v where $v : t$.

(b) if τ is a nominal link type, i.e. $\tau = T_N @ S$, then $V = \mathsf{ref}(id) \diamond W$ where $N := T' \in \Delta$, $\Delta \vdash_\mu W : (T'' @ S)$, $T_N @ S = (T'_N @\{\}) \uplus (T'' @ S)$, and $T = T' \uplus T''$.

(3) If $\Delta \vdash_\mu W : R$, then $W = \{l_1^{u_1} := U_1, \ldots, l_n^{u_n} := U_n, (@l'_1)^{u'_1} := U'_1, \ldots, (@l'_k)^{u'_k} := U'_k\}$, where $R = T @ S$, $T = \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\}$, $S = \{l'_1 : t_1^{m'_1}, \ldots, l'_n : t_n^{m'_n}\}$, $\forall_{i,1\leq i\leq n}.\Delta \vdash_\mu U_i : \tau_i^{m_i}$, and $\forall_{i,1\leq j\leq k}.\Delta \vdash_\mu U'_j : t_j^{m'_j}$.

Proof. Directly by induction on the typing derivations for values.
(1) Direct inversion
(2)(a) Direct inversion.
(2)(b)(i) Direct inversion, then apply (VT-ref),
(2)(b)(ii) Directly by inversion.
(3) W can always be broken into two parts – object components with plain labels and object components with link property labels. Then the rest follows by inversion on the typing rules. □

B.3 Properties of Object Unions and Type Unions

Lemma B.5 (Type Union Associativity). $\tau \uplus (R_1 \uplus R_2) = (\tau \uplus R_1) \uplus R_2$

Proof. Both sides are merging the components of τ, R_1 and R_2. On both sides, the type components of R_2 may override the keys of both R_1 and τ, and the type components of R_1 may override the type components of τ. That is, both sides contain (1) all the type components of R_2, (2) all the type components of R_1 that are not in R_2, (3) all the type components of τ that are not in either R_1 or R_2. Thus, both sides are equal. □

Lemma B.6 (Object Union Typing). *If $\Delta \vdash_\mu W : R_1$, and $\Delta \vdash_\mu W' : R_2$, then $\Delta \vdash_\mu W \uplus W' : R_1 \uplus R_2$.*

Proof. By inversion on the typing rules, $W \uplus W'$ will have all the labels of W' and the associated types, and then the labels of W that are not in W'. The result follows by direct case analysis.

More specifically,

Let $\Delta \vdash_\mu W : \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\} @\{l'''_1 : (\sigma_1)^{m'''_1}, \ldots, l'''_q : (\sigma_q)^{m'''_1}\}$,

then by inversion, $W = \{l_1^{u_1} := U_1, \ldots, l_n^{u_n} := U_n, @l_1'''^{(u'''_1)} := U'''_1, \ldots, @l_q'''^{(u'''_q)} := U'''_q\}$.

Similarly, let $\Delta \vdash_\mu W' : \{l'_1 : (\tau'_1)^{m'_1}, \ldots, l'_k : (\tau'_k)^{m'_k}\} @\{l''_1 : (\sigma'_1)^{m''_1}, \ldots, l''_p : (\sigma'_p)^{m''_p}\}$,

then by inversion, $W = \{l_1'^{(u_1)} := U'_1, \ldots, l_k'^{(u_n)} := U'_n, @l_1''^{(u''_1)} := U''_1, \ldots, @l_p''^{(u''_p)} := U''_p\}$.

We have $W \uplus W' = \big((\uplus_{i,1\leq i\leq k}\{l_i'^{(u'_i)} := U'_i\}) \uplus (\uplus_{j,1\leq j\leq n}\{l_j^{m_j} := U_j \mid \forall i.l_j \neq l'_i\})\big)$
$\uplus\big((\uplus_{i,1\leq i\leq p}\{@l_i''^{(u''_i)} := U''_i\}) \uplus (\uplus_{i,1\leq j\leq q}\{@l_j'''^{(u'''_i)} := U'''_i \mid \forall i.@l''_j \neq @l'''_i\})\big)$,

where none of the $\uplus$ in the right-hand-side of the above definition overwrites attributes.

Then by definition, $\Delta \vdash_\mu W \uplus W' : R_1 \uplus R_2$. □

B.4 Lemmas Related to Value Typing

Lemma B.7 (Value Set Projection Typing). *If $\Delta \vdash_\mu U : \tau^m$, and for each $V_i \in U$, $proj_\mu(L, V_i) = U_i$ and $\Delta \vdash_\mu U_i : \sigma^{m'}$, then $\Delta \vdash_\mu \cup_i U_i : \sigma^{m\times m'}$.*

Proof. By inversion on the first premise, we have in total m such V_i's, then the union will have cardinality $m \times m'$ by Lemma B.8. □

Lemma B.8 (Value Set Union Cardinality).
(1) If $\Delta \vdash_\mu U_1 : \tau^{m_1}$ and $\Delta \vdash_\mu U_2 : \tau^{m_2}$, then $\Delta \vdash_\mu U_1 \cup U_2 : \tau^{m_1+m_2}$.
(2) If $\Delta \vdash_\mu U_i : \tau^{m_1}$, then $\Delta \vdash_\mu \cup_i(U_i) : \tau^{m_1 \times m_2}$, where $lb(m_2) \leq \|\{U_i\}_i\| \leq ub(m_2)$.

Proof. Both theorems directly follow from the definition of union and cardinality arithmetic. □

Lemma B.9 (Value Typing Invariance). *For any $\mu : \Delta$ and $\mu' : \Delta$ such that μ and μ' agree on the typing for reference values, i.e. $\forall_{id}.(id, N_1, W_1) \in \mu \wedge (id, N_2, W_2) \in \mu' \implies N_1 = N_2$,*
(1) If $\Delta \vdash_\mu W : R$, then $\Delta \vdash_{\mu'} W : R$.
(2) If $\Delta \vdash_\mu U : \tau^m$, then $\Delta \vdash_{\mu'} U : \tau^m$.
(3) If $\Delta \vdash_\mu V : \tau$, then $\Delta \vdash_{\mu'} V : \tau$.

Proof. By induction on the typing derivation.
(1) directly follows from (2), and (2) directly follows from (3), both by IH. Now we show (3).
Case $V = v$, then the result is direct.
Case $V = \mathrm{ref}(id) \diamond W$, then the result directly follows by IH on W, and the agreement of μ and μ' on reference typing. □

The following lemma will only hold without *delete* expressions.

Lemma B.10 (Storage Typing Monotonicity). *If $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e \searrow \mu' \parallel U$, and if $(id, N, W) \in \mu$, then $(id, N, W') \in \mu'$ for some W'.*

Proof. No rule changes the typing of the objects in the store, and no rule removes objects from the store. □

B.5 Preservation Lemmas

Lemma B.11 (Preservation of Object Type Identity). *If $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e \searrow \mu' \parallel U$, $\mu : \Delta$, $\mu' : \Delta$ and if both $(id, N, W) \in \mu$ and $(id, N', W') \in \mu'$, then $N = N'$, and the labels of W and W' coincide.*

Proof. This is proved by direct observation that no rules ever update the type name of any id that is already in the database store. □

Lemma B.12 (Projection Typing). *We have the following theorems regarding the operation $proj_\mu$. Given $\mu_0' : \Delta$, and μ agrees with μ_0' on all objects in μ_0', (the $proj_\mu$ operation is parameterized by μ_0'),*
(1) If $\Delta \vdash_\mu V : T_N\, @\{\ldots, l_i : t_i^{m_i}, \ldots\}$, and $U_i = proj_\mu(@l_i, V)$, then $\Delta \vdash_\mu U_i : t_i^{m_i}$.
(2) If $\Delta \vdash_\mu V : \{\ldots, l_i : \tau_i^{m_i}, \ldots\}_N\, @\, S$, and $U_i = proj_\mu(l_i, V)$, then $\Delta \vdash_\mu U_i : \tau_i^{m_i}$.

Proof. (1) By the definition of $proj_\mu$ operation and inversion on typing, $V = \mathrm{ref}(id) \diamond W$, where by Theorem B.4 and (WT-obj), let $W' = \{l_i^{u_i} := U_i \mid @l_i^{u_i} := U_i \in W\}$, then $\Delta \vdash_\mu W' : \{\ldots, l_i : t_i^{m_i}, \ldots\}\, @\{\}$. By inversion, there exists object component $l_i^{u_i} := U_i$ in W' where $proj_\mu(@l_i, V) = U_i$. By inversion on the typing of W', $\Delta \vdash_\mu U_i : t_i^{m_i}$.

(2) By the definition of $proj_\mu$, $V = \mathrm{ref}(id) \diamond W$, and by inversion on typing, $\Delta \vdash_\mu W : T''\, @\, S$, $(id, N, W') \in \mu$, $N := T' \in \Delta$, $\{\ldots, l_i : \tau_i^{m_i}, \ldots\} = T' \uplus T''$. Either the label l_i is present in W (l_i is present in T'') or not (l_i is present in T').
   (a) If $l_i^{u_i} := U_i \in W$, then $proj_\mu(l_i, V) = U_i$, and by inversion on typing, $\Delta \vdash_\mu U_i : \tau^{m_i}$.

(b) If l_i is not in W, $(id, N', W'') \in \mu'_0$ or not, and $N = N'$ because of the agreement.

(i) If $(id, N, W'') \notin \mu'_0$, then by inversion on typing $l_i^{u_i} := U_i \in W'$. Then $proj_\mu(l_i, V) = U_i$, and by inversion on typing, $\Delta \vdash_\mu U_i : \tau^{m_i}$.

(ii) If $(id, N, W'') \in \mu'_0$, then by inversion on typing $l_i^{u_i} := U_i \in W''$. Then $proj_\mu(l_i, V) = U_i$, and by inversion on typing, $\Delta \vdash_\mu U_i : \tau^{m_i}$.

□

Lemma B.13 (Storage Coercion Typing). *For a schema object type T (the target types of T are either primitive or of the form $N @ S$), if $\Delta \vdash_\mu W : T @\{\}$ and $W \triangleleft T \Rightarrow W' : \Delta$, then $\Delta \vdash_\mu W' : T @\{\}$.*

Proof. We show this by induction on the object type T.

By definition of the typing rules, let $W = \{l_1^{u_1} := U_1, \ldots, l_n^{u_1} := U_n\}$, $T = \{l_1 : \tau_1^{m_1}, \ldots l_n : \tau_n^{m_n}\}$, and $W' = \{l_1^{u_1} := U'_1, \ldots, l_n^{u_n} := U'_n\}$. By inversion on $\Delta \vdash_\mu W : T @\{\}$, $\Delta \vdash_\mu U_i : \tau_i^{m_i}$.

To show $\Delta \vdash_\mu W' : T @\{\}$, it suffices to show that the $\Delta \vdash_\mu U'_i : \tau_i^{m_i}$

Case τ_i is a primitive type, the result follows directly by assumption as $U'_i = U_i$.

Case τ_i is a link type $N_i @ S_i$,

$U' = \{\text{ref}(id) \diamond W'_j \mid \text{ref}(id_j) \diamond W_j \in U_i\}$ where $W'_j = \{@l_j'^{(u_j)} := U''_j \mid @l_j'^{(u_j)} := U''_j \in W_j\}$

By inversion, for all j, $\Delta \vdash_\mu \text{ref}(id_j) \diamond W_j : \tau_i$, where $\tau_i = (N_i @\{\}) \uplus (T'_i @ S_i)$. Then, by definition, $\Delta \vdash_\mu W'_j : \{\} @ S_i$, and $\Delta \vdash_\mu \text{ref}(id) \diamond W'_j : N_i @ S_i$. The result follows by (UT-vset).

□

Lemma B.14 (Function Application Typing). *Given $f : [\tau_1^{p_1}, \ldots, \tau_n^{p_n}] \longrightarrow \tau^m$, if $\forall i. \Delta \vdash_\mu U_i : \tau_i^{m_i}$ and $p_i \triangleright m_i \rightsquigarrow m'_i$, and $U' = f^*(U_1, \ldots, U_n)$, then $\Delta \vdash_\mu U' : \tau^{m'_1 \times \cdots \times m'_n \times m}$.*

Proof. According to the broadcasting semantics, construct $C_i = \begin{cases} \{\{V_i\}\} \ \forall V_i \in U_i & \text{if } p_i = 1 \\ \{\{\}\} & \text{if } p_i = ? \wedge U_i = \{\} \\ \{\{V_i\}\} \ \forall V_i \in U_i & \text{if } p_i = ? \wedge U_i \neq \{\} \\ \{U_i\} & \text{if } p_i = * \end{cases}$,

then for $\forall i. \forall U_j \in C_i. \Delta \vdash_\mu U_j : \tau_i^{m_i}$ where $m_i = \begin{cases} (=1) & \text{if } p_i = 1 \\ (\leq 1) & \text{if } p_i = ? \\ (*) & \text{if } p_i = * \end{cases}$, and $m'_1, \ldots, m'_n$ are the cardinality modes of $C_1, \ldots, C_n$.

Then $U' = \cup\{f(U_1, \ldots, U_n) \mid U_1 \in C_1, \ldots, U_n \in C_n\}$

Assume the underlying evaluation function f^* can act on values of the following form:

(1) If the modifier is 1, then the function is expecting a singleton set
(2) If the modifier is ?, then the function is expecting a singleton set or an empty set
(3) If the modifier is $*$, then the function is expecting any set

Given that a primitive evaluation will return a result set of type τ^m, the entire aggregate is deemed to have type $\tau^{m'_1 \times \ldots m'_n \times m}$. □

B.6 Preservation

Lemma B.15 (View/Shape Typing). *Given $\Delta; \Gamma \vdash s : \tau \Rightarrow \tau', \mu : \Delta, \Omega : \Gamma$, if $\Delta \vdash_\mu V : \tau$ and $V' = view^\Omega(s, V) : \mu \mapsto \mu'$, then $\Delta \vdash_{\mu'} V' : \tau'$ and $\mu' : \Delta$.*

Proof. The proof of this lemma references Theorem B.16 in a justified manner: the size of the subject expression/shape decreases.

Let $s = \{L'_1 = x.e_1, \ldots, L'_k = x.e_k\}$.

By inversion on $\Delta;\Gamma \vdash s : \tau \Rightarrow \tau'$, $\forall_{i,1\le i\le k}.\Delta;\Gamma,x:\tau \vdash e_i : (\tau_i')^{m_i'}$, $T = \{l_i' : (\tau_i)^{m_i'} \mid L_i' = l_i\}$, $S = \{l_i' : (t_i)^{m_i'} \mid L_i' = @l_i, \tau_i = t_i\}$, and $\tau' = \tau \uplus (T @ S)$.

Given $\Delta \vdash_\mu V : \tau$, and $\tau = T_N' @ S'$, V must be $\mathsf{ref}(id) \diamond W$.

By the definition of $V' = \mathit{view}^\Omega(s,V) : \mu \mapsto \mu'$, we have $\Delta;\Gamma,x \mapsto \{V\} \vdash \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U_i'$. By Theorem B.16 on each e_i, we have $\Delta \vdash_{\mu_i} U_i' : (\tau_i')^{m_i'}$, and $\mu' : \Delta$.

Let $W_s = \{(L_1')^{visible} := U_1', \ldots, (L_k')^{visible} := U_k'\}$, by Lemma B.9 (1), Lemma B.11, and (WT-obj-2), we have $\Delta \vdash_{\mu'} W_s : T @ S$.

By inversion on $\Delta \vdash_\mu V : \tau$, if $(id, N, W_{id}) \in \mu$, then $\Delta \vdash_\mu W : R$, and $\tau = (N @\{\}) \uplus R$.

By definition, $V' = \mathsf{ref}(id) \diamond (W' \uplus W_s)$, where W' is W with all properties set to invisible. Since visibility doesn't affect typing, all judgments regarding W now hold for W'.

By Lemma B.6, $\Delta \vdash_{\mu'} W \uplus W_s : R \uplus (T @ S)$

By rule (VT-obj), $\Delta \vdash_{\mu'} \mathsf{ref}(id) \diamond (W' \uplus W_s) : (N @\{\}) \uplus (R \uplus (T @ S))$.

By Lemma B.5 $(N @\{\}) \uplus (R \uplus (T @ S)) = ((N @\{\}) \uplus R) \uplus (T @ S) = \tau \uplus (T @ S) = \tau'$.

By rule (VT-obj), $\Delta \vdash_{\mu'} V' : \tau'$.

□

In the following theorem, we follow Lemma B.11, we assume that μ agrees with μ_0 on the type of all objects.

Theorem B.16 (Preservation). *If $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e \searrow \mu' \parallel U$, and $\Delta;\Gamma \vdash e : \tau^m$, where $\Omega : \Gamma$, $\mu : \Delta$, and $\mu_0' : \Delta$, then $\Delta \vdash_{\mu'} U : \tau^m$, and $\mu' : \Delta$.*

Proof. By induction on the big step evaluation relation. The proof for $\mu' : \Delta$ is elided if it is trivial or directly follows from IH.

Case (E-val), $e = v$, by inversion we have $v : t$ and then by (T-val).

Case (E-var), $e = x$, by assumption $\Omega : \Gamma$.

Case (E-lprop), $e = e_1 @ l$, and $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e_1 \searrow \mu' \parallel U$, by inversion, $\Delta;\Gamma \vdash e_1 : (T_N @\{\ldots, l_i : t_i^{m_i}, \ldots\})^m$, by IH, U has cardinality m and for each $V_i \in U$, $\Delta \vdash_\mu V_i : T_N @\{\ldots, l_i : t_i^{m_i}, \ldots\}$. Let $U_i' = \mathit{proj}_{\mu'}(@l, V_i)$, by Lemma B.12 (1), $\Delta \vdash_\mu U_i' : t_i^{m_i}$. By Lemma B.7, we have $\Delta \vdash_{\mu'} \cup_i(U_i') : t_i^{m \times m_i}$.

Case (E-select), $e = \mathit{select}\ e_1$, directly by IH.

Case (E-shape), $e = e_1\ s$, directly by IH, an iterative application of Lemma B.15, and because the view^Ω operation is applied elementwise, the cardinality is preserved by (UT-obj).

Case (E-proj), $e = e_1 \cdot l_i$, and $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e_1 \searrow \mu' \parallel U_1$. By inversion on the typing rules, we have $\Delta;\Gamma \vdash e_1 : (T_N @ S)^m$, where $T = \{\ldots, l_i : \tau_i^{m_i}, \ldots\}$. By IH, we have $\Delta \vdash_\mu U_1 : (T_N @ S)^m$. Since we are joining together $\mathit{proj}_\mu(l_i, V_j)$ for each $V_j \in U_2$, the result follows from Lemma B.12 (2) and Lemma B.7.

Case (E-detached), $e = \mathit{detached}\ e_1$, directly by IH.

Case (E-union), $e = e_1 \cup e_2$, the result follows by IH and then Lemma B.8.

Case (E-empty-set), $e = \{\}_\tau$, we have $\mu' = \mu$ and $\Delta \vdash_{\mu'} \{\} : \tau^{(\le 1)}$ holds.

Case (E-with), $e = \mathit{with}(e_1; x.e_2)$, $\Omega \vdash_\Delta^{\mu_0'} \mu \parallel e_1 \searrow \mu' \parallel U_1$, and $\Omega, x \mapsto U_1 \vdash_\Delta^{\mu_0'} \mu' \parallel e_1 \searrow \mu'' \parallel U_2$, By inversion on typing, $\Delta;\Gamma \vdash e_1 : \tau_2^{m_2}$, and $\Gamma, x : \tau_2^{m_2} \vdash e_2 : \tau^m$. By IH, $\Delta \vdash_\mu U_1 : {\tau_2}^{m_2}$. Since $\Omega : \Gamma$, $\Omega, x \mapsto U_1 : \Gamma, x : {\tau_2}^{m_2}$. Also by IH, $\mu' : \Delta$. Then by IH, $\Delta \vdash_\mu U_2 : \tau^m$ and $\mu'' : \Delta$.

Case (E-obj), $e = N$, we have for all $(id, N, W) \in \mu_0'$, $\Delta \vdash_\mu \mathsf{ref}(id) \diamond \{\} : N @\{\}$, by $\mu_0' : \Delta$, and VT-ref. The result follows by UT-vset.

Case (E-insert), $e = \mathit{insert}\ N\{l_1 := e_1, \ldots, l_n := e_n\}$, $\mu_0 = \mu$, $\forall_i.\ \Omega \vdash \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U_i$, $W = \{l_1^u := U_1, \ldots, l_n^u := U_n\}$, $W \triangleleft T \Rightarrow W' : \Delta$, $N := T \in \Delta$. By IH on e_i's, Lemma B.9 (3), and Lemma B.11 $\Delta \vdash_{\mu'} W : T @\{\}$. By Lemma B.13, $\Delta \vdash_\mu W' : T @\{\}$. By definition, have $\mu' \cup \{(id, N, W')\} : \Delta$. By (VT-obj) and (UT-obj), $\Delta \vdash_{\mu' \cup \{(id,N,W')\}} \mathsf{ref}(id) \diamond \{\} : \tau^{(=1)}$.

Case (E-IF), $e = if\ e_1\ then\ e_2\ else\ e_3$, $\Omega \vdash^{\mu'_0}_{\Delta} \mu \parallel e_1 \searrow \mu_0 \parallel U'$, $\forall i.\Omega \vdash^{\mu'_0}_{\Delta} \mu_{i-1} \parallel e_2/e_3 \searrow \mu_i \parallel U_i$. By inversion on typing, $\Delta; \Gamma \vdash e_1 : bool^m$, $\Delta; \Gamma \vdash e_2 : \tau^{[i_1,i_2]}$, $\Delta; \Gamma \vdash e_3 : \tau^{[i_3,i_4]}$. By IH, $\Delta \vdash_{\mu_0} U' : bool^m$, and $\forall i.U_i : \tau^{[min(i_1,i_3),max(i_2,i_4)]}$. Then $\Delta \vdash_{\mu'} \cup_i(U_i) : \tau^{m \times [min(i_1,i_3),max(i_2,i_4)]}$ by definition.

Case (E-FOR), $e = \text{for}(e_1; x.e_2)$ this case is almost the same as (E-WITH), except the body e_2 is evaluated once per element in the result set of U', similar to the (E-IF) case. Thus, the result $\cup_i(U_i)$ satisfies $\Delta \vdash_\mu \cup_i(U_i) : \tau^{m_1 \times m_2}$, where $\Delta; \Gamma \vdash e_1 : \tau_2^{m_1}$ and $\Delta; \Gamma, x : \tau_2^{(=1)} \vdash e_2 : \tau^{m_2}$.

Case (E-OFOR-1), $e = \text{optional\_for}(e_1; x.e_2)$ where $\Omega \vdash^{\mu'_0}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel \{\}$, by inversion on typing and IH, $\Delta; \Gamma \vdash e_1 : \tau_2^{[0,i_2]}$, $\Delta; \Gamma, x : (\tau_1)^{(\leq 1)} \vdash e_2 : \tau_2^{m_2}$. By IH, $\Delta \vdash_{\mu'} U : \tau_2^{m_2}$. The result follows by T-CARD-SUB because $m_2 \leq [1, i_2] \times m_2$.

Case (E-OFOR-2), $e = \text{optional\_for}(e_1; x.e_2)$ where $\Omega \vdash^{\mu'_0}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel \{V_1, \ldots V_n\}$ and $n > 0$, By inversion on typing and IH, all U_i as a result of evaluating e_2 with $x \mapsto \{V_i\}$ satisfies $\Delta \vdash_{\mu_i} U_i : \tau^{m_2}$. The aggregated result has $\Delta \vdash_{\mu'} \cup_i U_i : \tau^{m_2 \times (=n)}$. Then the result follows by (T-CARD-SUB). This case is similar to (E-FOR).

Case (E-FUNC), $e = f(e_1, \ldots, e_n)$, where $\forall i.\Omega \vdash^{\mu'_0}_{\Delta} \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U_i$. By inversion on typing, $\forall i.\Delta; \Gamma \vdash e_i : \tau_i^{m_i}$. By IH, $\forall i.\Delta \vdash_\mu U_i : \tau_i^{m_i}$. The result follows by Lemma B.14.

Case (E-BACKLINK), $e = e_1 \cdot_{\leftarrow} l\ [is\ N]$, the result follows directly by assumption as we are computing $\text{ref}(id) \diamond \{\}$ where $(id, N, W) \in \mu'_0$.

Case (E-FILT), $e = e_1\ filter(x.e_2)\ order(x.e_3)$, the result directly follows by IH on e_1 evaluation, as the result is just a subset of e_1 result.

Case (E-UPDATE), $e = update\ e_1\ s$, where $\Omega \vdash^{\mu'_0}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U_1$. By inversion on typing $\Delta; \Gamma \vdash e_1 : (T_N\ @\{\})^m$. By IH, $\Delta \vdash_\mu U_1 : (T_N\ @\{\})^m$. The updated μ' satisfies $\mu' : \Delta$ by Lemma B.15 and Lemma B.13. The returned result typing follows from Lemma B.15.

□

B.7 Progress Lemmas

LEMMA B.17 (PROJECTION PROGRESS). *Given $\mu : \Delta$ and $\mu_0 : \Delta$, we have*
(1) If $\Delta \vdash_\mu V : \{\ldots, l_i : \tau_i^{m_i}, \ldots\}_N\ @\ S$, then there exists U such that $proj_\mu(l_i, V) = U$.
(2) If $\Delta \vdash_\mu V : T_N\ @\{\ldots, l_i : t_i^{m_i}, \ldots\}$, then there exists U such that $proj_\mu(@l_i, V) = U$.

PROOF. (1) By inversion, $V = \text{ref}(id) \diamond W$, $(id, N, W') \in \mu$, $\Delta \vdash_\mu W : R$, and $(N\ @\{\}) \uplus R = \{\ldots, l_i : \tau_i^{m_i}, \ldots\}_N\ @\ S$. The object type component $l_i : \tau_i^{m_i}$ is either in R or in N.

(a) If $l_i : \tau_i^{m_i}$ is in R, there is a there is a corresponding object value component $l_i^{u_i} := U_i$ in either W.

(b) Otherwise $l_i : \tau_i^{m_i}$ is not in R but in N, i.e., l_i is in T for $N := T \in \Delta$.
Either $(id, N, W'') \in \mu_0$ or not.

(i) If $(id, N, W'') \in \mu_0$, there is $l_i \in W''$.

(ii) If $(id, N, W'') \notin \mu_0$, there is $l_i \in W'$. In all cases, the $proj_\mu$ operation can progress.

(2) By inversion, $V = \text{ref}(id) \diamond W$, $\Delta \vdash_\mu W : T'\ @\{\ldots, l_i : t_i^{m_i}, \ldots\}$, by inversion on the typing rule, the label $@l$ must be in W, and thus $proj_\mu$ can progress.

□

LEMMA B.18 (STORAGE COERCION PROGRESS). *If $\Delta \vdash_\mu W : T\ @\{\}$, where T is a schema type, then there exists W' such that $W \lhd T \Rightarrow W'$.*

PROOF. A schema type means that the target types of T are either primitive or $N\ @\ S$.

For a type component $l_i^{u_i} : \tau_i^{m_i} \in T$, by inversion on the typing rule, there exists an object component $l_i := U_i \in W$, and $\Delta \vdash_\mu U_i : \tau_i^{m_i}$. We show that the corresponding object component $l_i^{u_i} := U'_i$ can always be constructed.

(1) case τ_i is primitive, the result is defined directly.
(2) case $\tau_i = N @ S$, for each $V \in U_i$, by inversion on the typing rule $V = \mathsf{ref}(id) \diamond W''$, the result U'_i can always be constructed as $W_s = \{@l_i'^{(u'_i)} := U''_i \mid @l_i'^{(u'_i)} := U''_i \in W''\}$ can always be constructed.

□

Lemma B.19 (Function Application Progress). *Given $f : [\tau_1^{p_1}, \ldots, \tau_n^{p_n}] \to \tau^m$, if $\forall i.\Delta \vdash_\mu U_i : \tau_i^{m_i}$, then there exists a U' such that and $U' = f^*(U_1, \ldots, U_n)$.*

Proof. Assume the underlying evaluation function can always compute a result on well-typed arguments, and then the result U' can always be constructed.

□

Lemma B.20 (View/Shape Result Form). *Given $V' = view^\Omega(s, V) : \mu \mapsto \mu'$, if $V = \mathsf{ref}(id) \diamond W$, then $V' = \mathsf{ref}(id) \diamond W'$ for some W'.*

Proof. Directly by definition. □

B.8 Progress

Lemma B.21 (View/Shape Progress). *If $\Delta \vdash_\mu V : \tau$, $\Delta; \Gamma \vdash s : \tau \Rightarrow \tau'$, given $\mu : \Delta$, there exists μ' and V' such that $V' = view^\Omega(s, V) : \mu \mapsto \mu'$.*

Proof. This lemma invokes Theorem B.22 in a justified way, the shape/expression is always smaller.

Let $s = \{L'_1 := x.e_1, \ldots, L'_n := x.e_n\}$.

Let $\mu_0 = \mu$. By inversion on the typing for s, we have $\Delta; \Gamma, x : \tau \vdash e_i : \tau_i^{m_i}$.

By definition, we have $\Omega, x \mapsto \{V\} : \Delta, x : \tau^m$. By Theorem B.22, there exists μ_i and U'_i's such that $\Omega, x \mapsto \{V\} \vdash_\Delta^{\mu_0} \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U'_i$. μ' is μ_n. By inversion on typing, V is a reference value $\mathsf{ref}(id) \diamond W$, and we can construct V' according to the definition.

□

We follow Lemma B.11 and assume μ agrees with μ_0 on the type of all objects.

Theorem B.22 (Progress/Normalization). *If $\Delta; \Gamma \vdash e : \tau^m$, $\Omega : \Gamma$, $\mu : \Delta$, and $\mu'_0 : \Delta$, then there exists μ' and U such that $\Omega \vdash_\Delta^{\mu'_0} \mu \parallel e \searrow \mu' \parallel U$.*

Proof. By induction on the typing derivation.

Case (T-card-sub), $e = e_1$, the result follows by IH.

Case (T-var), $e = x$, the result follows by $\Omega : \Gamma$.

Case (T-obj), $e = N$, the result follows by (E-obj), which always applies.

Case (T-val), $e = v$, the result follows by (E-val).

Case (T-shape), $e = e_1\,s$ where $\Delta; \Gamma \vdash e_1 : \tau^m$. By IH, we have $\Omega \vdash_\Delta^{\mu'_0} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By Theorem B.16, $\Delta \vdash_\mu U'' : \tau^m$. The result follows by repeated application of Lemma B.21 and Lemma B.15.

Case (T-union), $e = e_1 \cup e_2$, the result follows directly by IH.

Case (T-empty-set), $e = \{\}_\tau$, the result follows by (E-empty-set).

Case (T-func), $e = f(e_1, \ldots, e_n)$, by IH on each e_i, there exists U_i for each e_i such that $\Omega \vdash \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U_i$. By inversion on typing and Theorem B.16, $\Delta \vdash_\mu U_i : \tau_i^{m_i}$, and $f : [\tau_1^{m_1}, \ldots, \tau_n^{m_n}] \to \tau^m$. Since function arguments are either primitive types or nominal links without link properties, the result follows by Lemma B.19.

Case (T-proj), $e = e_1 \cdot l$, where $\Delta; \Gamma \vdash e_1 : (T_N \mathbin{@} S)^m$. By IH on e_1, we have $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$, where $\Delta \vdash_{\mu''} U'' : (T_N \mathbin{@} \{\})^m$ by Theorem B.16. Then the result follows by Lemma B.17 (1) on each element in U''.

Case (T-lprop), $e = e_1 \mathbin{@} l$, where $\Delta; \Gamma \vdash e_1 : (T_N \mathbin{@} \{\ldots, l_i : \tau_i^{m_i}, \ldots\})^m$. By IH, $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By Theorem B.16, $\Omega \vdash_{\mu''} U'' : (T_N \mathbin{@} \{\ldots, l_i : \tau_i^{m_i}, \ldots\})^m$. The result follows by Lemma B.17 (2) on each $V_i \in U''$.

Case (T-backlink), $e = e_1 \cdot_{\leftarrow} l \; [is\, N]$, where $\Delta; \Gamma \vdash e_1 : (T_N \mathbin{@} S)^m$. By IH, $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By inversion, U'' is a set of $\mathrm{ref}(id) \diamond W$'s. Theorem B.4 (3) and $\mu_0 : \Delta$ guarantees that label l is always present in W where $(id, N, W) \in \mu_0$. Then the result U can always be constructed by (E-backlink).

Case (T-select), directly by IH.

Case (T-detached), directly by IH.

Case (T-with), $e = \mathrm{with}(e_1; x.e_2)$, where $\Delta; \Gamma \vdash e_1 : \tau_2^{m_2}$ and $\Delta; \Gamma, x : \tau_2^{m_2} \vdash e_2 : \tau^m$. By IH on e_1, there exists U'' such that $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By Theorem B.16, $\Delta \vdash_{\mu''} U'' : \tau_2^{m_2}$, and $\mu'' : \Delta$. Then, $(\Omega, x \mapsto U'') : (\Gamma, x : \tau_2^{m_2})$. By IH on e_2, there exists U such that $\Omega, x \mapsto U'' \vdash^{\mu_0'}_{\Delta} \mu'' \parallel e_2 \searrow \mu' \parallel U$.

Case (T-for), $e = \mathrm{for}(e_1; x.e_2)$. This case is similar to (T-with), except that when e_1 evaluates to U'', we extend Ω with $x \mapsto \{V_i\}$ for each $V_i \in U''$, and derive $(\Omega, x \mapsto \{V_i\}) : (\Gamma, x : \tau_2^{(=1)})$. The rest is exactly similar to the (T-with) case.

Case (T-ofor-1), $e = \mathrm{optional\_for}(e_1; x.e_2)$, where $\Delta; \Gamma \vdash e_1 : \tau_2^{[0,i_2]}$ and $\Delta; \Gamma, x : \tau_2^{(\leq 1)} \vdash e_2 : \tau^{[1,i_2] \times m}$. By IH on e_1, there exists U'' such that $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By Theorem B.16, $\Delta \vdash_{\mu''} U'' : \tau_2^{[0,i_2]}$, and $\mu'' : \Delta$. We have either $U'' = \{\}$ or not. If $U'' = \{\}$, the rule (E-ofor-1) applies by $(\Omega, x \mapsto \{\}) : (\Gamma, x : \tau_2^{(\leq 1)})$, and IH. If $U'' \neq \{\}$, for each $V_i \in U''$, the rule (E-ofor-2) applies by $(\Omega, x \mapsto \{V_i\}) : (\Gamma, x : \tau_2^{(\leq 1)})$, and IH.

Case (T-ofor-2), $e = \mathrm{optional\_for}(e_1; x.e_2)$. This case is similar to (T-ofor-2), except that only the rule (E-ofor-2) applies because the result of e_1, U'' will not be $\{\}$ by Theorem B.4 (1).

Case (T-if), $e = \mathit{if}\; e_1 \;\mathit{then}\; e_2 \;\mathit{else}\; e_3$, where $\Delta; \Gamma \vdash e_1 : \mathit{bool}^{m_1}$. By IH on e_1, there exists U'' such that $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By Theorem B.16, and then Theorem B.4 (1) and (2)(a), U'' will be a set of values consisting of *true* or *false*. Then the result follows by IH on e_2 and e_3, with the rule (E-if).

Case (T-filt), $e = e_1 \,\mathit{filter}(x.e_2)\; \mathit{order}(x.e_3)$, where $\Delta; \Gamma \vdash e_1 : \tau^{m_1}$, $\Delta; \Gamma, x : \tau^{(=1)} \vdash e_2 : \mathit{bool}^{m_2}$, and $\Delta; \Gamma, x : \tau^{(=1)} \vdash e_3 : \tau_3{}^{(\leq 1)}$. By IH on e_1, $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$. By Theorem B.16, $\Delta \vdash_{\mu''} U'' : \tau^{m_1}$. We have $(\Omega, x \mapsto V_i) : \Gamma, x : \tau^{(=1)}$ for each V_i in U''. Then the result follows by IH on e_2 and e_3, and (E-filt). Theorem B.4 (2)(a) ensures that the result of e_2 has *true* and *false* values. *orderby* can be applied because all objects can be ordered.

Case (T-insert), $e = \mathit{insert}\; N\, \{l_1 := e_1, \ldots, l_n := e_n\}$, where $\Delta; \Gamma \vdash e_i : \tau_i^{m_i}$, and $N := \{l_1 : \tau_1^{m_1}, \ldots, l_n : \tau_n^{m_n}\} \in \Delta$. By IH on e_1 through e_n, if $\mu_0 = \mu$, there exists μ_1 through μ_n such that $\Omega \vdash^{\mu_0'}_{\Delta} \mu_{i-1} \parallel e_i \searrow \mu_i \parallel U_i$. By Theorem B.16, $\Delta \vdash_{\mu_i} U_i : \tau_i^{m_i}$. By Lemma B.9 (3) and Lemma B.11, $\Delta \vdash_{\mu_n} U_i : \tau_i^{m_i}$. Let $W = \{l_1 := U_1, \ldots, l_n := U_n\}$, we have $\Delta \vdash_{\mu_n} W : T \mathbin{@} \{\}$. By Lemma B.18, there exists W' such that $W \lhd \tau \Rightarrow W'$. Then the result follows by (E-insert).

Case (T-update), $e = \mathit{update}\; e_1\, s$, where $\Delta; \Gamma \vdash e_1 : (T_N \mathbin{@} \{\})^m$, and $\Delta; \Gamma \vdash s : (T_N \mathbin{@} \{\}) \Rightarrow (T_N \mathbin{@} \{\})$, $N := T \in \Delta$. By IH on e_1, Theorem B.16, $\Omega \vdash^{\mu_0'}_{\Delta} \mu \parallel e_1 \searrow \mu'' \parallel U''$, where U'' is a set of $\mathit{ref}(id_i) \diamond W_i$'s, and $\Delta \vdash_{\mu''} U'' : (T_N \mathbin{@} \{\})^m$. By repeated application of Lemma B.21, and Lemma B.15, for each $\mathrm{ref}(id_i) \diamond W_i \in U''$, we have $V_i = \mathit{view}^{\Omega}(s, \mathrm{ref}(id_i) \diamond W_i) : \mu_{i-1} \mapsto \mu_i$, where

$\Delta \vdash_{\mu_i} V_i : T_N\, @\{\}$. By Lemma B.20, each $V_i = \mathrm{ref}(id_i) \diamond W_i'$ for some W_i'. By Lemma B.9 (3) and Lemma B.11, $\Delta \vdash_{\mu_n} V_i : (T_N\, @\{\})$, where n is the cardinality of U''. For each V_i, suppose $\Delta \vdash_{\mu_n} W_i' : R$, then by inversion $(N\, @\{\}) \uplus R = T_N\, @\{\}$. Given $(id_i, N, W) \in \mu_n$, by Lemma B.6, we have $\Delta \vdash_{\mu_n} W \uplus W_i' : (T\, @\{\})$. By Lemma B.18, there exists W_i'' such that $W \uplus W_i \triangleleft T \Rightarrow W_i''$. Then the result follows by (E-UPDATE).

$\square$