Under consideration for publication in J. Functional Programming

Capability-based Localization of Distributed and Heterogeneous Queries – extended version

JOÃO COSTA SECO
NOVA LINCS – Universidade Nova de Lisboa, Lisboa

PAULO FERREIRA, and HUGO LOURENÇO
OutSystems, Lisboa

(*e-mail:* joao.seco@fct.unl.pt, paulo.ferreira@outsystems.com, hugo.lourenco@outsystems.com)

Abstract

One key aspect of data-centric applications is the manipulation of data stored in persistent repositories, which is moving fast from querying a centralized relational database to the ad-hoc combination of constellations of data sources.

The extension of general purpose languages with query operations is increasingly popular, as a tool to improve reasoning and optimizing capabilities of interpreters and compilers. However, not much is being done to integrate and orchestrate different and separate sources of data. We present a data manipulation language that abstracts the nature and location of data-sources. We define its semantics and a type directed query localization mechanism to be used in development tools for heterogeneous environments to efficiently compile them into native queries. We introduce a localization procedure based on rewriting of query expressions that is confluent, terminating and provides the maximum mapping between site capabilities and the structure of the query. We provide formal type safety results that support the sound distribution of query fragments over remote sites.

Our approach is also suitable for an interactive query construction environment by rich user interfaces that provide immediate feedback on data manipulation operations. This approach is currently the base for the data layer of a development platform for mobile and web applications.

1 Introduction

The state of the art on development of data-centric web, cloud, and mobile applications, is highly based on the use of frameworks, tools, languages and abstractions, specially designed to hide many development and runtime details. One of the key aspects is the safe and easy manipulation of persistent data repositories, usually performed with the help of abstractions like object mappings (e.g. Java JPA), or specialized query languages like Microsoft LINQ.

Obvious benefits are obtained by typefuly integrating query languages in the host programming languages, thus increasing the validation and optimizing power of interpreters and compilers (Serrano *et al.*, 2006; Cooper *et al.*, 2007; Fu *et al.*, 2013; Chlipala, 2015). However, the data manipulation paradigm is moving fast from querying a single data repository to combining data coming from a constellation of data sources. The emergence

of big data has contributed to this shift with the proliferation of multiple, disjoint sources of real-time data. As a consequence, heterogeneous queries are pervasive, in scenarios like medical databases and search engines, web service orchestrations, mobile applications, and web or cloud applications that enrich their interfaces with remote web services. Such queries are usually accomplished with ad-hoc code, that many times is inefficient, error prone, and highly resistant to being changed.

An urgent need arises for development platforms that integrate and query different and separate data sources, in a typeful and seamless way. The wide range of skills needed e.g. to query a relational database, efficiently combine the results with a web-service response, and then produce a map-reduce algorithm to join and filter the results in a NoSQL database, is not part of the skill-set of the average developer. Moreover, such an approach contrasts with the data integration efforts of hiding different sources behind a common interface in a very expressive, but predefined way (cf. (Halevy *et al.*, 2006)).

This paper introduces a model for a data manipulation language for heterogeneous data-centric environments, and a compilation method based on type and location information on data-sources. We define a model to generate specialized and distributed querying code for each (remote) data source, and the corresponding in-memory post-processing code. We model each kind of database system (relational or NoSQL), parameterized data repository (web services), or in-memory data, by a set of capabilities (e.g. to join collections, group by arbitrary expressions, nest results, filter), that guide the way operations are split between locations (Vassalos & Papakonstantinou, 2000). Languages like Microsoft LINQ do allow for several kinds of data sources to be involved in a query, but, in their case, the default execution includes fetching all data first and then combining the pieces in a centralized location. Our model decentralizes parts of a query as in (Wong, 2000), and can be extended with compile time normalization (Cheney *et al.*, 2013) and runtime optimizing strategies like (Grade *et al.*, 2013).

This paper extends and refines the approach presented in (Seco *et al.*, 2015). Our construction and query combination model is designed from first principles, targeting a general model of data sources, from relational data to nested collections (e.g. (Colby, 1989; Cheney *et al.*, 2014)). We explore a novel language operation, introduced in (Seco *et al.*, 2015), whose semantics is the in-place modification of nested data, given a tree-like path (cf. XPath (Clark & DeRose, 1999; Cheney *et al.*, 2014)). This operation can either be applied as an in-memory step or be re-written during the code generation process, and incorporated into the target query code, to be executed remotely. This operation is particularly useful both in supporting the visual counterpart of this model, that supports the incremental and interactive construction of nested queries with immediate feedback on results, and in providing a compositional and incremental way of building queries.

We refine the query transformation process presented in (Seco *et al.*, 2015) by decomposing it into three separate phases, and analyzing the contribution of each step in a precise and separate way. We also add extra formal results and proofs. The first phase of our transformation inserts explicit projection operations in a abstract query expression, and eliminates unnecessary code, to adjust a query to its concrete usage type. This can be seen as a compiler related technique of function inlining and specialization. Then, in a second phase we orderly annotate all subexpression abstract nodes with site locations by means of a (label) rewriting system, according to each location's capabilities. This

second phase uses an intermediate representation for joins that helps distributing the inner queries by the available locations. A third phase is used to translate the located query to the initial syntax and places the necessary remote invocation expressions. Besides the modular design, compared to our previous work, this new approach also provides a uniform process that deals with inner queries in filters and group criteria expressions.

Our approach is being used as the model of an industrial grade development platform for mobile and web applications, the OutSystems Platform (OutSystems, 2016), where different kinds of data-sources can be used in a typeful way (Cardelli, 1989), and where the data manipulation language provides type safety and language integration to developers, while it is compiled in a type preserving way to the state of the art database systems and their native query languages.

In the remainder of the paper we introduce the language by means of a running example (Section 3), that we then use to also illustrate the localization process, presented in Section 6. We formalize the operational semantics of language λ_{CDL} and its type system in Sections 4 and 5. The localization process, divided into three phases, is proven sound with relation to the language semantics. Formal results are presented together with summarized proofs, however they are expanded and presented in full in appendix B.

2 Syntax

In this section, we introduce the data manipulation language (λ_{CDL}), whose expressions and types are defined by the syntax given in Figure 1. An example using this language is presented in section 3. The core expression language is a typed lambda calculus, with base values (*num*, *bool*, *string*, *date*) and the corresponding predefined operations (abstracted as op), also with records and multisets, and equipped with a data manipulation language fragment, capable of querying nested structured data repositories (cf. relational databases, structured JSON data objects, etc.), similar to works using NRC (Buneman *et al.*, 1995; Cheney *et al.*, 2014). Our language is based on a set of predefined named data sources t, variables x, y, z, and record labels a, b. We use the list notation $[\overline{v}]$ to denote the bag construction $[v_1] \uplus [v_2] \ldots \uplus [v_n]$. We assume given a finite set of predefined location identifiers ℓ, for sites hosting data sources.

Our type language includes basic types for integer numbers, strings, and dates. We follow standard lines to type records, multisets, and abstractions. Our language includes also a special type to describe queries, that are first-class values in the language.

We extend the core calculus with query operations, starting by an expression that represents queries on parameterized data sources ($\mathsf{db}_\ell(t, \overline{e})$), and base queries that are directly defined by base language expressions, yielding their denoted values ($\mathsf{return}\ e$). We introduce a general iteration operation, over a set of joined inner queries ($\overline{e}$), of the form ($\mathsf{foreach}_c\ \{\ \overline{x \leftarrow e}\ \}\ e'$), using cursors ($\overline{x}$), and filtered by a condition (c). We introduce an operation, of the form ($\mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow e\ \}$), that groups the results of an inner query (e) by a set of computed criteria ($\overline{a = e}$) where the label to access the details of each group is also given (b). This operation corresponds to the specification of nested query results, regardless of the underlying support. Cursor x is bound in the grouping criteria expressions $\overline{e}$. We also include an explicit projection operation ${}^{\tau}\pi^{\sigma}(e)$, from type σ to type τ that is

τ, σ	::=		(*Types*)
		$\mathsf{num} \mid \mathsf{bool} \mid \mathsf{string} \mid \mathsf{date}$	(Basic Types)
	|	$\langle \overline{a : \tau} \rangle$	(Record Type)
	|	τ^*	(Multiset Type)
	|	$\tau \to \sigma$	(Function Type)
	|	$\mathscr{Q}(\tau)$	(Query Type)
e, c	::=		(*Expressions*)
		num	(Numeric)
	|	$bool$	(Boolean)
	|	$string$	(String)
	|	$date$	(Date)
	|	e op e'	(Base Value Operations)
	|	x	(Identifier)
	|	$\lambda x{:}\tau.e$	(Abstraction)
	|	$e\ e$	(Call)
	|	$\langle \overline{a = e} \rangle$	(Record)
	|	$e \oplus e$	(Record Concatenation)
	|	$e.a$	(Field Selection)
	|	$\emptyset$	(Empty List)
	|	$[e]$	(Singleton List)
	|	$e \uplus e$	(List Concatenation)
	|	$\mathsf{exec}\ x : \tau = e\ \mathsf{in}\ e$	(Query execution)
	|	$\mathsf{db}_\ell(t, \overline{e})$	(Data source)
	|	$\mathsf{foreach}_c \{ \overline{x \leftarrow e} \}\ e$	(Iteration operation)
	|	$\mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \}$	(Group-by operation)
	|	$\mathsf{do}\ e_{\downarrow p}\{e\}$	(At operation)
	|	$\mathsf{return}\ e$	(Return operation)
	|	${}^{\tau}\pi^{\sigma}(e)$	(Projection operation)
	|	$[\, e \,]_\ell$	(Remote evaluation)
p	::=		(*Paths*)
		ε	(Empty path)
	|	$.a \cdot p$	(Record path)
	|	$/p$	(List path)

Fig. 1. Syntax of Expressions

defined for expressions yielding record or list values. Finally, expression $[\, e \,]_\ell$ represents the remote evaluation of a query expression e in a site identified by location ℓ.

In order to manipulate and transform structured nested data, we introduce a general purpose operation that operates deep in the nested query results. The operation, of the form ($\mathsf{do}\ e_{\downarrow p}\{e'\}$), applies the abstraction denoted by expression e to the fragments of the resulting values of query e' identified by path p. Paths, where a is a record label and p a complete path, specify the traversal of a data structure composed of records ($.a \cdot p$), and lists ($/p$). We sometimes abbreviate the traversal of a list of records ($/.a \cdot p$) with ($/a \cdot p$). This so called "at" operation allows in-place modification of parts of nested results, by iterating or filtering them, joining them with other data-sources, or grouping them with local criteria. We adopt this kind of operation as a generalization of functional map operations in semi-structured manipulation languages. The "at" operation allows for query simplifications by

$teams = \mathsf{db}_{\texttt{SALESDB}}(\texttt{Team})$

id	name
1	Alpha
2	Bravo
3	Charlie

Fig. 2. Teams - SALESDB

transporting the operation closer to the specified location, can be factored together with other operations, and can be compiled into imperative style query languages, as the ones found in No-SQL data stores (e.g. local storage with indexedDB).

We define queries as logically separated values, that can be gradually composed by query operations and executed separately (cf. staged computations (Davies & Pfenning, 2001; Cheney *et al.*, 2013)). The base constructors have the form $\mathsf{return}\ e$ and $\mathsf{db}_\ell(t,\overline{e})$, and the expression $\mathsf{exec}\ x : \tau = e\ \mathsf{in}\ e'$ represents the execution of the query denoted by e, having a usage type τ, binding its results to x in e'. In this way we are able to conveniently model a type based query localization and optimization procedure (based on the usage type) just by isolating query typed values. A query whose result type is τ, is described by means of a special type $\mathscr{Q}(\tau)$. The query resulting data is then obtained by the explicit evaluation of the query expression in a exec expression. For the sake of simplicity, we write $\mathsf{run}\ e$ to abbreviate $\mathsf{exec}\ x : \tau = e\ \mathsf{in}\ x$, and $\mathsf{db}_\ell(t)$ to abbreviate $\mathsf{db}_\ell(t,\langle\rangle)$ when t has type $\langle\rangle \to \tau$, where $\langle\rangle$ is the empty record type used here to represent the unit type.

3 Example

To illustrate and motivate the language semantics, we use the running example below. Consider a mobile application that organizes the daily job of field technicians in a telecom company. Its core data is stored in two separate cloud based relational databases named SALESDB and SAP, as depicted in Figures 2, 3, and 4, and whose schemas are as follows:

- $\texttt{Team} : \langle\rangle \to \tau_T^*$ where $\tau_T = \langle \mathit{id}\colon \mathsf{num},\ \mathit{name}\colon \mathsf{string}\rangle$
- $\texttt{Job} : \langle\rangle \to \tau_J^*$ where $\tau_J = \langle \mathit{id}\colon \mathsf{num},\ \mathit{title}\colon \mathsf{string},\ \mathit{teamId}\colon \mathsf{num},\ \mathit{clientId}\colon \mathsf{num},\ \mathit{date}\colon \mathsf{date},\ \mathit{time}\colon \mathsf{num}\rangle$
- $\texttt{Client} : \langle\rangle \to \tau_C^*$ where $\tau_C = \langle \mathit{id}\colon \mathsf{num},\ \mathit{name}\colon \mathsf{string},\ \mathit{address}\colon \mathsf{string}\rangle$

The system also uses a geolocation web service, named GEO, to obtain the GPS coordinates for a given street address, which is specified by the following function type:

- $\texttt{Coords} : \mathsf{string} \to \tau_L$ where $\tau_L = \langle \mathit{lat}\colon \mathsf{num},\ \mathit{lng}\colon \mathsf{num}\rangle$

A developer needs to know the tasks assigned to a team in a given date, e.g., May 8. So, she gradually builds a query. The first step is to join the tables Team, Job, and Client,

$$jobs = \mathsf{db}_{\texttt{SALESDB}}(\texttt{Job})$$

id	title	teamId	clientId	date	time
1	Check WiFi	1	2	8/5	10
2	Replace phone	1	3	8/5	11
3	Setup TV	2	1	8/5	10
4	Install router	1	4	8/5	14
5	Replace cable	3	4	10/5	9

Fig. 3. Jobs - SALESDB

$$clients = \mathsf{db}_{\texttt{SAP}}(\texttt{Client})$$

id	name	address
1	Helen	75 Globe Road, London
2	Ive	58 Pitfold Road, London
3	James	4 Dean's Court, London
4	Lewis	25 Ebury Bridge Road, London

Fig. 4. Clients - SAP

Figure 5, using a foreach expression, a basic filter, and record constructs.

$$work = \mathsf{foreach}_{t.id=j.teamId \wedge j.clientId=c.id \wedge j.date=8/5} \left\{ \begin{array}{l} t \leftarrow teams, \\ j \leftarrow jobs, \\ c \leftarrow clients \end{array} \right\}$$
$$\langle team = t,\ job = j,\ client = c \rangle$$

Next, the developer groups the results by team's name, with a groupby expression, Figure 6. The result is a nested collection of records, each containing a team's name, and a list of records (job, team, and client).

$$workByTeam = \mathsf{groupby}^{name=x.team.name}_{details} \{\ x \leftarrow work\ \}$$

In our example, we still need the GPS coordinates of each client's address. To obtain them, we call the Coords web-service for each one of the addresses, by modifying the current query with an in-place operation using path $/details$. See Figure 7 for the data resulting from

$$addLoc = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \} \\ (y \oplus \langle loc = \mathsf{run}\ \mathsf{db}_{\texttt{GEO}}(\texttt{Coords}, y.client.address)\rangle)$$
$$withLoc = \mathsf{do}\ addLoc_{\downarrow/details}\{workByTeam\}$$

Our approach is suited to a scenario of a visual manipulation language where a user interface can be designed to naturally define "at" operations, by pointing to the displayed data. It is also useful in an incremental query composition environment where the original data is originally nested, via other queries or web services, and the developer writes refinements over existing queries, in opposition to modify the initial query to include the new column.

$$work = \mathsf{foreach}_{t.id=j.teamId \wedge j.clientId=c.id \wedge j.date=8/5} \{\, t \leftarrow teams, j \leftarrow jobs, c \leftarrow clients \,\} \\ \langle team = t, job = j, client = c \rangle$$

team		job				client		
id	name	id	title	clientId	time	id	name	address
1	Alpha	1	Check WiFi	2	10	2	Ive	58 Pitfold Road, London
1	Alpha	2	Replace phone	3	11	3	James	4 Dean's Court, London
2	Bravo	3	Setup TV	1	10	1	Helen	75 Globe Road, London
1	Alpha	4	Install router	4	14	4	Lewis	25 Ebury Bridge Road, London

Fig. 5. Work assignment for May 8

$$workByTeam = \mathsf{groupby}^{name=x.team.name}_{details} \{\, x \leftarrow work \,\}$$

name	details							
	team	job				client		
	...	id	title	clientId	time	id	name	address
Alpha	...	1	Check WiFi	2	10	2	Ive	58 Pitfold Road, London
	...	2	Replace phone	3	11	3	James	4 Dean's Court, London
	...	4	Install router	4	14	4	Lewis	25 Ebury Bridge Road, London
Bravo	...	3	Setup TV	1	10	1	Helen	75 Globe Road, London

Fig. 6. Group by team's name

Given the complete, expanded query

$$\begin{array}{l} \mathsf{do}_{\downarrow/details} \\ \quad (\lambda x.\mathsf{foreach}\,\{\, y \leftarrow x \,\}\ (y \oplus \langle loc = \mathsf{run}\ \mathsf{db}_{\mathtt{GEO}}(\mathtt{Coords}, y.client.address) \rangle)) \\ \{ \\ \quad \mathsf{groupby}^{name=x.team.name}_{details} \{\, x \leftarrow \\ \qquad \mathsf{foreach}_{t.id=j.teamId \wedge j.clientId=c.id \wedge j.date=8/5} \left\{ \begin{array}{l} t \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Team}), \\ j \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Job}), \\ c \leftarrow \mathsf{db}_{\mathtt{SAP}}(\mathtt{Client}) \end{array} \right\} \\ \qquad\qquad \langle team = t,\ job = j,\ client = c \rangle \\ \quad \} \\ \} \end{array}$$

the best way to orchestrate this query is to dispatch the join between the `Team` and `Job` tables to the `SALESDB` database server running SQL, while the join with the `Client` table needs to be done in memory since the data comes from location `SAP`, a different database server. The `Coords` web-service must also be called in memory. Moreover, we aim at using (typing) information about the concrete usage of data. For instance, if a given client application is not using the GPS coordinates, the call to the `Coords` web service can be safely discarded and a significant amount of processing time can be spared.

In the following sections we define the semantics of the language, and the corresponding typing relation.

$withLoc = \mathsf{do}\ addLoc_{\downarrow/details}\{workByTeam\}$ where
$addLoc = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \}\ (y \oplus \langle loc = \mathsf{run}\ \mathsf{db}_{\mathtt{GEO}}(\mathtt{Coords}, y.client.address)\rangle))$

name	details						
	team	job	client			loc	
	...	clientId ...	id	name	address	lat	lng
Alpha	...	2 ...	2	Ive	58 Pitfold Road, London	51.45	0.02
	...	3 ...	3	James	4 Dean's Court, London	51.52	-0.15
	...	4 ...	4	Lewis	25 Ebury Bridge Road, London	51.50	-0.15
Bravo	...	1 ...	1	Helen	75 Globe Road, London	51.52	-0.05

Fig. 7. Get address coordinates

Values	u, v	::=	$\lambda x.e$
		|	$\langle \overline{a = u} \rangle$
		|	$\emptyset$
		|	$[v]$
		|	$v \uplus v$
		|	*num*
		|	*bool*
		|	*string*
		|	*date*
		|	r
Query values	r, s	::=	$\mathsf{db}_\ell(t, \overline{v})$
		|	$\mathsf{foreach}_c\ \{\ \overline{x \leftarrow r}\ \}\ e$
		|	$\mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow r\ \}$
		|	$\mathsf{do}\ (\lambda x.e)_{\downarrow p}\{r\}$
		|	$\mathsf{return}\ v$
		|	${}^{\tau}\pi^{\sigma}(r)$
		|	$[\ r\ ]_\ell$

Fig. 8. Language Values

4 Semantics of λ_{CDL}

The operational semantics for λ_{CDL} is defined by a big-step relation on expressions with relation to a state ($\mathscr{S}$), representing referred data repositories. We write $\langle\!\langle e \rangle\!\rangle$ to denote the computed value of an expression e, defined by the grammar in Figure 8, and define it using the cases in Figures 9 and 10. The evaluation of query expressions corresponds to the staging of queries, that are afterwards executed with relation to the given state, by means of an exec expression. In our scenario, this corresponds to executing queries in remote database systems. We use sets ($\{\overline{e}\}$) and multi-sets ($[\overline{e}]$), with list comprehension notation, as the basis to define the semantics of executing query values r, by the relation $[\![r]\!]$, defined in Figure 11. Our approach is inspired by works like (Buneman *et al.*, 1994; Buneman *et al.*, 1995; Jones & Wadler, 2007; Cheney *et al.*, 2014).

The call-by-value semantics of expressions is straightforwardly defined in the structure of the expressions in most of the cases, hence we avoid a detailed explanation. Instead, we

$$
\begin{array}{ll}
\langle\!\langle v \rangle\!\rangle = v & \\
\langle\!\langle e \text{ op } e' \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle \text{ op } \langle\!\langle e' \rangle\!\rangle & \\
\langle\!\langle e\ e' \rangle\!\rangle = \langle\!\langle e''\{^{v}/_{x}\} \rangle\!\rangle & \text{where } \langle\!\langle e \rangle\!\rangle = \lambda x{:}\tau.e'' \\
 & \qquad \langle\!\langle e' \rangle\!\rangle = v \\
\langle\!\langle \langle \overline{a = e} \rangle \rangle\!\rangle = \langle \overline{a = \langle\!\langle e \rangle\!\rangle} \rangle & \\
\langle\!\langle e.a \rangle\!\rangle = v & \text{where } \langle\!\langle e \rangle\!\rangle = \langle a = v, \ldots \rangle \\
\langle\!\langle e \oplus e' \rangle\!\rangle = \langle \overline{a = v}, \overline{b = u} \rangle & \text{where } \langle\!\langle e \rangle\!\rangle = \langle \overline{a = v} \rangle \\
 & \qquad \langle\!\langle e' \rangle\!\rangle = \langle \overline{b = u} \rangle \\
\langle\!\langle [e] \rangle\!\rangle = [\langle\!\langle e \rangle\!\rangle] & \\
\langle\!\langle e \uplus e' \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle \uplus \langle\!\langle e' \rangle\!\rangle & \\
\langle\!\langle {}^{\tau\to\sigma}\pi^{\tau'\to\sigma'}(\lambda x{:}\tau'.e) \rangle\!\rangle = \lambda x{:}\tau.({}^{\sigma}\pi^{\sigma'}(e)) & \\
\langle\!\langle {}^{\langle \overline{a:\tau} \rangle}\pi^{\langle \overline{a:\tau}, \overline{b:\sigma} \rangle}(\langle \overline{a = u}, \overline{b = v} \rangle) \rangle\!\rangle = \langle \overline{a = u} \rangle & \\
\langle\!\langle {}^{\tau^*}\pi^{\sigma^*}(\emptyset) \rangle\!\rangle = \emptyset & \\
\langle\!\langle {}^{\tau^*}\pi^{\sigma^*}([v]) \rangle\!\rangle = [{}^{\tau}\pi^{\sigma}(v)] & \\
\langle\!\langle {}^{\tau^*}\pi^{\sigma^*}(u \uplus v) \rangle\!\rangle = {}^{\tau^*}\pi^{\sigma^*}(u) \uplus {}^{\tau^*}\pi^{\sigma^*}(v) & \\
\langle\!\langle {}^{\tau}\pi^{\sigma}(e) \rangle\!\rangle = \langle\!\langle {}^{\tau}\pi^{\sigma}(\langle\!\langle e \rangle\!\rangle) \rangle\!\rangle & \\
\langle\!\langle \mathsf{exec}\ x = e\ \mathsf{in}\ e' \rangle\!\rangle = \langle\!\langle e'\{^{v}/_{x}\} \rangle\!\rangle & \text{where } \langle\!\langle e \rangle\!\rangle = r \\
 & \qquad [\![r]\!] = v
\end{array}
$$

Fig. 9. Operational semantics for expressions

$$
\begin{array}{l}
\langle\!\langle \mathsf{db}_{\ell}(t, \overline{e}) \rangle\!\rangle = \mathsf{db}_{\ell}(t, \overline{\langle\!\langle e \rangle\!\rangle}) \\
\langle\!\langle \mathsf{foreach}_{c}\{\ \overline{x \leftarrow e}\ \}\ e' \rangle\!\rangle = \mathsf{foreach}_{c}\left\{\ \overline{x \leftarrow \langle\!\langle e \rangle\!\rangle}\ \right\}\ e' \\
\langle\!\langle \mathsf{groupby}_{b}^{\overline{a=e}}\{\ x \leftarrow e'\ \} \rangle\!\rangle = \mathsf{groupby}_{b}^{\overline{a=e}}\{\ x \leftarrow \langle\!\langle e' \rangle\!\rangle\ \} \\
\langle\!\langle \mathsf{do}\ e_{\downarrow p}\{e'\} \rangle\!\rangle = \mathsf{do}\ e_{\downarrow p}\{\langle\!\langle e' \rangle\!\rangle\} \\
\langle\!\langle \mathsf{return}\ e \rangle\!\rangle = \mathsf{return}\ \langle\!\langle e \rangle\!\rangle \\
\langle\!\langle [\ e\ ]_{\ell} \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle
\end{array}
$$

Fig. 10. Operational semantics for query expressions

describe the cases of non-standard constructs. For instance, we resort to a native definition of the semantics for predefined operations (op). For instance, in the case of a projection operation, the base cases are defined on record values, by removing the fields filtered out, and the projection operator commutes with all other operators like abstraction and list construction. Notice that a projection operation is only meaningful if the resulting type is strictly a supertype of the source type. Moreover, note that if a projection operation is applied to a query, it is staged and thus promoted to a query value itself. In the case of expression $\mathsf{exec}\ x : \tau = e\ \mathsf{in}\ e'$, it first evaluates (stages) the query value denoted by e, and proceeds with the evaluation of e' binding x to the results of the query (cf. (Davies & Pfenning, 2001)). This is an extension point of the language that we use to introduce the typed compilation procedure, that transforms queries before actually executing them.

The language fragment that represent query operations is interpreted by the top-level semantic function ($\langle\!\langle e \rangle\!\rangle$), by the cases in Figure 10, thus producing closed query values.

$$[\![\mathsf{db}_\ell(t,\overline{v})]\!] = (\mathscr{S}(t))(\overline{v})$$

$$[\![\mathsf{foreach}_c\ \{\ \overline{x \leftarrow r}\ \}\ e]\!] = [\ e\{\overline{u}/\overline{x}\} \mid \overline{u \in [\![r]\!]},\ c\{\overline{u}/\overline{x}\}\ ]$$

$$[\![\mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow r\ \}]\!] = [\ k \oplus \langle b = details_k \rangle \mid k \in keys\ ]$$
where
$keys = \{\langle \overline{a = e_a\{u/x\}} \rangle \mid u \in [\![r]\!]\}$
$details_k = [u \mid u \in [\![r]\!], \langle \overline{a = e_a\{u/x\}} \rangle = k]$

$$[\![\mathsf{do}\ e_{\downarrow\varepsilon}\{r\}]\!] = [\![s]\!]$$
where
$s = \langle\!\langle e\ (\mathsf{return}\ [\![r]\!]) \rangle\!\rangle$

$$[\![\mathsf{do}\ e_{\downarrow.a\cdot p}\{r\}]\!] = \langle a = [\![\mathsf{do}\ e_{\downarrow p}\{\mathsf{return}\ u\}]\!], \overline{b = v} \rangle$$
where
$[\![r]\!] = \langle a = u, \overline{b = v} \rangle$

$$[\![\mathsf{do}\ e_{\downarrow/p}\{r\}]\!] = [\ [\![\mathsf{do}\ e_{\downarrow p}\{\mathsf{return}\ u\}]\!] \mid u \in [\![r]\!]\ ]$$

$$[\![^\tau\pi^\sigma(r)]\!] = \langle\!\langle {}^\tau\pi^\sigma([\![r]\!]) \rangle\!\rangle$$

$$[\![[\ r\ ]_\ell]\!] = [\![r]\!]$$

Fig. 11. Operational semantics for query values

The semantics of executing query values (Figure 11), states that a data source invocation ($\mathsf{db}_\ell(t,\overline{v})$) is represented by directly accessing state $\mathscr{S}$, and calling the data source end point with the given parameters. This general model using sources with parameters allows the representation of both web-services that require parameters, and database tables which do not. The execution of an iteration operation (foreach) includes joining the results of inner queries, and then producing and filtering a value for each tuple. Group-by operations (groupby) compute the unique values given by the grouping criteria (i.e. the keys), and use them to produce a nested structure, which pairs each key with a details field containing all the original values that are grouped under it.

The semantics of operations of the form $\mathsf{do}\ e_{\downarrow p}\{r\}$, is defined by case analysis of the path given. In the case of an empty path (ε), the operation is mapped onto applying the abstraction denoted by expression e to the results of query r. The case of a list path ($/p$), which corresponds to a map operation, recursively follows the remaining path for each of the elements in the collection. The case of record traversal ($.a \cdot p$) specifies the navigation in the structure of the target value and recursively follows the remaining path.

5 Typing

In order to typecheck λ_{CDL} expressions, we recall the types introduced in Figure 1,

$$\tau, \sigma\ ::=\quad \mathsf{num} \mid \mathsf{bool} \mid \mathsf{string} \mid \mathsf{date} \mid \langle \overline{a : \tau} \rangle \mid \tau^* \mid \tau \to \sigma \mid \mathscr{Q}(\tau)$$

$$\frac{\Delta \vdash e_i : \tau_i \quad {}_{i=1..n}}{\Delta, t : \overline{\tau} \to \tau \vdash \mathsf{db}_\ell(t, \overline{e}) : \mathscr{Q}(\tau)} \ (\textsc{Source})$$

$$\frac{\Delta \vdash e_i : \mathscr{Q}(\tau_i^*) \ {}_{i=1..n} \quad \Delta, \overline{x : \tau} \vdash e' : \mathsf{bool} \quad \Delta, \overline{x : \tau} \vdash e'' : \sigma}{\Delta \vdash \mathsf{foreach}_{e'} \{ \overline{x \leftarrow e} \} \ e'' : \mathscr{Q}(\sigma^*)} \ (\textsc{Select})$$

$$\frac{\Delta \vdash e : \mathscr{Q}(\tau^*) \quad \Delta, x : \tau \vdash e_i : \sigma_i \quad {}_{i=1..n}}{\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)} \ (\textsc{Group})$$

$$\frac{\Delta \vdash e : \tau}{\Delta \vdash \mathsf{return}\ e : \mathscr{Q}(\tau)} \ (\textsc{Return}) \qquad \frac{\Delta \vdash e : \mathscr{Q}(\tau') \to \mathscr{Q}(\sigma') \quad \Delta \vdash e' : \mathscr{Q}(\tau)}{\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathscr{Q}(\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\})} \ (\textsc{At})$$

$$\frac{\Delta \vdash e : \mathscr{Q}(\sigma') \quad \sigma' \leq \sigma \quad \Delta, x : \sigma \vdash e' : \tau}{\Delta \vdash \mathsf{exec}\ x : \sigma = e\ \mathsf{in}\ e' : \tau} \ (\textsc{Exec})$$

$$\frac{\Delta \vdash e : \sigma \quad \sigma \leq \tau}{\Delta \vdash {}^{\tau}\pi^{\sigma}(e) : \tau} \ (\textsc{Project}) \qquad \frac{\Delta \vdash e : \tau}{\Delta \vdash [\, e \,]_\ell : \tau} \ (\textsc{Remote})$$

Fig. 12. Typing relation (Selected rules)

that include basic types for integer numbers, strings, and dates, to match our running example, record types, multiset types, and abstractions. We also assume a predefined typing relation for all predefined operations on base values. Recall that a query whose result type is τ, is described by means of a special type $\mathscr{Q}(\tau)$, and that its resulting data is only obtained by the explicit evaluation of the query expression in a exec expression. We inductively define the typing relation for λ_{CDL} in terms of the judgment $\Delta \vdash e : \tau$, according to the rules in Figure 12. We focus only on part of the type system, and omit standard rules from the body of the paper. The complete set of rules of the type system can be found in appendix A. Rules for the functional fragment of the language are quite standard, and are combined with rules for query expressions, projection and remote execution of expressions. Regarding queries, rule (Source) ensures that all data sources are properly accessed, according to the function type signature given by the typing environment Δ. The expression is typed as a query that returns the prescribed type in the function signature. Iteration operations, in rule (Select), are typed so that cursors, representing elements of the results of inner queries, and that conditions and select expression are well typed. Also, in group operations, rule (Group), the inner query must be well typed and the group criteria expressions given the corresponding cursor type. Return operations are typed so that any value can be used as a query, rule (Return). Rule (At) types an operation that is applied, in-place, deep in the structure of a query. We define below a type transformation function that, given a path, follows it through the structure of the type, and when matching the type at the end of the path, applies a direct type transformation in-place, called *type-at*.

Definition 5.1 (*Type at*)
For all types $\tau, \tau', \sigma, \sigma'$, paths p and labels a, the *type-at*, written $\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\}$, operation is defined inductively in the size of path p.

$$\begin{array}{rcl} \tau_{\downarrow \varepsilon}\{^{\sigma}/_{\tau}\} & \triangleq & \sigma \\ \tau^*{}_{\downarrow /}\{^{\sigma}/_{\tau}\} & \triangleq & \sigma^* \\ (\langle a : \tau \rangle \oplus \sigma)_{\downarrow . a \cdot p}\{^{\sigma'}/_{\tau'}\} & \triangleq & (\langle a : \tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\} \rangle \oplus \sigma) \end{array}$$

Operation *type-at* applies a query transformation operation, of type $\mathcal{Q}(\tau') \to \mathcal{Q}(\sigma')$, and the operation on types $\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\}$, validates and transforms the necessary "deep" transformation of the target query.

The typing of the query execution expressions, rule (EXEC), checks that a query is executed and used according to the expression's type annotation, representing the actual usage of the results. The remote execution of expressions, rule (REMOTE), checks that the remotely executed (sub) expression is well typed. Finally, the (PROJECT) rule checks that a projection is only performed when the subtyping relationship, defined below, is guaranteed.

Subtyping We also consider the universal (transitive) subtyping relation for function, record, queries, and lists, introduced in typing by means of rule (SUB).

$$\tau \leq \tau \qquad \frac{\tau_i \leq \tau'_i \quad i = 0..n}{\langle \overline{a : \tau}, \overline{b : \sigma} \rangle \leq \langle \overline{a : \tau'} \rangle} \qquad \frac{\tau' \leq \tau \quad \sigma \leq \sigma'}{\tau \to \sigma \leq \tau' \to \sigma'} \qquad \frac{\tau \leq \sigma}{\tau^* \leq \sigma^*} \qquad \frac{\tau \leq \tau'}{\mathcal{Q}(\tau) \leq \mathcal{Q}(\tau')}$$

We introduce subtyping in the language as a way to express the soundness invariant of the code transformation defined ahead. However, in some of the database systems we are considering, type coercion under this universal subtyping is not automatic. Thus, in source programs we always consider typing as being derived without the subsumption rule (SUB).

Example Here, we illustrate the application of the typing relation presented here to our running example. In order to type the query definitions in Figures 2 to 7, let the typing context $\Delta \triangleq \{$ `Team` $: \langle\rangle \to \tau_T^*$, `Job` $: \langle\rangle \to \tau_J^*$, `Client` $: \langle\rangle \to \tau_C^*$, `Coords` $:$ `string` $\to \tau_L \}$, with τ_T, τ_J, τ_C, and τ_L as defined in section 3. Then, we have:

- $\Delta \vdash \mathit{teams} : \mathcal{Q}(\tau_T^*)$
- $\Delta \vdash \mathit{jobs} : \mathcal{Q}(\tau_J^*)$
- $\Delta \vdash \mathit{clients} : \mathcal{Q}(\tau_C^*)$
- $\Delta \vdash \mathit{work} : \mathcal{Q}(\langle \mathit{team} : \tau_T, \mathit{job} : \tau_J, \mathit{client} : \tau_C \rangle^*)$
- $\Delta \vdash \mathit{workByTeam} : \mathcal{Q}(\langle \mathit{name} : \mathit{string}, \mathit{details} : \langle \mathit{team} : \tau_T, \mathit{job} : \tau_J, \mathit{client} : \tau_C \rangle^* \rangle^*)$
- $\Delta \vdash \mathit{withLoc} : \mathcal{Q}(\langle \mathit{name} : \mathit{string}, \mathit{details} : \langle \mathit{team} : \tau_T, \mathit{job} : \tau_J, \mathit{client} : \tau_C, \mathit{loc} : \tau_L \rangle^* \rangle^*)$

Given the above subtyping relation and the *type-at*, we prove an intermediate result about the covariance of the at operation.

Lemma 5.2 (*Type-At is Covariant*)
For all types $\tau, \sigma, \delta, \delta', \delta''$ and paths p, if $\tau \leq \sigma$ and $\delta' \leq \delta''$ then $\tau_{\downarrow p}\{^{\delta'}/_{\delta}\} \leq \sigma_{\downarrow p}\{^{\delta''}/_{\delta}\}$.

Proof. According to the Definition 5.1, the substitution occurs only on positive positions, thus the covariance is directly proved by induction on the definition. □

In order to prove soundness of the type system, we also state a derivable weakening property (both width and depth), based on the following auxiliary definition:

Definition 5.3 (*Environment Subtyping*)
For all typing environments Δ, Δ', we write $\Delta' \leq \Delta$, if and only if $Dom(\Delta') = Dom(\Delta)$, $\forall_{y \in Dom(\Delta)}. \Delta'(y) \leq \Delta(y)$.

Lemma 5.4 (*Weakening*)
For all typing environments Δ, Δ', expressions e, and types τ, σ, if $\Delta \vdash e : \tau$ and $x \notin FV(e)$ then $\Delta', x : \sigma \vdash e : \tau'$ with $\tau' \leq \tau$, and $\Delta' \leq \Delta$.

Proof. The proof follows by induction on the derivation of the typing relation, and having in mind that the in-place substitution of types (the *type-at* operation) is covariant in rule (AT), and using the transitivity of subtyping in rules (EXEC) and (PROJECT). □

We prove the soundness to the typing relation with relation to the operational semantics following standard lines, in Theorem 5.5.

Theorem 5.5 (*Type Soundness*)
1. If $\Delta \vdash e : \tau$ and $\langle\!\langle e \rangle\!\rangle = v$ then $\Delta \vdash v : \tau'$ with $\tau' \leq \tau$.
2. If $\Delta \vdash r : \mathscr{Q}(\tau)$ and $[\![r]\!] = v$ then $\Delta \vdash v : \tau'$ with $\tau' \leq \tau$.

Proof. The proof follows by induction on the typing and type transformation definitions, and supports the usual properties of absence of runtime errors for terminating expressions. See appendix B for the detailed proof. □

6 Localization

Optimizations are a well-known problem in relational databases, with many variants (Silberschatz *et al.*, 2006) that shape the execution plan in order to optimize the usage of memory and CPU time. In a distributed and heterogeneous setting, the criteria to optimize a query's execution plan are somewhat different. The way different data sources are interplayed can shorten the execution time of a query in a significant way because the determining factor is no longer memory usage and CPU time, but the amount of data that is interchanged through the network (Taylor, 2010), the number of locations visited, and the native capabilities used on each database system or data repository.

We next extend the data manipulation language introduced in section 2 with a location and type based transformation process for queries. Queries are transformed in such a way that subexpressions are grouped to be shipped to remote locations, and executed in the most efficient way possible. We use knowledge about the capabilities of each remote site (Papakonstantinou *et al.*, 1998), in order to place the operations as close as possible to the origin of the data. The parts of a query that can be computed remotely are grouped and dispatched, and an *in-memory* post-processing phase is generated to complete the job, in the starter location. We leverage not only on the locations of data sources, but also on the actual usage of data, which is expressed as type information. The transformation process prunes the query tree, to avoid fetching unnecessary data, and eliminates all remote invocations that have impact on the processing time but do not influence the query result. We

divide the compilation process into the use of type information to prune parts of the query and the eager localization of the query components. For an optimized distributed execution we foresee that we can use orthogonal strategies to efficiently execute it (e.g. (Grade *et al.*, 2013) and (Taylor, 2010)).

We improve and refine the process presented in (Seco *et al.*, 2015) and present a query transformation process consisting of three separate and orthogonal steps. The first phase corresponds to the pruning of the query expression based on the result usage type. The process takes as input a query expression and the corresponding usage type and recursively transforms it by either erasing unnecessary subexpressions or explicitly inserting projection expressions in the query code. The second phase of the process is based on a rewriting system on expressions. Each (sub)expression node in a query is annotated with a location such that the whole expression is executable in the starting location. The rewriting process then refines the location of expression nodes based on their intrinsic capabilities and the locations assigned to its children nodes. We prove that the rewriting function is monotone, and hence the process stops when a fixed point is reached. This phase uses a special representation and organization of iteration operation binders, so that binders and conditions can be grouped according to their intrinsic locations. Finally, the third phase of our transformation process is designed to explicitly produce located query code, by introducing remote calls when needed and expanding groups of binders into located subqueries.

One important aspect on our setting is that it does not change the structure of the query, it is based solely only on the location of subexpressions. Standard use of a cost model may lead to further optimizations in the execution of query fragments in remote nodes.

6.1 Phase I: Usage based projection

The first step of our compilation process consists on recursively transforming query expressions, by inserting explicit projection operations, and trimming record construction operations to adjust them to the actual usage type. We define a type directed projection relation, represented by the judgment,

$$\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma$$

that denotes the transformation of expression e to expression e', based on the expression type τ, the actual usage type σ, the typing environment Δ that maps all free variables of e to their type, and the usage typing environment Γ that maps all free variables in e' to their actual usage. Algorithmically, the rules should be read as if the typing environment Δ, type τ and usage type σ are given as input, and the algorithm's outputs are the transformed expression e' and its usage typing environment Γ.

Definition 6.1 (*Type Directed Projection*)
We inductively define the type directed projection relation, written $\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma$, by the rules in Figures 13 and 14.

The expected soundness invariant in the projection relation, is that the expression type τ is a subtype of its possible usages σ. This is expressed and verified by the soundness

$$\Delta;\emptyset \vdash num : \mathsf{Num} \Rightarrow num : \mathsf{Num} \qquad (\pi\text{−NUM})$$

$$\Delta;\emptyset \vdash bool : \mathsf{Bool} \Rightarrow bool : \mathsf{Bool} \qquad (\pi\text{−BOOL})$$

$$\Delta;\emptyset \vdash date : \mathsf{Date} \Rightarrow date : \mathsf{Date} \qquad (\pi\text{−DATE})$$

$$\Delta;\emptyset \vdash string : \mathsf{String} \Rightarrow string : \mathsf{String} \qquad (\pi\text{−STRING})$$

$$\Delta, x:\tau;\ \emptyset, x:\tau \vdash x:\tau \Rightarrow x:\tau \qquad (\pi\text{−ID})$$

$$\frac{\tau < \tau'}{\Delta, x:\tau;\ \emptyset, x:\tau' \vdash x:\tau \Rightarrow {}^{\tau'}\pi^{\tau}(x):\tau'} \qquad (\pi\text{−ID-SUB})$$

$$\frac{\Delta, x:\tau;\Gamma, x:\tau \vdash e:\sigma \Rightarrow e':\sigma'}{\Delta;\Gamma \vdash (\lambda x{:}\tau.e):\tau\to\sigma \Rightarrow (\lambda x{:}\tau.e'):\tau\to\sigma'} \qquad (\pi\text{−ABSTRACTION})$$

$$\frac{\Delta;\Gamma \vdash e:\delta\to\tau \Rightarrow e'':\delta'\to\sigma \quad \Delta;\Gamma \vdash e':\delta \Rightarrow e''':\delta'}{\Delta;\Gamma \vdash (e\ e'):\tau \Rightarrow (e''\ e'''):\sigma} \qquad (\pi\text{−APPLICATION})$$

$$\frac{\Delta;\Gamma \vdash e_i:\tau_i \Rightarrow e_i'':\tau_i'\ \ {\scriptstyle i=0..n}}{\Delta;\Gamma \vdash \langle \overline{a=e}, \overline{b=e'}\rangle : \langle \overline{a:\tau}, \overline{b:\sigma}\rangle \Rightarrow \langle \overline{a=e''}\rangle : \langle \overline{a:\tau'}\rangle} \qquad (\pi\text{−RECORD})$$

$$\frac{\Delta;\Gamma \vdash e_i:\tau_i \Rightarrow e_i':\sigma_i\ \ {\scriptstyle i=1..2}}{\Delta;\Gamma \vdash e_1 \oplus e_2 : \tau_1 \oplus \tau_2 \Rightarrow e_1' \oplus e_2' : \sigma_1 \oplus \sigma_2} \qquad (\pi\text{−CONCAT})$$

$$\frac{\Delta;\Gamma \vdash e:\tau \Rightarrow e':\sigma}{\Delta;\Gamma \vdash [e]:\tau^* \Rightarrow [e']:\sigma^*} \qquad (\pi\text{−SINGLETON})$$

$$\frac{\Delta;\Gamma \vdash e_i:\tau^* \Rightarrow e_i':\sigma^*\ \ {\scriptstyle i=1..2}}{\Delta;\Gamma \vdash e_1 \uplus e_2 : \tau^* \Rightarrow e_1' \uplus e_2' : \sigma^*} \qquad (\pi\text{−APPEND})$$

Fig. 13. Type Directed Projection transformation (I)

Lemma 6.2 below. Notice that for all basic values, rules (π−NUM), (π−BOOL), (π−DATE), (π−STRING), types and expressions are not changed. In the case of identifiers, a projection operation is introduced, only if the types strictly differ, rules (π−ID) and (π−ID-SUB). Recall that the projection operation is also defined for abstraction values and corresponds to projecting its resulting value, Figure 9. In the case of function literals, the projection is expanded to the function body. The case of record literal expressions, the filtered out field expressions are simply omitted (since we are in a purely functional setting), rule (π−RECORD), while the case for concatenation of records splits the required usage between both its record expressions, rule (π−CONCAT). Notice that $\tau \oplus \sigma$ denotes the

$$\frac{\Delta(t) = \overline{\sigma} \to \tau \qquad \Delta;\Gamma \vdash e_i : \sigma \Rightarrow e_i' : \sigma \quad i = 1..n \quad \tau \leq \tau'}{\Delta;\Gamma \vdash \mathsf{db}_\ell(t,\overline{e}) : \mathcal{Q}(\tau) \Rightarrow {}^{\mathcal{Q}(\tau')}\pi^{\mathcal{Q}(\tau)}(\,\mathsf{db}_\ell(t,\overline{e'})\,) : \mathcal{Q}(\tau')} \quad (\pi\text{-SOURCE})$$

$$\frac{\begin{array}{c}\Delta,\overline{x:\delta};\Gamma,\overline{x:\delta'} \vdash c : \mathsf{bool} \Rightarrow c' : \mathsf{bool} \qquad \Delta,\overline{x:\delta};\Gamma,\overline{x:\delta''} \vdash e : \sigma \Rightarrow e'' : \sigma' \\ \Delta;\Gamma \vdash e_i : \mathcal{Q}(\delta_i{}^*) \Rightarrow e_i' : \mathcal{Q}(\delta_i'''{}^*)\ {}_{i=1..n} \qquad \overline{\delta'''} \leq \overline{\delta'} \qquad \overline{\delta'''} \leq \overline{\delta''}\end{array}}{\Delta;\Gamma \vdash \mathsf{foreach}_c\,\{\,\overline{x \leftarrow e}\,\}\ e : \mathcal{Q}(\sigma^*) \Rightarrow \mathsf{foreach}_{c'}\,\{\,\overline{x \leftarrow e'}\,\}\ e'' : \mathcal{Q}(\sigma'^*)} \quad (\pi\text{-SELECT})$$

$$\frac{\begin{array}{c}\langle\overline{a:\sigma}, b:\tau^*\rangle^* \leq \delta \\ \Delta,x:\tau;\Gamma,x:\tau' \vdash e_i : \sigma_i \Rightarrow e_i' : \sigma_i\ {}_{i=1..n} \qquad \Delta;\Gamma \vdash e : \mathcal{Q}(\tau^*) \Rightarrow e' : \mathcal{Q}(\tau'^*)\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\,x \leftarrow e\,\} : \mathcal{Q}(\langle\overline{a:\sigma}, b:\tau^*\rangle^*) \Rightarrow \\ \qquad {}^{\mathcal{Q}(\delta)}\pi^{\mathcal{Q}(\langle\overline{a:\sigma},b:\tau^*\rangle^*)}(\mathsf{groupby}_b^{\overline{a=e'}}\{\,x \leftarrow e'\,\}) : \mathcal{Q}(\delta)\end{array}} \quad (\pi\text{-GROUP})$$

$$\frac{\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma}{\Delta;\Gamma \vdash \mathsf{return}\ e : \mathcal{Q}(\tau) \Rightarrow \mathsf{return}\ e' : \mathcal{Q}(\sigma)} \quad (\pi\text{-RETURN})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : \mathcal{Q}(\tau') \to \mathcal{Q}(\sigma') \Rightarrow e'' : \mathcal{Q}(\tau'') \to \mathcal{Q}(\sigma'') \\ \Delta;\Gamma \vdash e' : \mathcal{Q}(\tau) \Rightarrow e''' : \mathcal{Q}(\delta') \quad \delta = \delta'_{\downarrow p}\{^{\tau''}/_{\sigma''}\}\end{array}}{\Delta;\Gamma \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathcal{Q}(\tau_{\downarrow p}\{^{\tau'}/_{\sigma'}\}) \Rightarrow \mathsf{do}\ e''_{\downarrow p}\{e'''\} : \mathcal{Q}(\delta)} \quad (\pi\text{-DO})$$

$$\frac{\Delta;\Gamma \vdash e : \mathcal{Q}(\sigma'') \Rightarrow e'' : \mathcal{Q}(\sigma) \quad \Delta,x:\sigma;\Gamma,x:\sigma' \vdash e' : \tau \Rightarrow e''' : \tau'}{\Delta;\Gamma \vdash \mathsf{exec}\ x:\sigma = e\ \mathsf{in}\ e' : \tau \Rightarrow \mathsf{exec}\ x:\sigma = e''\ \mathsf{in}\ e''' : \tau'} \quad (\pi\text{-EXEC})$$

$$\frac{\tau \leq \tau' \qquad \Delta;\Gamma \vdash e : \sigma \Rightarrow e' : \tau'}{\Delta;\Gamma \vdash {}^{\tau}\pi^{\sigma}(e) : \tau \Rightarrow e' : \tau'} \quad (\pi\text{-PROJECT})$$

Fig. 14. Type Directed Projection transformation (II)

concatenation of disjoint record types, in the same lines of record values concatenation. List construction and concatenation result directly from the type directed projection of their sub-expressions, by rules ($\pi-$SINGLETON) and ($\pi-$APPEND).

Query expressions are recursively transformed according to the usage type, and projections are inserted when no transformation is possible or the inner query results are needed for the current operation. For instance, in rule ($\pi-$SOURCE), a projection is always inserted, although it can be compiled to code when transformed into native query code. Rule ($\pi-$SELECT) propagates the usage of query cursors, given by the transformation of the select expression ($\overline{\delta''}$), and the transformation of the query condition ($\overline{\delta'}$), into the transformation of the inner query expressions. Recall that we algorithmically interpret the right hand side type of the transformation judgment as an input, denoting the target type for the projection. We interpret the types in environment Γ, as outputs of the transformation algorithm. The target to transform the inner queries is $\overline{\delta'''}$, which is the greatest lower bound of δ' and δ'', the subtype that supports the typing of both condition and select expressions. Notice also types $\overline{\delta}$ which are the types of the query cursors. In rule ($\pi-$GROUP), the usage of attributes is not changed to avoid interfering with the

groupby results, hence an explicit projection must be used. The usage type of an "at" operation, in rule ($\pi-$DO), influences both the abstraction being applied as well as the target query. Given the application path and the full usage type, we are able to decompose it and determine the desired return type for the abstraction, and infer the usage type of the target query. This is a rule whose implementation is not direct and requires type inference reasoning to reach a final projection of the code. Rule ($\pi-$EXEC) must use the annotated type as guide for the usage, the compliance of both query and continuation expressions is given by the relation invariant. Also, type σ'' in the premise corresponds to the typing of the expression without subsumption (see remark about subtyping in section 5). Technically, a stronger result of typing environment contraction is needed to drop the annotation in the expression. ($\pi-$PROJECT) incorporates the explicit projection in the code and recursively simplifies the code.

Lemma 6.2 (*Type Soundness - Phase I*)
If $\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma$ then $\tau \leq \sigma$, $\Delta \leq \Gamma$, and $\Delta \vdash e' : \sigma$.

Proof. We prove this result by simple induction on the size of the derivations and using subsumption in the cases where a common supertype is needed. See the detailed proof in appendix B. □

Lemma 6.3 (*Semantic Preservation - Phase I*)
If $\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma$ then ${}^{\sigma}\pi^{\tau}(\langle\!\langle e \rangle\!\rangle) = \langle\!\langle e' \rangle\!\rangle$.

Proof. Many cases are solved by simple induction on the size of the derivation, while the cases where an explicit projection operation is inserted, the result is reached by the definition of the semantics of the projection operation. □

This code transformation results in a trimmed down version of the resulting data, while maintaining the soundness of the original program, namely by taking into account all intermediate results needed to compute the result and not shown in the final results. It follows a type and language based technique, similar to the ones used by compilers to detect dead code.

Example Recall the query *withLoc* from section 3

$$\begin{aligned} addLoc &= \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \} \\ &\qquad (y \oplus \langle loc = \mathsf{run}\ \mathsf{db}_{\mathtt{GEO}}(\mathtt{Coords}, y.client.address)\rangle) \\ withLoc &= \mathsf{do}\ addLoc_{\downarrow/details}\{workByTeam\} \end{aligned}$$

In section 5 we have determined its type to be $\mathscr{Q}(\tau^*)$, with

$$\begin{aligned} \tau &= \langle name : string, details : \tau_d^* \rangle \\ \tau_d &= \langle team : \tau_T, job : \tau_J, client : \tau_C, loc : \tau_L \rangle \end{aligned}$$

Consider an usage of this query where we don't the need the GEO location information, i.e., let the actual usage type be $\mathscr{Q}(\sigma^*)$, with

$$\begin{aligned} \sigma &= \langle name : string, details : \sigma_d^* \rangle \\ \sigma_d &= \langle team : \tau_T, job : \tau_J, client : \tau_C \rangle \end{aligned}$$

It is possible to observe that by (1) applying ($\pi-$ID), ($\pi-$RECORD) and ($\pi-$CONCAT), then (2) ($\pi-$SELECT) and ($\pi-$ABSTRACTION), and finally (3) (π-GROUP) and (π-DO):

$$
\begin{array}{l}
1)\ \Delta, x : \mathscr{Q}(\sigma_d{}^*), y : \sigma_d;\ \emptyset, y : \sigma_d \vdash \\
\quad y \oplus \langle loc = \mathsf{run}\ \mathsf{db}_{\mathtt{GEO}}(\mathtt{Coords}, y.client.address)\rangle : \tau_d \Rightarrow \\
\quad y \oplus \langle\rangle : \sigma_d \\
2)\ \Delta;\emptyset \vdash addLoc : \mathscr{Q}(\sigma_d^*) \to \mathscr{Q}(\tau_d^*) \Rightarrow addLoc' : \mathscr{Q}(\sigma_d^*) \to \mathscr{Q}(\sigma_d^*) \\
3)\ \Delta;\emptyset \vdash withLoc : \mathscr{Q}(\tau^*) \Rightarrow withLoc' : \mathscr{Q}(\sigma^*)
\end{array}
$$

where

$$
\begin{array}{l}
addLoc' = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \}\ y \oplus \langle\rangle \\
withLoc' = \mathsf{do}\ addLoc'_{\downarrow/details}\{workByTeam\}
\end{array}
$$

Note that *addLoc* is doing nothing. During compilation we can detect and omit patterns like these.

6.2 Phase II: Localization of expression nodes

The second phase of the localization and optimization process is responsible for assigning concrete locations to each of the query expression nodes. We assume a finite set of site locations $\mathscr{L}$, and a lattice $(\mathscr{L} \cup \{\top, \bot\}, \sqsubseteq)$. Location $\top$ represents *in-memory* execution, where all query operations can be evaluated. Location $\bot$ is assigned to expressions that can be computed or transported to any location (e.g. literals, identifiers, etc). We define the usual order relation between $\bot$ and $\top$ and all other locations

$$\forall_{\ell \in \mathscr{L}}.\ \ell \sqsubseteq \top \wedge \bot \sqsubseteq \ell$$

Since at this stage we are ignoring site delegation, all locations other than $\bot$ and $\top$, representing the different sites locations of the system, are kept unrelated. Query delegation between sites are an interesting extension of this work that is out of the scope of this work. We give hints on how details should be worked out to support delegation along the technical parts of the paper, but do not really take them into account in the formal results.

We consider also a set of predefined predicates to specify capabilities of locations. The truth value of the predicates is predetermined and immutable. The selection of predicates used here is inspired on the concrete experience of developing a DSL (OutSystems, 2016) for data manipulation, and is adapted to the set of operations that is included in the language. We say that proposition $\mathtt{can\_group}(\ell)$ holds if the database engine running at location ℓ is able to execute a group-by operation with aggregation of results, as in relational databases. Predicate $\mathtt{can\_nestgroups}(\ell)$ holds for locations (ℓ) running database engines which have support for the nested grouping operations, i.e. return a query together with the details of its groups. This is the case of some NoSQL databases such as MongoDB. Predicate $\mathtt{can\_join}(\ell)$ states that the database repository at location ℓ supports the joining of two (or more) sources given a condition, and $\mathtt{can\_iterate}(\ell)$ indicates that it supports the iteration of a list and the computing of a given expression on all elements of a query. Predicates $\mathtt{can\_lambda}(\ell)$ and $\mathtt{can\_call}(\ell)$ refer to the definition and use of abstractions. As for predicates $\mathtt{can\_createrecords}(\ell)$ refers to handling of record expressions, and $\mathtt{can\_createlists}(\ell)$ refers to handling of list expressions.

Notice that predefined functions can be encoded in native operations (op), each one with a different capability. For instance, SQL databases provide function `NOW()` and MongoDB provides specialized operators such as `$near` to compare GPS coordinates. Common operations must be encoded in a single (abstract) operation and then compiled differently on each source. As an extra example, consider a classic REST interface, yielding a JSON object. None of the above predicates holds since the interface's only capability is to return the data. The extension of this relation to a meta-level, between operations and locations, is out of the scope of this work, and will be pursued in the future.

We define an intermediate format for expressions to support this second transformation phase. We use location labeled expressions $e_\ell \in \mathscr{E}$, where e is an expression given by the syntax of Figure 1, where each subexpression is labeled with a location from $\mathscr{L} \cup \{\top, \bot\}$. We then define a localization system on labeled expressions by means of a rewriting system, as follows:

Definition 6.4 (*Localization system*)
We define the localization system as a rewrite system $(\mathscr{E}, \rightsquigarrow)$, on location labeled expressions $\mathscr{E}$, and a relation $\rightsquigarrow$ defined by the rewriting rules of Figures 15 and 16.

The rewriting rules used above are designed to update the locations assigned to expressions, that indicate where they may or should be evaluated, according to each site capabilities and maximizing the query code discharged to the remote sites – without actually changing the query structure. Operations such as record concatenation and append are encoded in the case the general operation op.

The rewriting system starts in a given initial state defined below, and runs until a fixed point is reached. We prove ahead that this relation is confluent, and hence the localization system always terminates. Literals are labeled with location $\bot$ in the initial state, meaning that they can be computed in any location. The remaining expression nodes are labeled with the $\top$ location in the initial state, which means that all can be evaluated *in-memory*. The rewrite system will converge to assign each expression with a more specific evaluation location. The locations obtained in the final stage of the localization process are then used by the process phase III to produce located code. Data source expressions nodes are explicitly given location ℓ in the expression syntax, even if the whole node containing argument expressions is given location $\top$. The initial labeled expression is therefore $(\mathsf{db}_\ell(t, \overline{e})_\top)$, where $\top$ corresponds to the location where the computation of arguments should be performed, and location ℓ corresponds only to the site where data is located and from where data transmission occurs. In this intermediate form we also introduce a specialized and flexible labeling scheme for expression binders in foreach expressions (following the earlier approach of (Seco *et al.*, 2015)). This includes a partition of binders by location, and an association of boolean conditions — parts of the original condition in the conjunctive form — to each one of the binder partitions. Hence, a foreach expression takes the transient syntactical form

$$\mathsf{foreach}_c \left\{ \overline{y \leftarrow e}, \overline{(x \leftarrow e_\ell, c)_\ell} \right\} e$$

where there is a set of ungrouped binders ($\overline{y}$) that correspond to the binders (still) in the $\top$ location, and a set of partitions containing binders attached to conditions that only refer to

$$e_\ell \ \mathrm{op}_m \ e'_{\ell'} \rightsquigarrow e_\ell \ \mathrm{op}_{m'} \ e'_{\ell'} \quad (\text{when } m' = \ell \sqcap \ell' \wedge \texttt{can\_op}(m')) \qquad (\rightsquigarrow\textsc{Op})$$

$$(\lambda x.e_\ell)_m \rightsquigarrow (\lambda x.e_\ell)_{m'} \quad (\text{when } m' = \ell \wedge \texttt{can\_lambda}(m')) \qquad (\rightsquigarrow\textsc{Lambda})$$

$$(e_\ell \ e'_{\ell'})_m \rightsquigarrow (e_\ell \ e'_{\ell'})_{m'} \quad (\text{when } m' = \ell \sqcap \ell' \wedge \texttt{can\_call}(m')) \qquad (\rightsquigarrow\textsc{Call})$$

$$\langle \overline{a = e_\ell} \rangle_m \rightsquigarrow \langle \overline{a = e_\ell} \rangle_{m'} \quad (\text{when } m' = \sqcap \overline{\ell} \wedge \texttt{can\_createrecords}(m')) \qquad (\rightsquigarrow\textsc{Record})$$

$$(e_\ell.a)_m \rightsquigarrow (e_\ell.a)_{m'} \quad (\text{when } m' = \ell \wedge \texttt{can\_createrecords}(m')) \qquad (\rightsquigarrow\textsc{Field})$$

$$[e_\ell]_m \rightsquigarrow [e_\ell]_{m'} \quad (\text{when } m' = \ell \wedge \texttt{can\_createlists}(\ell)) \qquad (\rightsquigarrow\textsc{Singleton})$$

Fig. 15. Location rewriting rules (I)

the corresponding identifiers ($\overline{x}$). In the general form, there are (possibly empty) partitions for all possible locations in $\mathscr{L}$ (except for $\top$ or $\bot$). This transient form is compiled back to the original representation in the third step of our query transformation process (section 6.3) to:

$$\mathsf{foreach}_{\overline{c\{^{z.x}/_x\}}} \left\{ \overline{y \leftarrow e}, \overline{z \leftarrow \left[\ \mathsf{foreach}_c \left\{ \overline{x \leftarrow [\ e\ ]_\ell} \right\} \ \langle \overline{x = x} \rangle \ \right]_\ell} \right\} \ \overline{e\{^{z.x}/_x\}}$$

thus transforming groups of binders into subqueries, and adjusting both the conditions and the select expression with explicit substitutions. This intermediate form can be easily encoded into the original language, but it technically helps us to prove the confluence of our definition up to equivalence of partitions (binders and conditions).

Formally, the initial labeling of locations to expressions is defined as follows

Definition 6.5 (*Initial Expression Labeling*)

1. Literals are labeled with location $\bot$.
2. All other expressions are labeled with location $\top$.
3. Binders in foreach expressions are initially ungrouped, and groups of binders for all possible locations in $\mathscr{L}$ are given an empty set of binders and condition $\texttt{true}_\bot$.

Given the initial state for location labeled expressions, we introduce the rewriting rules in Figures 15 and 16. Most rules are of the form $e_m \rightsquigarrow e_{m'}$ where the final location m' is the least upper bound of all locations in the subexpressions of e, restricted with extra conditions on the capabilities of the target site. This evolution is the basis for our confluence and termination results. Notice for instance in rule ($\rightsquigarrow$Op) that if the chosen site m', where the operands can both be computed, has the capability to perform a certain operation, then the whole expression gets localized there. The same happens in all the rules in Figure 15. Notice that the old location is ignored on all steps of the rewriting process.

$$(\mathsf{db}_\ell(t,\overline{e_\ell}))_m \rightsquigarrow (\mathsf{db}_\ell(t,\overline{e_\ell}))_{m'} \quad (\text{when } m' = \sqcap\overline{\ell}\sqcap\ell) \qquad (\rightsquigarrow\textsc{Source})$$

$${}^{\tau}\pi^{\sigma}_{m}(e_\ell) \rightsquigarrow {}^{\tau}\pi^{\sigma}_{\ell}(e_\ell) \quad (\text{when } \texttt{can\_project}(\ell)) \qquad (\rightsquigarrow\textsc{Project})$$

$$(\mathsf{return}\ e_\ell)_m \rightsquigarrow (\mathsf{return}\ e_\ell)_\ell \qquad (\rightsquigarrow\textsc{Return})$$

$$\mathsf{foreach}_{c_\ell\wedge d}\left\{\ \overline{x\leftarrow e}, \overline{(z\leftarrow e', c'')_{\ell''}}\ \right\}\ e \rightsquigarrow \mathsf{foreach}_{d}\left\{\ \overline{x\leftarrow e}, (\overline{y\leftarrow e}, c'\wedge c_\ell)_{\ell'}, \overline{(z\leftarrow e', c'')_{\ell''}}\ \right\}\ e$$
$$(\text{when } \ell\sqsubseteq\ell' \wedge FV(c)\subseteq\{\overline{y}\} \wedge \texttt{can\_iterate}(\ell')) \qquad (\rightsquigarrow\textsc{Filter})$$

$$\mathsf{foreach}_{d}\left\{\ w\leftarrow e_{\ell'}, \overline{x\leftarrow e}, (\overline{y\leftarrow e}, c')_{\ell'}, \overline{(z\leftarrow e', c'')_{\ell''}}\ \right\}\ e \rightsquigarrow$$
$$\mathsf{foreach}_{d\{{}^{w_{\ell'}}/_{w_\top}\}}\left\{\ \overline{x\leftarrow e}, (w\leftarrow e_{\ell'}, \overline{y\leftarrow e}, c')_{\ell'}, \overline{(z\leftarrow e', c'')_{\ell''}}\ \right\}\ e\{{}^{w_{\ell'}}/_{w_\top}\}$$
$$(\text{when } \texttt{can\_iterate}(\ell') \wedge (\texttt{can\_join}(\ell')\vee|\{\overline{y},w\}|=1)) \qquad (\rightsquigarrow\textsc{Binder})$$

$$(\mathsf{foreach}_{\mathsf{true}}\ \{\ (\overline{x\leftarrow e_\ell}, c_{\ell'})_{\ell''}\ \}\ e_\ell)_m \rightsquigarrow (\mathsf{foreach}_{\mathsf{true}}\ \{\ (\overline{x\leftarrow e_\ell}, c_{\ell'})_{\ell''}\ \}\ e_\ell)_{m'}$$
$$(\text{when } m' = \ell\sqcap\ell'\sqcap\ell'' \wedge \texttt{can\_iterate}(m')\wedge(\texttt{can\_join}(m')\vee|\overline{x}|=1)) \qquad (\rightsquigarrow\textsc{Select})$$

$$(\mathsf{groupby}_b^{\overline{a=e_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_m \rightsquigarrow (\mathsf{groupby}_b^{\overline{a=e\{{}^{x_{\ell'}}/_{x_\top}\}_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_m \qquad (\rightsquigarrow\textsc{Group-Cursor})$$

$$(\mathsf{groupby}_b^{\overline{a=e_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_m \rightsquigarrow (\mathsf{groupby}_b^{\overline{a=e_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_{m'}$$
$$(\text{when } m' = \sqcap\overline{\ell}\sqcap\ell' \wedge \texttt{can\_nestgroups}(m')) \qquad (\rightsquigarrow\textsc{Nesting})$$

$${}^{\langle\overline{a:\tau}\rangle}\pi^{\langle\overline{a:\tau},\overline{b:\tau'}\rangle}((\mathsf{groupby}_b^{\overline{a=e_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_m) \rightsquigarrow {}^{\langle\overline{a:\tau}\rangle}\pi^{\langle\overline{a:\tau},\overline{b:\tau'}\rangle}((\mathsf{groupby}_b^{\overline{a=e_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_{m'})$$
$$(\text{when } m' = \sqcap\overline{\ell}\sqcap\ell' \wedge \texttt{can\_group}(m')) \qquad (\rightsquigarrow\textsc{Group})$$

Fig. 16. Location rewriting rules (II)

In the case of query expressions, Figure 16, we have that the location of a data source expression $((\mathsf{db}_\ell(t,\overline{e_\ell}))_m)$ is given by the least upper bound of the data source location and its arguments $\sqcap\overline{\ell}\sqcap\ell$, by rule ($\rightsquigarrow$SOURCE). This kind of rewriting rules propagate the locations of data sources from the expression leafs, up the abstract syntax tree, according to the capabilities of remote sites. Rules ($\rightsquigarrow$PROJECT), ($\rightsquigarrow$SINGLETON) and ($\rightsquigarrow$RETURN) are particular cases of the pattern used in Figure 15, propagating the location of the single inner expression directly to the top level expression.

Special treatment is given to binders in foreach expressions, which are grouped according to the locations of the corresponding inner queries, based on the intermediate representation described above. Rule ($\rightsquigarrow$BINDER) encodes the actual grouping of binders, while Rule ($\rightsquigarrow$FILTER) distributes filter conditions in the foreach expression according to the locations of the bound cursors and their usage. A fully localized binder list is finally captured by rule ($\rightsquigarrow$SELECT) provided that the appointed remote site has the capability to iterate and join data sources, and that the condition and select expression can also be evaluated there. The rules for group-by operations cover two possible cases regarding nested data. If nested data is required, rule ($\rightsquigarrow$NESTING) can be applied only when the location has the capability of processing nested data, hence discharging the whole operation of the remote site. In the case of nested data being discarded by an explicit projection, rule ($\rightsquigarrow$GROUP) can be applied on sites where the grouping operation does not provide nested buckets of detail rows, but are able to group data anyway. This effect is provided by the syntactic pattern on rule ($\rightsquigarrow$GROUP) detecting the pattern containing a projection. Note also that identifiers are initially located at $\top$, the expressions where they are used are also, by definition, located at $\top$. Rules ($\rightsquigarrow$GROUP-CURSOR) and ($\rightsquigarrow$BINDER) localize the usages of specific cursors (from a groupby or foreach, respectively) to the location of the corresponding source, enabling the specific localization of the expressions using its identifier.

In order to state the soundness property of the localization system we need a few auxiliary results, that help establishing the system's invariant. Consider the following definitions on locations of expressions,

Definition 6.6 (*Locations of Strict Sub-Expressions*)
For any expression e, $L(e)$ denotes all the locations in strict sub-expressions of e. The particular case of $L(\mathsf{db}_\ell(t,\overline{e_\ell}))$ also includes the location ℓ.

This definition is used to define the systems' invariant, that is that the location of each expression is "higher" in our lattice than the locations of all subexpressions.

Definition 6.7 (*Minimal Distribution*)
A labeled expression e_m is *minimally distributed*, if all its strict sub-expressions are minimally distributed and $m \sqsubset \top$ implies that $m = \sqcap L(e)$.

The notion of minimal distribution is at the core of the rewriting system's invariant. Each defined rewriting step preserves that property as stated in Lemma 6.10. In particular, the initial state of a location labeled expression is *minimally distributed*, according to Definition 6.7, as stated in Lemma 6.8.

Lemma 6.8 (*Minimally Distributed Initial Labeling*)
Initially labeled expressions are minimally distributed.

Proof. The lemma is proven by induction on the expression structure and analysis of the two cases in Definition 6.5. Notice that the terminal cases in the expression syntax, except the identifiers, are labeled with $\bot$ in the initial state, and all other expressions are labeled with $\top$. In the case of $e_\top$ we have a false hypothesis in the definition's implication, and hence

the invariant trivially holds. In the case of $e_\perp$ we have an empty set of strict subexpressions, and the join of an empty set is $\perp$, as required. □

The soundness of our localization system is established by a preservation property of the system's invariant, the minimal distribution (Definition 6.7), throughout the rewriting process (Lemma 6.10 below).

Lemma 6.9 (*Fixed Distribution*)
If e_m is minimally distributed, $m \sqsubset \top$ and $e_m \rightsquigarrow {e'}_{m'}$, then $m' = m$.

Proof. Proven by case analysis of the possible rewriting reductions, leveraging the fact that rules that change m don't change e, and are precisely those that have $m' = \sqcap L(e')$. □

Lemma 6.10 (*Preservation of Minimal Distribution*)
If e_m is minimally distributed and $e_m \rightsquigarrow {e'}_{m'}$, then ${e'}_{m'}$ is minimally distributed.

Proof. Proven by induction on the expression structure and case analysis of the possible rewriting reductions. Rule ($\rightsquigarrow$FILTER) doesn't change m and only moves its strict sub-expressions. Rules ($\rightsquigarrow$BINDER) and ($\rightsquigarrow$GROUP-CURSOR) are similar, but change the location of some sub-expressions to a necessarily lower one. In reductions where sub-expressions are changed, such as ($\rightsquigarrow$GROUP), we proceed by case analysis of m, and Lemma 6.9. In the remaining cases, minimal distribution is given directly by definition, as $m' = \sqcap L(e)$. □

Another important property of the rewriting system is confluence, stated in Theorem 6.16 below. Consider first some auxiliary results, among which: the termination of the system (Lemma 6.14) and the local confluence property (Lemma 6.15) of the rewriting relation when restricted to our invariant.

Corollary 6.11 (*Transitive Preservation of Minimal Distribution*)
If expression e_m is minimally distributed and $e_m \rightsquigarrow^* {e'}_{m'}$, then ${e'}_{m'}$ is minimally distributed.

Proof. Trivially by repeated application of Lemma 6.10. □

Lemma 6.12 (*Well-Locatedness*)
If e_ℓ is minimally distributed, then $\sqcap L(e) \sqsubseteq \ell$.

Proof. Follows directly by case analysis of ℓ and Definition 6.7. □

Lemma 6.13 (*Location Monotonicity*)
If expression e_m is minimally-distributed and $e_m \rightsquigarrow {e'}_{m'}$, then $m' \sqsubseteq m$.

Proof. Follows directly by case analysis of m and Lemma 6.9. □

Lemma 6.14 (*Termination*)
The localization relation $\rightsquigarrow$ is terminating for minimally distributed expressions.

Proof. Consider as induction measure the lexicographic order of the all linearized locations in an expression, the number of ungrouped binders in a foreach expression and its number of ungrouped conditions.

By Lemma 6.10, we know that for any minimally distributed expression e_m, the derivation $e_m \rightsquigarrow {e'}_{m'}$ leads to another minimally distributed expression. By Lemma 6.12, we

know then that $\sqcap L(e') \sqsubseteq m'$ and by Lemma 6.13 we know that $m' \sqsubseteq m$. When applied, rules ($\rightsquigarrow$OP) to ($\rightsquigarrow$RETURN), ($\rightsquigarrow$SELECT), ($\rightsquigarrow$NESTING), and ($\rightsquigarrow$GROUP) have $\sqcap L(e) \sqsubset m'$ (or they are not applied at all). In the normal form $\sqcap L(e) = m'$ or the site does not have the needed capabilities. In the case of rules ($\rightsquigarrow$BINDER) and ($\rightsquigarrow$FILTER), the distance measure stays the same but the number of ungrouped binders or ungrouped conditions in the foreach expression decreases, respectively. □

Lemma 6.15 (*Local Confluence*)
Relation $\rightsquigarrow$ is locally confluent for minimally distributed expressions: for any minimally distributed expressions e_ℓ, $e_{1\ell_1}$ and $e_{2\ell_2}$ such that $e_\ell \rightsquigarrow e_{1\ell_1}$ and $e_\ell \rightsquigarrow e_{2\ell_2}$, there exists an expression $e'_{\ell'}$ such that $e_{1\ell_1} \rightsquigarrow^* e'_{\ell'}$ and $e_{2\ell_2} \rightsquigarrow^* e'_{\ell'}$

Proof. By case analysis of the initial expression e_ℓ and the possible $e_{1\ell_1}$ and $e_{2\ell_2}$ pairs.

We have to consider the particular cases of the (few) overlapping rewriting rules, but the remaining are proven using the same proof strategy. Overlapping rewritings by rules ($\rightsquigarrow$GROUP), ($\rightsquigarrow$GROUP-CURSOR) and ($\rightsquigarrow$NESTING) don't change the expression structure, and localize the groupby expression at the exact same location. Contrary to ($\rightsquigarrow$SELECT), Rules ($\rightsquigarrow$BINDER) and ($\rightsquigarrow$FILTER) both rewrite foreach expressions with ungrouped binders, but they commute as they move different sub-expressions (binders and conditions, respectively) to binder groups which are guaranteed to be unique per location.

Cases where $e_{1\ell_1}$ and $e_{2\ell_2}$ are obtained by reduction of independent sub-expressions commute, so its just a matter of rewriting the other sub-expression accordingly.

The remaining cases are simply proofs that reduction of the top-level expression commutes with the reduction of one of the sub-expressions in zero or more steps. Since all rewriting rules that change ℓ don't change e, and forget it in favor of $\sqcap L(e)$, re-applying the top-level reduction after the sub-expression reduction ensures both converge. If the rewriting requires a specific capability, it is ensured by leveraging Lemma 6.9. For the full proof, check appendix B. □

Theorem 6.16 (*Confluence*)
Relation $\rightsquigarrow$ is confluent for minimally distributed expressions.

Proof. By Lemma 6.10, we know that minimal distribution is preserved through reductions of $\rightsquigarrow$. Since $\rightsquigarrow$ is both locally confluent (Lemma 6.15) and terminating (Lemma 6.14), by Newman's Lemma (Newman, 1942) we can say that $\rightsquigarrow$ is confluent for minimally distributed expressions. □

We now illustrate the rewriting process using our running example.

Example Recall the query *work* from Section 3

$$\mathsf{foreach}_{t.id=j.teamId \,\wedge\, j.clientId=c.id \,\wedge\, j.date=8/5} \left\{ \begin{array}{l} t \leftarrow \mathtt{db}_{\mathtt{SALESDB}}(\mathtt{Team}), \\ j \leftarrow \mathtt{db}_{\mathtt{SALESDB}}(\mathtt{Job}), \\ c \leftarrow \mathtt{db}_{\mathtt{SAP}}(\mathtt{Client}) \end{array} \right\}$$
$$\langle team = t,\ job = j,\ client = c \rangle$$

The localization of this query would begin by labeling it according to Definition 6.5.

$$\left(\mathsf{foreach}_{filter}\left\{\begin{array}{l} t \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Team})_{\top}, \\ j \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Job})_{\top}, \\ c \leftarrow \mathsf{db}_{\mathtt{SAP}}(\mathtt{Client})_{\top} \end{array}\right\} select\right)_{\top}$$

$$\begin{aligned} \text{where } filter = {} & (t_{\top}.id_{\top} = j_{\top}.teamId_{\top})_{\top} \\ & \wedge (j_{\top}.clientId_{\top} = c_{\top}.id_{\top})_{\top} \\ & \wedge (j_{\top}.date_{\top} = 8/5_{\bot})_{\top} \\ select = {} & \langle team = t_{\top},\ job = j_{\top},\ client = c_{\top}\rangle_{\top} \end{aligned}$$

Since the initial location of all identifiers is $\top$, repeated rewriting of both the foreach's filter and its select expression would keep everything localized at $\top$. Rewriting by applying rule ($\rightsquigarrow$SOURCE), however, would locate all the db expressions in their respective locations.

$$\mathsf{foreach}_{filter}\left\{\begin{array}{l} t \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Team})_{\mathtt{SALESDB}}, \\ j \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Job})_{\mathtt{SALESDB}}, \\ c \leftarrow \mathsf{db}_{\mathtt{SAP}}(\mathtt{Client})_{\mathtt{SAP}} \end{array}\right\} select$$

The localization process would then proceed by rewriting with the rule ($\rightsquigarrow$BINDER) repeatedly, which would group foreach's binders by location, and appropriately locate the usages of their cursors, t, j and c, in the $filter$ and $select$ expressions.

$$\mathsf{foreach}_{filter}\left\{\begin{array}{l} \left(\begin{array}{l} t \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Team})_{\mathtt{SALESDB}}, \\ j \leftarrow \mathsf{db}_{\mathtt{SALESDB}}(\mathtt{Job})_{\mathtt{SALESDB}}, \\ \mathsf{true}_{\bot} \end{array}\right)_{\mathtt{SALESDB}} \\ \left(\begin{array}{l} c \leftarrow \mathsf{db}_{\mathtt{SAP}}(\mathtt{Client})_{\mathtt{SAP}}, \\ \mathsf{true}_{\bot} \end{array}\right)_{\mathtt{SAP}} \end{array}\right\} select$$

$$\begin{aligned} \text{where } filter = {} & (t_{\mathtt{SALESDB}}.id_{\top} = j_{\mathtt{SALESDB}}.teamId_{\top})_{\top} \\ & \wedge (j_{\mathtt{SALESDB}}.clientId_{\top} = c_{\mathtt{SAP}}.id_{\top})_{\top} \\ & \wedge (j_{\mathtt{SALESDB}}.date_{\top} = 8/5_{\bot})_{\top} \\ select = {} & \langle team = t_{\mathtt{SALESDB}},\ job = j_{\mathtt{SALESDB}},\ client = c_{\mathtt{SAP}}\rangle_{\top} \end{aligned}$$

At this point, any rewritings of $select$ expression by rule ($\rightsquigarrow$RECORD) will maintain location $\top$, as the *team* and *job* fields are located in SALESDB, while the *client* field is located in SAP. However, repeated rewriting of the $filter$ expression using rules ($\rightsquigarrow$FIELD) and ($\rightsquigarrow$OP) would result in

$$\begin{aligned} & (t.id_{\mathtt{SALESDB}} = j.teamId_{\mathtt{SALESDB}})_{\mathtt{SALESDB}} \\ & \wedge (j.clientId_{\mathtt{SALESDB}} = c.id_{\mathtt{SAP}})_{\top} \\ & \wedge (j.date_{\mathtt{SALESDB}} = 8/5_{\bot})_{\mathtt{SALESDB}} \end{aligned}$$

which would allow for the localization of the first and third comparisons to the SALESDB binder group, using rule (⇝FILTER).

$$\mathsf{foreach}_{filter}\left\{\begin{array}{l}\left(\begin{array}{l}t \leftarrow \mathtt{db}_{\mathtt{SALESDB}}(\mathtt{Team})_{\mathtt{SALESDB}},\\ j \leftarrow \mathtt{db}_{\mathtt{SALESDB}}(\mathtt{Job})_{\mathtt{SALESDB}},\\ (t.id_{\mathtt{SALESDB}} = j.teamId_{\mathtt{SALESDB}})_{\mathtt{SALESDB}}\\ \wedge\ (j.date_{\mathtt{SALESDB}} = 8/5_{\perp})_{\mathtt{SALESDB}}\end{array}\right)_{\mathtt{SALESDB}}\\ \\ \left(\begin{array}{l}c \leftarrow \mathtt{db}_{\mathtt{SAP}}(\mathtt{Client})_{\mathtt{SAP}},\\ \mathsf{true}_{\perp}\end{array}\right)_{\mathtt{SAP}}\end{array}\right\} select$$

where $filter = (j_{\mathtt{SALESDB}}.clientId_{\mathtt{SALESDB}} = c_{\mathtt{SAP}}.id_{\mathtt{SAP}})_{\top}$

$$select = \langle team = t_{\mathtt{SALESDB}},\ job = j_{\mathtt{SALESDB}},\ client = c_{\mathtt{SAP}}\rangle_{\top}$$

At this point, no other rewriting would change the labeled query, so we reached the normal form of the query expression. If we were localizing the full *withLoc* query introduced in Section 3, we'd reach the exact same result for the foreach expression, but would also be able to localize the Coords web-service call at location GEO, as expected.

6.3 Phase III: Finalizing

The third, and final phase of the process consists in transforming the labeled expression format back to the regular syntax, while ensuring that all operations are indeed evaluated at the most appropriate location.

The labeled (query) expressions (r) are transformed, via function $(\!|\, r \,|\!)$, defined in Figures 17 and 18, in such a way that remote execution expressions, of the form $[\, e \,]_{\ell}$, are explicitly placed when crossing the border of a location, and so that binder groups in foreach queries are rewritten as full-fledged remote inner queries (recall the intermediate format description in section 6.2). We inductively define this function by case analysis and ensure all transitions to a different location (other than $\perp$) are enclosed by a remote execution expression, and that a transition from a location ℓ to $\perp$ is ignored, as expressions labeled at $\perp$ can be evaluated anywhere. This general policy is captured by the auxiliary function $e \uparrow_{\ell}^{\ell'}$, and based on the invariant that $\ell \sqsubseteq \ell'$, established by the rewriting system (Lemma 6.10).

To accommodate the transformation process, we extend the operational semantics so that all three phases described here are called in sequence. We need to annotate the evaluation functions with a location $\langle\!\langle e \rangle\!\rangle_{\ell}$, $[\![e]\!]_{\ell}$, and ensure that query results are produced using the properly transformed version of the query (see Figure 9). The initial location of every evaluation is $\top$. The cases where the annotation is manipulated in any significant way are the cases of remote invocation $[\, e \,]_{\ell}$, that switches the current execution location according to the expression's annotation,

$$[\![[\, e \,]_{\ell}]\!]_{\ell'} = [\![e]\!]_{\ell}$$

$$
\begin{array}{rcl}
(\!|\, num_{\perp} \,|\!)_{\ell} & \triangleq & num \\
(\!|\, bool_{\perp} \,|\!)_{\ell} & \triangleq & bool \\
(\!|\, string_{\perp} \,|\!)_{\ell} & \triangleq & string \\
(\!|\, date_{\perp} \,|\!)_{\ell} & \triangleq & date \\
(\!|\, \emptyset_{\perp} \,|\!)_{\ell} & \triangleq & \emptyset \\
(\!|\, (e_{\ell'} \text{ op } e'_{\ell''})_m \,|\!)_{\ell} & \triangleq & ((\!|\, e_{\ell'} \,|\!)_m \text{ op } (\!|\, e'_{\ell''} \,|\!)_m) \uparrow^{\ell}_{m} \\
(\!|\, x_m \,|\!)_{\ell} & \triangleq & x \uparrow^{\ell}_{m} \\
(\!|\, (\lambda x{:}\tau.e_{\ell'})_m \,|\!)_{\ell} & \triangleq & (\lambda x{:}\tau.(\!|\, e_{\ell'} \,|\!)_m) \uparrow^{\ell}_{m} \\
(\!|\, (e_{\ell'}\ e'_{\ell''})_m \,|\!)_{\ell} & \triangleq & ((\!|\, e_{\ell'} \,|\!)_m\ (\!|\, e'_{\ell''} \,|\!)_m) \uparrow^{\ell}_{m} \\
(\!|\, \langle \overline{a = e_{\ell'}} \rangle_m \,|\!)_{\ell} & \triangleq & \langle \overline{a = (\!|\, e_{\ell'} \,|\!)_m} \rangle \uparrow^{\ell}_{m} \\
(\!|\, (e_{\ell'}.a)_m \,|\!)_{\ell} & \triangleq & (\!|\, e_{\ell'} \,|\!)_m.a \uparrow^{\ell}_{m} \\
(\!|\, [e_{\ell'}]_m \,|\!)_{\ell} & \triangleq & [(\!|\, e_{\ell'} \,|\!)_m] \uparrow^{\ell}_{m} \\
(\!|\, (e_{\ell} \uplus e_{\ell'})_m \,|\!)_{\ell} & \triangleq & (\!|\, e_{\ell'} \,|\!)_m \uplus (\!|\, e'_{\ell'} \,|\!)_m \uparrow^{\ell}_{m} \\
(\!|\, (\mathsf{db}_{\ell'}(t, \overline{e_{\ell''}}))_m \,|\!)_{\ell} & \triangleq & \mathsf{db}_{\ell'}(t, \overline{(\!|\, e_{\ell''} \,|\!)_m}) \uparrow^{\ell}_{m} \\
(\!|\, {}^{\tau}\pi^{\sigma}_{m}(e_{\ell'}) \,|\!)_{\ell} & \triangleq & {}^{\tau}\pi^{\sigma}((\!|\, e_{\ell'} \,|\!)_m) \uparrow^{\ell}_{m} \\
(\!|\, (\mathsf{return}\ e_{\ell'})_m \,|\!)_{\ell} & \triangleq & (\mathsf{return}\ (\!|\, e_{\ell'} \,|\!)_m) \uparrow^{\ell}_{m} \\
(\!|\, (\mathsf{foreach}_c \left\{ \overline{x \leftarrow e}, \overline{(y \leftarrow e'_{\ell'}, c'')_{\ell''}} \right\} e)_m \,|\!)_{\ell} & \triangleq & \\
\left(\mathsf{foreach}_{(\!|\, c \,|\!)_m} \left\{ \overline{x \leftarrow (\!|\, e \,|\!)_m}, z \leftarrow (\!|\, (\mathsf{foreach}_{c''} \left\{ \overline{y \leftarrow e'_{\ell'}} \right\} \langle \overline{y = y} \rangle)_{\ell''} \,|\!)_m \right\} (\!|\, e\{\overline{z.y}/\overline{y}\} \,|\!)_m \right) \uparrow^{\ell}_{m} & & \\
(\!|\, (\mathsf{groupby}^{\overline{a=e_{\ell}}}_{b}\{\, x \leftarrow e_{\ell'} \,\})_m \,|\!)_{\ell} & \triangleq & \mathsf{groupby}^{\overline{a=(\!|\, e_{\ell} \,|\!)_m}}_{b}\{\, x \leftarrow (\!|\, e_{\ell'} \,|\!)_m \,\} \uparrow^{\ell}_{m} \\
(\!|\, ([\, e \,]_{\ell'})_m \,|\!)_{\ell} & \triangleq & [\, (\!|\, e \,|\!)_{\ell'} \,]_{\ell'} \uparrow^{\ell}_{m} \\
(\!|\, \mathsf{do}\ (e_{\ell})_{\downarrow p}\{e_{\ell'}\}_m \,|\!)_{\ell} & \triangleq & \mathsf{do}\ ((\!|\, e_{\ell} \,|\!)_m)_{\downarrow p}\{(\!|\, e_{\ell'} \,|\!)_m\} \uparrow^{\ell}_{m} \\
(\!|\, (\mathsf{exec}\, x : \tau = e\ \mathsf{in}\ e')_m \,|\!)_{\ell} & \triangleq & (\mathsf{exec}\, x : \tau = (\!|\, e \,|\!)_m\ \mathsf{in}\ (\!|\, e' \,|\!)_m) \uparrow^{\ell}_{m}
\end{array}
$$

Fig. 17. Location labeling erasure function.

$$
\begin{array}{rcll}
e \uparrow^{\ell'}_{\ell} & \triangleq & e & \text{with } \ell = \ell' \text{ or } \ell = \perp \\
e \uparrow^{\ell'}_{\ell} & \triangleq & [\, e \,]_{\ell} & \text{otherwise (i.e. } \ell \neq \perp \text{ and } \ell \sqsubset \ell')
\end{array}
$$

Fig. 18. Location labeling placing function.

and the case of the $\mathsf{exec}\, x : \sigma = e\ \mathsf{in}\ e'$ expression that uses the current location annotation to transform the inner expression.

$$
\langle\!\langle \mathsf{exec}\, x : \sigma = e\ \mathsf{in}\ e' \rangle\!\rangle_{\ell} = \langle\!\langle e'\{^{v}/_{x}\} \rangle\!\rangle_{\ell} \quad \text{where } r = \langle\!\langle e \rangle\!\rangle_{\ell}
$$

$$
\begin{array}{l}
\emptyset; \Gamma \vdash r : \tau \Rightarrow r' : \sigma \\
r' \rightsquigarrow^{*} r'' \\
r''' = (\!|\, r'' \,|\!)_{\ell} \\
v = [\![r''']\!]_{\ell}
\end{array}
$$

We have presented results that support the soundness of the two first steps (Lemma 6.2 and Lemma 6.10). Phase III maintains the whole structure of the query, which does not

```
class DBData { public string Name; }

return ExecuteQuery<DBData>(
  @"SELECT Team.Name FROM Team
  INNER JOIN Job ON Team.Id = Job.TeamId
  WHERE Job.Date = '8/5' GROUP BY Team.Name");
```

Fig. 19. Code for Figure 7, using only top level data.

raise any soundness issue. To ensure the whole process is sound, it would be interesting to ensure that operations are only executed in sites with the right capabilities. We leave that as future work.

The application of this transformation process ends with the compilation of the query and corresponding generation of native query languages for each database system running in the remote locations. Code located at location $\top$ is translated into a general purpose language (Java, C♯, or Javascript), while others include languages like SQL, LINQ, or Javascript using MongoDB API and operators. Notice that we are not introducing any kind of centralized middleware system that interprets a general query language, serving the results to a client. Instead, we devise a method that allows true data mashups, with the adaptation and discharging of query fragments from the client device or system, to the native remote database systems, and providing glue code as necessary.

Example When compiling a query we take advantage of the way its results are being used. As an example, consider two possible usages of the *withLoc* query from Figure 7. First, consider that we only want to display the name of the teams that have any work to do. We thus compile the query using $\tau = \langle name : String \rangle^*$ as the target usage type. In this process we can safely ignore the `Client` table and the calls to the `Coords` service, resulting in the (abbreviated) query:

$$\left[\begin{array}{l} \mathsf{groupby}^{name=x.team.name}_{details} \{\, x \leftarrow \\ \quad \mathsf{foreach}_{t.id=j.teamId \wedge j.date=8/5} \left\{ \begin{array}{l} t \leftarrow teams, \\ j \leftarrow jobs \end{array} \right\} \langle team = e, job = j \rangle \\ \} \end{array} \right]_{\texttt{SALESDB}}$$

Notice that the group-by operation is compiled and localized in the `SALESDB` database, since the usage does not refer to group details nor the `Client` table, which resides in a different database. Notice also that the *addLoc* term is projected, as in the example in section 6.1, to

$$addLoc' = \lambda x.\mathsf{foreach}\, \{\, y \leftarrow x \,\}\; y \oplus \langle\rangle$$

which in practice is doing nothing. During the code generation phase we detect and remove patterns like these. For the sake of simplicity, the abbreviated query above is shown using this optimization. This query can then be used to produce the C# code shown in Figure 19. If we instead compile it with relation to type

$$\begin{aligned}\tau = \langle & \mathit{name} : \mathsf{string}, \\ & \mathit{details} : \langle \mathit{job} : \langle \mathit{title} : \mathsf{string} \rangle, \\ & \qquad \mathit{client} : \langle \mathit{name} : \mathsf{string} \rangle, \\ & \qquad \mathit{loc} : \langle \mathit{lat} : \mathsf{num}, \mathit{lng} : \mathsf{num} \rangle^* \rangle^* \rangle\end{aligned}$$

then we no longer can omit the call to the `GEO` service. Furthermore, the group-by operation needs to be performed in memory.

The resulting code is shown in Figure 20. Although neither the `Job.ClientId` nor `Client.Id` fields are present in the usage type, we need to fetch them because they're used by the in-memory join operation. Similarly, we need to fetch the client's address because it is needed for the `Coords` service. For simplicity reasons we have omitted the code required to remove these fields from the result.

We leave further query optimization for future work. Looking at the code in Figure 20, it is obvious that fetching all clients is not a very efficient approach. We can take advantage of the results returned by the query in the `SALESDB` database to restrict the data to be fetched from from the `SAP` database, e.g. by introducing an "IN" condition.

7 Related Work

Unlike many DSLs for the development of complete data-centric applications (Fu *et al.*, 2013; Cooper *et al.*, 2007), we focus on the problem of typeful integration of data sources, as in (Lindley & Cheney, 2012), but dealing with the particular aspect of distributing and optimizing queries, by means of code specialization, given a particular usage. Our DSL is similar to (Cheney *et al.*, 2013), in the sense that both allow for the composition of staged values (queries, in our case), and for the separate compilation and execution of queries. Instead of focusing on SQL, however, our proposal targets a flexible nesting base model (as (Buneman *et al.*, 1995)), that fits several variants of data repositories, from relational databases, to NoSQL document based repositories, to parameterizable web services. Additionally, we naturally deal with raw nested data (Colby, 1989), by means of our in-place modification operation.

Our work is related to the composition of higher order queries, and higher order manipulation of XML data (Robie *et al.* , 2014; Benzaken *et al.*, 2003). We use the uniform and compositional mechanism of in-place modifications, that applies to all kinds of repositories, and is suitable to query simplification, and compilation to efficient imperative query languages that manipulate data by reference (e.g. IndexedDB).

Native capabilities of data repositories have been used to generate queries. In the case of (Vassalos & Papakonstantinou, 2000), capabilities are captured using description logics, to solve the problem of answering queries by combining existing repositories. Our goal is different, as we limit the capabilities to the language operations, and avoid using the semantics of the schema. We use capabilities to produce a sound distribution of general purpose operations across remote sites, and then use dedicated compilation strategies. (Vassalos & Papakonstantinou, 2000) define a general query generation strategy, without seeking the optimization of code and sites visited based on usage. Our approach falls into the category of light-weight compiler and code specialization procedures, and not in the category of semantic based code generation tools.

```
class Client {
  public int Id;
  public string Name;
  public string Address;
}

class Team {
  public string Name;
}

class Job {
  public string Title;
  public int ClientId;
}

class SalesDBData {
  public Team Team;
  public Job Job;
}

class SAPData {
  public Client Client;
}
```

```
class JoinData {
  public Team Team;
  public Job Job;
  public Client Client;
}

class Location {
  public float Lat;
  public float Lng;
}

class Detail {
  public Client Client;
  public Job Job;
  public Location Loc;
}

class QueryData {
  public IEnumerable<Detail>
      Details;
  public string Name;
}
```

```
var salesDBData = ExecuteQuery<SalesDBData>(
  @"SELECT Team.Name, Job.Title, Job.ClientId FROM Team
  INNER JOIN Job ON Team.Id = Job.TeamId
  WHERE Job.Date = '8/5'");
var sapData = ExecuteQuery<SAPData>(
  @"SELECT Client.Id, Client.Name, Client.Address FROM CLIENT");
var joinData = salesDBData.Join(sapData,
   x => x.ClientId,
   y => y.Id,
   (x, y) => new JoinData {
      Team = x.Team,
      Job = x.Job,
      Client = y.Client
   });
return joinData.GroupBy(
    elem => new { Name = elem.Team.Name },
    elem => elem,
    (key, elems) => new QueryData() {
      Name = key.Name,
      Details = elems.Select(a => new Detail() {
        Client = a.Client,
        Job = a.Job,
        Loc = GEO.Coords(a.Client.Address)
      })
   });
```

Fig. 20. Generated code for query in Figure 7, using job's title, the client's name and coordinates.

Related work includes systems that integrate, behind a single interface, several data based systems (e.g. (Halevy *et al.*, 2006)). We address a simpler, and yet relevant scenario, that is how to integrate data sources via a query compiler for applications, that typically are already capable of orchestrating several data sources. This approach lets the developer seamlessly access and combine data of different natures and sources, without worrying about the efficiency of the orchestration and distribution of the final query, nor knowing the specific (native) query languages involved. Query systems like Kleisli (Wong, 2000), already pursue these goals of abstraction in a direction very similar to ours, providing an high-level query language for nested collections and optimized distribution of queries to external data sources. Nonetheless, the developer still needs to be aware of the specific query languages being integrated. Our work provides a single query language over the participating data sources, whose capabilities are described in a principled way, allowing for the uniform treatment of sub-queries and sub-expressions across the whole query.

8 Final Remarks

Querying data in heterogeneous and distributed environments is arguably a skillful task, when building all kinds of service and data-centric applications. We believe that is crucial to introduce new data manipulation languages for nested collections that allow the orchestration and abstraction of a number heterogeneous data sources remotely, and served by different kinds of database systems, yielding different capability sets. In this work, we leverage on the abstraction of such capabilities, and on a type based compilation and optimization algorithm, to attain such an objective. Our core goal is the production of specialized code for each specific data usage and each kind of database engine. We eagerly project and aggregate operations as close to the data sources as possible, and fall back to in-memory data processing when needed. Our approach differs from existing middleware systems that provide heterogeneous and distributed queries: we provide a compile time strategy of code specialization, orchestrated by the client application, which is not feasible in a general purpose situation. Our approach seeks the elimination of (useless) code, and the corresponding (expensive) remote invocations, associated with an optimized query execution plan, obtained by discharging as much work as possible to external systems.

We extend and generalize the initial work presented in (Seco *et al.*, 2015), by modularizing the optimization procedure in such a way that the non-trivial task of distributing and gluing query parts, is achievable by a series of simple code transformations. Other extensions to the optimization process such as the ones listed below, and to the language itself (i.e. new query operations) should fall into the same architecture seamlessly. Moreover, we added the formal methodology to prove the soundness of the whole process.

Two immediate extensions that arise from this work are the delegation between sites, and the introduction of a cost model to the optimization procedure. The implementation of site delegation, where parts of queries are exchanged and combined, results from a richer lattice of locations and capabilities. Our setting can be uniformly extended to cope with delegation, although some work is needed to adapt the formal results. Traditional optimization approaches, based on the reordering of join clauses, are orthogonal to our approach, that focuses on localization of sub-queries based on site capabilities. Nevertheless, the association of a traditional cost model to our localization and capability based algorithm is

also interesting. In (Taylor, 2010), remote execution of subqueries already incorporates a transmission cost, which can easily extend and interplay in the binder grouping mechanism that we have introduced to produce the best combined result.

Finally, recall that our model is the base for a new visual data manipulation language in the OutSystems platform, one that allows the gradual construction of queries with immediate feedback to developers. In this scenario, developers are abstracted from the real usages of their queries, and therefore code specialization is a highly desired feature. Future work also includes the definition of the query rewriting mechanism that simplifies the deep data manipulation operations on nested data. We foresee an approach that builds and extends existing works on normalization of queries (Cooper, 2009; Cheney *et al.*, 2013), to be able to deal with more instances of the localization problem.

Acknowledgments We thank the anonymous reviewers for the insightful comments. João Costa Seco is supported by NOVA LINCS ref. UID/CEC/04516/2013.

References

Benzaken, Véronique, Castagna, Giuseppe, & Frisch, Alain. (2003). Cduce: An xml-centric general-purpose language. *Pages 51–63 of: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. New York, NY, USA: ACM.

Buneman, Peter, Libkin, Leonid, Suciu, Dan, Tannen, Val, & Wong, Limsoon. (1994). Comprehension syntax. *Sigmod rec.*, **23**(1), 87–96.

Buneman, Peter, Naqvi, Shamim, Tannen, Val, & Wong, Limsoon. (1995). Principles of programming with complex objects and collection types. *Theor. comput. sci.*, **149**(1), 3–48.

Cardelli, Luca. (1989). Typeful programming. *IFIP State of the Art Reports (Formal Description of Programming Concepts)*, 431–507.

Cheney, James, Lindley, Sam, & Wadler, Philip. (2013). A practical theory of language-integrated query. *Pages 403–416 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM.

Cheney, James, Lindley, Sam, & Wadler, Philip. (2014). Query shredding: Efficient relational evaluation of queries over nested multisets. *Pages 1027–1038 of: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. New York, NY, USA: ACM.

Chlipala, Adam. (2015). Ur/web: A simple model for programming the web. *Pages 153–165 of: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. New York, NY, USA: ACM.

Clark, James, & DeRose, Steven J. (1999). *XML Path Language (XPath) Version 1.0*.

Colby, Latha S. (1989). A recursive algebra and query optimization for nested relations. *Pages 273–283 of: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. New York, NY, USA: ACM.

Cooper, Ezra. (2009). The script-writer's dream: How to write great sql in your own language, and be sure it will succeed. *Pages 36–51 of: Proceedings of the 12th International Symposium on Database Programming Languages*. DBPL '09. Berlin, Heidelberg: Springer-Verlag.

Cooper, Ezra, Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2007). Links: Web programming without tiers. *Pages 266–296 of: Proceedings of the 5th International Conference on Formal Methods for Components and Objects*. FMCO'06. Berlin, Heidelberg: Springer-Verlag.

Davies, Rowan, & Pfenning, Frank. (2001). A modal analysis of staged computation. *J. acm*, **48**(3), 555–604.

Fu, Yupeng, Ong, Kian Win, & Papakonstantinou, Yannis. (2013). Declarative Ajax Web Applications through SQL++ on a Unified Application State. *Proceedings of Intern. Symposium on Database Programming Languages*.

Grade, Nuno, Ferrão, Lúcio, & Seco, João Costa. (2013). Optimizing Data Queries Over Heterogeneous Sources. *Proceedings of the 5th Simpósio de Informática*.

Halevy, Alon, Rajaraman, Anand, & Ordille, Joann. (2006). Data integration: The teenage years. *Pages 9–16 of: Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB '06. VLDB Endowment.

Jones, Simon Peyton, & Wadler, Philip. (2007). Comprehensive comprehensions. *Pages 61–72 of: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. New York, NY, USA: ACM.

Lindley, Sam, & Cheney, James. (2012). Row-based effect types for database integration. *Pages 91–102 of: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. New York, NY, USA: ACM.

Newman, M. H. A. (1942). On theories with a combinatorial definition of "equivalence". *Annals of math.*, **43**(2), 223–243.

OutSystems. (2016). *Using Aggregates - Fetch and Display Data from the Database. Tech. Documentation*.

Papakonstantinou, Yannis, Gupta, Ashish, & Haas, Laura. (1998). Capabilities-based query rewriting in mediator systems. *Distributed and parallel databases*, **6**(1), 73–110.

Robie, Jonathan, *et al.* . (2014). *XQuery 3.0: An XML Query Language*.

Seco, João Costa, Lourenço, Hugo, & Ferreira, Paulo. (2015). A common data manipulation language for nested data in heterogeneous environments. *Pages 11–20 of: Proceedings of the 15th Symposium on Database Programming Languages*. DBPL 2015.

Serrano, Manuel, Gallesio, Erick, & Loitsch, Florian. (2006). Hop: a language for programming the web 2.0. *Pages 975–985 of: Companion to the 21th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Silberschatz, Abraham, Korth, Henry, & Sudarshan, S. (2006). *Database systems concepts*. 5 edn. New York, NY, USA: McGraw-Hill, Inc.

Taylor, Robert. (2010). *Query optimization for distributed database systems*. Master thesis, University of Oxford.

Vassalos, Vasilis, & Papakonstantinou, Yannis. (2000). Expressive capabilities description languages and query rewriting algorithms. *The journal of logic programming*, **43**(1), 75 – 122.

Wong, Limsoon. (2000). Kleisli, a functional query system. *J. funct. program.*, **10**(1), 19–56.

A Type System

$$\Delta \vdash num : \mathsf{Num}\ (\textsc{Num}) \qquad \Delta \vdash bool : \mathsf{Bool}\ (\textsc{Bool}) \qquad \Delta \vdash string : \mathsf{String}\ (\textsc{String})$$

$$\Delta \vdash date : \mathsf{Date}\ (\textsc{Date}) \qquad \frac{\Delta \vdash \mathrm{op} : \tau' \to \tau'' \to \tau \quad \Delta \vdash e : \tau' \quad \Delta \vdash e' : \tau''}{\Delta \vdash e\ \mathrm{op}\ e' : \tau}\ (\textsc{Op})$$

$$\Delta, x : \tau \vdash x : \tau\ (\textsc{Id}) \qquad \frac{\Delta, x : \tau \vdash e : \sigma}{\Delta \vdash \lambda x{:}\tau.e : \tau \to \sigma}\ (\textsc{Fun}) \qquad \frac{\Delta \vdash e' : \tau \quad \Delta \vdash e : \tau \to \sigma}{\Delta \vdash e\ e' : \sigma}\ (\textsc{App})$$

$$\frac{\Delta \vdash e : \sigma \quad \sigma \leq \tau}{\Delta \vdash e : \tau}\ (\textsc{Sub}) \qquad \frac{\Delta \vdash e_i : \tau_i \ \ _{i=0..n}}{\Delta \vdash \langle \overline{a = e} \rangle : \langle \overline{a : \tau} \rangle}\ (\textsc{Record})$$

$$\frac{\Delta \vdash e : \langle a : \tau, \overline{b : \sigma} \rangle}{\Delta \vdash e.a : \tau}\ (\textsc{Field}) \qquad \frac{\Delta \vdash e : \langle \overline{a : \tau} \rangle \quad \Delta \vdash e' : \langle \overline{b : \sigma} \rangle}{\Delta \vdash e \oplus e' : \langle \overline{a : \tau}, \overline{b : \sigma} \rangle}\ (\textsc{Concat})\ \overline{a} \# \overline{b}$$

$$\Delta \vdash \emptyset : \tau^*\ (\textsc{Empty}) \qquad \frac{\Delta \vdash e : \tau}{\Delta \vdash [e] : \tau^*}\ (\textsc{Singleton}) \qquad \frac{\Delta \vdash e : \tau^* \quad \Delta \vdash e' : \tau^*}{\Delta \vdash e \uplus e : \tau^*}\ (\textsc{Append})$$

$$\frac{\Delta \vdash e_i : \tau_i \quad _{i=1..n}}{\Delta, t : \overline{\tau} \to \tau \vdash \mathsf{db}_\ell(t, \overline{e}) : \mathscr{Q}(\tau)}\ (\textsc{Source})$$

$$\frac{\Delta \vdash e_i : \mathscr{Q}(\tau_i^*) \ \ _{i=1..n} \quad \Delta, \overline{x : \tau} \vdash e' : \mathsf{bool} \quad \Delta, \overline{x : \tau} \vdash e'' : \sigma}{\Delta \vdash \mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow e}\ \}\ e'' : \mathscr{Q}(\sigma^*)}\ (\textsc{Select})$$

$$\frac{\Delta \vdash e : \mathscr{Q}(\tau^*) \quad \Delta, x : \tau \vdash e_i : \sigma_i \quad _{i=1..n}}{\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow e\ \} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)}\ (\textsc{Group})$$

$$\frac{\Delta \vdash e : \tau}{\Delta \vdash \mathsf{return}\ e : \mathscr{Q}(\tau)}\ (\textsc{Return}) \qquad \frac{\Delta \vdash e : \mathscr{Q}(\tau') \to \mathscr{Q}(\sigma') \quad \Delta \vdash e' : \mathscr{Q}(\tau)}{\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathscr{Q}(\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\})}\ (\textsc{At})$$

$$\frac{\Delta \vdash e : \mathscr{Q}(\sigma') \quad \sigma' \leq \sigma \quad \Delta, x : \sigma \vdash e' : \tau}{\Delta \vdash \mathsf{exec}\ x : \sigma = e\ \mathsf{in}\ e' : \tau}\ (\textsc{Exec})$$

$$\frac{\Delta \vdash e : \sigma \quad \sigma \leq \tau}{\Delta \vdash {}^{\tau}\pi^{\sigma}(e) : \tau}\ (\textsc{Project}) \qquad \frac{\Delta \vdash e : \tau}{\Delta \vdash [\,e\,]_\ell : \tau}\ (\textsc{Remote})$$

B Formal Results

Lemma B.1 (*Substitution Lemma*)

If $\Delta, x : \tau \vdash e : \sigma$ and $\Delta \vdash v : \tau$ then $\Delta \vdash e\{^{v}/_{x}\} : \sigma'$ with $\sigma' \leq \sigma$.

Proof. By induction on the size of the type derivation, performing case analysis on the last typing rule applied.

$\Delta \vdash v : \tau$. H0.

Case (NUM).
$\Delta \vdash num : \mathsf{num}$. (NUM).
$\Delta \vdash num\{^{v}/_{x}\} : \mathsf{num}$. (SUBST).

Cases (BOOL), (STRING), (DATE). Similar.

Case (ID).
Assuming $x \neq y$, by alpha-renaming.
$\Delta, x : \tau \vdash y : \sigma$. H1.
$\Delta(y) = \sigma$. Invert (ID) on H1.
$\Delta \vdash y : \sigma$. (ID).
$\Delta \vdash y\{^{v}/_{x}\} : \sigma$. (SUBST).
$\sigma \leq \sigma$

Case (FUN).
Assuming $x \neq y$, by alpha-renaming.
$\Delta, x : \tau \vdash \lambda y.e : \sigma_1 \to \sigma_2$. H1.
$\Delta, x : \tau, y : \sigma_1 \vdash e : \sigma_2$. Invert (FUN) on H1.
$\Delta, y : \sigma_1, x : \tau \vdash e : \sigma_2$. (EXCHANGE).
$\Delta, y : \sigma_1 \vdash v : \tau$. Apply (WEAKENING) on H0.
$\Delta, y : \sigma_1 \vdash e\{^{v}/_{x}\} : \sigma_2'$. Induction.
$\sigma_2' \leq \sigma_2$
$\Delta \vdash \lambda y.e\{^{v}/_{x}\} : \sigma_1 \to \sigma_2'$. (FUN).
$\Delta \vdash (\lambda y.e)\{^{v}/_{x}\} : \sigma_1 \to \sigma_2'$. (SUBST).
$\sigma_1 \to \sigma_2' \leq \sigma_1 \to \sigma_2'$

Case (APP).
$\Delta, x : \tau \vdash e\ e' : \sigma$. H1.

$\Delta, x : \tau \vdash e : \tau' \to \sigma$. Invert (APP) on H1.
$\Delta \vdash e\{^{v}/_{x}\} : \tau'' \to \sigma'$.
$\tau'' \to \sigma' \leq \tau' \to \sigma$ Induction.

$\tau' \leq \tau''$
$\sigma' \leq \sigma$

$\Delta, x : \tau \vdash e' : \tau'.$ Invert (APP) on H1.
$\Delta \vdash e'\{^{v}/_{x}\} : \tau'''.$
$\tau''' \leq \tau'$ Induction.

$\tau''' \leq \tau''$
$\Delta \vdash e'\{^{v}/_{x}\} : \tau'.$
$\Delta \vdash (e\{^{v}/_{x}\})\ (e'\{^{v}/_{x}\}) : \sigma'.$ (APP).
$\Delta \vdash (e\ e')\{^{v}/_{x}\} : \sigma'.$ (SUBST).

Case (RECORD).
$\Delta, x : \tau \vdash \langle \overline{a = e} \rangle : \langle \overline{a : \sigma} \rangle.$ H1.
$\Delta, x : \tau \vdash \overline{e : \sigma}.$ Invert (RECORD) on H1.
$\Delta \vdash \overline{e\{^{v}/_{x}\} : \sigma'}.$ Induction.
$\sigma'_i \leq \sigma_i$
$\Delta \vdash \langle \overline{a = e\{^{v}/_{x}\}} \rangle : \langle \overline{a : \sigma'} \rangle.$ (RECORD).
$\Delta \vdash \langle \overline{a = e} \rangle \{^{v}/_{x}\} : \langle \overline{a : \sigma'} \rangle.$ (SUBST).
$\langle \overline{a : \sigma'} \rangle \leq \langle \overline{a : \sigma} \rangle$

Case (CONCAT).
$\Delta, x : \tau \vdash e \uplus e' : \sigma.$ H1.

$\Delta, x : \tau \vdash e : \sigma.$ Invert (CONCAT) on H1.
$\Delta \vdash e\{^{v}/_{x}\} : \sigma'.$ Induction.
$\Delta \vdash e\{^{v}/_{x}\} : \sigma.$ Subsumption.
$\sigma' \leq \sigma$

$\Delta, x : \tau \vdash e' : \sigma''.$ Invert (CONCAT) on H1.
$\Delta \vdash e'\{^{v}/_{x}\} : \sigma''.$ Induction.
$\sigma'' \leq \sigma$
$\Delta \vdash e'\{^{v}/_{x}\} : \sigma.$ Subsumption.
$\Delta \vdash (e\{^{v}/_{x}\}) \uplus (e'\{^{v}/_{x}\}) : \sigma.$ (CONCAT).
$\Delta \vdash (e \uplus e')\{^{v}/_{x}\} : \sigma.$ (SUBST).

Case (EXEC).
Assuming $x \neq y$, by alpha-renaming.
$\Delta, x : \tau \vdash \mathsf{exec}\ y = e\ \mathsf{in}\ e' : \sigma.$ H1.

$\Delta, x : \tau \vdash e : \mathscr{Q}(\sigma').$ Invert (EXEC) on H1.
$\Delta \vdash e\{^{v}/_{x}\} : \sigma'.$ Induction.

$\Delta, y : \sigma' \vdash v : \sigma.$ Apply (WEAKENING) on H0.
$\Delta, x : \tau, y : \sigma' \vdash e' : \sigma.$ Invert (EXEC) on H1.
$\Delta, y : \sigma', x : \tau \vdash e' : \sigma.$ (EXCHANGE).
$\Delta, y : \sigma' \vdash e'\{^{v}/_{x}\} : \sigma.$ Induction.

$\Delta \vdash \mathsf{exec}\ y = e\{^v/_x\}\ \mathsf{in}\ e'\{^v/_x\} : \sigma.$ (EXEC).
$\Delta \vdash (\mathsf{exec}\ y = e\ \mathsf{in}\ e')\{^v/_x\} : \sigma.$ (SUBST).

Case (SELECT).
Assuming $x \notin \overline{y}$, by alpha-renaming.
$\Delta, x : \tau \vdash \mathsf{foreach}_{e'}\ \{\ \overline{y \leftarrow e}\ \}\ e'' : \mathcal{Q}(\sigma^*).$ H1.

$\Delta, x : \tau \vdash \overline{e_i : \mathcal{Q}(\tau_i{}^*)}.$ Invert (SELECT) on H1.
$\Delta \vdash \overline{e_i\{^v/_x\} : \mathcal{Q}(\tau_i{}^*)}.$ Induction.

$\Delta, x : \tau, \overline{y_i : \tau_i} \vdash e' : \mathsf{bool}.$ Invert (SELECT) on H1.
$\Delta, \overline{y_i : \tau_i}, x : \tau \vdash e' : \mathsf{bool}.$ (EXCHANGE).
$\Delta, \overline{y_i : \tau_i} \vdash v : \tau.$ Apply (WEAKENING) on H0.
$\Delta, \overline{y_i : \tau_i} \vdash e'\{^v/_x\} : \mathsf{bool}.$ Induction.

$\Delta, x : \tau, \overline{y_i : \tau_i} \vdash e'' : \sigma.$ Invert (SELECT) on H1.
$\Delta, \overline{y_i : \tau_i}, x : \tau \vdash e'' : \sigma.$ (EXCHANGE).
$\Delta, \overline{y_i : \tau_i} \vdash v : \tau.$ Apply (WEAKENING) on H0.
$\Delta, \overline{y_i : \tau_i} \vdash e''\{^v/_x\} : \sigma.$ Induction.

$\Delta \vdash \mathsf{foreach}_{e'\{^v/_x\}}\ \left\{\ \overline{y \leftarrow e\{^v/_x\}}\ \right\}\ e''\{^v/_x\} : \mathcal{Q}(\sigma^*).$ (SEL.).
$\Delta \vdash (\mathsf{foreach}_{e'}\ \{\ \overline{y \leftarrow e}\ \}\ e'')\{^v/_x\} : \mathcal{Q}(\sigma^*).$ (SUBST).

Case (GROUP).
Assuming $x \neq y$, by alpha-renaming.
$\Delta, x : \tau \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\ y \leftarrow e\ \} : \mathcal{Q}(\langle \overline{a : \sigma}, b : \tau_y{}^* \rangle^*).$ H1.

$\Delta, x : \tau \vdash e : \mathcal{Q}(\tau_y{}^*).$ Invert (GROUP) on H1.
$\Delta \vdash e\{^v/_x\} : \mathcal{Q}(\tau_y{}^*).$ Induction.

$\Delta, x : \tau, y : \tau_y \vdash \overline{e_i : \sigma_i}.$ Invert (GROUP) on H1.
$\Delta, y : \tau_y, x : \tau \vdash \overline{e_i : \sigma_i}.$ (EXCHANGE).
$\Delta, y : \tau_y \vdash v : \tau.$ Apply (WEAKENING) on H0.
$\Delta, y : \tau_y \vdash \overline{e_i\{^v/_x\} : \sigma_i}.$ Induction.

$\Delta \vdash \mathsf{groupby}_b^{\overline{a=e\{^v/_x\}}}\{\ y \leftarrow e\{^v/_x\}\ \} : \mathcal{Q}(\langle \overline{a : \sigma}, b : \tau_y{}^* \rangle^*).$ (GROUP).
$\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\ y \leftarrow e\ \}\{^v/_x\} : \mathcal{Q}(\langle \overline{a : \sigma}, b : \tau_y{}^* \rangle^*).$ (SUBST).

Case (AT).
$\Delta, x : \tau \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathcal{Q}(\sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\}).$ H1.

$\Delta, x : \tau \vdash e : \mathcal{Q}(\tau') \to \mathcal{Q}(\sigma').$ Invert (AT) on H1.
$\Delta \vdash e\{^v/_x\} : \mathcal{Q}(\tau') \to \mathcal{Q}(\sigma').$ Induction.

$\Delta, x : \tau \vdash e' : \mathcal{Q}(\sigma).$ Invert (AT) on H1.

$\Delta \vdash e'\{^{v}/_{x}\} : \mathcal{Q}(\sigma)$. Induction.

$\Delta \vdash \mathsf{do}\ e\{^{v}/_{x}\}_{\downarrow p}\{e'\{^{v}/_{x}\}\} : \mathcal{Q}(\sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\})$. (AT).
$\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{e'\}\{^{v}/_{x}\} : \mathcal{Q}(\sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\})$. (SUBST).

Case (RETURN).
$\Delta, x:\tau \vdash \mathsf{return}\ e : \mathcal{Q}(\sigma)$. H1.
$\Delta, x:\tau \vdash e : \sigma$. Invert (RETURN) on H1.
$\Delta \vdash e\{^{v}/_{x}\} : \sigma$. Induction.
$\Delta \vdash \mathsf{return}\ (e\{^{v}/_{x}\}) : \sigma$. (RETURN).
$\Delta \vdash (\mathsf{return}\ e)\{^{v}/_{x}\} : \sigma$. Invert (SUBST).

Case (SOURCE).
$\Delta, x:\tau \vdash \mathsf{db}_{\ell}(t,\overline{e}) : \mathcal{Q}(\sigma')$. H1.
$\Delta(t) = \overline{\sigma} \to \sigma'$. Invert (SOURCE) on H1.
$\Delta, x:\tau \vdash \overline{e:\sigma}$. Invert (SOURCE) on H1.
$\Delta \vdash \overline{e\{^{v}/_{x}\} : \sigma}$. Induction.
$\Delta \vdash \mathsf{db}_{\ell}(t, \overline{e\{^{v}/_{x}\}}) : \mathcal{Q}(\sigma')$. (SOURCE).
$\Delta \vdash \mathsf{db}_{\ell}(t,\overline{e})\{^{v}/_{x}\} : \mathcal{Q}(\sigma')$. Invert (SUBST).

□

We assume given a typing relation for the state.

Definition B.2 (*Typing states*)

If $\Delta \vdash \mathscr{S}$, $[\![\mathsf{db}_{\ell}(t,\overline{v})]\!] = v$, $\Delta(t) = \overline{\tau} \to \tau$ and $\Delta \vdash v_i : \tau_i$ then $\Delta \vdash v : \tau$.

In the following proofs, we consider the following typing rule for set and multi-set (bag) comprehensions..

Definition B.3 (*Typing comprehensions*)

$$\frac{\Delta \vdash e_i : \tau_i^* \quad \Delta, \overline{x:\tau} \vdash c : \mathsf{bool} \quad \Delta, \overline{x:\tau} \vdash e : \sigma}{\Delta \vdash \{\ e \mid \overline{x \in e}, c\ \} : \sigma^*}\ (\text{SET-C})$$

$$\frac{\Delta \vdash e_i : \tau_i^* \quad \Delta, \overline{x:\tau} \vdash c : \mathsf{bool} \quad \Delta, \overline{x:\tau} \vdash e : \sigma}{\Delta \vdash [\ e \mid \overline{x \in e}, c\ ] : \sigma^*}\ (\text{BAG-C})$$

Lemma B.4 (*Typing Run*)
If $\Delta \vdash e : \mathcal{Q}(\tau)$ then $\Delta \vdash \mathsf{run}\ e : \tau$.

Proof.
$\Delta \vdash e : \mathcal{Q}(\tau)$ H0.
$\Delta, x:\tau \vdash x : \tau$ (ID).

$\Delta \vdash \mathsf{exec}\ x = e\ \mathsf{in}\ x : \tau$ (EXEC).
$\Delta \vdash \mathsf{run}\ e : \tau$ By definition.
□

Lemma B.5 (*Semantics of Run*)
$\langle\!|\mathsf{run}\ e|\!\rangle = [\![\langle\!|e|\!\rangle]\!]$

Proof.
$\langle\!|\mathsf{exec}\ x = e\ \mathsf{in}\ x|\!\rangle = x\{^{[\![\langle\!|e|\!\rangle]\!]}/_{x}\}$ Definition of $\langle\!|\rangle$.
$\langle\!|\mathsf{exec}\ x = e\ \mathsf{in}\ x|\!\rangle = [\![\langle\!|e|\!\rangle]\!]$ (SUBST).
$\langle\!|\mathsf{run}\ e|\!\rangle = [\![\langle\!|e|\!\rangle]\!]$ By definition.
□

Theorem 5.5 (Type Soundness).
If $\Delta \vdash \mathscr{S}$ then

1. If $\Delta \vdash e : \tau$ and $\langle\!|e|\!\rangle = v$ then $\Delta \vdash v : \tau'$ with $\tau' \leq \tau$.
2. If $\Delta \vdash r : \mathscr{Q}(\tau)$ and $[\![r]\!] = v$ then $\Delta \vdash v : \tau'$ with $\tau' \leq \tau$.

Note: $\mathscr{S}$ is omitted in the $[\![]\!]$ and $\langle\!|\rangle$ relations used in the whole paper (and proofs) because it is always constant.
Proof. The two cases are proven by mutual induction on the size of the evaluation derivation. Case 1 is proven by case analysis on the last typing rule applied.

Case Literals and (ID). N/A (no reduction).

Case op.
$\Delta \vdash e\ \mathrm{op}\ e' : \tau$
$\Delta \vdash \mathrm{op} : \tau' \to \tau'' \to \tau$ H1.
$\Delta \vdash e : \tau'$ H2.
$\Delta \vdash e' : \tau''$ H3.
$\Delta \vdash \langle\!|e|\!\rangle : \sigma'$
$\sigma' \leq \tau'$ IND H2.
$\Delta \vdash \langle\!|e'|\!\rangle : \sigma''$
$\sigma'' \leq \tau''$ IND H3.

Case (FUN). By definition of $\langle\!|\rangle$.

Case (APP).
$\Delta \vdash e\ e' : \sigma$. H0.
$\Delta \vdash e' : \tau$
$\Delta \vdash e : \tau \to \sigma$ Invertion of (APP).
$\langle\!|e\ e'|\!\rangle = v'$ H1.
$\langle\!|e|\!\rangle = \lambda x.e''$
$\langle\!|e'|\!\rangle = v$
$\langle\!|e''\{^{v}/_{x}\}|\!\rangle = v'$ Invertion of (H1).

$\Delta \vdash \lambda x.e'' : \tau \to \sigma$ Induction.
$\Delta \vdash v : \tau$ Induction.
$\Delta \vdash e''\{^{v}/_{x}\} : \sigma$ Lemma B.1.
$\Delta \vdash v' : \sigma$ Induction.

Case (RECORD). Similar, by induction.

Case (CONCAT). Similar, by induction.

Case (SOURCE).
$\Delta \vdash \mathsf{db}_\ell(t,\overline{e}) : \mathscr{Q}(\tau)$ H0.
$\langle\!|\mathsf{db}_\ell(t,\overline{e})|\!\rangle = \mathsf{db}_\ell(t,\overline{v})$ H1.
$\Delta \vdash e_i : \tau_i$ Inversion of (SOURCE).
$\langle\!|e_i|\!\rangle = v_i$ Inv. of definition of $\langle\!|\,|\!\rangle$ in H1.
$\Delta \vdash v_i : \tau_i$ Induction.
$\Delta \vdash \mathsf{db}_\ell(t,\overline{v}) : \mathscr{Q}(\tau)$ (SOURCE).

Case (SELECT).
$\Delta \vdash \mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow e}\ \}\ e'' : \mathscr{Q}(\sigma^*)$ H0.
$\langle\!|\mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow e}\ \}\ e''|\!\rangle = \mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow r}\ \}\ e''$ H1.
$\Delta \vdash e_i : \mathscr{Q}(\tau_i{}^*)$
$\Delta, \overline{x : \tau} \vdash e' : \mathsf{bool}$
$\Delta, \overline{x : \tau} \vdash e'' : \sigma$ Invertion of (SELECT).
$\langle\!|e_i|\!\rangle = r_i$ Inv. of definition of $\langle\!|\,|\!\rangle$ in H1.
$\Delta \vdash r_i : \mathscr{Q}(\tau_i{}^*)$ Induction.
$\Delta \vdash \mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow r}\ \}\ e'' : \mathscr{Q}(\sigma^*)$ (SELECT).

Case (GROUP).
$\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow e\ \} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)$ H0.
$\langle\!|\mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow e\ \}|\!\rangle = \mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow r\ \}$ H1.
$\Delta \vdash e : \mathscr{Q}(\tau^*)$
$\Delta, x : \tau \vdash e_i : \sigma_i$ Invertion of (GROUP).
$\langle\!|e|\!\rangle = r$ Inv. of definition of $\langle\!|\,|\!\rangle$ in H1.
$\Delta \vdash r : \mathscr{Q}(\tau^*)$ Induction.
$\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow r\ \} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)$ (GROUP).

Case (RETURN).
$\Delta \vdash \mathsf{return}\ e : \mathscr{Q}(\tau)$ H0.
$\langle\!|\mathsf{return}\ e|\!\rangle = \mathsf{return}\ v'$ H1.
$\Delta \vdash e : \tau$ Invert H0.
$\langle\!|e|\!\rangle = v'$ Inv. of definition of $\langle\!|\,|\!\rangle$ in H1.
$\Delta \vdash v' : \tau$ Induction.
$\Delta \vdash \mathsf{return}\ v' : \mathscr{Q}(\tau)$ (RETURN).

Case (AT).
$\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathscr{Q}(\tau_{\downarrow p}\{^{\tau'}/_{\sigma'}\})$ H0.

$\langle\!\langle \mathsf{do}\ e_{\downarrow p}\{e'\}\rangle\!\rangle = \mathsf{do}\ e_{\downarrow p}\{r\}$ H1.
$\Delta \vdash e : \mathscr{Q}(\tau') \to \mathscr{Q}(\sigma')$
$\Delta \vdash e' : \mathscr{Q}(\tau)$ Invertion of (AT).
$\langle\!\langle e'\rangle\!\rangle = r$ Inv. of definition of $\langle\!\langle\rangle\!\rangle$ in H1.
$\Delta \vdash r : \mathscr{Q}(\tau)$ Induction.
$\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{r\} : \mathscr{Q}(\tau_{\downarrow p}\{{}^{\tau'}/_{\sigma'}\})$ (AT).

Case (EXEC).
$\Delta \vdash \mathsf{exec}\ x = e\ \mathsf{in}\ e' : \tau$ H0.
$\langle\!\langle \mathsf{exec}\ x = e\ \mathsf{in}\ e'\rangle\!\rangle = v'$ H1.
$\Delta \vdash e : \mathscr{Q}(\sigma)$ §2.
$\Delta, x : \sigma \vdash e' : \tau$ Invertion of (EXEC).
$\langle\!\langle e\rangle\!\rangle = r$
$[\![r]\!] = v$ §2.
$\langle\!\langle e'\{{}^{v}/_{x}\}\rangle\!\rangle = v'$ Inv. of definition of $\langle\!\langle\rangle\!\rangle$ in H1.
$\Delta \vdash v : \sigma$ Case 2 on §1 and §2 .
$\Delta \vdash e'\{{}^{v}/_{x}\} : \tau$ Lemma B.1.
$\Delta \vdash v' : \tau$ Induction.

Case 2 is proven by induction on the size evaluation derivation $[\![r]\!]$ and case analysis of the last typing rule used.

Case (SOURCE).
$\Delta \vdash \mathsf{db}_\ell(s, \overline{v}) : \mathscr{Q}(\tau)$ H0.
$\Delta(t) = \overline{\tau} \to \tau$
$\Delta \vdash v_i : \tau_i$ Invertion of (SOURCE).
$[\![\mathsf{db}_\ell(t, \overline{v})]\!] = v$ H1.
$\Delta \vdash v : \tau$ Definition B.2.

Case (SELECT).
$\Delta \vdash \mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow r}\ \}\ e'' : \mathscr{Q}(\sigma^*)$ H0.
$[\![\mathsf{foreach}_{e'}\ \{\ \overline{x \leftarrow r}\ \}\ e'']\!] = [\ v''_{\overline{u}} \mid \overline{u \in v}, v'_{\overline{u}}\ ]$ H1.
$\Delta \vdash r_i : \mathscr{Q}(\tau_i{}^*)$ Invertion of (SELECT).
$[\![r_i]\!] = v_i$ Inv. of definition of $[\![]\!]$ in H1.
$\Delta \vdash v_i : \tau_i{}^*$ §1. By induction.
$\Delta \vdash u_{i_j} : \tau_i$ §1 and $v_i = \overline{u_i}$

$\Delta, \overline{x : \tau} \vdash e' : \mathsf{bool}$ Invertion of (SELECT).
$\Delta \vdash e'\{{}^{\overline{u}}/_{\overline{x}}\} : \mathsf{bool}$ By Lemma B.1 for all u_{i_j}.
$\langle\!\langle e'\{{}^{\overline{u}}/_{\overline{x}}\}\rangle\!\rangle = v'_{\overline{u}}$ Inv. of definition of $[\![]\!]$ in H1.
$\Delta \vdash v'_{\overline{u}} : \mathsf{bool}$ Case 1.

$\Delta, \overline{x : \tau} \vdash e'' : \sigma$ Invertion of (SELECT).
$\Delta \vdash e''\{{}^{\overline{u}}/_{\overline{x}}\} : \sigma$ By Lemma B.1.
$\langle\!\langle e''\{{}^{\overline{u}}/_{\overline{x}}\}\rangle\!\rangle = v''_{\overline{u}}$ Inv. of definition of $[\![]\!]$ in H1.

$\Delta \vdash v''_u : \sigma$ Case 1.

$\Delta \vdash [\, v''_u \mid \overline{u \in v}, v'_u \,] : \sigma^*$ (BAG-C).

Case (GROUP).
$\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}}\{\, x \leftarrow r \,\} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)$ H0.
$[\![\mathsf{groupby}_b^{\overline{a=e}}\{\, x \leftarrow r \,\}]\!] = [k \oplus \langle b = dtls_k \rangle \mid k \in ks]$ H1.
$\Delta \vdash r : \mathscr{Q}(\tau^*)$ Invertion of (GROUP).
$[\![r]\!] = v$ Inv. of definition of $[\![]\!]$ in H1.
$\Delta \vdash v : \tau^*$ Induction.

$\Delta \vdash u : \tau$ $\forall u \in v$

$\Delta, x : \tau \vdash e_i : \sigma_i$ Invertion of (GROUP).
$\Delta \vdash e_i\{^u/_x\} : \sigma_i$ By Lemma B.1.
$\langle\!\langle e_i\{^u/_x\} \rangle\!\rangle = v'_i$ Inv. of definition of $[\![]\!]$ in H1.
$\Delta \vdash v'_i : \sigma_i$ Case 1.

$ks = \{ \langle \overline{a = \langle\!\langle e\{^u/_x\} \rangle\!\rangle} \rangle \mid u \in v \}$
$\Delta \vdash ks : \langle \overline{a : \sigma} \rangle^*$ (SET-C).

$\Delta \vdash k : \langle \overline{a : \sigma} \rangle$ $k \in ks$.
$dtls_k = [u \mid u \in v, \langle \overline{a = \langle\!\langle e\{^u/_x\} \rangle\!\rangle} \rangle = k]$
$\Delta \vdash dtls_k : \tau^*$ (BAG-C).

$\Delta \vdash [k \oplus \langle b = dtls_k \rangle \mid k \in ks] : \langle \overline{a : \sigma}, b : \tau^* \rangle^*$ (BAG-C).

Case (RETURN).
$\Delta \vdash \mathsf{return}\ v : \mathscr{Q}(\tau)$ H0.
$[\![\mathsf{return}\ v]\!] = v$ H1.
$\Delta \vdash v : \tau$ Invert (RETURN).

Case (AT).
$\Delta \vdash \mathsf{do}\ e_{\downarrow p}\{r\} : \mathscr{Q}(\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\})$ H0.
$\Delta \vdash r : \mathscr{Q}(\tau)$
$\Delta \vdash e : \mathscr{Q}(\tau') \to \mathscr{Q}(\sigma')$ Invert (AT)
$\Delta \vdash [\![r]\!] : \tau$ Induction on H1, §1.

SubCase $p = \varepsilon$.
$[\![\mathsf{do}\ e_{\downarrow \varepsilon}\{r\}]\!] = [\![\langle\!\langle e\ (\mathsf{return}\ [\![r]\!]) \rangle\!\rangle]\!]$ Definition of $[\![]\!]$.
$\Delta \vdash \mathsf{return}\ [\![r]\!] : \mathscr{Q}(\tau)$ (RETURN).
$\tau = \tau'$ Invertion of H0, definition 5.1.
$\Delta \vdash e\ (\mathsf{return}\ [\![r]\!]) : \mathscr{Q}(\sigma')$ (APP).
$\langle\!\langle e\ (\mathsf{return}\ [\![r]\!]) \rangle\!\rangle = r'$ Definition of $\langle\!\langle\rangle\!\rangle$.
$\Delta \vdash r' : \mathscr{Q}(\sigma')$ Case 1.

$[\![r']\!] = v'$ By definition.
Note: $[\![r']\!]$ is a subderivation of $[\![\langle\!\langle e(\mathsf{return}\ [\![r]\!])\rangle\!\rangle]\!]$
$\Delta \vdash v' : \sigma'$ Induction.
$\Delta \vdash v' : \tau_{\downarrow_\varepsilon}\{^{\sigma'}/_{\tau}\}$ By definition 5.1.

SubCase $p = .ap'$.
$[\![\mathsf{do}\ e_{\downarrow.ap}\{r\}]\!] = \langle a = [\![\mathsf{do}\ e_{\downarrow p}\{\mathsf{return}\ u\}]\!], \overline{b = v}\rangle$
$\Delta \vdash u : \tau$ §1.
$\tau_{\downarrow.ap}\{^{\sigma'}/_{\tau'}\} = \langle a : \sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\}, \overline{b : \tau}\rangle$
$\Delta \vdash [\![\mathsf{do}\ e_{\downarrow p}\{\mathsf{return}\ u\}]\!] : \sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\}$ (RETURN) and (AT)
$[\![\mathsf{do}\ e_{\downarrow p}\{\mathsf{return}\ u\}]\!] = v$
$\Delta \vdash v : \sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\}$ Induction
$\Delta \vdash \langle a = v, \overline{b = v}\rangle : \langle a : \sigma_{\downarrow p}\{^{\sigma'}/_{\tau'}\}, \overline{b : \tau}\rangle$
$\Delta \vdash \langle a = v, \overline{b = v}\rangle : \tau_{\downarrow.ap}\{^{\sigma'}/_{\tau'}\}$ (RECORD)

SubCase $p = /p'$.
$[\![\mathsf{do}\ e_{\downarrow/p'}\{r\}]\!] = [\ [\![\mathsf{do}\ e_{\downarrow p'}\{\mathsf{return}\ u\}]\!] \mid u \in [\![r]\!]\ ]$
$\Delta \vdash [\![r]\!] : \tau$ Induction.
$\tau = \tau''^{*}$ Invertion of (AT).
$\tau''^{*}_{\downarrow/p'}\{^{\sigma'}/_{\tau'}\} = (\tau''_{\downarrow p'}\{^{\sigma'}/_{\tau'}\})^{*}$
$\Delta \vdash [\![\mathsf{do}\ e_{\downarrow p'}\{\mathsf{return}\ u\}]\!] : \sigma_{\downarrow p'}\{^{\sigma'}/_{\tau'}\}$ (RETURN) and (AT)
$[\![\mathsf{do}\ e_{\downarrow p'}\{\mathsf{return}\ u_i\}]\!] = v_i$ $\forall u_i \in \overline{u}, v_i \in \overline{v}$
$\Delta \vdash v_i : \sigma_{\downarrow p'}\{^{\sigma'}/_{\tau'}\}$ Induction
$\Delta \vdash [\ v_i \mid v \in \overline{v}\ ] : (\sigma_{\downarrow p'}\{^{\sigma'}/_{\tau'}\})^{*}$ (LIST)
$\Delta \vdash [\ v_i \mid v \in \overline{v}\ ] : \tau_{\downarrow/p'}\{^{\sigma'}/_{\tau'}\}$

Case (EXEC).
$\Delta \vdash \mathsf{exec}\ x : \sigma = e\ \mathsf{in}\ e' : \tau$
$\Delta \vdash e : \mathscr{Q}(\sigma')$ H1.
$\sigma' \leq \sigma$ H2.
$\Delta, x : \sigma \vdash e' : \tau$ H3.
$\Delta \vdash \langle\!\langle e \rangle\!\rangle : \mathscr{Q}(\sigma'')$
$\mathscr{Q}(\sigma'') \leq \mathscr{Q}(\sigma')$ IND H1.
$\sigma'' \leq \sigma$
$\Delta \vdash e''\{^{v}/_{x}\} : \tau'$
$\tau' \leq \tau$ SUBST.
$\Delta \vdash \langle\!\langle e''\{^{v}/_{x}\}\rangle\!\rangle : \tau''$ IND.
$\tau'' \leq \tau'$
$\tau'' \leq \tau$ □

Theorem 6.2 (Type Preservation - Phase I).

If $\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma$ then $\tau \leq \sigma$, $\Delta \leq \Gamma$, and $\Delta \vdash e' : \sigma$.

Proof. We prove this lemma by induction on the size of the derivation of $\Delta,\Gamma \vdash e : \tau \Rightarrow e' : \sigma$ and analysing the last rule applied.

Case base values $(num, bool, date, string)$.
$e = e'$ and $\tau = \sigma$.

Case Identifier.

SubCase $\tau = \tau'$.
$\Delta, x:\tau;\ \emptyset, x:\tau \vdash x : \tau \Rightarrow x : \tau$ H1.

SubCase $\tau \neq \tau'$.
$\Delta, x:\tau;\ \emptyset, x:\tau' \vdash x : \tau \Rightarrow {}^{\tau'}\pi^{\tau}(x) : \tau'$ H1.
$\tau \leq \tau'$ H2.
$\Delta, x:\tau \vdash x : \tau$ §1., (ID)
$\Delta, x:\tau \vdash {}^{\tau'}\pi^{\tau}(x) : \tau'$ §H2, §1, (PROJECTION)
$\Delta, x:\tau \leq \emptyset, x:\tau'$ (ENVSUB).

Case Abstraction.
$\Delta;\Gamma \vdash (\lambda x{:}\tau.e) : \tau \to \sigma \Rightarrow (\lambda x{:}\tau.e') : \tau \to \sigma'$
$\Delta, x:\tau;\Gamma, x:\tau \vdash e : \sigma \Rightarrow e' : \sigma'$ H1.
$\sigma \leq \sigma'$ IND.
$\Delta, x:\tau \vdash e' : \sigma'$ IND.
$\Delta, x:\tau \leq \Gamma, x:\tau$ IND.
$\Delta \vdash \lambda x{:}\tau.e' : \tau \to \sigma'$ ABSTRACTION.
$\Delta \leq \Gamma$ (ENVSUB).

Case Application.
$\Delta;\Gamma \vdash (e\ e') : \tau \Rightarrow (e''\ e''') : \sigma$
$\Delta,\Gamma \vdash e : \delta \to \tau \Rightarrow e'' : \delta' \to \sigma$ H1.
$\Delta \vdash e' : \delta \Rightarrow e''' : \delta'$ H2.
$\Delta \vdash e'' : \delta' \to \sigma$ §1.
$\Delta \vdash e''' : \delta'$ §2.
$\Delta \leq \Gamma$ IND.
$\Delta \vdash (e''\ e''') : \sigma$ §1, §2, and (APPLICATION)

Case Record.
$\Delta;\Gamma \vdash \langle \overline{a = e}, \overline{b = e'} \rangle : \langle \overline{a : \tau}, \overline{b : \sigma} \rangle \Rightarrow \langle \overline{a = e''} \rangle : \langle \overline{a : \tau'} \rangle$
$\Delta;\Gamma \vdash e_i : \tau_i \Rightarrow e''_i : \tau'_i \ _{i=1..n}$ H1.
$\Delta \vdash e''_i : \tau'_i \ _{i=1..n}$ §1.
$\Delta \leq \Gamma$ IND.
$\Delta \vdash \langle \overline{a = e''} \rangle : \langle \overline{a : \tau'} \rangle$ §1 and (RECORD)

Case Concat.
$\Delta;\Gamma \vdash e_1 \oplus e_2 : \tau_1 \oplus \tau_2 \Rightarrow e'_1 \oplus e'_2 : \sigma_1 \oplus \sigma_2$
$\Delta;\Gamma \vdash e_i : \tau_i \Rightarrow e'_i : \sigma_i \ _{i=1..n}$ H1.

$\Delta \vdash e'_i : \sigma_i \ _{i=1..n}$ §1.
$\Delta \leq \Gamma$
$\tau_i \leq \sigma_i \ _{i=1..n}$ IND, §2.
$\tau_1 \oplus \tau_2 \leq \sigma_1 \oplus \sigma_2$ §2 and (SUB RECORD).
$\Delta \vdash e'_1 \oplus e'_2 : \sigma_1 \oplus \sigma_2$ §1 and (CONCAT).

Case Singleton.
$\Delta;\Gamma \vdash [e] : \tau^* \Rightarrow [e'] : \sigma^*$
$\Delta;\Gamma \vdash e : \tau \Rightarrow e' : \sigma$ H1.
$\Delta \vdash e' : \sigma$ IND, §1.
$\tau \leq \sigma$ IND, §2.
$\Delta \leq \Gamma$ IND.
$\tau^* \leq \sigma^*$ §1 and (SUB LIST).
$\Delta \vdash [e'] : \sigma^*$ §1 and (SINGLETON).

Case Append.
$\Delta;\Gamma \vdash e_1 \uplus e_2 : \tau^* \Rightarrow e'_1 \uplus e'_2 : \sigma^*$
$\Delta;\Gamma \vdash e_i : \tau^* \Rightarrow e'_i : \sigma^* \ _{i=1..n}$ H1.
$\Delta \vdash e'_i : \sigma^* \ _{i=1..n}$ §1.
$\Delta \leq \Gamma$
$\tau^* \leq \sigma^*$ IND.
$\Delta \vdash e'_1 \uplus e'_2 : \sigma^*$ §1 and (APPEND).

Case Data source.
$\Delta;\Gamma \vdash \mathsf{db}_\ell(t,\overline{e}) : \mathscr{Q}(\tau) \Rightarrow {}^{\mathscr{Q}(\tau')}\pi^{\mathscr{Q}(\tau)}(\ \mathsf{db}_\ell(t,\overline{e'})\ ) : \mathscr{Q}(\tau')$
$\Delta(t) = \overline{\sigma} \to \tau$ H1.
$\Delta;\Gamma \vdash e_i : \sigma \Rightarrow e'_i : \sigma \quad i = 1..n$ H2.
$\tau \leq \tau'$ H3.
$\Delta \leq \Gamma$
$\Delta \vdash e'_i : \sigma \quad i = 1..n$ IND.
$\Delta \vdash \mathsf{db}_\ell(t,\overline{e'}) : \mathscr{Q}(\tau)$ DATA SOURCE.
$\mathscr{Q}(\tau) \leq \mathscr{Q}(\tau')$ H3 and SUB QUERY
$\Delta \vdash {}^{\mathscr{Q}(\tau')}\pi^{\mathscr{Q}(\tau)}(\ \mathsf{db}_\ell(t,\overline{e'})\ ) : \mathscr{Q}(\tau')$ PROJECTION.

Case Select.
$\Delta;\Gamma \vdash \mathsf{foreach}_c\ \{\ \overline{x \leftarrow e}\ \}\ e : \mathscr{Q}(\sigma^*) \Rightarrow \mathsf{foreach}_{c'}\ \{\ \overline{x \leftarrow e'}\ \}\ e'' : \mathscr{Q}(\sigma'^*)$
$\Delta;\Gamma \vdash e_i : \mathscr{Q}(\delta_i^*) \Rightarrow e'_i : \mathscr{Q}(\delta_i'''^*) \ _{i=1..n}$ H1.
$\Delta,\overline{x : \delta};\Gamma,\overline{x : \delta'} \vdash c : \mathsf{bool} \Rightarrow c' : \mathsf{bool}$ H2.
$\Delta,\overline{x : \delta};\Gamma,\overline{x : \delta''} \vdash e : \sigma \Rightarrow e'' : \sigma'$ H3.
$\overline{\delta''' \leq \delta'}$ H4.
$\overline{\delta''' \leq \delta''}$ H5.
$\overline{\delta''' \leq \delta}$
$\Delta \leq \Gamma$
$\Delta \vdash e'_i : \mathscr{Q}(\delta_i'''^*) \ _{i=1..n}$ H1 and IND,§1.
$\Delta,\overline{x : \delta} \leq \Gamma,\overline{x : \delta'}$

$\Delta, \overline{x : \delta} \vdash c' : \mathsf{bool}$ H2 and IND, §2.
$\Delta, \overline{x : \delta} \leq \Gamma, \overline{x : \delta''}$
$\Delta, \overline{x : \delta} \vdash e'' : \sigma'$ H3 and IND, §3.
$\Delta, \overline{x : \delta'''} \vdash c' : \mathsf{bool}$ §2 and WEAKENING
$\Delta, \overline{x : \delta'''} \vdash e'' : \sigma'$ §3 and WEAKENING
$\Delta \vdash \mathsf{foreach}_{c'} \{ \overline{x \leftarrow e'} \}\ e'' : \mathscr{Q}(\sigma'^*)$ SELECT.

Case Group.

$\Delta;\Gamma \vdash \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*) \Rightarrow {}^{\mathscr{Q}(\delta)}\pi^{\mathscr{Q}(\langle \overline{a:\sigma}, b:\tau^* \rangle^*)}(\mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \}) : \mathscr{Q}(\delta)$
$\langle \overline{a : \sigma}, b : \tau^* \rangle^* \leq \delta$ H1.
$\Delta, x : \tau; \Gamma, x : \tau_i' \vdash e_i : \sigma_i \Rightarrow e_i : \sigma_i \quad i=1..n$ H2.
$\Delta;\Gamma \vdash e : \mathscr{Q}(\tau^*) \Rightarrow e : \mathscr{Q}(\tau^*)$ H3.
$\Delta, x : \tau \vdash e_i : \sigma_i \quad i=1..n$ H2 and IND
$\Delta \vdash e : \mathscr{Q}(\tau^*)$ H3 and IND
$\Delta \vdash \mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \} : \mathscr{Q}(\langle \overline{a : \sigma}, b : \tau^* \rangle^*)$ (GROUP).
$\Delta \vdash {}^{\mathscr{Q}(\delta)}\pi^{\mathscr{Q}(\langle \overline{a:\sigma}, b:\tau^* \rangle^*)}(\mathsf{groupby}_b^{\overline{a=e}} \{ x \leftarrow e \}) : \mathscr{Q}(\delta)$ (PROJECTION).

Case Return.

$\Delta, \Gamma \vdash \mathsf{return}\ e : \mathscr{Q}(\tau) \Rightarrow \mathsf{return}\ e' : \mathscr{Q}(\sigma)$
$\Delta, \Gamma \vdash e : \tau \Rightarrow e' : \sigma$ H1.
$\tau \leq \sigma$
$\Delta \leq \Gamma$
$\Delta \vdash e' : \sigma$ IND.
$\Delta \vdash \mathsf{return}\ e' : \mathscr{Q}(\sigma)$ RETURN.

Case At.

$\Delta, \Gamma \vdash \mathsf{do}\ e_{\downarrow p}\{e'\} : \mathscr{Q}(\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\}) \Rightarrow \mathsf{do}\ e''_{\downarrow p}\{e'''\} : \mathscr{Q}(\delta)$
$\Delta, \Gamma \vdash e : \mathscr{Q}(\tau') \to \mathscr{Q}(\sigma') \Rightarrow e'' : \mathscr{Q}(\tau'') \to \mathscr{Q}(\sigma'')$ H1.
$\Delta \vdash e' : \mathscr{Q}(\tau) \Rightarrow e''' : \mathscr{Q}(\delta')$ H2.
$\delta = \delta'_{\downarrow p}\{^{\sigma''}/_{\tau''}\}$ H3.
$\Delta \vdash e'' : \mathscr{Q}(\tau'') \to \mathscr{Q}(\sigma'')$
$\mathscr{Q}(\tau') \to \mathscr{Q}(\sigma') \leq \mathscr{Q}(\tau'') \to \mathscr{Q}(\sigma'')$ H1 and IND
$\Delta \vdash e''' : \mathscr{Q}(\delta')$
$\tau \leq \delta'$ H2 and IND
$\sigma' \leq \sigma''$
$\tau'' \leq \tau$
$\Delta \vdash \mathsf{do}\ e''_{\downarrow p}\{e'''\} : \mathscr{Q}(\delta)$
$\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\} \leq \delta'_{\downarrow p}\{^{\sigma''}/_{\tau''}\}$ LEMMA 5.2.

Case Exec.

$\Delta;\Gamma \vdash \mathsf{exec}\ x : \sigma = e\ \mathsf{in}\ e' : \tau \Rightarrow \mathsf{exec}\ x : \sigma = e''\ \mathsf{in}\ e''' : \tau'$
$\Delta;\Gamma \vdash e : \mathscr{Q}(\sigma'') \Rightarrow e'' : \mathscr{Q}(\sigma)$ H1.
$\Delta, x : \sigma; \Gamma, x : \sigma' \vdash e' : \tau \Rightarrow e''' : \tau'$ H2.
$\sigma \leq \sigma'$

$\Delta \vdash e'' : \mathscr{Q}(\sigma)$ H1 and IND
$\tau \leq \tau'$
$\Delta, x : \sigma \vdash e''' : \tau'$ H2 and IND
$\Delta \vdash \mathsf{exec}\ x : \sigma = e''\ \mathsf{in}\ e''' : \tau'$ EXEC.

Case Project.
$\Delta;\Gamma \vdash {}^{\tau}\pi^{\sigma}(e) : \tau \Rightarrow e' : \tau'$
$\Delta;\Gamma \vdash e : \sigma \Rightarrow e' : \tau'$ H1.
$\tau \leq \tau'$ H2.
$\Delta \vdash e' : \tau'$ H1 and IND
□

Lemma 6.15 (Local Confluence). Relation $\rightsquigarrow$ is locally confluent for minimally distributed expressions: for any minimally distributed expressions e_ℓ, $e_{1\ell_1}$ and $e_{2\ell_2}$ such that $e_\ell \rightsquigarrow e_{1\ell_1}$ and $e_\ell \rightsquigarrow e_{2\ell_2}$, there exists an expression $e'_{\ell'}$ such that $e_{1\ell_1} \rightsquigarrow^* e'_{\ell'}$ and $e_{2\ell_2} \rightsquigarrow^* e'_{\ell'}$

Proof. By case analysis of the initial expression e and the possible e_1 and e_2 pairs. Note that:

- Cases where $e_1 = e_2$ are ommitted, since the lemma trivially holds.
- Cases where e_1 and e_2 are obtained by the reduction of independent sub-expressions are also ommitted, as confluence is just a matter of applying both reductions in any order .

Case $(\lambda x.e_\ell)_m$.

The possible reductions are:

1. $(\lambda x.e_\ell)_\ell$ if $\mathsf{has\_lambdas}(\ell)$ (by rule ($\rightsquigarrow$LAMBDA))
2. $(\lambda x.e'_{\ell'})_m$ if $e_\ell \rightsquigarrow e'_{\ell'}$

Analysis of the possible pairs:

- Pair of 1. and 2.:
  — If $\ell = \top$, then for the expression to be well-located, $m = \top$.

    $(\lambda x.e_\top)_\top \rightsquigarrow_{sub} (\lambda x.e'_{\ell'})_\top$

  — Otherwise if $\ell \sqsubset \top$, then by Lemma 6.9 we know that $\ell' = \ell$. Therefore $\mathsf{has\_lambdas}(\ell')$ holds, since $\mathsf{has\_lambdas}(\ell)$ is a premise.

    $(\lambda x.e_\ell)_\ell$
    $\rightsquigarrow_{sub} (\lambda x.e'_{\ell'})_\ell$
    $\rightsquigarrow_{Lambda} (\lambda x.e'_{\ell'})_{\ell'}$ if $\mathsf{has\_lambdas}(\ell')$

    $(\lambda x.e'_{\ell'})_m$
    $\rightsquigarrow_{Lambda} (\lambda x.e'_{\ell'})_{\ell'}$ if $\mathsf{has\_lambdas}(\ell')$

Case $e_{1\ell_1}\ \mathrm{op}_m\ e_{2\ell_2}$.

The possible reductions are:

1. $e_{1\ell_1}\ \mathrm{op}_{\ell_1\sqcap\ell_2}\ e_{2\ell_2}$ if $\mathsf{can\_op}(\ell_1\sqcap\ell_2)$ (by rule ($\rightsquigarrow$OP))
2. $e'_{1\ell'_1}\ \mathrm{op}_m\ e_{2\ell_2}$ if $e_{1\ell_1}\rightsquigarrow e'_{1\ell'_1}$
3. $e_{1\ell_1}\ \mathrm{op}_m\ e'_{2\ell'_2}$ if $e_{2\ell_2}\rightsquigarrow e'_{2\ell'_2}$

Analysis of the possible pairs:

- Pair of 1. and 2.:
  — If $\ell_1\sqcap\ell_2=\top$, then for the expression to be well-located, $m=\top$.

    $$e_{1\ell_1}\ \mathrm{op}_\top\ e_{2\ell_2}\rightsquigarrow_{sub} e'_{1\ell'_1}\ \mathrm{op}_\top\ e_{2\ell_2}$$

  — Otherwise if $\ell_1\sqcap\ell_2\sqsubset\top$, then $\ell_1\sqsubset\top$ and by Lemma 6.9 we know that $\ell'_1=\ell_1$. Therefore $\ell'_1\sqcap\ell_2=\ell_1\sqcap\ell_2$, implying $\mathsf{can\_op}(\ell'_1\sqcap\ell_2)$ holds.

    $$\begin{array}{l} e_{1\ell_1}\ \mathrm{op}_{\ell_1\sqcap\ell_2}\ e_{2\ell_2}\\ \rightsquigarrow_{sub} e'_{1\ell'_1}\ \mathrm{op}_{\ell_1\sqcap\ell_2}\ e_{2\ell_2}\\ \rightsquigarrow_{Op} e'_{1\ell'_1}\ \mathrm{op}_{\ell'_1\sqcap\ell_2}\ e_{2\ell_2}\quad \text{if } \mathsf{can\_op}(\ell'_1\sqcap\ell_2)\end{array}$$

    $$\begin{array}{l} e'_{1\ell'_1}\ \mathrm{op}_m\ e_{2\ell_2}\\ \rightsquigarrow_{Op} e'_{1\ell'_1}\ \mathrm{op}_{\ell'_1\sqcap\ell_2}\ e_{2\ell_2}\quad \text{if } \mathsf{can\_op}(\ell'_1\sqcap\ell_2)\end{array}$$

- Pair of 1. and 3.: similar to the previous.

Cases $(e_\ell\ e'_{\ell'})_m$, $\langle\overline{a=e_\ell}\rangle_m$, $e_\ell.a_m$, $(\mathsf{return}\ \ e_\ell)_m$ or $(\mathsf{groupby}_b^{\overline{a=e_\ell}}\{\ x\leftarrow e_{\ell'}\ \})_m$, $[e_\ell]_m$.

The same proof strategy as the previous cases applies.

Case $\mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e_{n\ell_n})_m$.

The possible reductions are:

1. $\mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e_{n\ell_n})_{\sqcap\overline{\ell}\sqcap\ell}$
2. $\mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e'_{i\ell'_i},\ldots,e_{n\ell_n})_m$ if $e_{i\ell_i}\rightsquigarrow e'_{i\ell'_i}$

Analysis of the possible pairs:

- Pair of 1. and 2.:

  $$\begin{array}{l} \mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e_{n\ell_n})_{\sqcap\overline{\ell}\sqcap\ell}\\ \rightsquigarrow_{sub} \mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e'_{i\ell'_i},\ldots,e_{n\ell_n})_{\sqcap\overline{\ell}\sqcap\ell}\\ \rightsquigarrow_{Source} \mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e'_{i\ell'_i},\ldots,e_{n\ell_n})_{\ell_1\sqcap\ldots\sqcap\ell'_i\sqcap\ldots\sqcap\ell_n\sqcap\ell}\end{array}$$

  $$\begin{array}{l} \mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e'_{i\ell'_i},\ldots,e_{n\ell_n})_m\\ \rightsquigarrow_{Source} \mathsf{db}_\ell(t,e_{1\ell_1},\ldots,e'_{i\ell'_i},\ldots,e_{n\ell_n})_{\ell_1\sqcap\ldots\sqcap\ell'_i\sqcap\ldots\sqcap\ell_n\sqcap\ell}\end{array}$$

Case ${}^{\tau}\pi^{\sigma}_{m}(e_\ell)$.

The possible reductions are:

1. ${}^{\tau}\pi^{\sigma}_{\ell}(e_\ell)$ if $\mathsf{can\_project}(\ell)$
2. ${}^{\tau}\pi^{\sigma}_{m}(e'_{\ell'})$ if $e_\ell \rightsquigarrow e'_{\ell'}$
3. ${}^{\tau}\pi^{\sigma}_{m}(e_{\sqcap\overline{\ell}\sqcap\ell_g})$ if $e = \mathsf{groupby}^{\overline{a=e_\ell}}_{b}\{\, x \leftarrow e_{g\ell_g} \,\} \wedge \mathsf{can\_group}(\sqcap\overline{\ell}\sqcap\ell_g) \wedge \tau = \cdots \wedge \sigma = \ldots$

Analysis of the possible pairs:

- Pairs of 1. and 2., and 1. and 3. are proven using the same proof strategy as previous expressions.

- Pair of 2. and 3. is proven by case analysis of $e_\ell \rightsquigarrow e'_{\ell'}$:
  1. If e_ℓ reduces by rule ($\rightsquigarrow$NESTING), $e' = e$ and $\ell' = \sqcap\overline{\ell}\sqcap\ell_g$, implying 2. and 3. are already the same.
  2. If e_ℓ reduces by $e_{g\ell_g} \rightsquigarrow e'_{g\ell'_g}$, we proceed by case analysis of ℓ_g, using the same proof strategy as previous expressions.
  3. If e_ℓ reduces by $e_{i\ell_i} \rightsquigarrow e'_{i\ell'_i}$, we proceed by case analysis of ℓ_i, using the same proof strategy as previous expressions.

Case $(\mathsf{foreach}_{c_n}\left\{\, \overline{x \leftarrow e_\ell}^{\,k}_{1}, \overline{(x \leftarrow e_\ell, c_n)_\ell} \,\right\} e_\ell)_m$.

The possible reductions are:

1. $(\mathsf{foreach}_{c_n}\left\{\, (\overline{x \leftarrow e_\ell}, c_{n_i})_{\ell_i}, \overline{(x \leftarrow e_\ell, c_n)_\ell} \,\right\} e_\ell)_{\ell_i\sqcap\ell}$ if $k = 0$
2. $(\mathsf{foreach}_{d_{2n_2}}\left\{\, \overline{x \leftarrow e_\ell}^{\,k}_{1}, \overline{(x \leftarrow e_\ell, c_n)_\ell}, (x \leftarrow e_\ell, c_{j_{n_j}} \wedge d_{1n_1})_{\ell_j} \,\right\} e_\ell)_m$
   if $c_n = d_{1n_1} \wedge d_{2n_2}$
3. $(\mathsf{foreach}_{c_n}\left\{\, \overline{x \leftarrow e_\ell}^{\,k-1}_{1}, \overline{(x \leftarrow e_\ell, c_n)_\ell}, (x_k \leftarrow e_{k\ell_j}, \overline{x \leftarrow e_\ell}, c_{j_{n_j}})_{\ell_j} \,\right\} e_\ell)_m$
   if $\ell_k = \ell_j$
4. $(\mathsf{foreach}_{c_n}\left\{\, \overline{x \leftarrow e_\ell}^{\,k}_{1}, \overline{(x \leftarrow e_\ell, c_n)_\ell} \,\right\} e'_{\ell'})_m$ if $e_\ell \rightsquigarrow e'_{\ell'}$
5. $(\mathsf{foreach}_{c'_{n'}}\left\{\, \overline{x \leftarrow e_\ell}^{\,k}_{1}, \overline{(x \leftarrow e_\ell, c_n)_\ell} \,\right\} e_\ell)_m$ if $c_n \rightsquigarrow c'_{n'}$
6. $(\mathsf{foreach}_{c_n}\left\{\, \overline{x \leftarrow e_\ell}^{\,i-1}_{1}, x_i \leftarrow e'_{i\ell'_i}, \overline{x \leftarrow e_\ell}^{\,k}_{i+1}, \overline{(x \leftarrow e_\ell, c_n)_\ell} \,\right\} e_\ell)_m$ if $e_{i\ell_i} \rightsquigarrow e'_{i\ell'_i}$
7. $(\mathsf{foreach}_{c_n}\left\{\, \overline{x \leftarrow e_\ell}^{\,k}_{1}, \overline{(x \leftarrow e_\ell, c_n)_\ell}, (\overline{x \leftarrow e_\ell}, c'_{j_{n'_j}})_{\ell_j} \,\right\} e_\ell)_m$ if $c_{j_{n_j}} \rightsquigarrow c'_{j_{n'_j}}$
8. Reduction of sub-expressions inside grouped binders (similar to the above).

Analysis of the possible pairs:

- Case 1. and 3.: impossible because 1. requires $k = 0$ and 3. requires $k > 0$.
- Case 1. and sub-expression cases: 1. doesn't change sub-expressions so they can always reduce, and the only sub-expression that affects the result of 1. is e_ℓ, which we can show using the usual proof strategy won't affect the final $\ell_i \sqcap \ell$.

- Case 3. and sub-expression cases: 3. doesn't change locations, and doesn't change sub-expressions other than moving the outer condition $c_{\ell'}$. Moving the outer condition doesn't impact it's own reduction.

□