JFP **27**, e26, 36 pages, 2017. © Cambridge University Press 2017
doi:10.1017/S095679681700017X

Capability-based localization of distributed and heterogeneous queries

JOÃO COSTA SECO
NOVA LINCS – Universidade Nova de Lisboa, Lisboa, Portugal
PAULO FERREIRA and HUGO LOURENÇO
OutSystems, Lisboa, Portugal
(*e-mails:* 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, especially 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, which 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), parametrized data repository (web services), or in-memory data, by a set of capabilities (e.g., to join collections, group-by arbitrary expressions, nest results and 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 focuses on the decentralization of parts of a query as in Wong (2000), but in a way that is guided by the internal capabilities of each site, and that includes their specialization according to data usage. We intentionally view this as orthogonal and composable in a pipeline of optimizations to other processes that seek the parallel execution of independent query parts, or that reorder operations by analysing developer hints or data profiles, which may not be available in such heterogeneous environments.

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.

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

We refine the query transformation process presented in Seco *et al.* (2015) by decomposing it into three separate phases, and analysing 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 an 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 a companion technical report (Seco *et al.*, 2017).

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 as 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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

Fig. 1. Syntax of expressions.

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 parametrized 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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

an explicit projection operation ${}^{\tau}\pi^{\sigma}(e)$, from type σ to type τ that is 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, from query to query, denoted by expression e to the sub-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 is designed to allow for query rewriting manipulations that simplify it and transport operations closer to the specified location. These optimizations can be considered together with other operations, and can also be compiled into imperative style query languages, such as the ones found in No-SQL data stores (e.g., local storage with indexedDB, manipulated by JavaScript programs).

We define queries as logically separated values, described by types of the form $\mathcal{Q}(\tau)$, 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 $\mathcal{Q}(\tau)$. 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 2–4, whose schemas are as follows:

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

id	name
1	Alpha
2	Bravo
3	Charlie

Fig. 2. Teams – SALESDB.

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

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.

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 $\tau_L = \langle lat : \mathsf{num},\ lng : \mathsf{num} \rangle$

A developer wants to list 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, Figure 5, using a foreach expression, a basic filter and a record constructor expression:

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

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,

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

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.

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 useful in the scenario of 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 modifying the initial query to include the new column. It is also suited for a visual manipulation environment where a user interface can be designed to naturally define 'at' operations, by pointing to the displayed data. Notice that the path */details* refers to a sub-query whose results is a list, and hence, we need a foreach expressions to join new data to all its elements.

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

$withLoc = \mathsf{do}\ addLoc_{\downarrow/details}\{workByTeam\}$ 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.

The complete, expanded, query is the following:

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

The query refers to multiple data sources, located at different sites (`SALESDB`, `SAP`, `GEO`). Intuitively, 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 performed in the caller site, using in-memory data structures, since the data comes from location `SAP`, a different database server. The `Coords` web-service must also be called and its results processed in memory by the query starting location, after the remaining results are fetched and explicitly joined. 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 and data transmission time can be spared.

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

4 Semantics of λ_{CDL}

The semantics for λ_{CDL} is inductively defined with relation to a state ($\mathscr{S}$) that contains all the referred data repositories. We define $\langle\!\langle e \rangle\!\rangle$, a function that denotes the computed value of an expression e, using the cases in Figures 9 and 10. The syntax of values is defined by the grammar in Figure 8. The evaluation of query expressions corresponds to the staging of queries, that are afterwards executed with relation to

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

Fig. 8. Language values.

$$
\begin{array}{lll}
\langle\!\langle v \rangle\!\rangle = v & & \\
\langle\!\langle e\ \mathrm{op}\ e' \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle\ \mathrm{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'' & \\
 & \quad\ \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 & \\
 & \quad\ \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 : \tau = e\ \mathsf{in}\ e' \rangle\!\rangle = \langle\!\langle e'\{^{v}/_{x}\} \rangle\!\rangle & \text{where } \langle\!\langle e \rangle\!\rangle = r & \\
 & \quad\ [\![r]\!] = v &
\end{array}
$$

Fig. 9. Semantics of expressions.

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 functional approach to

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

$$\langle\!\langle \mathsf{db}_\ell(t,\overline{e}) \rangle\!\rangle = \mathsf{db}_\ell(t,\overline{\langle\!\langle e \rangle\!\rangle})$$

$$\langle\!\langle \mathsf{foreach}_c\ \{\ \overline{x \leftarrow e}\ \}\ e' \rangle\!\rangle = \mathsf{foreach}_c \left\{ \overline{x \leftarrow \langle\!\langle e \rangle\!\rangle} \right\}\ e'$$

$$\langle\!\langle \mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow e'\ \} \rangle\!\rangle = \mathsf{groupby}_b^{\overline{a=e}}\{\ x \leftarrow \langle\!\langle e' \rangle\!\rangle\ \}$$

$$\langle\!\langle \mathsf{do}\ e_{\downarrow p}\{e'\} \rangle\!\rangle = \mathsf{do}\ \langle\!\langle e \rangle\!\rangle_{\downarrow p}\{\langle\!\langle e' \rangle\!\rangle\}$$

$$\langle\!\langle \mathsf{return}\ e \rangle\!\rangle = \mathsf{return}\ \langle\!\langle e \rangle\!\rangle$$

$$\langle\!\langle [\ e\ ]_\ell \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle$$

Fig. 10. Semantics of query expressions.

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

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

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

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

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

$$[\![\mathsf{return}\ v]\!] = v$$

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

Fig. 11. Semantics of query values.

defining the semantics is inspired by works like (Buneman *et al.*, 1994, 1995; Peyton 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 describe the cases of non-standard constructs. For instance, we resort to a native definition of the semantics for predefined operations (op) on values of basic types. 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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

exec $x : \tau = e$ 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). We use this expression as an extension point where to introduce the typed compilation procedure that transforms queries before actually executing them.

The language fragment that represents 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. The semantics of executing query values (Figure 11) states that a data source invocation ($\mathsf{db}_\ell(t, \bar{v})$) 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 that 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. Projection operations (${}^{\tau}\pi^{\sigma}(r)$) convert the provided value to the target type τ, removing the (possibly nested) fields that are no longer present. The return operation (return) simply yields its inner value, and the remote evaluation ($[\![\, r \,]\!]_\ell$) sends the query r to be executed in the location ℓ.

The semantics of operations of the form 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

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

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. The typing relation for λ_{CDL} is defined inductively in terms of the judgement $\Delta \vdash e : \tau$, according to the rules in Figure 12. We focus mainly on the rules for typing queries, as the 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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 12. Typing relation.

by the typing environment Δ. The data source access expression is typed as a query that returns values of the prescribed type in its 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 the base cases of typing, they allow that any expression can be used in the context of a query, rule (Return). Rule (At) types an operation that is applied, in-place, deep in the

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 type transformation in-place, called *type-at*, and defined below

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:

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

Since the 'at' operation applies a query transformation of type $\mathcal{Q}(\tau') \to \mathcal{Q}(\sigma')$ on a given path, the *type-at* operation on types $\tau_{\downarrow p}\{^{\sigma'}/_{\tau'}\}$ perform the corresponding 'deep' transformation of the target query type. The type transformation is only defined in cases where there is a perfect match at the target subtype pointed by path p. To better represent the intuition of staged query values, we designed the 'at' operation as a query transformation function, from query to query, as an alternative to the more general form $\tau' \to \mathcal{Q}(\sigma')$, targeting the common use cases of deeply nested filter, join and group-by operations in an interactive query construction environment. In future work, this will allow us to extend the system with optimization procedures that rewrite and reorder operations. This is still not part of the present semantics where inner queries are evaluated first and the operation is applied deep into the structure of the query results.

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):

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

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 2–7, 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` :

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

$\mathsf{string} \to \tau_L\}$, with τ_T, τ_J, τ_C and τ_L as defined in Section 3. Then, we have

- $\Delta \vdash teams : \mathcal{2}(\tau_T^*)$
- $\Delta \vdash jobs : \mathcal{2}(\tau_J^*)$
- $\Delta \vdash clients : \mathcal{2}(\tau_C^*)$
- $\Delta \vdash work : \mathcal{2}(\langle team : \tau_T, job : \tau_J, client : \tau_C\rangle^*)$
- $\Delta \vdash workByTeam : \mathcal{2}(\langle name : string, details : \langle team : \tau_T, job : \tau_J, client : \tau_C\rangle^*\rangle^*)$
- $\Delta \vdash withLoc : \mathcal{2}(\langle name : string, details : \langle team : \tau_T, job : \tau_J, client : \tau_C, loc : \tau_L\rangle^*\rangle^*)$

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 $\tau \leqslant \sigma$ and $\delta' \leqslant \delta''$, then $\tau_{\downarrow p}\{^{\delta'}/_{\delta}\} \leqslant \sigma_{\downarrow p}\{^{\delta''}/_{\delta}\}$.

Proof. According to 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 $\Delta' \leqslant \Delta$, if and only if $Dom(\Delta') = Dom(\Delta)$, $\forall_{y \in Dom(\Delta)}.\ \Delta'(y) \leqslant \Delta(y)$.

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 $\tau' \leqslant \tau$, and $\Delta' \leqslant \Delta$.

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 $\tau' \leqslant \tau$.
2. If $\Delta \vdash r : \mathcal{2}(\tau)$ and $[\![r]\!] = v$ then $\Delta \vdash v : \tau'$ with $\tau' \leqslant \tau$.

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 Seco *et al.* (2017) 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,

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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; Grade *et al*., 2013), the number of locations visited, the local indexing structures on each location (Wong, 2000) 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 further optimize and efficiently execute it (e.g., Grade *et al*., 2003; Taylor, 2010). We also identify each remote location as containing different data sources and capabilities, which rules out optimizations that seek to parallelize (similar) parts of a query to different nodes.

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 sub-queries.

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

in remote nodes. This step is orthogonal to our current focus, and should be easily incorporated as an extra query processing phase in our pipeline.

6.1 Phase I: Usage-based projection

The first step in 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 judgement,

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

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, 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 Lemma 6.2. 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 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 (π-Singleton) and (π-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 (π-Source), a projection is always inserted, although it can be compiled to code when transformed into native query code. Rule (π-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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 13. Type directed projection transformation (I).

judgement 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. Given that our type system does not include a $\top$ type, we have no specific projection transformation for queries bound to a cursor that is not used in both the select and condition expressions. In rule (π-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 (π-AT), influences both the abstraction being applied and the target

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

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

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

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

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

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

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

Fig. 14. Type directed projection transformation (II).

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 (π-EXEC) must use the annotated type as guide for the usage, with the compliance of both query and continuation expressions being 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). Notice that since the annotated type is invariant in the rule, we are effectively blocking optimization opportunities arising from x not being bound in e'. The same limitation applies to rule (π-ABSTRACTION). Technically, a stronger result of typing environment contraction is needed to drop the annotation in the expression, thus allowing optimization of those scenarios. Rule (π-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 $\tau \leqslant \sigma$, $\Delta \leqslant \Gamma$, and $\Delta \vdash e' : \sigma$.

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 Seco *et al.* (2017). □

Lemma 6.3 (*Semantic preservation – phase 1*)
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 leads to a trimmed down version of the resulting data, while maintaining the soundness of the original program. Namely, it preserves all intermediate information not present in the final result but required to compute it. 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

$$addLoc = \lambda x.\mathsf{foreach}\ \{\ y \leftarrow x\ \} \\ (y \oplus \langle loc = \mathsf{run}\ \mathsf{db}_{\mathtt{GEO}}(\mathtt{Coords}, y.client.address)\rangle)$$

$$withLoc = \mathsf{do}\ addLoc_{\downarrow/details}\{workByTeam\}$$

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

$$\tau = \langle name : string, details : \tau_d^* \rangle$$
$$\tau_d = \langle team : \tau_T, job : \tau_J, client : \tau_C, loc : \tau_L \rangle$$

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

$$\sigma = \langle name : string, details : \sigma_d^* \rangle$$
$$\sigma_d = \langle team : \tau_T, job : \tau_J, client : \tau_C \rangle$$

It is possible to observe that by (1) applying (π-Id), (π-Record) and (π-Concat), then (2) (π-Select) and (π-Abstraction) and finally (3) (π-Group) and (π-At):

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

where

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

$$withLoc' = \mathsf{do}\ addLoc'_{\downarrow/details}\{workByTeam\}$$

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 in the starting location, 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 is an interesting extension of this model 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 domain-specific language (DSL) (OutSystems, 2016) for data manipulation, and is adapted to the set of operations that is included in the language. We say that proposition `can_group`(ℓ) 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 `can_nestgroups`(ℓ) holds for locations (ℓ) running database engines that 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 `can_join`(ℓ) states that the database repository at location ℓ supports the joining of two (or more) sources given a condition, and `can_iterate`(ℓ) indicates that it supports the iteration of a list and the computing of a given expression on all elements of a query. Predicates `can_lambda`(ℓ) and `can_call`(ℓ) refer to the definition and use of abstractions. As for predicates `can_createrecords`(ℓ) refers to handling of record expressions, and `can_createlists`(ℓ) 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 labelled expressions $e_\ell \in \mathscr{E}$, where e is an expression

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

given by the syntax of Figure 1, where each subexpression is labelled with a location from $\mathscr{L} \cup \{\top, \bot\}$. We then define a localization system on labelled 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 labelled 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. Precisely because the localization rules do not change the query structure, queries of similar semantics or intention will possibly result in a different number of remote calls to the same sites, depending on their actual structure. For instance, nesting iteration operations with sources in different sites will partition the application of the localization rules, limiting their scope. Using 'at' operations also creates a localization silo inside their query transformation functions, since they are always executed *in-memory*. We foresee the combination of our rewriting system with a query normalization technique, that for instance eliminates unnecessary 'at' operations, similar to the one in Cheney *et al.* (2013), as a possible solution to be explored in future work.

The rewriting system starts in a given initial state defined below, and runs until a fix point is reached. We prove ahead that this relation is confluent, and hence the localization system always terminates. Literals are labelled with location $\bot$ in the initial state, meaning that they can be computed in any location. The remaining expression nodes are labelled 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 labelled 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 labelling 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:

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

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 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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

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

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

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

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

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

Fig. 15. Location rewriting rules (I).

transient form is compiled back to the original representation in the third step of our query transformation process (Section 6.3) to

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

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 labelling of locations to expressions is defined as follows.

Definition 6.5 (*Initial expression labelling*)

1. Literals are labelled with location $\bot$.
2. All other expressions are labelled 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 $\mathsf{true}_\bot$.

Given the initial state for location labelled expressions, we introduce the rewriting rules in Figures 15 and 16, where operations such as record concatenation and list append are encoded in the case of the general operation op. 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.

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

Fig. 16. Location rewriting rules (II).

rules propagate the locations of data sources from the expression leafs, up the abstract syntax tree, according to the capabilities of remote sites. Rules (⇝PROJECT), (⇝SINGLETON) and (⇝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 (⇝BINDER) encodes the actual grouping of binders, while rule (⇝FILTER) distributes filter conditions in the foreach expression

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

according to the locations of the bound cursors and their usage. A fully localized binder list is finally captured by rule (⇝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 (⇝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 (⇝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 (⇝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 (⇝GROUP-CURSOR) and (⇝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 establish 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 labelled 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 labelled expression is *minimally distributed*, according to Definition 6.7, as stated in Lemma 6.8.

Lemma 6.8 (*Minimally distributed initial labelling*)
Initially labelled 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 labelled with $\bot$ in the initial state, and all other expressions are labelled 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 $e_\bot$, we have an empty set of strict subexpressions, and the join of an empty set is $\bot$, 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).

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 the case analysis of the possible rewriting reductions, leveraging the fact that rules that change m do not 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 (⇝Filter) does not change m and only moves its strict sub-expressions. Rules (⇝Binder) and (⇝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 (⇝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 are: 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 (⇝Op) to (⇝Return), (⇝Select), (⇝Nesting) and (⇝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 (⇝Binder) and (⇝Filter), the distance measure stays the same but the number of

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 the 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) do not 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 that 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 it is 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 do not change e, and forget it in favour 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. □

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

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

The localization of this query would begin by labelling it according to Definition 6.5:

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

$$\begin{aligned} \text{where } \mathit{filter} &= (t_{\top}.\mathit{id}_{\top} = j_{\top}.\mathit{teamId}_{\top})_{\top} \\ &\quad \wedge (j_{\top}.\mathit{clientId}_{\top} = c_{\top}.\mathit{id}_{\top})_{\top} \\ &\quad \wedge (j_{\top}.\mathit{date}_{\top} = 8/5_{\bot})_{\top} \\ \mathit{select} &= \langle \mathit{team} = t_{\top},\ \mathit{job} = j_{\top},\ \mathit{client} = c_{\top} \rangle_{\top} \end{aligned}$$

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:

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

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:

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

$$\begin{aligned} \text{where } \mathit{filter} &= (t_{\mathtt{SALESDB}}.\mathit{id}_{\top} = j_{\mathtt{SALESDB}}.\mathit{teamId}_{\top})_{\top} \\ &\quad \wedge (j_{\mathtt{SALESDB}}.\mathit{clientId}_{\top} = c_{\mathtt{SAP}}.\mathit{id}_{\top})_{\top} \\ &\quad \wedge (j_{\mathtt{SALESDB}}.\mathit{date}_{\top} = 8/5_{\bot})_{\top} \\ \mathit{select} &= \langle \mathit{team} = t_{\mathtt{SALESDB}},\ \mathit{job} = j_{\mathtt{SALESDB}},\ \mathit{client} = c_{\mathtt{SAP}} \rangle_{\top} \end{aligned}$$

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

$$\begin{aligned} &(t.\mathit{id}_{\mathtt{SALESDB}} = j.\mathit{teamId}_{\mathtt{SALESDB}})_{\mathtt{SALESDB}} \\ &\wedge (j.\mathit{clientId}_{\mathtt{SALESDB}} = c.\mathit{id}_{\mathtt{SAP}})_{\top} \\ &\wedge (j.\mathit{date}_{\mathtt{SALESDB}} = 8/5_{\bot})_{\mathtt{SALESDB}} \end{aligned}$$

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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

$$\mathsf{foreach}_{filter}\left\{\begin{array}{l}\left(\begin{array}{l}t \leftarrow \mathtt{db}_{\mathtt{SALESDB}}(\mathtt{Team})_{\mathtt{SALESDB}},\\ j \leftarrow \mathtt{db}_{\mathtt{SALESDB}}(\mathtt{Job})_{\mathtt{SALESDB}},\\ (t.id_{\mathtt{SALESDB}} = j.teamId_{\mathtt{SALESDB}})_{\mathtt{SALESDB}}\\ \wedge\ (j.date_{\mathtt{SALESDB}} = 8/5_{\bot})_{\mathtt{SALESDB}}\end{array}\right)_{\mathtt{SALESDB}}\\[2ex] \left(\begin{array}{l}c \leftarrow \mathtt{db}_{\mathtt{SAP}}(\mathtt{Client})_{\mathtt{SAP}},\\ \mathsf{true}_{\bot}\end{array}\right)_{\mathtt{SAP}}\end{array}\right\} select$$

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 labelled query, so we reached the normal form of the query expression. If we were localizing the full *withLoc* query introduced in Section 3, we would 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 labelled expression format back to the regular syntax, while ensuring that all operations are indeed evaluated at the most appropriate location.

The labelled (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 that all transitions to a different location (other than $\bot$) are enclosed by a remote execution expression, and that a transition from a location ℓ to $\bot$ is ignored, as expressions labelled at $\bot$ 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 switch the current execution location according to the expression's annotation:

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

$$
\begin{array}{rcl}
(\!|\, num_{\bot} \,|\!)_{\ell} & \triangleq & num \\
(\!|\, bool_{\bot} \,|\!)_{\ell} & \triangleq & bool \\
(\!|\, string_{\bot} \,|\!)_{\ell} & \triangleq & string \\
(\!|\, date_{\bot} \,|\!)_{\ell} & \triangleq & date \\
(\!|\, \emptyset_{\bot} \,|\!)_{\ell} & \triangleq & \emptyset \\
(\!|\, (e_{\ell'} \ \mathrm{op}\ e'_{\ell''})_m \,|\!)_{\ell} & \triangleq & ((\!|\, e_{\ell'} \,|\!)_m \ \mathrm{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/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 labelling erasure function.

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

Fig. 18. Location labelling 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 (Lemmas 6.2 and 6.10). Phase III maintains the whole structure of the query, which does not raise any soundness issue. To ensure the whole process is sound, it would

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

```
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.

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

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

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 are 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 the `SAP` database, e.g., by introducing an 'IN' condition.

7 Related work

Unlike many DSLs for the development of complete data-centric applications (Cooper *et al*., 2007; Fu *et al*., 2013), 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 parametrizable 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 (Benzaken *et al*., 2003; Robie *et al*., 2014). We introduce the uniform and compositional mechanism of in-place modifications that applies to all kinds of repositories, is suitable to query simplification and can be compiled into efficient imperative code that manipulates data by reference (e.g., IndexedDB).

Native capabilities of data repositories have been used to generate queries in heterogeneous environments. 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 is not in the category of semantic-based code generation tools.

Related work includes systems that integrate, behind a single interface, several data-based systems (e.g., Halevy *et al*. 2006), acting as a middleware layer. We

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

```
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.

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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 a high-level query language for nested collections and optimized distribution of queries to external data sources. Our work also provides a single query language over the participating data sources, and deals with the problem of maximizing the distribution of query subexpressions. However, we shape the optimized query along different optimization axes such as the capabilities of target locations and usage type, and not towards execution strategies like lazy and concurrent evaluation. Furthermore, our principled description of location capabilities allows for the uniform and extensible treatment of sub-queries and sub-expressions across the whole query. To the extent of our understanding, in Wong (2000), the use of specific drivers to different database systems assumes that all target database systems are treated alike, with the same capabilities. Also, in many of the examples shown, the developer still needs to be aware of the specific query languages being integrated (SQL).

There are similar aspects between the location modalities in ML5 (Murphy VII *et al*., 2008) and the inferred labelling of expressions that we implement. Our work is mainly focused on an eager algorithm for location inference, shaped by the set of capabilities exposed on each remote location. It is clear that we can directly encode the location of data sources in ML5, and that several aspects match directly, as the "shamrock" modality with our labelling matches with the $\perp$ location. However, we believe that the specialized constraints on language capabilities we introduce would only be possible in an extended ML5 setting where language operations are tied to locations by typing axioms.

It is foreseeable that the formal framework provided by general models for distributed computing, such as the Ambient Calculus (Cardelli *et al*., 2002) or the modal logic behind ML5 programming languages, can be extended to prove a stronger soundness property of our approach at a meta-level, ensuring that code transformations always produce queries that only use the capabilities available in the locations where the data collection and manipulation operations are set to be executed. We leave this study for a subsequent work. Note that in our setting any unlabelled well-typed query is correct with relation to this invariant, data source operations are executed remotely and the manipulation of results occurs in-memory at the $\top$ location.

8 Final remarks

Querying data in heterogeneous and distributed environments is arguably a skilful task, when building all kinds of service and data-centric applications. We believe that it is crucial to introduce new data manipulation languages for nested collections that allow the orchestration and abstraction of a number heterogeneous data sources

https://doi.org/10.1017/S095679681700017X Published online by Cambridge University Press

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, on each particular use of a query. It is not feasible for a middleware system, containing predefined queries, to automatically adapt to any possible use of the query resulting data. We further seek an associate optimized query execution plan, obtained by discharging as much work as possible to external systems. Notice that our focus is not on the traditional (vertical) reordering of query operations, which we find to be orthogonal to our approach and whose combination can only increase the efficiency of queries even further. Nevertheless, we find the maximal grouping of partial join operations in terms of location within a single existing node. Also, the efficient, optimized and adaptive remote invocation of query parts can be orchestrated orthogonally (Grade *et al*., 2013).

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 sequence 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. Also, the application of a more general normalization technique, like the one in Cheney *et al*. (2013), to queries can lead to better results in terms of optimizing query execution time, and to making the localization more agnostic to the particular query structure. 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. 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 that allowed the improvement of this work. João Costa Seco is supported by NOVA LINCS ref. UID/CEC/04516/2013 and PTDC/EEI-CTP/4293/2014.

References

Benzaken, V., Castagna, G. & Frisch, A. (2003) CDuce: An XML-centric general-purpose language. In Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP '03). New York, NY, USA: ACM, pp. 51–63.

Buneman, P., Libkin, L., Suciu, D., Tannen, V., & Wong, L. (1994) Comprehension syntax. *ACM SIGMOD Rec.* **23**(1), 87–96.

Buneman, P., Naqvi, S., Tannen, V., & Wong, L. (1995) Principles of programming with complex objects and collection types. *Theor. Comput. Sci.* **149**(1), 3–48.

Cardelli, L. (1989) Typeful programming. In *IFIP State of the Art Reports (Formal Description of Programming Concepts)*, Neuhold, Erich J. & Paul, M. (eds). New York, NY, USA: Springer-Verlag, pp. 431–507.

Cardelli, L., Ghelli, G., & Gordon, A. D. (2002) Types for the ambient calculus. *Inform. Comput.* **177**(2), 160–194.

Cheney, J., Lindley, S. & Wadler, P. (2013) A practical theory of language-integrated query. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13), New York, NY, USA: ACM, pp. 403–416.

Cheney, J., Lindley, S., & Wadler, P. (2014) Query shredding: Efficient relational evaluation of queries over nested multisets. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14). New York, NY, USA: ACM, pp. 1027–1038.

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

Clark, J., & DeRose, S. J. (1999) Path Language (XPath) Version 1.0. Available at: `www.w3.org/TR/xpath/`

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

Cooper, E. (2009) The Script-Writer's dream: How to write great SQL in your own language, and be sure it will succeed. In Proceedings of the 12th International Symposium on Database Programming Languages (DBPL '09). Berlin, Heidelberg: Springer-Verlag, pp. 36–51.

Cooper, E., Lindley, S., Wadler, P., & Yallop, J. (2007) Links: Web programming without tiers. In Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO '06). Berlin, Heidelberg: Springer-Verlag, pp. 266–296.

Davies, R., & Pfenning, F. (2001) A modal analysis of staged computation. *J. ACM* **48**(3), 555–604.

Fu, Y., Ong, K. W., & Papakonstantinou, Y. (2013) Declarative ajax web applications through SQL++ on a unified application state. In Proceedings of the International Symposium on Database Programming Languages.

Grade, N., Ferrão, L., & Seco, J. C. (2013) Optimizing data queries over heterogeneous sources. In Proceedings of the 5th Simpósio de Informática, INForum.

Halevy, A., Rajaraman, A. & Ordille, J. (2006) Data integration: The teenage years. In Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06). VLDB Endowment, pp. 9–16.

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

Murphy VII, T., Crary, K. & Harper, R. (2008) Type-safe distributed programming with ML5. In *Trustworthy Global Computing: Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, Barthe, G. & Fournet, C. (eds). Berlin, Heidelberg: Springer, pp. 108–123.

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

OutSystems. (2016) Using aggregates – Fetch and display data from the database. Technical documentation. Available at: `www.outsystems.com/search/Fetch+and+Display+Data+from+the+Database`

Papakonstantinou, Y., Gupta, A. & Haas, L. (1998) Capabilities-based query rewriting in mediator systems. *Distrib.Parallel Databases* **6**(1), 73–110.

Peyton Jones, S. & Wadler, P. (2007) Comprehensive comprehensions. In Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07). ACM, pp. 61–72.

Robie, J. *et al.* (2014) XQuery 3.0: An XML query language. Available at: `www.w3.org/TR/xquery-30/`

Seco, J. C., Lourenço, H., & Ferreira, P. (2015) A common data manipulation language for nested data in heterogeneous environments. In Proceedings of the 15th Symposium on Database Programming Languages (DBPL '15), pp. 11–20.

Seco, J. C., Ferreira, P. & Lourenço, H. (2017) *Capability-based Localization of Distributed and Heterogeneous Queries – Extended version with proofs*. Technical Report, NOVA University of Lisbon. Available at: `http://ctp.di.fct.unl.pt/~jcs/papers/jfp-big-data-tech-report.pdf`

Serrano, M., Gallesio, E. & Loitsch, F. (2006) Hop: A language for programming the web 2.0. In Companion to the 21th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 975–985.

Silberschatz, A., Korth, H. & Sudarshan, S. (2006) *Database Systems Concepts*, 5 ed. New York, NY, USA: McGraw-Hill.

Taylor, R. (2010) *Query Optimization for Distributed Database Systems*. Master Thesis, University of Oxford, University of Oxford.

Vassalos, V. & Papakonstantinou, Y. (2000) Expressive capabilities description languages and query rewriting algorithms. *J. Log. Program.* **43**(1), 75–122.

Wong, L. (2000) Kleisli, a functional query system. *J. Funct. Program.* **10**(1), 19–56.