# An Algebraic Foundation for Knowledge Graph Construction

Sitt Min Oo[1] and Olaf Hartig[2]

[1] Ghent University - imec, Ghent, Belgium
`x.sittminoo@ugent.be`
[2] Linköping University, Linköping, Sweden
`olaf.hartig@liu.se`

**Abstract.** Although they exist since more than ten years already, have attracted diverse implementations, and have been used successfully in a significant number of applications, declarative mapping languages for constructing knowledge graphs from heterogeneous types of data sources still lack a solid formal foundation. This makes it impossible to introduce implementation and optimization techniques that are provably correct and, in fact, has led to discrepancies between different implementations. Moreover, it precludes studying fundamental properties of different languages (e.g., expressive power). To address this gap, this paper introduces a language-agnostic algebra for capturing mapping definitions. As further contributions, we show that the popular mapping language RML can be translated into our algebra (by which we also provide a formal definition of the semantics of RML) and we prove several algebraic rewriting rules that can be used to optimize mapping plans based on our algebra.

## 1 Introduction

Knowledge graphs (KGs) are often created or populated by extracting and transforming data from other sources of structured or semi-structured data. One approach to this end is to employ a mapping engine for which the desired mapping from the source data to the KG can be specified using a declarative language. This approach has gained traction in recent years, with applications across diverse domains such as COVID research [17], railway networks [16], and incident management in ICT systems [22]. To provide a basis for this approach, several declarative mapping languages, in particular for RDF-based KGs, have been proposed, including languages that focus only on a single type of input data sources (e.g., R2RML for relational databases [21]) and, more interestingly, also languages designed for multiple types of input data sources [23]. The latter can be classified further into mapping-focused extensions of query languages and dedicated languages designed solely for the declarative description of mappings.

So far, most of these mapping languages—in particular, the dedicated ones—are defined only informally. Yet, such an informal definition can easily lead to discrepancies between different implementations. For instance, for RML [4,5], which has several implementations [1,6,9,20,18], the introduction of conformance tests [8] has revealed various cases in which different implementations fail such a test [7]. It may be argued that the developers of these implementations interpreted the informal definition of the language differently. Without a formal definition it is impossible to introduce implementation and optimization techniques that are provably correct, or to properly compare languages or language features in terms of fundamental properties, such as their expressive power. Therefore, the goal of our work in this paper is to provide a formal foundation for such declarative mapping languages. Especially, instead of focusing on a single specific language, we aim for a more general formalization approach.

Our main contribution is a language-agnostic algebra to capture definitions of mappings from heterogeneous types of data sources to RDF-based KGs. The basis of this mapping algebra is a variation of the relational data model (Section 3), for which we define five types of operators that can be arbitrarily combined into algebra expressions (Section 4). Due to its language-agnostic nature, the algebra can serve as a foundation to various mapping languages formally. As our second contribution, we demonstrate this benefit by providing an algorithm that translates mappings defined using RML into our algebra (Section 5). Through this algorithm, we show not only that our algebra is at least as expressive as RML, but we also provide a formal definition of the semantics of RML. To the best of our knowledge, this is the first formal approach to capture the RML semantics. Another important value of having an algebra that captures declarative mapping definitions is that it may be used as the basis of a systematic and well-defined approach to plan and to optimize the execution of KG construction processes in mapping engines. Related to this option, we make our third contribution: We show several algebraic equivalences which can be used as rewriting rules to optimize mapping plans that are based on our algebra (Section 6).

## 2 Related Work

As mentioned above, declarative mapping languages for KG construction can be classified into languages that are extensions of other types of languages (mainly, query languages), with XSPARQL [2], SPARQL-Generate [11], and Facade-X [3] being examples of such "repurposed mapping languages" [23], and "dedicated mapping languages" [23] that have been created specifically for the purpose of KG construction, with R2RML [21] and RML [4] as examples.

Given our focus on formal approaches to capture such languages, for the repurposed ones we notice they can rely on the formalization of their underlying language. For instance, XSPARQL combines the semantics of XQuery and SPARQL for mapping between XML and RDF data [2], and to also support relational databases, the authors extend their work by utilizing the semantics of relational algebra [12]. Similarly, the formalization of SPARQL-Generate [11] builds on the algebra-based formalization of SPARQL and, thus, is similar in spirit to our approach, which we see as more relevant for dedicated mapping languages.

When considering dedicated mapping languages, to the best of our knowledge, only R2RML has a formally defined semantics. That is, Sequeda et al. [19] provide such a formalization by means of 57 Datalog rules. As this definition is for R2RML, it is limited to mappings from relational databases to RDF, whereas our formalization in this paper can capture mappings of different types of sources of structured and semi-structured data. Moreover, we argue that an algebraic formalization such as ours has the additional advantage that it can also be utilized to represent a form of logical execution plans in mapping engines.

Besides R2RML, a few studies with formalizations related to RML have been published in recent years [9,14]. Iglesias et al. formalize RML-based mapping processes based on Horn clauses [9], where the focus of this formalization is to define optimization techniques presented by the authors. Lastly, an initial study to capture the semantics of mappings in a language-agnostic manner through algebraic mapping operators [14] has a very similar goal as our work in this paper. Yet, in contrast to our work, the authors do not cover all aspects of popular dedicated mapping language such as RML (e.g., joining data from multiple sources) and several of their definitions remain informal or rely on undefined concepts.

## 3 Data Model

This section introduces the data model based on which our mapping algebra is defined. Hence, this model captures the types of intermediate results of any mapping process described by our formalism. Informally, the data model is a version of the relational data model, restricted to a special kind of relations, that we call *mapping relations* and that contain RDF terms as possible values.

As a preliminary step for defining the model formally, we introduce relevant concepts of RDF: We let $\mathcal{S}$ be the countably infinite set of strings (sequences of Unicode code points) and $\mathcal{I}$ be the subset of $\mathcal{S}$ that consists of all strings that are valid IRIs. Moreover, $\mathcal{L}$ is a countably infinite set of pairs $(lex, dt) \in \mathcal{S} \times \mathcal{I}$, which we call *literals*. For each such literal $\ell = (lex, dt)$, $lex$ is its *lexical form* and $dt$ is its *datatype IRI*.[3] Furthermore, we assume a countably infinite set $\mathcal{B}$ of *blank nodes*, which is disjoint from both $\mathcal{S}$ and $\mathcal{L}$. The set $\mathcal{T}$ of all *RDF terms* is defined as $\mathcal{T} = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$, and we write $\bar{\mathcal{T}}$ to denote the set of all possible sequences of terms; i.e., each element of $\bar{\mathcal{T}}$ is a sequence of elements of $\mathcal{T}$. As usual, an *RDF triple* is a tuple $(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times \mathcal{T}$, an *RDF graph* is a set of RDF triples, and an *RDF dataset* is a set $\{G_{\text{dflt}}, (n_1, G_1), \ldots, (n_m, G_m)\}$ where $G_{\text{dflt}}, G_1, \ldots, G_m$ are RDF graphs, $m \geq 0$, and $n_i \in \mathcal{I} \cup \mathcal{B}$ for all $i \in \{1, \ldots, m\}$.

As additional ingredients for defining mapping relations, we assume a countably infinite set $\mathcal{A}$ of *attributes* and a special value $\epsilon$ that is not an RDF term (i.e., $\epsilon \notin \mathcal{T}$) and that shall be used to capture processing errors. The last ingredient are *mapping tuples*, which are the type of tuples used in mapping relations.

**Definition 1.** A **mapping tuple** is a partial function $t : \mathcal{A} \to \mathcal{T} \cup \{\epsilon\}$.

We say that two mapping tuples $t$ and $t'$ are *compatible* if, for every attribute $a \in \text{dom}(t) \cap \text{dom}(t')$, it holds that $t(a) = t'(a)$. Given two mapping tuples $t$ and $t'$ that are compatible, we write $t \cup t'$ to denote the mapping tuple $t''$ that is the merge of $t$ and $t'$; that is, $\text{dom}(t'') = \text{dom}(t) \cup \text{dom}(t')$ and, for every attribute $a \in \text{dom}(t'')$, $t''(a) = t(a)$ if $a \in \text{dom}(t)$, and $t''(a) = t'(a)$ otherwise.

Now we are ready to define our notion of a mapping relation.

[3] To avoid overly complex formulas in this paper, we ignore the option of literals that have a language tag. Yet, the formalism can easily be extended to cover them.

**Table 1.** Mapping relation $r$ discussed in Examples 1 and 2.

| | $a_\mathrm{x}$ | $a_\mathrm{s}$ | $a_\mathrm{p}$ | $a_\mathrm{o}$ | $a_\mathrm{g}$ |
|---|---|---|---|---|---|
| $t$ | ("12", xsd:integer) | ex:alice | foaf:knows | ex:bob | rr:defaultGraph |
| $t'$ | ex:alice | ex:alice | ("knows", xsd:string) | ex:charles | ex:g1 |
| $t''$ | $\epsilon$ | ex:bob | foaf:name | ("Bob", xsd:string) | ex:g2 |

**Definition 2.** A **mapping relation** $r$ is a tuple $(A, I)$, where $A \subset \mathcal{A}$ is a finite, non-empty set of attributes and $I$ is a set of mapping tuples such that, for every such tuple $t \in I$, it holds that $\mathrm{dom}(t) = A$. We call $A$ the **schema** of the mapping relation and $I$ is the **instance** of the mapping relation.

For the sake of conciseness, we sometimes write $t \in r$ to denote that $t$ is a mapping tuple in the instance $I$ of a mapping relation $r = (A, I)$.

*Example 1.* Consider the mapping relation $r = (A, I)$ that is represented in a tabular form in Table 1. It holds that $A = \{a_\mathrm{x}, a_\mathrm{s}, a_\mathrm{p}, a_\mathrm{o}, a_\mathrm{g}\}$ and $I$ contains three mapping tuples, $t$, $t'$, and $t''$, such that $t(a_\mathrm{x})$ is the literal $(\texttt{"12"}, \texttt{xsd:integer})$, $t'(a_\mathrm{x})$ is the IRI `ex:alice`, $t''(a_\mathrm{x}) = \epsilon$, etc.

While mapping relations are intermediate results of mapping processes described by our algebra, the ultimate goal of such processes is to produce an RDF dataset. Therefore, we also need to define how mapping relations are mapped into RDF datasets. To this end, we assume four special attributes: $a_\mathrm{s}, a_\mathrm{p}, a_\mathrm{o}, a_\mathrm{g} \in \mathcal{A}$. Every mapping relation whose schema contains these attributes is mapped to an RDF dataset by using the values that the tuples in the relation have for these attributes. That is, if the values that a mapping tuple has for $a_\mathrm{s}$, $a_\mathrm{p}$, and $a_\mathrm{o}$ form a valid RDF triple, then the graph determined by the $a_\mathrm{g}$-value of the tuple contains this triple. Formally, we define this approach as follows.

**Definition 3.** Let $r = (A, I)$ be a mapping relation with $a_\mathrm{s}, a_\mathrm{p}, a_\mathrm{o}, a_\mathrm{g} \in A$, and

$$I_\mathrm{valid} = \big\{ t \in I \,\big|\, \big(t(a_\mathrm{s}), t(a_\mathrm{p}), t(a_\mathrm{o})\big) \text{ is an RDF triple and } t(a_\mathrm{g}) \in \mathcal{I} \cup \mathcal{B} \big\} \text{ and}$$
$$N = \big\{ t(a_\mathrm{g}) \in \mathcal{I} \cup \mathcal{B} \,\big|\, t \in I_\mathrm{valid} \text{ and } t(a_\mathrm{g}) \text{ is not the IRI } \texttt{rr:defaultGraph} \big\}.$$

The **RDF dataset resulting from** $r$ is the RDF dataset

$$D = \{G_\mathrm{dflt}\} \cup \big\{ (n, G_n) \,\big|\, n \in N \big\},$$

where, for every $n \in N$,

$$G_n = \big\{ \big(t(a_\mathrm{s}), t(a_\mathrm{p}), t(a_\mathrm{o})\big) \,\big|\, t \in I_\mathrm{valid} \text{ such that } t(a_\mathrm{g}) = n \big\} \text{ and}$$
$$G_\mathrm{dflt} = \big\{ \big(t(a_\mathrm{s}), t(a_\mathrm{p}), t(a_\mathrm{o})\big) \,\big|\, t \in I_\mathrm{valid} \text{ such that } t(a_\mathrm{g}) \text{ is } \texttt{rr:defaultGraph} \big\}.$$

*Example 2.* For the mapping relation in Example 1 it holds that $I_\mathrm{valid} = \{t, t''\}$ and $N = \{\texttt{ex:g2}\}$. Therefore, the RDF dataset resulting from this mapping relation is $D = \big\{G, (\texttt{ex:g2}, G')\big\}$ with $G = \big\{(\texttt{ex:alice}, \texttt{foaf:knows}, \texttt{ex:bob})\big\}$ and $G' = \big\{(\texttt{ex:bob}, \texttt{foaf:name}, (\texttt{"Bob"}, \texttt{xsd:string}))\big\}$. Notice that $t' \notin I_\mathrm{valid}$ because $\big(t'(a_\mathrm{s}), t'(a_\mathrm{p}), t'(a_\mathrm{o})\big)$ is not an RDF triple (since $t'(a_\mathrm{p})$ is a literal).

## 4 Mapping Algebra

This section introduces our mapping algebra which consists of five types of operators. The output of each such operator is a mapping relation and the inputs (if any) are mapping relations as well. Hence, the operators can be combined into arbitrarily complex algebra expressions, where each such expression defines a mapping from one or more sources of (semi-)structured data into an RDF dataset.[4]

### 4.1 Source Operator

The basis of every expression that defines a mapping using our algebra are operators that establish a relational view of each source of input data. From the perspective of the algebra, these *source operators* are nullary operators; they do not have any mapping relation as input. The mapping relation that such an operator creates as output depends on the data of the corresponding data source and can be specified based on a language appropriate to identify and extract relevant pieces of data from the data source. As our mapping algebra should be applicable for various types of data sources, for which different such languages are suitable, we introduce an abstraction of this extraction process and define our notion of source operators based on this abstraction.

Our abstraction considers two infinite sets, $\mathcal{D}$ and $\mathcal{Q}$, which are disjoint. $\mathcal{D}$ is the set of all possible *data objects*. In practice, there are different kinds of data objects, even within the context of the same type of data source, and data objects may be nested within one another. For instance, each line of a CSV file may be seen as a data object, and so may each value within such a line, as well as the CSV file as a whole. In our abstraction, any particular kind of data objects is considered as a specific subset of $\mathcal{D}$.

The set $\mathcal{Q}$ captures possible *query languages* to retrieve data objects from other data objects. We consider each such language $L \in \mathcal{Q}$ as a set, where every element $q \in L$ is one of the expressions admitted by the language. In practice, each such language is designed to be used for a specific kind of data objects (i.e., a specific subset of $\mathcal{D}$). Therefore, we define the notion of the *types of data sources* considered for the source operators of our algebra as a structure that specifies the relevant kinds of data objects and the corresponding evaluation semantics of the languages to be used for data sources of such a type.

**Definition 4.** A **source type** is a tuple $(\mathcal{D}^{\text{ds}}, \mathcal{D}^{\text{c1}}, \mathcal{D}^{\text{c2}}, L, L', \mathit{eval}, \mathit{eval}', \mathit{cast})$ where

- $\mathcal{D}^{\text{ds}} \subseteq \mathcal{D}, \quad \mathcal{D}^{\text{c1}} \subseteq \mathcal{D}, \quad \mathcal{D}^{\text{c2}} \subseteq \mathcal{D}$,
- $L$ and $L'$ are query languages in $\mathcal{Q}$,
- *eval* is a function $\mathit{eval}\colon \mathcal{D}^{\text{ds}} \times L \rightarrow \overline{\mathcal{D}^{\text{c1}}}$,
- *eval*$'$ is a partial function $\mathit{eval}'\colon \mathcal{D}^{\text{ds}} \times \mathcal{D}^{\text{c1}} \times L' \rightarrow \overline{\mathcal{D}^{\text{c2}}}$, and
- *cast* is a partial function $\mathit{cast}\colon \mathcal{D}^{\text{c2}} \rightarrow \mathcal{L}$,

where $\overline{\mathcal{D}^{\text{c1}}}$, resp. $\overline{\mathcal{D}^{\text{c2}}}$, is the set of all sequences of data objects in $\mathcal{D}^{\text{c1}}$, resp. $\mathcal{D}^{\text{c2}}$.

**Definition 5.** A **data source** $s$ is a tuple $(\mathit{type}, D)$ in which *type* is a source type $(\mathcal{D}^{\text{ds}}, \mathcal{D}^{\text{o1}}, \mathcal{D}^{\text{o2}}, L, L', \mathit{eval}, \mathit{eval}', \mathit{cast})$ and $D \in \mathcal{D}^{\text{ds}}$.

[4] We assume here that the mapping relation resulting from the root operator of the expression contains the aforementioned attributes $a_{\text{s}}$, $a_{\text{p}}$, $a_{\text{o}}$, and $a_{\text{g}}$ (see Definition 3).

As per Definitions 4 and 5, the set $\mathcal{D}^{\mathsf{ds}}$ of a source type determines the kind of data objects that the data sources of this type may provide access to. For instance, for CSV-based data sources this may be the (infinite) set of all possible CSV files. The functions *eval* and *eval′* are meant to define the evaluation semantics of the languages $L$ and $L'$, respectively. Notice that the domain of *eval′* consists of 3-tuples that contain an additional data object as their second element, which is meant as a context object for the evaluation. The idea of the source operators shall be to use $L$ to specify relevant context objects and, then, to use $L'$ to retrieve values for creating a mapping tuple per context object.

While a formal definition of concrete source types is beyond the scope of this paper, the following two examples outline informally how JSON-based data sources and CSV-based data sources may be captured within our abstraction.

*Example 3.* To define $type_{\mathsf{json}} = (\mathcal{D}^{\mathsf{ds}}_{\mathsf{json}}, \mathcal{D}^{\mathsf{c1}}_{\mathsf{jp}}, \mathcal{D}^{\mathsf{c2}}_{\mathsf{jp}}, L_{\mathsf{jp}}, L'_{\mathsf{jp}}, eval_{\mathsf{jp}}, eval'_{\mathsf{jp}}, cast_{\mathsf{json}})$ as a source type for JSON-based data sources, we let $\mathcal{D}^{\mathsf{ds}}_{\mathsf{json}}$ be the infinite set of all possible JSON documents, both $\mathcal{D}^{\mathsf{c1}}_{\mathsf{jp}}$ and $\mathcal{D}^{\mathsf{c2}}_{\mathsf{jp}}$ are the infinite set of all possible elements that may occur within JSON documents (i.e., $\mathcal{D}^{\mathsf{c1}}_{\mathsf{jp}} = \mathcal{D}^{\mathsf{c2}}_{\mathsf{jp}}$), $L_{\mathsf{jp}}$ is the set of all JSONPath expressions that begin with the selector for the root element (i.e., `$`), and $L'_{\mathsf{jp}}$ are all JSONPath expressions that do not begin with `$`. Then, $eval_{\mathsf{jp}}$ and $eval'_{\mathsf{jp}}$ capture the evaluation semantics of JSONPath. That is, for a JSON document $D \in \mathcal{D}^{\mathsf{ds}}_{\mathsf{json}}$ and a query $q \in L_{\mathsf{jp}}$, $eval_{\mathsf{jp}}(D, q)$ returns the elements of $D$ selected by $q$. If $q' \in L'_{\mathsf{jp}}$, and assume $d \in \mathcal{D}^{\mathsf{c1}}_{\mathsf{jp}}$ is a JSON object within $D$, then $eval'_{\mathsf{jp}}(D, d, q')$ returns the elements of $D$ that can be reached from $d$ as per the traversal specified by $q'$. Finally, $cast_{\mathsf{json}}$ maps JSON values to RDF literals. For instance, string values may be mapped to `xsd:string` literals, numeric values that are integers may be mapped to `xsd:integer` literals, etc.

*Example 4.* As a source type for CSV-based data sources, we may introduce $type_{\mathsf{csv}} = (\mathcal{D}^{\mathsf{ds}}_{\mathsf{csv}}, \mathcal{D}^{\mathsf{c1}}_{\mathsf{csv}}, \mathcal{D}^{\mathsf{c2}}_{\mathsf{csv}}, L_{\mathsf{csv}}, L'_{\mathsf{csv}}, eval_{\mathsf{csv}}, eval'_{\mathsf{csv}}, cast_{\mathsf{csv}})$ as follows. $\mathcal{D}^{\mathsf{ds}}_{\mathsf{csv}}$ is the infinite set of all possible CSV files, $\mathcal{D}^{\mathsf{c1}}_{\mathsf{csv}}$ is the infinite set of all possible rows of CSV files, $\mathcal{D}^{\mathsf{c2}}_{\mathsf{csv}} \subset \mathcal{S}$ is the infinite set of strings that may be values within CSV files, and the $cast_{\mathsf{csv}}$ function maps all these string values in $\mathcal{D}^{\mathsf{c2}}_{\mathsf{csv}}$ to `xsd:string` literals. As the language $L_{\mathsf{csv}}$, we may use the singleton set that contains (only) the empty string $\varepsilon$, and $L'_{\mathsf{csv}} \subset \mathcal{S}$ is the set of all strings that may be used as column names in CSV files. Informally, $eval_{\mathsf{csv}}$ and $eval'_{\mathsf{csv}}$ may then be defined as follows. For every CSV file $D \in \mathcal{D}^{\mathsf{ds}}_{\mathsf{csv}}$, $eval_{\mathsf{csv}}(D, \varepsilon)$ returns all rows of $D$, and for every such row $d \in eval_{\mathsf{csv}}(D, \varepsilon)$ and every string $c$ that is a column name in $D$, $eval'_{\mathsf{csv}}(D, d, c)$ is the value that the row $d$ has in column $c$.

*Example 5.* As the basis of a running example for the rest of this section, we assume a data source $s_{\mathsf{ex}} = (type_{\mathsf{csv}}, D_{\mathsf{ex}})$ where $D_{\mathsf{ex}}$ is a CSV file with four columns, named $\mathsf{id}$, $\mathsf{firstname}$, $\mathsf{lastname}$, and $\mathsf{age}$, and the following two rows:

$$(\mathtt{1}, \mathtt{Alice}, \mathtt{Lee}, \mathtt{23}) \quad \text{and} \quad (\mathtt{2}, \mathtt{Bob}, \mathtt{Malice}, \mathtt{unknown}).$$

If we write $d_1$ to denote the first of these rows, then $eval'_{\mathsf{csv}}(D_{\mathsf{ex}}, d_1, \mathsf{firstname})$ is the string `Alice`, and $cast_{\mathsf{csv}}(\mathtt{Alice})$ is the RDF literal $(\mathtt{"Alice"}, \mathtt{xsd{:}string})$.

Now we are ready to define our notion of source operators. While they are nullary operators, they are specified using three parameters: the data source from which the data for the produced mapping relation is extracted, a query that selects relevant context objects from the data source, and a map that associates attributes with queries, where these queries are meant to select the values that a mapping tuple produced for a context object has for the attributes. Formally, we define the mapping relation produced by a source operator as follows.

**Definition 6.** Let $s = (type, D)$ be a data source with $type = (\mathcal{D}^{\mathsf{ds}}, \mathcal{D}^{\mathsf{c1}}, \mathcal{D}^{\mathsf{c2}}, L, L', eval, eval', cast)$, let $q \in L$, and let $\mathbb{P}$ be a partial function $\mathbb{P} \colon \mathcal{A} \to L'$. The $(q, \mathbb{P})$**-specific mapping relation obtained from** $s$, denoted by $\mathrm{Source}^{(s,q,\mathbb{P})}$, is the mapping relation $(A, I)$ such that $A = \mathrm{dom}(\mathbb{P})$ and

$$I = \big\{\{a_1 \to cast(v_1), \dots, a_n \to cast(v_n)\} \bigm| d \text{ is in } \bar{O} \text{ and } ((a_1, v_1), \dots, (a_n, v_n)) \in X_d\big\},$$

where $\bar{O} = eval(D, q)$ and

$$X_d = \bigtimes_{a \in \mathrm{dom}(\mathbb{P})} \big\{(a, v) \bigm| v \text{ is in } eval'(D, d, q') \text{ with } q' = \mathbb{P}(a)\big\}.$$

*Example 6.* The result of $\mathrm{Source}^{(s_{\mathsf{ex}},\varepsilon,\mathbb{P}_{\mathsf{ex}})}$ with source $s_{\mathsf{ex}}$ of Example 5 and $\mathbb{P}_{\mathsf{ex}} = \{a_1 \to \mathsf{id}, a_2 \to \mathsf{firstname}, a_3 \to \mathsf{age}\}$ is the relation $\big(\{a_1, a_2, a_3\}, \{t_1, t_2\}\big)$ with

$$t_1 = \big\{a_1 \to (\texttt{"1"}, \texttt{xsd:string}), a_2 \to (\texttt{"Alice"}, \texttt{xsd:string}), a_3 \to (\texttt{"23"}, \texttt{xsd:string})\big\}$$ and $$t_2 = \big\{a_1 \to (\texttt{"2"}, \texttt{xsd:string}), a_2 \to (\texttt{"Bob"}, \texttt{xsd:string}), a_3 \to (\texttt{"unknown"}, \texttt{xsd:string})\big\}.$$

### 4.2 Extend Operator

Extend operators are unary operators that can be used to extend a mapping relation with an additional attribute. The values that the tuples of the relation shall have for this attribute are specified via a so-called *extend expression* that can be formed using *extension functions*. In the following, we first introduce the notions of such functions and such expressions, and then define extend operators.

**Definition 7.** An **extension function** is a function $f$ of the form

$$f \colon (\mathcal{T} \cup \{\epsilon\}) \times \dots \times (\mathcal{T} \cup \{\epsilon\}) \to (\mathcal{T} \cup \{\epsilon\}).$$

*Example 7.* A concrete example of such an extension function may be a unary function called `toInt` that converts `xsd:string` literals representing integers into `xsd:integer` literals. Formally, for every $v \in (\mathcal{T} \cup \epsilon)$, we define:

$$\texttt{toInt}(v) = \begin{cases} (lex, \texttt{xsd:integer}) & \text{if } v \text{ is a literal } (lex, dt) \text{ such that } lex \text{ is in the lexical space of the datatype denoted by the IRI } \texttt{xsd:integer} \text{ and } dt \text{ is the IRI } \texttt{xsd:string}, \\ \epsilon & \text{else.} \end{cases}$$

**Definition 8. Extend expressions** are defined recursively as follows:

1. Every RDF term in $\mathcal{T}$ is an extend expression.

2. Every attribute in $\mathcal{A}$ is an extend expression.
3. If $\varphi_1, \ldots, \varphi_n$ are extend expressions and $f$ is an extension function (as per Definition 7), then the tuple $(f, \varphi_1, \ldots, \varphi_n)$ is an extend expression.

For every extend expression $\varphi$, we write $\mathrm{attrs}(\varphi)$ to denote the set of all attributes mentioned in $\varphi$. Formally, this set is defined recursively as follows.

1. If $\varphi$ is an RDF term, then $\mathrm{attrs}(\varphi) = \emptyset$.
2. If $\varphi$ is an attribute $a \in \mathcal{A}$, then $\mathrm{attrs}(\varphi) = \{a\}$.
3. If $\varphi$ is of the form $(f, \varphi_1, \ldots, \varphi_n)$, then $\mathrm{attrs}(\varphi) = \bigcup_{i \in \{1,\ldots,n\}} \mathrm{attrs}(\varphi_i)$.

While Definition 8 introduces the syntax of extend expressions, their interpretation as a function to obtain values for mappings tuples is defined as follows.

**Definition 9.** Let $\varphi$ be an extend expression and $t$ be a mapping tuple. The **evaluation of** $\varphi$ **over** $t$, denoted by $\mathit{eval}(\varphi, t)$, is either an RDF term or the error value ($\epsilon$), and is determined recursively as follows:

$$\mathit{eval}(\varphi, t) = \begin{cases} \varphi & \text{if } \varphi \text{ is an RDF term,} \\ t(\varphi) & \text{if } \varphi \in \mathcal{A} \text{ and } \varphi \in \mathrm{dom}(t), \\ f\big(\mathit{eval}(\varphi_1, t), \ldots, \mathit{eval}(\varphi_n, t)\big) & \text{if } \varphi \text{ is of the form } (f, \varphi_1, \ldots, \varphi_n) \\ & \text{such that } f \text{ is an } n\text{-ary function,} \\ \epsilon & \text{else.} \end{cases}$$

*Example 8.* If $\varphi_{\mathsf{ex}}$ is the extend expression $(\texttt{toInt}, a_3)$, for the tuples $t_1$ and $t_2$ of Example 6, $\mathit{eval}(\varphi_{\mathsf{ex}}, t_1)$ is the literal $(\texttt{"23"}, \texttt{xsd:integer})$ and $\mathit{eval}(\varphi_{\mathsf{ex}}, t_2) = \epsilon$.

Now we can define the semantics of extend operators as follows.

**Definition 10.** Let $r = (A, I)$ be a mapping relation, $a$ be an attribute that is not in $A$ (i.e., $a \in \mathcal{A} \setminus A$), and $\varphi$ be an extend expression. The $(a, \varphi)$**-extension of** $r$, denoted by $\mathrm{Extend}^{a}_{\varphi}(r)$, is the mapping relation $(A', I')$ such that $A' = A \cup \{a\}$ and $I' = \big\{ t \cup \{a \to \mathit{eval}(\varphi, t)\} \mid t \in I \big\}$.

*Example 9.* We may extend the mapping relation $r$ of Example 6 with an attribute $a_4$ for which each of the two tuples of $r$ gets the value obtained by applying the extend expression $\varphi_{\mathsf{ex}}$ of Example 8. This operation can be captured as $\mathrm{Extend}^{a_4}_{\varphi_{\mathsf{ex}}}(r)$ and results in the mapping relation $\big(\{a_1, a_2, a_3, a_4\}, \{t'_1, t'_2\}\big)$ with

$$t'_1 = t_1 \cup \{a_4 \to (\texttt{"23"}, \texttt{xsd:integer})\} \quad \text{and} \quad t'_2 = t_2 \cup \{a_4 \to \epsilon\}.$$

While the example shows how extend operators can be used to convert literals from one datatype to another, other use cases include: adding the same constant RDF term to all tuples of a mapping relation, creating IRIs from other attribute values, and applying arbitrary functions to one or more other attribute values.

### 4.3 Relational Algebra Operators

In addition to the operators introduced above, any operator of the relational algebra can also be used for mapping relations. For instance, the following definition introduces the notion of *projection*, adapted to mapping relations.

**Definition 11.** Let $r = (A, I)$ be a mapping relation and $P \subseteq A$ be a non-empty subset of the attributes of $r$. The **$P$-specific projection of $r$**, denoted by $\text{Project}^P(r)$, is the mapping relation $(P, I')$ such that $I' = \{t[P] \mid t \in I\}$, where, for every mapping tuple $t \in I$, $t[P]$ denotes the mapping tuple $t'$ that is the restriction of $t$ to $P$; i.e., $\text{dom}(t') = P$ and $t'(a) = t(a)$ for all $a \in P$.

*Example 10.* Let $r$ be the mapping relation of Example 9. Assuming that further steps of a potential mapping process that involve this relation use only the attributes $a_2$ and $a_4$ of $r$, we may project away the rest of the attributes. To this end, we use $\text{Project}^{\{a_2,a_4\}}(r)$ and obtain the relation $\big(\{a_2, a_4\}, \{t''_1, t''_2\}\big)$ with

$$t''_1 = \big\{a_2 \to (\texttt{"Alice"}, \texttt{xsd:string}),\ a_4 \to (\texttt{"23"}, \texttt{xsd:integer})\big\} \text{ and}$$
$$t''_2 = \big\{a_2 \to (\texttt{"Bob"}, \texttt{xsd:string}),\ a_4 \to \epsilon\big\}.$$

As two more examples of adopting traditional relational algebra operators for mapping relations, we present the following two definitions.

**Definition 12.** Let $r_1 = (A_1, I_1)$ and $r_2 = (A_2, I_2)$ be mapping relations such that $A_1 \cap A_2 = \emptyset$, and $\mathbb{J} \subseteq A_1 \times A_2$. The **$\mathbb{J}$-based equijoin of $r_1$ and $r_2$**, denoted by $\text{EqJoin}^{\mathbb{J}}(r_1, r_2)$, is the mapping relation $(A, I)$ such that $A = A_1 \cup A_2$ and

$$I = \{t_1 \cup t_2 \mid t_1 \in I_1 \text{ and } t_2 \in I_2 \text{ such that } t_1(a_1) = t_2(a_2) \text{ for all } (a_1, a_2) \in \mathbb{J}\}.$$

**Definition 13.** The **union** of mapping relations $r_1 = (A_1, I_1)$ and $r_2 = (A_2, I_2)$ with $A_1 = A_2$, denoted by $\text{Union}(r_1, r_2)$, is the mapping relation $(A_1, I_1 \cup I_2)$.

## 5 Algebra-Based Definition of RML

This section introduces an algorithm to convert mappings defined using the mapping language RML (specifically, version 1.1.2 [5]) into our mapping algebra. This algorithm shows that *our algebra is at least as expressive as RML*. Moreover, through this algorithm, in combination with the algebra, we provide a *formal definition of the semantics of RML mappings*. We emphasize that this formal semantics coincides with the informally-defined semantics of RML v1.1.2 [5], which we have verified by running the official RML test cases [8] on a prototypical implementation of our approach.[5] Hereafter, we describe the algorithm step by step, while a complete pseudocode representation is given as Algorithm 1. For the description, we assume familiarity with the concepts of RML [4,5,10].

***Input and Output.*** Since RML mappings are captured as RDF graphs that use the RML ontology [10], the main input to our algorithm is such an RDF graph (we assume it uses IRIs of the RML ontology only in the way as intended by that ontology). A second input is an IRI considered as a base for resolving relative IRIs [5]. As a third input, we assume an injective function $S2B \colon \mathcal{S} \to \mathcal{B}$ that maps every string to a unique blank node (which shall become relevant later). The output is a mapping relation that is obtained by applying operators of our algebra and that can be converted into an RDF dataset as per Definition 3.

[5] `https://github.com/s-minoo/eswc-2025-poster-algebra-implementation`

**Algorithm 1:** Translates an RML mapping into our mapping algebra.

```
Input: $G$ - an RDF graph (assumed to describe an RML mapping)
       $base$ - an IRI considered as base IRI
       $S2B$ - an injective function that maps every string to a unique blank node
Output: a mapping relation with attributes $a_s$, $a_p$, $a_o$, and $a_g$
1  $G_{NF} \leftarrow$ normalized version of $G$ (apply the update queries of Appendix A to $G$)
2  $\omega_{spog} \leftarrow$ the empty mapping relation $(\{a_s, a_p, a_o, a_g\}, \emptyset)$
3  foreach $u_{tm} \in \mathcal{I} \cup \mathcal{B}$ for which there exists $o \in \mathcal{I} \cup \mathcal{B}$ such that $G_{NF}$ contains
       the triple $(u_{tm}, \texttt{rr:predicateObjectMap}, o)$ do
4      $\omega \leftarrow \mathrm{Source}^{(s,q,\mathbb{P})}$, where $(s,q) = \textsc{SrcAndRootQuery}(u_{tm}, G_{NF})$  // Definition 14
           and $\mathbb{P} = \textsc{ExtractQueries}(u_{tm}, G_{NF})$  // Algorithm 2
5      $u_{sm} \leftarrow o$, where $o \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{tm}, \texttt{rr:subjectMap}, o) \in G_{NF}$
6      $u_{pom} \leftarrow o$, where $o \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{tm}, \texttt{rr:predicateObjectMap}, o) \in G_{NF}$
7      $u_{pm} \leftarrow o$, where $o \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{pom}, \texttt{rr:predicateMap}, o) \in G_{NF}$
8      $u_{om} \leftarrow o$, where $o \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{pom}, \texttt{rr:objectMap}, o) \in G_{NF}$
9      $\omega \leftarrow \mathrm{Extend}^{a_s}_{\varphi}(\omega)$, where $\varphi = \textsc{CreateExtExpr}(u_{sm}, G_{NF}, base, S2B, \mathbb{P})$  // Alg.3
10     $\omega \leftarrow \mathrm{Extend}^{a_p}_{\varphi}(\omega)$, where $\varphi = \textsc{CreateExtExpr}(u_{pm}, G_{NF}, base, S2B, \mathbb{P})$
11     if there is a $u_{ptm} \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{om}, \texttt{rr:parentTriplesMap}, u_{ptm}) \in G_{NF}$ then
12         $\omega' \leftarrow \mathrm{Source}^{(s',q',\mathbb{P}')}$, where $(s',q') = \textsc{SrcAndRootQuery}(u_{ptm}, G_{NF})$
               and $\mathbb{P}' = \textsc{ExtractQueries}(u_{ptm}, G_{NF})$  // Algorithm 2
13         $\mathbb{J} \leftarrow \emptyset$
14         foreach $u_{jc} \in \mathcal{I} \cup \mathcal{B}$ for which $(u_{om}, \texttt{rr:joinCondition}, u_{jc}) \in G_{NF}$ do
15             $a \leftarrow \mathbb{P}^{-1}(lex)$, where $(u_{jc}, \texttt{rr:child}, o) \in G_{NF}$ with $o = (lex, dt) \in \mathcal{L}$
16             $a' \leftarrow \mathbb{P}'^{-1}(lex)$, where $(u_{jc}, \texttt{rr:parent}, o) \in G_{NF}$ with $o = (lex, dt) \in \mathcal{L}$
17             $\mathbb{J} \leftarrow \mathbb{J} \cup \{(a, a')\}$
18         $\omega \leftarrow \mathrm{EqJoin}^{\mathbb{J}}(\omega, \omega')$
19         $u_{sptm} \leftarrow o$, where $o \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{ptm}, \texttt{rr:subjectMap}, o) \in G_{NF}$
20         $\omega \leftarrow \mathrm{Extend}^{a_o}_{\varphi}(\omega)$, where $\varphi = \textsc{CreateExtExpr}(u_{sptm}, G_{NF}, base, S2B, \mathbb{P}')$
21     else
22         $\omega \leftarrow \mathrm{Extend}^{a_o}_{\varphi}(\omega)$, where $\varphi = \textsc{CreateExtExpr}(u_{om}, G_{NF}, base, S2B, \mathbb{P})$
23     if there is a $u_{gm} \in \mathcal{I} \cup \mathcal{B}$ such that $(u_{sm}, \texttt{rr:graphMap}, u_{gm}) \in G_{NF}$ or
           $(u_{pom}, \texttt{rr:graphMap}, u_{gm}) \in G_{NF}$ then
24         $\omega \leftarrow \mathrm{Extend}^{a_g}_{\varphi}(\omega)$, where $\varphi = \textsc{CreateExtExpr}(u_{gm}, G_{NF}, base, S2B, \mathbb{P})$
25     else
26         $\omega \leftarrow \mathrm{Extend}^{a_g}_{\varphi}(\omega)$, where $\varphi$ is the IRI $\texttt{rr:defaultGraph}$
27     $\omega_{spog} \leftarrow \mathrm{Union}\big(\omega_{spog}, \mathrm{Project}^{\{a_s,a_p,a_o,a_g\}}(\omega)\big)$
28 return $\omega_{spog}$
```

***Normalization Phase.*** To minimize the variations of RML descriptions to be considered, the algorithm begins by converting the given RML mapping into a *normal form* in which i) no shortcut properties are used, ii) all referencing object maps have a join condition, and iii) every triples map has only a single predicate-object map with a single predicate map, a single object map, and at most one graph map. Every RML mapping can be converted into this normal form by applying the following sequence of rewriting steps, which we have adopted from

Rodríguez-Muro and Rezk [15], with adaptations specific to RML. While we present these steps informally, Appendix A captures them formally in terms of update queries. None of these steps changes the meaning of the defined mapping.

1. Every class IRI definition is expanded into a predicate-object map.
2. All shortcut properties for constant-valued term maps are expanded.
3. Predicate-object maps with multiple predicate maps or multiple object maps are duplicated to have a single predicate map and a single object map each.
4. Referencing object maps without a join condition are replaced by an object map with the subject IRI of the parent triples map as the reference value.
5. Every triples map with multiple predicate-object maps is duplicated such that each of the resulting triples maps has a single predicate-object map only.
6. Every triples map with multiple graph maps is duplicated such that each of the resulting triples maps has one of these graph maps.

***Main Loop.*** After normalization, the algorithm iterates over all triples maps defined by the (normalized) RML mapping (lines 3–27). For each of them, it combines algebra operators, as described below, to define a mapping relation that, in the end, has the attributes $a_\mathrm{s}$, $a_\mathrm{p}$, $a_\mathrm{o}$, and $a_\mathrm{g}$. These mapping relations are combined using union (line 27), resulting in a mapping relation (line 28) that captures the complete RDF dataset defined by the given RML mapping.

***Translation of the Logical Source.*** As the first step for each triples map, the algorithm creates a source operator (line 4). The first two parameters for this operator—the data source $s$ and the query expression $q$ for selecting relevant context objects (cf. Definition 6)—depend on the logical source defined for the triples map. Since RML is extensible regarding the types of logical sources, we have to abstract the creation of these two parameters by the following function.

**Definition 14.** Given an IRI or blank node $u_\mathsf{tm}$ (assumed to denote an RML triples map) and an RDF graph $G$ (assumed to describe an RML mapping, including $u_\mathsf{tm}$), $\textsc{SrcAndRootQuery}(u_\mathsf{tm}, G)$ is a pair $(s, q)$ of a data source $s = (\mathit{type}, D)$ with $\mathit{type} = (\mathcal{D}^\mathrm{ds}, \mathcal{D}^\mathrm{c1}, \mathcal{D}^\mathrm{c2}, L, L', \mathit{eval}, \mathit{eval}', \mathit{cast})$ and a query expression $q \in L$, as extracted from the description of the logical source of $u_\mathsf{tm}$ in $G$.

*Example 11.* Given an RDF graph $G_\mathsf{ex}$ that contains the following RML mapping, $\textsc{SrcAndRootQuery}(\texttt{ex:tm}, G_\mathsf{ex}) = (s'_\mathsf{ex}, \varepsilon)$ where $s'_\mathsf{ex} = (\mathit{type}_\mathsf{csv}, D'_\mathsf{ex})$ such that $\mathit{type}_\mathsf{csv}$ is the source type for CSV-based data sources (cf. Example 4), $D'_\mathsf{ex}$ is the CSV file mentioned in the second triple, and $\varepsilon$ is the empty string.

```
ex:tm rml:logicalSource [ rml:source "data.csv";  rml:referenceFormulation ql:CSV ];
      rr:subjectMap [ rr:template "http://example.com/person_{ID}";  rr:termType rr:IRI ];
      rr:predicateObjectMap [ rr:predicate rdfs:label;
                              rr:objectMap [rml:reference "Name"] ].
```

The third parameter for the source operator—the partial function $\mathbb{P}$ that maps attributes to query expressions (cf. Definition 6)—is populated by Algorithm 2, which extracts the query expressions from the term maps of the given triples map. In particular, the term maps may use query expressions as references (lines 3–4 of Algorithm 2), in string templates (lines 5–6), and as child and parent queries of join conditions (lines 7–10). To focus only on the term maps of the given triples map, the extraction is restricted to the relevant subgraph of the given RDF graph (line 1 of Algorithm 2), which we define formally as follows.

**Algorithm 2:** ExtractQueries - extract query expressions from triples maps.

```
    Input: $u_{\mathsf{tm}}$ - IRI or blank node of the triples map to extract queries from
           $G$ - an RDF graph (assumed to contain a *normalized* RML description)
    Output: an injective partial function $\mathbb{P}$ that maps attributes to query expressions
1   $G_{\mathsf{tm}} \leftarrow$ the $u_{\mathsf{tm}}$-rooted subgraph of $G$                // Definition 15
2   $Q \leftarrow \emptyset$                                // Initially empty set of query expressions
3   forall $s \in \mathcal{I} \cup \mathcal{B}$ and $o \in \mathcal{L}$ for which there is a triple $(s, \texttt{rml:reference}, o) \in G_{\mathsf{tm}}$ do
4       $Q \leftarrow Q \cup \{lex\}$, where $o = (lex, dt)$
5   forall $s \in \mathcal{I} \cup \mathcal{B}$ and $o \in \mathcal{L}$ for which there is a triple $(s, \texttt{rr:template}, o) \in G_{\mathsf{tm}}$ do
6       $Q \leftarrow Q \cup Q'$, where $o = (lex, dt)$ and $Q'$ is the set of all substrings of $lex$ that
                       are enclosed by "{" and "}"
7   $T_{\mathrm{c}} \leftarrow \{(s,p,o) \in G_{\mathsf{tm}} \,|\, p$ is the IRI $\texttt{rr:child}$ and $o \in \mathcal{L}\}$
8   $T_{\mathrm{p}} \leftarrow \{(s,p,o) \in G \,|\, p$ is the IRI $\texttt{rr:parent}, o \in \mathcal{L},$ and there is an $s' \in \mathcal{I} \cup \mathcal{B}$ s.t.
                       $(s', \texttt{rr:joinCondition}, s) \in G$ and $(s', \texttt{rr:parentTriplesMap}, u_{\mathsf{tm}}) \in G\}$
9   foreach triple $(s,p,o) \in (T_{\mathrm{c}} \cup T_{\mathrm{p}})$ do
10      $Q \leftarrow Q \cup \{lex\}$, where $o = (lex, dt)$
11  $\mathbb{P} \leftarrow$ (initially) empty function from attributes to query expressions
12  foreach query expression $q \in Q$ do
13      $\mathbb{P} \leftarrow \mathbb{P} \cup \{a \rightarrow q\}$, where $a$ is a fresh attribute from $\mathcal{A} \setminus \{a_{\mathrm{s}}, a_{\mathrm{p}}, a_{\mathrm{o}}, a_{\mathrm{g}}\}$ that
                       has not been used before in the whole translation process
14  return $\mathbb{P}$
```

**Definition 15.** Let $G$ be an RDF graph and $u \in \mathcal{I} \cup \mathcal{B}$. The **$u$-rooted subgraph of $G$** is an RDF graph $G' \subseteq G$ that is defined recursively as follows:
1. $G'$ contains every triple $(s,p,o) \in G$ for which it holds that $s = u$.
2. $G'$ contains every triple $(s,p,o) \in G$ for which there is already another triple $(s',p',o') \in G'$ such that $s = o'$ and $p'$ is not the IRI `rr:parentTriplesMap`.

*Example 12.* For the RDF graph $G_{\mathsf{ex}}$ that contains the RML mapping of Example 11, ExtractQueries($\texttt{ex:tm}, G_{\mathsf{ex}}$) returns $\mathbb{P} = \{a_1 \rightarrow \mathsf{ID}, a_2 \rightarrow \mathsf{Name}\}$, where $a_1$ and $a_2$ are arbitrary attributes such that $a_1 \neq a_2$ and $a_1, a_2 \notin \{a_{\mathrm{s}}, a_{\mathrm{p}}, a_{\mathrm{o}}, a_{\mathrm{g}}\}$.

***Translation of the Subject Map and the Predicate Map.*** The source operator constructed in the previous step creates a mapping relation with values obtained via the extracted query expressions (along the lines of Example 6). This relation can now be extended by using these values to create RDF terms as defined by the term maps of the triples map. To this end, for the subject (term) map and the predicate (term) map, the algorithm adds an extend operator, respectively (lines 9–10 in Algorithm 1). The construction of the extend expressions of these operators (see Definitions 8 and 10) is captured in Algorithm 3 and uses the following extension functions, which are defined formally in Appendix B.

– `toIRI` converts string literals representing IRIs into these IRIs.
– $\texttt{toBNode}^{S2B}$, parameterized by the aforementioned function $S2B$ (see the discussion of the input of Algorithm 1), converts string literals into blank nodes.
– `toLiteral` is a binary extension function that converts string literals into literals with a datatype IRI that is given as the second argument.

**Algorithm 3:** CreateExtExpr - creates an extend expression for a term map.

```
Input: $u$ - IRI or blank node of the term map to create the extend expression for
       $G$ - an RDF graph (assumed to describe the term map $u$)
       $base$ - an IRI considered as base IRI
       $S2B$ - an injective function that maps every string to a unique blank node
       $\mathbb{P}$ - an injective partial function that maps attributes to query expressions
Output: an extend expression (as per Definition 8)
if $G$ contains a triple $(u, \texttt{rr:constant}, o)$ then return $o$
$\varphi \leftarrow$ initially empty extend expression
if $G$ contains a triple $(u, \texttt{rr:reference}, o)$ such that $o$ is a literal $(lex, dt)$ then
    $\varphi \leftarrow a$, where $a$ is the attribute in $\mathrm{dom}(\mathbb{P})$ such that $\mathbb{P}(a) = lex$
else if $G$ contains a triple $(u, \texttt{rr:template}, o)$ such that $o$ is a literal $(lex, dt)$ then
    $\bar{S} \leftarrow \textsc{Split}(lex)$                                   // Definition 16
    $\varphi \leftarrow (s_1, \texttt{xsd:string})$, where $s_1$ is the first element in $\bar{S}$
    foreach $s_i$ in $\bar{S}$ without the first element do        // Iterate based on the order in $\bar{S}$
        if $s_i$ is a string that starts with "{" and ends with "}" then
            $\varphi \leftarrow \texttt{concat}\big(\varphi, \mathbb{P}^{-1}(q)\big)$, where $q$ is $s_i$ without enclosing "{" and "}"
        else
            $\varphi \leftarrow \texttt{concat}\big(\varphi, (s_i, \texttt{xsd:string})\big)$
if $G$ contains the triple $(u, \texttt{rr:termType}, \texttt{rr:BlankNode})$ then return $\texttt{toBNode}^{S2B}(\varphi)$
else if $G$ contains $(u, \texttt{rr:datatype}, o)$ with $o \in \mathcal{I}$ then return $\texttt{toLiteral}(\varphi, o)$
else if $G$ contains $(u, \texttt{rr:termType}, \texttt{rr:Literal})$ then return $\texttt{toLiteral}(\varphi, \texttt{xsd:string})$
else if $G$ contains $(t, \texttt{rr:objectMap}, u)$ with $t \in \mathcal{I} \cup \mathcal{B}$ and $G$ contains
        $(u, \texttt{rml:reference}, o)$ with $o \in \mathcal{L}$ then return $\texttt{toLiteral}(\varphi, \texttt{xsd:string})$
else return $\texttt{toIRI}(\varphi, base)$
```

– `concat` is a binary extension function that concatenates two string literals.

The extend expression created by Algorithm 3 depends on whether the given term map is constant-valued (line 1), reference-valued (lines 3–4), or template-valued (lines 5–12). In the latter case, the corresponding template string is first split into a sequence of substrings (line 6), defined as follows.

**Definition 16.** For every string $tmpl \in \mathcal{S}$, we write $\textsc{Split}(tmpl)$ to denote a sequence $\bar{S}$ of strings that is constructed by the following two steps.

1. Partition $tmpl$ into a sequence $\bar{S}$ of *query substrings* and *normal substrings* in the order in which these substrings appear in $tmpl$ from left to right, where
   – a *query substring* is a substring starting with an opening curly brace (i.e., "{") and ending with a closing curly brace (i.e., "}"), and
   – a *normal substring* is every substring between two query substrings, as well as the one before the first and the one after the last query substring.
2. If $\bar{S}$ is the empty sequence, insert the empty string as a new substring into $\bar{S}$.

The extend expression is then created to concatenate the resulting substrings (lines 7–12), where every query substring "$\{q\}$" is replaced by the attribute $a$ for which it holds that $\mathbb{P}(a) = q$, because that attribute holds the values that the source operator obtains for the query expression $q$.

*Example 13.* Assume $b_{\mathsf{sm}}$ and $b_{\mathsf{om}}$ are the blank nodes denoting the subject map and the object map in RDF graph $G_{\mathsf{ex}}$ in Example 11, respectively, *base* is an arbitrary base IRI, and $\mathbb{P} = \{a_1 \rightarrow \mathsf{ID},\ a_2 \rightarrow \mathsf{Name}\}$ as in Example 12. Then, $\textsc{CreateExtExpr}(b_{\mathsf{om}}, G_{\mathsf{ex}}, \mathit{base}, \mathit{S2B}, \mathbb{P})$ returns the extend expression $\mathtt{toLiteral}(a_2, \mathtt{xsd{:}string})$. Moreover, $\textsc{CreateExtExpr}(b_{\mathsf{sm}}, G_{\mathsf{ex}}, \mathit{base}, \mathit{S2B}, \mathbb{P})$ returns the extend expression $\mathtt{toIRI}\big(\mathtt{concat}\big(\ell, a_1\big), \mathit{base}\big)$ where $\ell$ is the literal $($`"http://example.com/person_"`, `xsd:string`$)$.

***Translation of the Object Map.*** For object (term) maps, the algorithm also has to consider the case of referencing object maps (lines 11–20 in Algorithm 1), which capture a join with RDF terms defined by the subject map of another triples map. In this case, another source operator needs to be added (for this other triples map) and the mapping relation created by that source operator needs be joined with the mapping relation produced before (line 18). To this end, an equijoin operator is used, for which the join attributes are determined based on the join condition defined for the referencing object map (lines 13–17). Ordinary object maps are handled like subject and predicate maps before (line 22).

***Translation of the Graph Map.*** Finally, if the triples map has a graph map, the algorithm adds an extend operator, with attribute $a_{\mathrm{g}}$, to create IRIs defined by this graph map (lines 23–24). If the triples map has no graph map, an extend operator that assigns $a_{\mathrm{g}}$ to the constant IRI `rr:defaultGraph` is added (line 26).

## 6 Algebraic Equivalences

In addition to providing a basis to formally define the semantics of a mapping language such as RML, our mapping algebra may also be used as a foundation for implementing such languages in mapping tools. More precisely, such tools may use our algebra as a form of language to internally represent the plans that they create for converting data according to a given mapping description, and then they may use algebraic equivalences to rewrite initial mapping plans into more efficient ones (with the provable guarantee that the rewritten plans produce the same result!). While an extensive study of such optimization opportunities is beyond the scope of this paper, in this section we show a number of such equivalences and describe how they may be relevant for a plan optimizer.

### 6.1 Projection Pushing

The first group of equivalences focuses on pushing project operators deeper into a given plan. This may be useful to reduce the size of intermediate mapping relations in terms of values for attributes that are not used anymore later in the plan. Dropping such attributes from intermediate mapping relations as early as possible reduces the memory space required for executing the mapping plan.

As a typical example of an algebraic equivalence that an optimizer may use for this purpose, consider the following result, which shows that a project operator over an extend-based subexpression may be pushed into this subexpression if the set of projection attributes contains both the extend attribute and the attributes mentioned by the corresponding extend expression (if any).[6]

[6] We provide the proof of Proposition 1 in Appendix C. The following propositions in this section can be shown in the same way.

**Proposition 1.** Let $r = (A, I)$ be a mapping relation, $a \in \mathcal{A}$ be an attribute that is not in $A$ (i.e., $a \notin A$), $\varphi$ be an extend expression, and $P \subseteq A$ be a non-empty subset of the attributes of $r$. If $(\mathrm{attrs}(\varphi) \cap A) \subseteq P$, then it holds that

$$\mathrm{Project}^{P \cup \{a\}}\big(\mathrm{Extend}^{a}_{\varphi}(r)\big) = \mathrm{Extend}^{a}_{\varphi}\big(\mathrm{Project}^{P}(r)\big).$$

To satisfy the condition for the equivalence in Proposition 1 and, thus, to facilitate projection pushing based on this equivalence, in some cases, it may be necessary to first add another projection with additional projection attributes. The following result shows an equivalence that can be used for this purpose.

**Proposition 2.** Let $r = (A, I)$ be a mapping relation and $P \subseteq A$ be a non-empty subset of the attributes of $r$. For every $P' \subseteq A$, it holds that

$$\mathrm{Project}^{P}(r) = \mathrm{Project}^{P}\big(\mathrm{Project}^{P \cup P'}(r)\big).$$

While further equivalences for projection pushing can be shown (e.g., into a join), we also want to highlight the existence of cases in which a project operator can be removed completely. For instance, every project operator with a source operator as input may be removed, as shown by the following result.

**Proposition 3.** Let $s = (type, D)$ be a data source with $type = (\mathcal{D}^{\mathsf{ds}}, \mathcal{D}^{\mathsf{c1}}, \mathcal{D}^{\mathsf{c2}}, L, L', eval, eval', cast)$, let $q \in L$, let $\mathbb{P}$ be a partial function $\mathbb{P}\colon \mathcal{A} \to L'$, and $P \subseteq \mathrm{dom}(\mathbb{P})$. It holds that $\mathrm{Project}^{P}\big(\mathrm{Source}^{(s,q,\mathbb{P})}\big) = \mathrm{Source}^{(s,q,\mathbb{P}')}$, where $\mathbb{P}'$ is the restriction of $\mathbb{P}$ to $P$; i.e., $\mathbb{P}'(a) = \mathbb{P}(a)$ for all $a \in \mathrm{dom}(\mathbb{P}') = P$.

### 6.2 Pushing or Pulling Extend Operators

The extended version of this paper shows a second group of equivalences which focus on moving extend operators either on top of a join-based subplan or under the corresponding join operator (i.e., into one of the inputs of the join) [13].

## 7 Concluding Remarks and Future Work

The work presented in this paper opens the door for several new directions of research in the domain of mappings-based KG construction. First and foremost, our approach may be used to formalize other mapping languages besides RML. In this context we emphasize that, while the five types of operators defined in this paper are sufficient to capture the expressive power of RML, the algebra may easily be extended with additional types of operators if needed for a particular language or use case. Similarly, the algebraic equivalences shown in this paper may be complemented with further equivalences. A natural next step then is to develop concrete implementation and optimization techniques based on the algebra and the equivalences. Given the language-agnostic nature of the algebra, it even becomes trivial for resulting engines to support multiple languages.

**Acknowledgments.** The presented work was supported by the imec.icon project PACSOI (HBC.2023.0752), which was co-financed by imec and VLAIO and brings together the following partners: FAQIR Foundation, FAQIR Institute, MoveUP, Byteflies, AContrario, and Ghent University – IDLab, and by the European Union's Horizon Europe research and innovation programme under grant agreement no. 101058682.

**Disclosure of Interests.** The authors have no competing interests to declare.

## References

1. Arenas-Guerrero, J., Chaves-Fraga, D., Toledo, J., Pérez, M.S., Corcho, O.: Morph-KGC: Scalable Knowledge Graph Materialization with Mapping Partitions. Semantic Web **15**(1), 1–20 (2022). `https://doi.org/10.3233/sw-223135`
2. Bischof, S., Decker, S., Krennwallner, T., Lopes, N., Polleres, A.: Mapping between RDF and XML with XSPARQL. Journal on Data Semantics **1**(3), 147–185 (2012). `https://doi.org/10.1007/s13740-012-0008-7`
3. Daga, E., Asprino, L., Mulholland, P., Gangemi, A.: Facade-X: An Opinionated Approach to SPARQL Anything. In: Proceedings of the 17th International Conference on Semantic Systems (SEMANTiCS). Studies on the Semantic Web, vol. 53, pp. 58–73. IOS Press (2021). `https://doi.org/10.3233/SSW210035`
4. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: Proceedings of the 7th Workshop on Linked Data on the Web (LDOW). CEUR Workshop Proceedings, vol. 1184. CEUR (2014), `http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf`
5. Dimou, A., Vander Sande, M., De Meester, B., Heyvaert, P., Delva, T.: RDF Mapping Language (RML). Version 1.1.2; online at `https://rml.io/specs/rml/v/1.1.2/` (Jun 2024)
6. Freund, M., Schmid, S., Dorsch, R., Harth, A.: FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer. In: Proceedings of the 21st Extended Semantic Web Conference (ESWC). pp. 40–56. Springer Nature Switzerland, Cham (2024). `https://doi.org/10.1007/978-3-031-60635-9_3`
7. Heyvaert, P., Dimou, A., Chaves-Fraga, D.: RML Implementation Report. Online at `https://rml.io/implementation-report/` (Feb 2022)
8. Heyvaert, P., Dimou, A., De Meester, B.: RML Test Cases. Online at `https://rml.io/test-cases/` (Mar 2019)
9. Iglesias, E., Jozashoori, S., Vidal, M.E.: Scaling Up Knowledge Graph Creation to Large and Heterogeneous Data Sources. Journal of Web Semantics **75** (2023). `https://doi.org/10.1016/j.websem.2022.100755`
10. Iglesias-Molina, A., Assche, D.V., Arenas-Guerrero, J., Meester, B.D., Debruyne, C., Jozashoori, S., Maria, P., Michel, F., Chaves-Fraga, D., Dimou, A.: The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF. In: Proceedings of the 22nd International Semantic Web Conference (ISWC). Lecture Notes in Computer Science, vol. 14266, pp. 152–175. Springer (2023). `https://doi.org/10.1007/978-3-031-47243-5_9`
11. Lefrançois, M., Zimmermann, A., Bakerally, N.: A SPARQL Extension for Generating RDF from Heterogeneous Formats. In: Proceedings of the 14th Extended Semantic Web Conference (ESWC). pp. 35–50. Springer International Publishing (2017). `https://doi.org/10.1007/978-3-319-58068-5_3`
12. Lopes, N., Bischof, S., Decker, S., Polleres, A.: On the Semantics of Heterogeneous Querying of Relational, XML and RDF Data with XSPARQL. In: Proceedings of the 15th Portuguese Conference on Artificial Intelligence (EPIA). pp. 10–13 (2011)
13. Min Oo, S., Hartig, O.: An Algebraic Foundation for Knowledge Graph Construction (Extended Version). CoRR **arXiv/2503.10385** (2025), online at `https://arxiv.org/abs/2503.10385`
14. Min Oo, Sitt and De Meester, Ben and Taelman, Ruben and Colpaert, Pieter: Towards Algebraic Mapping Operators for Knowledge Graph Construction. In:

Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks. CEUR Workshop Proceedings, vol. 3632. CEUR (2023), https://ceur-ws.org/Vol-3632/ISWC2023_paper_412.pdf

15. Rodríguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML Mappings. Journal of Web Semantics **33**, 141–169 (2015). https://doi.org/10.1016/j.websem.2015.03.001
16. Rojas, J.A., Aguado, M., Vasilopoulou, P., Velitchkov, I., Van Assche, D., Colpaert, P., Verborgh, R.: Leveraging Semantic Technologies for Digital Interoperability in the European Railway Domain. In: Proceedings of the 20th International Semantic Web Conference (ISWC). pp. 648–664. Springer (2021)
17. Sakor, A., Jozashoori, S., Niazmand, E., Rivas, A., Bougiatiotis, K., Aisopos, F., Iglesias, E., Rohde, P.D., Padiya, T., Krithara, A., Paliouras, G., Vidal, M.E.: Knowledge4COVID-19: A Semantic-Based Approach for Constructing a COVID-19 Related Knowledge Graph from Various Sources and Analyzing Treatments' Toxicities. Journal of Web Semantics **75** (2023). https://doi.org/10.1016/j.websem.2022.100760
18. Scrocca, M., Comerio, M., Carenini, A., Celino, I.: Turning Transport Data to Comply with EU Standards While Enabling a Multimodal Transport Knowledge Graph. In: Proceedings of the 19th International Semantic Web Conference (ISWC). pp. 411–429. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-62466-8_26
19. Sequeda, J.F.: On the Semantics of R2RML and its Relationship with the Direct Mapping. In: Proceedings of the ISWC 2013 Posters & Demonstrations Track. CEUR Workshop Proceedings, vol. 1035. CEUR (2013), https://ceur-ws.org/Vol-1035/iswc2013_poster_4.pdf
20. Sitt Min Oo, Haesendonck, G., De Meester, B., Dimou, A.: RMLStreamer-SISO: An RDF Stream Generator from Streaming Heterogeneous Data. In: Proceedings of the 21st International Semantic Web Conference (ISWC). pp. 697–713. Springer (2022). https://doi.org/10.1007/978-3-031-19433-7_40
21. Sundara, S., Das, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language. W3C recommendation, W3C (Sep 2012), https://www.w3.org/TR/2012/REC-r2rml-20120927/
22. Tailhardat, L., Chabot, Y., Troncy, R.: NORIA-O: An Ontology for Anomaly Detection and Incident Management in ICT Systems. In: Proceedings of hte 21st Extended Semantic Web Conference (ESWC). pp. 21–39. Springer Nature Switzerland (2024). https://doi.org/10.1007/978-3-031-60635-9_2
23. Van Assche, D., Delva, T., Haesendonck, G., Heyvaert, P., De Meester, B., Dimou, A.: Declarative RDF Graph Generation from Heterogeneous (Semi-)Structured Data: A Systematic Literature Review. Journal of Web Semantics **75** (2022). https://doi.org/10.1016/j.websem.2022.100753

## A SPARQL Update Queries to Create Normalized RML

```
DELETE { ?sm rr:class ?sm_class . }
INSERT { ?tm rr:predicateObjectMap [
                rr:predicateMap [
                    rr:termType rr:IRI ;
                    rr:constant rdf:type ] ;
                rr:objectMap [
                    rr:termType rr:IRI ;
                    rr:constant ?sm_class ] ] }
WHERE { ?tm rr:subjectMap ?sm .
        ?sm rr:class ?sm_class . }
```

**Query 1.** Normalization step 1 (expand shortcuts for class IRIs).

```
DELETE { ?tm rr:predicateObjectMap  ?pom.
         ?pom rr:predicateMap ?pm ;
              rr:objectMap ?om ;  rr:graphMap ?gm }
INSERT { ?tm rr:predicateObjectMap [
                      rr:predicateMap ?pm ;
                      rr:objectMap ?om ;
                      rr:graphMap ?gm ] }
WHERE {    ?tm rr:predicateObjectMap ?pom .
           ?pom rr:predicateMap ?pm ;
           ?pom rr:objectMap ?om
 OPTIONAL {?pom rr:graphMap ?gm} }
```

**Query 2.** Normalization step 3 (duplicate multi-predicate-object maps into singletons).

```
DELETE {  ?tm   rr:subject ?sm_constant .   ?pompm rr:predicate ?pm_constant .
          ?termMap rr:graph ?gm_constant .  ?pomom rr:object  ?om_constant .   }
INSERT { ?tm      rr:subjectMap   [ rr:constant ?sm_constant ].
         ?pompm   rr:predicateMap [ rr:constant ?pm_constant ].
         ?pomom   rr:objectMap    [ rr:constant ?om_constant ].
         ?termMap rr:graphMap     [ rr:constant ?gm_constant ]. }
WHERE { { ?tm      rr:subject   ?sm_constant }
  UNION { ?pompm   rr:predicate ?pm_constant }
  UNION { ?pomom   rr:object    ?om_constant }
  UNION { ?termMap rr:graph     ?gm_constant }  }
```

**Query 3.** Normalization step 2 (expand shortcuts for constant-valued term maps).

```
DELETE { ?om rr:parentTriplesMap ?ptm }
INSERT { ?om rr:reference  ?ref ;
             rr:template ?template;
             rr:constant ?const ;
             rr:termType rr:IRI . }
WHERE { ?om rr:parentTriplesMap ?ptm .
        ?ptm rr:subjectMap ?sm .
  OPTIONAL{ ?sm rr:reference ?ref }
  OPTIONAL{ ?sm rr:template ?template }
  OPTIONAL{ ?sm rr:constant ?const }
  FILTER NOT EXISTS { ?om rr:joinCondition ?jc } }
```

**Query 4.** Normalization step 4 (replace referencing object maps without join conditions).

```
DELETE { ?tm rdf:type rr:TriplesMap ;
             rml:logicalSource ?ls ;
             rr:subjectMap ?sm ;
             rr:predicateObjectMap ?pom }
INSERT { [] rml:logicalSource ?ls ;
            rr:subjectMap ?sm ;
            rr:predicateObjectMap ?pom }
WHERE { ?tm rml:logicalSource ?ls ;
            rr:subjectMap ?sm ;
            rr:predicateObjectMap ?pom }
```

**Query 5.** Normalization step 5 (duplicate triples maps that contain multiple pred.-object maps).

```
DELETE { ?tm rr:predicateObjectMap ?pom .
         ?pom rr:graphMap ?pom_gm .       }
INSERT { [] rml:logicalSource ?ls ;
            rr:subjectMap [
                 rr:reference ?ref ;
                 rr:template ?template ;
                 rr:constant ?const ;
                 rr:termType ?ttype ;
                 rr:graphMap ?sm_gm ;
                 rr:graphMap ?pom_gm ] ;
            rr:predicateObjectMap ?pom .  }
WHERE { ?tm rml:logicalSource ?ls ;
            rr:subjectMap ?sm ;
            rr:predicateObjectMap ?pom .
        ?pom rr:graphMap ?pom_gm .
  OPTIONAL { ?sm rr:graphMap ?sm_gm }
  OPTIONAL { ?sm rr:reference ?ref }
  OPTIONAL { ?sm rr:template ?template }
  OPTIONAL { ?sm rr:constant ?const }
  OPTIONAL { ?sm rr:termType ?ttype }     }
```

**Query 6.** Normalization step 6a (duplicate triples maps with subject maps where the predicate-object maps contain multiple graph maps).

```
DELETE { ?sm rr:graphMap ?gm1 . }
INSERT { [] rml:logicalSource ?ls ;
            rr:subjectMap [
                 rr:reference ?ref ;
                 rr:template ?template ;
                 rr:constant ?const ;
                 rr:termType ?ttype ;
                 rr:graphMap ?gm1 ] ;
            rr:predicateObjectMap ?pom }
WHERE { ?tm rml:logicalSource ?ls ;
            rr:subjectMap ?sm ;
            rr:predicateObjectMap ?pom .
        ?sm rr:graphMap ?gm1 ;
            rr:graphMap ?gm2
        FILTER ( ?gm1 != ?gm2 )
  OPTIONAL { ?sm rr:reference ?ref }
  OPTIONAL { ?sm rr:template ?template }
  OPTIONAL { ?sm rr:constant ?const }
  OPTIONAL { ?sm rr:termType ?ttype }  }
```

**Query 7.** Normalization step 6b (duplicate subject maps that contain multiple graph maps).

## B Definition of Relevant Extension Functions

For every $v \in (\mathcal{T} \cup \epsilon)$, $dt \in (\mathcal{T} \cup \epsilon)$, $v' \in (\mathcal{T} \cup \epsilon)$, and $base \in \mathcal{I}$, we define:

$$\texttt{toBNode}^{S2B}(v) = \begin{cases} S2B(lex) & \text{if } v \text{ is a literal } (lex, dt) \text{ s.t. } dt = \texttt{xsd:string}, \\ \epsilon & \text{else.} \end{cases}$$

$$\texttt{toLiteral}(v, dt) = \begin{cases} (\mathit{lex}, dt) & \text{if } v \text{ is a literal } (\mathit{lex}, dt') \text{ s.t. } dt = \texttt{xsd:string}, \\ \epsilon & \text{else.} \end{cases}$$

$$\texttt{toIRI}(v, \mathit{base}) = \begin{cases} \mathit{lex} & \text{if } v \text{ is a literal } (\mathit{lex}, dt) \text{ such that } \mathit{lex} \text{ is} \\ & \text{a valid IRI and } dt \text{ is the IRI } \texttt{xsd:string}, \\ \mathit{base} \circ \mathit{lex} & \text{if } v \text{ is a literal } (\mathit{lex}, dt) \text{ such that } \mathit{lex} \text{ is not} \\ & \text{a valid IRI, but } \mathit{base} \circ \mathit{lex} \text{ is a valid IRI, and} \\ & dt \text{ is the IRI } \texttt{xsd:string}, \\ \epsilon & \text{else.} \end{cases}$$

$$\texttt{concat}(v, v') = \begin{cases} (\mathit{lex} \circ \mathit{lex}', \texttt{xsd:string}) & \text{if } v \text{ and } v' \text{ are literals } (\mathit{lex}, dt) \text{ and} \\ & (\mathit{lex}', dt'), \text{ respectively, such that} \\ & dt = dt' = \texttt{xsd:string}, \\ \epsilon & \text{else,} \end{cases}$$

where $\mathit{lex} \circ \mathit{lex}'$ is the string obtained from concatenating the strings $\mathit{lex}$ and $\mathit{lex}'$.

## C Proof of Proposition 1

We assume that $(\mathrm{attrs}(\varphi) \cap A) \subseteq P$, and let $(A_1, I_1) = \mathrm{Project}^{P \cup \{a\}}\big(\mathrm{Extend}^a_\varphi(r)\big)$ and $(A_2, I_2) = \mathrm{Extend}^a_\varphi\big(\mathrm{Project}^P(r)\big)$. To show that $(A_1, I_1) = (A_2, I_2)$ we first note that, by Definition 11, it holds that $A_1 = P \cup \{a\}$, and by Definition 10, $A_2 = P \cup \{a\}$. Hence, $A_1 = A_2$ and, thus, it remains to show that $I_1 = I_2$.

We first show that $I_1 \subseteq I_2$, for which we consider an arbitrary mapping tuple $t \in I_1$ and show that $t \in I_2$. Given that $t \in I_1$, by Definition 11, it holds that i) $\mathrm{dom}(t) = P \cup \{a\}$ and ii) there exists a mapping tuple $t' \in \mathrm{Extend}^a_\varphi(r)$ such that $t(a') = t'(a')$ for all $a' \in P \cup \{a\}$. Since $t' \in \mathrm{Extend}^a_\varphi(r)$, by Definition 10, it holds that i) $\mathrm{dom}(t') = A \cup \{a\}$ and ii) there exists a mapping tuple $t'' \in r$ such that $\mathit{eval}(\varphi, t'') = t'(a)$ and $t''(a') = t'(a')$ for all $a' \in A$. Now, let $t'''$ be the restriction of $t''$ to $P$; i.e., $t''' = t''[P]$, which, by Definition 11, also means that $\mathrm{dom}(t''') = P$ and $t''' \in \mathrm{Project}^P(r)$. Then, since $(\mathrm{attrs}(\varphi) \cap A) \subseteq P$ and $t'''(a') = t''(a')$ for all $a' \in P$, it holds that $\mathit{eval}(\varphi, t''') = \mathit{eval}(\varphi, t'')$. As a consequence, there exists a tuple $t^{(4)} \in \mathrm{Extend}^a_\varphi\big(\mathrm{Project}^P(r)\big)$ such that i) $\mathrm{dom}(t^{(4)}) = \mathrm{dom}(t''') \cup \{a\}$, ii) $t^{(4)}(a) = t'(a)$, and iii) $t^{(4)}(a') = t'''(a')$ for all $a' \in P$. Since $\mathrm{dom}(t''') \cup \{a\} = P \cup \{a\} = \mathrm{dom}(t)$ and $t'(a) = t(a)$ and $t'''(a') = t''(a') = t'(a') = t(a')$ for all $a' \in P$, we can conclude that $t^{(4)} = t$ and, thus, $t \in \mathrm{Extend}^a_\varphi\big(\mathrm{Project}^P(r)\big)$; i.e., $t \in I_2$.

Now we show that $I_1 \supseteq I_2$, for which we consider a tuple $t \in I_2$ and show that $t \in I_1$. Given that $t \in I_2$, by Definitions 10 and 11, it holds that i) $\mathrm{dom}(t) = \{a\} \cup P$ and ii) there exists a mapping tuple $t' \in \mathrm{Project}^P(r)$ such that $t(a) = \mathit{eval}(\varphi, t')$ and $t(a') = t'(a')$ for all $a' \in P$. Since $t' \in \mathrm{Project}^P(r)$, it holds that i) $\mathrm{dom}(t') = P$ and ii) there exists a mapping tuple $t'' \in r$ such that $\mathrm{dom}(t'') = A$ and $t''(a') = t'(a')$ for all $a' \in P$. Now, let $t'''$ be the mapping tuple for which it holds that i) $\mathrm{dom}(t''') = A \cup \{a\}$, ii) $t'''(a) = \mathit{eval}(\varphi, t'')$, and iii) $t'''(a') = t''(a')$ for all $a' \in A$. Hence, by Definition 10, $t''' \in \mathrm{Extend}^a_\varphi(r)$. Next, let $t^{(4)}$ be the restriction of $t'''$ to $P \cup \{a\}$; i.e., $t^{(4)} = t'''[P \cup \{a\}]$, which, by Definition 11, means that $t^{(4)} \in \mathrm{Project}^{P \cup \{a\}}\big(\mathrm{Extend}^a_\varphi(r)\big)$. Putting everything together, we have that i) $\mathrm{dom}(t^{(4)}) = P \cup \{a\} = \mathrm{dom}(t)$, ii) $t^{(4)}(a) = t'''(a) = \mathit{eval}(\varphi, t'') = \mathit{eval}(\varphi, t') = t(a)$, and iii) $t^{(4)}(a') = t'''(a') = t''(a') = t(a)$ for all $a' \in P$. As a consequence, $t^{(4)} = t$ and, thus, $t \in \mathrm{Project}^{P \cup \{a\}}\big(\mathrm{Extend}^a_\varphi(r)\big)$; i.e., $t \in I_1$.