Correspondence and Translation for Heterogeneous Data *

Serge Abiteboul
I.N.R.I.A., Rocquencourt
France
(Serge.Abiteboul@inria.fr)

Sophie Cluet
I.N.R.I.A., Rocquencourt
France
(Sophie.Cluet@inria.fr)

Tova Milo
Tel-Aviv U.
Israel
(milo@math.tau.ac.il)

December 14, 2000

1 Introduction

A primary motivation for new database technology is to provide support for the broad spectrum of multimedia data available notably through the network. These data are stored under different formats: relational or ODMG [1] model (in databases), SGML or LaTex (documents), DX formats (data exchange formats in scientific data), Step (CAD/CAM data), etc. Their integration is a very active field of research and development (see for instance, for a very small sample, [13, 9, 10, 12, 11, 15, 25, 26]). In this paper, we provide a formal foundation to facilitate the integration of such heterogeneous data and the maintenance of heterogeneous replicated data.

A sound solution for a data integration task requires a clean abstraction of the different formats in which data are stored, and means for specifying the correspondences/relationships between data in different worlds and for translating data from one world to another. For that, we introduce a *middleware* data model that serves as a basis for the integration task, and *declarative rules* for specifying the integration.

The choice of the *middleware* data model is clearly essential. One common trend in data integration over heterogeneous models has always been to use an integrating model that encompasses the source models. We take an opposite approach here, i.e., our model is minimalist. The data structure we use consists of ordered labeled trees. We claim that this simple model is general enough to capture the essence of formats we are interested in. Even though a mapping from a richer data model to this model may loose some of the original semantics, the data itself is preserved and the integration with other data models is facilitated. Our model is similar to the one used in [10] and to the OEM model for unstructured data (see, e.g., [27, 26]). This is not surprising since the data formats that motivated these works are part of the formats that our framework intends to support. A difference with the OEM model is that we view the children of each vertex as ordered. This is crucial to describe lists, an essential component of DX formats. Also, [17] introduces BNF generated trees to unify hierarchical data models. However, due to the fixed number of children of a rule, collections are represented by left or right deep trees not suitable for the casual users.

A main contribution of the paper is in the declarative specification of correspondences between data in different worlds. For this we use datalog-style rules, enriched with, as a novel feature, *merge* and *cons* term constructors. The semantics of the rules takes into consideration the fact that some

*The work was partially supported by AFIRST and by the Israely Ministry of Science. A preliminary version of this work was presented in ICDT97 [5]

[1] ODMG stands for the *Object Data Management Group* ODMG that specifies the standard model for object-oriented databases.

internal vertices represent collections with specific properties (e.g., sets are insensitive to order and duplicates). We show that correspondences between data elements can be computed in polynomial time in many cases, and may require exponential time only when insensitivity to order or duplicates are considered.

Deriving correspondences within existing data is only one issue in a heterogeneous context. One would also want to translate data from one representation to another. Interestingly, we show that in most practical cases, translation rules can automatically be derived from the correspondence rules. Thus, a complete integration task (derivation of correspondences, transformation of data from one world to the other, incremental integration of a new bulk of data, etc.) can be specified using a *single* set of declarative rules. This is an important result. It simplifies the specification task and also helps in preventing inconsistencies in specifications.

It should be noted that the language we use to define correspondence rules is rather limited. Similar correspondences could be easily derived using more powerful languages previously proposed (e.g., LDL [8] or IQL [7]). But in these languages it would be much more difficult (sometimes impossible) to derive translation rules from given correspondence rules. Nevertheless, our language is expressive enough to describe many desired correspondences/translations.

As will be seen, correspondence rules have a very simple and intuitive graphical representation. Indeed, the present work served as the basis for building a data translation system system, where a specification of integration of heterogeneous data proceeds in two phases. In a first phase, data is abstracted to yield a tree-like representation that is hiding details unnecessary to the restructuring (e.g., tags or parsing information). In a second phase, available data is displayed in a graphical window and starting from that representation, the user can specify correspondences or derive data.

Motivations The work we present in this paper was motivated by an on-going project (see, [3, 4, 2]) whose goal was the integration of SGML documents [16] in an object-oriented database (namely O_2 [14]). We developed tools to load the documents in the database and down-load portions of the database as SGML documents. We also considered the maintenance of the same data in the two forms. For query optimization and update purposes, we soon realized that we had to maintain correspondences between these two representations of data. Two main conclusions were drawn from that experience:

1. The task was complicated by numerous technical aspects of the specific data sources that were not really relevant to the translation process (for example SGML parsing). This motivated the choice of a simple uniform middleware model. The formats we are interested in and most formats we are aware of can easily be mapped to this model. The implementation of this mapping has to be done only once. From there on, the person implementing the data integration process faces a uniform and simple tree representation of the data from both worlds.

2. If one starts with an arbitrary specification of a mapping/translation from one data representation to another, it is virtually impossible to invert that mapping, or to automatically derive correspondences between data elements. This motivated the development of a unique specification that serves all purposes – a specification of correspondence between the two representations, which under some reasonable restrictions can be transformed into specification of mappings/translations in both directions.

Our approach to the integration task is rule-based. Rules can be used in a number of ways within an integration process. Suppose for instance that the same data can be represented in

SGML files and in an object-oriented database (our original motivation). Then, one can use several sets of rules:

1. If we already have the data from both worlds, we need rules to specify the correspondences between data elements in the different worlds.

2. If we have SGML data, then we can use rules to derive OODB data.

3. Conversely, if we have OODB data, rules can specify the translation to SGML data.

4. Suppose that data is physically stored in only one form (say SGML), and the second representation is only virtual. Rules may allow to translate queries and updates specified on the virtual representation.

5. Finally, rules can be used to specify the propagation of (bulk) updates from one world to the other.

Obviously, one would like to avoid having to write rules for each one of these cases individually. Once the correspondence existing between two worlds has been specified via correspondence rules, the system should be able to derive new rules allowing the translation from one world to the other and vice-versa.

The contribution of this work is therefore in (1) the focus on a single data model supporting ordered data collections motivated by data exchange formats, (2) the definition of a single "all purpose" rule-based declarative specification, used for defining both correspondences between data elements and bi-directional data translation, where the defined correspondence can serve as basis for the other purposes mentioned above, and (3) the formal study of the computational cost of the integration task.

The paper is organized as follows. Section 2 introduces a core data model and Section 3 a core language for specifying correspondences. In Section 4, we extend the framework to better deal with collections. Section 5 deals with the translation problem. The last section is a conclusion. (Two technical proofs are given in Appendix A and B.)

2 The Data Model

To reason about the structure of data coming from various sources and on the possible correspondence between different pieces of data, we need a common data model in which the sources data can be naturally represented. Our goal is to provide a data model that allows declarative specifications of the correspondence between data stored in different worlds (DX, ODMG, SGML, etc.). In this section, we introduce the data model.

For illustration purposes, we use the following example. A simple instance of an SGML document is given in Figure 1. A tree representation of the document in our middleware model, together with correspondences between this tree and a forest representation of the reference for this document in an object database is given in Figure 2.

2.1 Data Forest

We assume the existence of some infinite sets: (i) **name** of names; (ii) **dom** of data values; and (iii) **vertex** of vertices. We also assume that each vertex has a unique identifier. Identifiers are denoted by $\&i$ where i is an integer. The identifier of a vertex $v \in$**vertex** is denoted $id(v)$. For

<article status=final>
<title> Correspondence and Translation for Heterogeneous Data < \title>
<author> Serge Abiteboul
<author> Sophie Cluet
<author> Tova Milo
<abstract>
A primary motivation for new database technology is to provide support
for the broad spectrum of multimedia data available notably through
...
< \abstract>
<body>
...
< \body>
< \article>

Figure 1: An instance of an SGML document

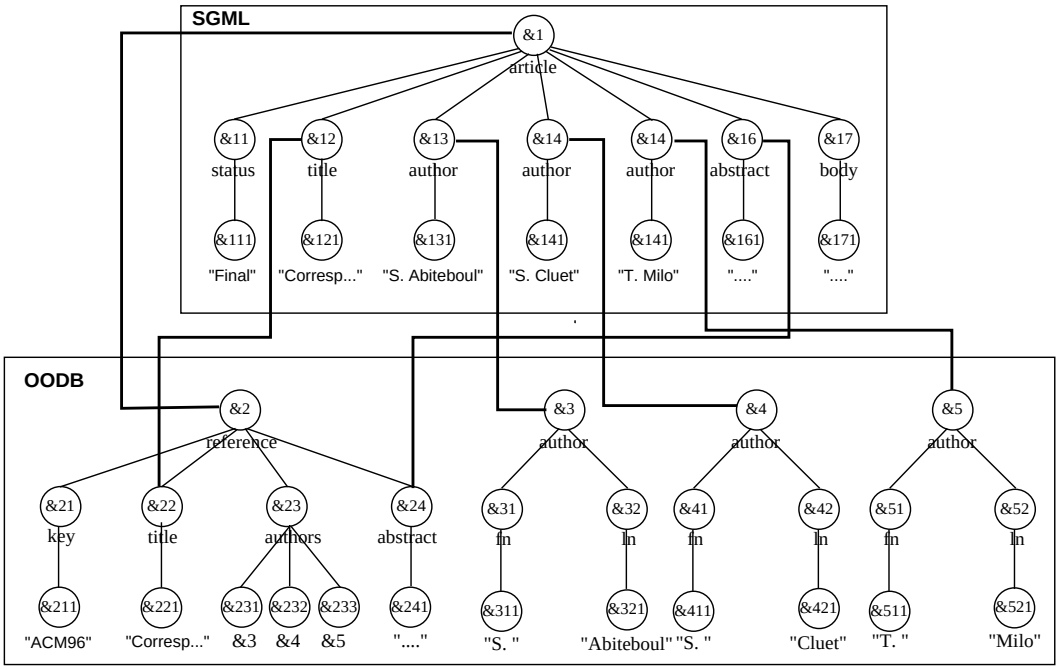

Figure 2: Correspondence between SGML file and an OODB

brevity, when things are clear from the context, we will sometimes abbreviate in the sequel $id(v)$ by $\&v$.

A *data forest* is a forest of *ordered labeled trees*. An *ordered labeled tree* is a tree with a labeling of vertices and for each vertex, an ordering of its children. The internal vertices of the trees have labels from **name** whereas the leaves have labels from **name** $\cup$ **dom** $\cup$ **vertex**. The only constraint is that if a vertex occurs as a leaf label, it should also occur as a vertex in the forest. Observe that this is a rather conventional tree structure. This is a data model in the spirit of the complex value model [22, 1, 14] and many others. It is particularly influenced by models for unstructured data [27, 26] and the tree model of [10]. A particularity is the ordering of vertices that is important to model data formats essentially described by files obeying a certain grammar (e.g., SGML).

Consider the data in Figure 2. Some leaves with labels in **dom** denote atomic values, e.g., "ACM96". Other leaves with labels in **vertex** are used as references to other objects, i.e., roots of trees. This is the case for leaf &232 which refers to the object corresponding to the tree for one author, "S. Cluet". Finally, internal leaves have labels in **name** and provide some semantic (type) information such as *author* or *title*.

Definition 2.1 A *data forest* F is a quadruple (V, E, L, ω), where

- (V, E) is a finite *ordered* forest with set V of vertices from **vertex** and set of edges E;
- ω the ordering function that orders the outgoing edges of each vertex (i.e., it maps the outgoing edges of the vertex to an initial segment of the natural numbers);
- L, the *labeling function*, maps some leaves in V to $V \cup$ **dom** $\cup$ **name**, and non-leaf vertices to **name**.

The set of vertices V of a forest F is denoted $vertex(F)$, and the sets of data values and names appearing in F are denoted $dom(F)$ and $label(F)$ respectively.

Remark 2.2 Observe that by definition, we allow a leaf to be mapped to a name. For all purposes, we may think of such leaves as internal vertices without children. This will turn useful to represent for instance the empty set or the empty list. In the following, we use the word *leaf* only to vertices v such that $L(v)$ is a vertex or is in **dom**.

We illustrate this notion, as well as the syntactic representations we use, in an example. Consider the graphical representation of the forest describing the OODB, shown in the lower part of Figure 2. A tabular representation of part of the same forest is given in Figure 3.

Finally, below is an equivalent textual representation:

&2 *reference* { &21 *key* { &211 "*ACM96*" { } },
&22 *title* { &221 "*Correspondence...*" { } },
&23 *authors* { &231 &3 { }, &232 &4 { }, &233 &5 { } },
&24 *abstract* { &241"..." { } } }
...

In this textual representation each vertex v is represented by its identifier $id(v)$, followed by the vertex label $L(v)$, followed by brackets containing a (possibly empty) sequence of representations for the vertex children.

For brevity, we will sometimes use in the sequel a more compact syntax and omit brackets when a vertex has a single or no children, and also omit vertex identifiers and labels when they are irrelevant for the discussion. For example the above reference tree may be abbreviated by

vertex	*label*	*children*
&2	*reference*	{&21, &22, &23, &24}
&21	*key*	{&211}
&211	"*ACM96*"	{ }
&22	*title*	{&221}
&221	"*Correspondence...*"	{ }
&23	*authors*	{&231, &232, &233}
&231	&3	{ }
&232	&4	{ }
&233	&5	{ }
&24	*abstract*	{&241}
&241	"..."	{ }
&3	*author*	{&31, &32}
&31	*fn*	{&311}
&311	"*S.*"	{ }
&32	*ln*	{&321}
&321	"*Abiteboul*"	{ }
...		

Figure 3: A Data Forest

&2 *reference* { *key* "*ACM96*",
&22 *title* "*Correspondence...*",
authors{ &3, &4, &5 },
&24 *abstract* "..." }.

To illustrate how data sources can be mapped into our middleware model, we now consider three common and essential kinds of data. The first concerns relational databases and more generally all simple table-based formats. The second is used for object-oriented databases, and more generally most graph formats. Finally, the last one concerns formats based on a BNF grammar description. In each case, the mapping from the original data to the middleware model is one-to-one, and an inverse mapping (decoding) can easily be obtained.

A relation can be represented by a tree whose root label is the relation name and which has as many children as rows in the relation. At depth 2, vertices represent rows and are labeled by the label "tuple". At depth 3, 4 and 5, vertices are labeled respectively by attribute names, types and values. (This representation is clearly redundant and an alternative representation that would avoid repeating the type can easily be specified.)

An object oriented database can be viewed as a cyclic graph. However, using object identifier one may easily represents a cyclic graph as a tree, see, e.g., [7]. We consider here the ODMG model and a possible representation for it. An object is represented by the root vertex of a tree representing the value of the object as follows:

- For an atomic value, we use a vertex labeled by the atomic type and whose unique child is labeled by the appropriate atomic value; and for references, we use a vertex labeled by the object identifier of the vertex corresponding to the referenced object.

- For tuples, we use a vertex labeled "tuple" with one child per attribute; the child, grand child and great grand child are labeled respectively with the attribute name, the type and the value as in the relation case.
- For collections (i.e. sets, lists, bags), we use a vertex labeled by the collection type (set, list, bag), with as many children as elements in the collection. Each such child node is labeled by the element type and has a single child labeled by the element value.

Class extents are represented by a particular tree whose root vertex is labeled with the keyword *class*. There is one child for each class extent labeled with the class name. The vertex for a class c has as many children as there are objects in the extent of c labeled by the identifiers of the vertices corresponding to the objects in the extent of c. Roots of persistence are handled similarly.

A document can be described by a simplified representation of its parse tree. The labels of the internal vertices (resp. leaves) represent the grammar non-terminal symbols (resp. tokens). SGML and HTML, among other formats, allow references to internal and external data. Parsers do not interpret these references. They usually consider them as strings. In our context, these references should be interpreted when possible. Like for object databases, the reference can be replaced by the identifier of the vertex containing the referred data.

Note that the only identification of data in the middleware model is given by the vertex identifiers. This means that it is the responsibility of the data sources (or the wrappers of these sources) to maintain the relationships between the exported data and the vertex identifiers. This relationship is not always needed (e.g., for a translation process), and may be of a fine or large grain depending on the application needs and the data source capacities.

The identification of data in the data sources may take various forms. For relational databases, it may be based on keys or on some internal address. For object databases, it may be based, for instance, on the internal or external object identifier, on a query yielding the particular object, or on keys as in the relational case. For files, it may, for instance, be based on an offset in the file, or on some identification of a vertex in the parse tree.

2.2 Correspondence

We are concerned with establishing/maintaining correspondences between objects. Some objects may come from one data source with a particular forest F_1, and others from another forest, say F_2. To simplify, we consider here that we have a single forest (that can be viewed as the union of all the data forests) and look for correspondences *within* the forest. If we feel it is essential to distinguish between the sources, we may assume that the vertices of each tree from a particular data source have the name of that source, e.g., F_1, F_2, as part of the label. We describe correspondences between objects using particular relations. This is illustrated first with an example.

Example 2.3 Consider the forest including the SGML and OODB trees of Figure 2.

We may want to have the following correspondences:

$$\{\ is(\&1,\&2),\ is(\&12,\&22),\ is(\&13,\&3),\ is(\&14,\&4),\ is(\&15,\&5),\ is(\&16,\&24)$$
$$concat(\text{“}S.Abiteboul\text{”}, \text{“}S.\text{”}, \text{“}Abiteboul\text{”}),\ \ concat(\text{“}S.Cluet\text{”}, \text{“}S.\text{”}, \text{“}Cluet\text{”}),$$
$$concat(\text{“}T.Milo\text{”}, \text{“}T.\text{”}, \text{“}Milo\text{”})\ \ \}.$$

Note that there is an essential difference between the two predicates above: *is* relates objects that represent the same real world entity, whereas *concat* is a standard concatenation predicate/function that is defined externally. The *is*-relationship is represented in Figure 2.

Definition 2.4 Let $\mathbf{R}$ be a relational schema. An $\mathbf{R}$-*correspondence* is a pair (F, I) where F is a data forest and I a relational instance over $\mathbf{R}$ with values in $vertex(F) \cup dom(F)$.

For instance, consider Example 2.3. Let $\mathbf{R}$ consist of a binary relation *is* and a ternary one *concat*. For the forest F and correspondences I as in the above example, (F, I) is an $\mathbf{R}$-correspondence. Note that we do not restrict our attention to 1-1 correspondences. The correspondence predicates may have arbitrary arity, and also, because of data duplication, some n-m correspondences may be introduced.

3 The Core Language

In this section, we introduce the core language. This is in the style of rule-based languages for objects, e.g., IQL [7], LDL [8], F-logic [20] and more precisely, of MedMaker [25]. The language we present in this section is tailored to correspondence derivation, and thus in some sense more limited. However, we will consider in a next section a powerful new feature.

We assume the existence of three infinite sorts: a sort **data-var** of data variables, **label-var** of label variables, and **vertex-var** of vertex variables. In the following examples, data and labels variables start with capitals (to distinguish them from names and data values); and vertex variables start with the character & followed by a capital letter.

Rules are built from correspondence literals and tree terms. Correspondence literals have the form $R(x_1, x_2, ..., x_n)$ where R is a relation name and $x_1, x_2, .., x_n$ are data/label/vertex variables/constants. Tree terms are of the form $\&X$ or $\&X\ L\ \{t_1, \ldots, t_n\}$, where $\&X$ is a vertex variable or constant, L is a label/data/vertex variable or constant[2], and $t_1, \ldots, t_n$ is a (possibly empty) list of tree terms. A rule is obtained by distinguishing some correspondence literals and tree terms to be in the body, and some to be in the head. Semantics of rules is given in the sequel.

As an example, consider the following rule that we name r_{so}. Recall that we use an abbreviated syntax so for example $\&X_2\ title\ X_3$ actually stands for $\&X_2\ title\ \{\ \&X3'\ \{\ X_3\ \}\ \}$.

$$
\begin{array}{lll}
 & & \&X_0\ article\ \{\ \&X_1, \\
 & & \qquad \&X_2\ title\ X_3, \\
 & & \qquad \&X_4\ author\ X_5,\ \&X_6\ author\ X_7,\ \&X_8\ author\ X_9, \\
 & & \qquad \&X_{10}\ abstract\ X_{11}, \\
 & & \qquad \&X_{12}\ \} \\
 & is(\&X_0, \&X_{13}) & \&X_{13}\ reference\ \{\ \&X_{14}, \\
 & is(\&X_2, \&X_{15}) & \qquad \&X_{15}\ title\ X_3, \\
r_{so}: & is(\&X_{10}, \&X_{19}) & \qquad authors\{\ \&X_{16}, \&X_{17}, \&X_{18}\ \}\ , \\
 & is(\&X_4, \&X_{16}) \quad \leftarrow & \qquad \&X_{19}\ abstract\ X_{11}\ \} \\
 & is(\&X_6, \&X_{17}) & \&X_{16}\ author\ \{\ fn\ X_{20},\ ln\ X_{21}\ \} \\
 & is(\&X_8, \&X_{18}) & \&X_{17}\ author\ \{\ fn\ X_{22},\ ln\ X_{23}\ \} \\
 & & \&X_{18}\ author\ \{\ fn\ X_{24},\ ln\ X_{25}\ \} \\
 & & \\
 & & concat(X_5, X_{20}, X_{21}) \\
 & & concat(X_7, X_{22}, X_{23}) \\
 & & concat(X_9, X_{24}, X_{25})
\end{array}
$$

[2]It is possible to add a requirement that L is allowed to be a data/vertex variable or constant only in leafs of the tree term. However, since this restriction follows anyway from the notion of *valuation* for tree terms, discussed later on, we chose to ignore this point here.

Note again the distinction between *concat* which is a predicate on data values and can be thought of as given by extension or computed externally, and the derived *is* correspondence predicate. As in the case of data forests, to obtain a shorter description of rules, we used a more compact syntax for tree terms.

More precisely, a *rule* consists of a body and a head. When a rule has only literals in its head, it is said to be a *correspondence rule*. We assume that all variables in the head of a correspondence rule also occur in the body.

We now define the semantics of correspondence rules. Intuitively, the tree terms in the body of rules are matched against portions of the forest having the structure described by the term. So we will be looking for assignments to the variables of a tree term, such that the term, under the assignment, "represents" some vertex in the forest (and additionally that the predicates in the body are satisfied by the assignment). Now, recall that when a leaf in the forest has a vertex identifier as a label, it basically represents a pointer/reference to the vertex having this identifier. For convenience, we may sometimes want to implicitly "traverse" this pointer in the tree term, and view the pointed vertex as a direct child. This motivates the following definition.

Definition 3.1 Let F be a data forest, and let t be a grounded tree term (i.e. a tree term with no variables). We say that t *represents* a vertex v in F if one of the following holds:

1. t has the form $\&x$ or $\&x\ l\ \{t_1, \ldots, t_n\}$ where $\&x = v$, and for the later case $l = L(v)$ and v exactly has n children $v_1, v_2, \ldots, v_n$ that are respectively represented by $t_1, t_2, \ldots, t_n$.

2. v is a single leaf vertex whose label is the identifier of a vertex v' in F, and (1) above holds for t and v'.

For example, the ground tree below represents the vertex &23 in the OODB forest of Figure 2. Note that &231 has been dereferenced (see item (2) in the above definition), and that the label and children of vertex &232 have been omitted (as in the first case of item (1) in the definition).

&23 *authors* { &3 *author* {&31 *fn* {&311 "*S*" {}}, &32 *ln* {&312 "*Abiteboul*" {}}},
&232,
&233 &5 {} }

The semantics of rules is defined as follows.

Definition 3.2 Given an instance (F, I) and some correspondence rule r, a *valuation* ν over (F, I) is a mapping over variables in r such that

1. ν maps data variables to $dom(F)$, label variables to $label(F)$, and vertex variables to $vertex(F)$.

2. For each term H in the body of r, one of the following holds for $\nu(H)$ the ground literal obtained from H by replacing every variable by its interpretation in ν.

   (a) H is a correspondence literal and $\nu(H)$ is true in I; or

   (b) H is a tree term and $\nu(H)$ represents some vertex in F.

We say that a correspondence $C(v_1, \ldots, v_k)$ is *derived from* (F, I) *using* r if $C(v_1, \ldots, v_k) = \nu(H)$ for some literal H in the head of r, and some valuation ν over (F, I).

Let $\mathcal{P}$ be a set of rules. Let $I' = \{H' \mid H' \text{ derived from } (F,I) \text{ using some } r \text{ in } \mathcal{P}\}$. Then, $(F, I \cup I')$ is denoted $T_{\mathcal{P}}(F,I)$. If $\mathcal{P}$ is recursive, we may be able to apply $T_{\mathcal{P}}$ to $T_{\mathcal{P}}(F,I)$ to derive new correspondences. The limit $T^{\omega}_{\mathcal{P}}(F,I) = \bigcup_{i \geq 0} T^{i}_{\mathcal{P}}(F,I)$, when it exists, of the application of $T_{\mathcal{P}}$ is denoted, $\mathcal{P}(F,I)$.

Theorem 3.3 For each (possibly recursive) finite set $\mathcal{P}$ of correspondence-rules and each data forest (F,I), $\mathcal{P}(F,I)$ is well-defined (in particular, the sequence of applications of $T_{\mathcal{P}}$ converges in a finite number of stages). Furthermore, $\mathcal{P}(F,I)$ can be computed in PTIME.

Proof: The number of correspondences that can be derived is polynomial. To prove the theorem we show that (1) a data forest can be represented by a relational instance whose size is linear in the size of the forest, and (2) for each rule r, and every correspondence literal in the head of r, the correspondences derived for the literal in each derivation step can be computed with a first-order formula. The limit $T^{\omega}_{\mathcal{P}}(F,I) = \bigcup_{i \geq 0} T^{i}_{\mathcal{P}}(F,I)$ then corresponds precisely to the inflationary fixpoint computation of the FO formula, hence the the theorem follows immediately from the properties of fixpoint computation for FO formulas, and in particular from the fact that this fixpoint can be computed in PTIME [6].

The relational instance consists of two relations: the 2-ary $Vertex$ relation storing for each vertex its identifier and label, and the 3-ary $Child$ relation storing for each vertex and each of its children the vertex id, the child id, and the index of the child in the ordered list of children.

The first order formulas are constructed as follows. Let r be a rule with terms $H_1, \ldots, H_k$ in the body, and let H be a correspondence literal in the head. For simplicity assume first that H does not contain constants (we will deal with those later). Let $x_1 \ldots x_n$ be the variables in H and let $x_1, \ldots, x_n, x_{n+1}, \ldots, x_m$ be the variables in the body of r. The FO formula has the form:

$$\{x_1, \ldots, x_n \mid \exists x_{n+1}, \ldots, \exists x_m \bigwedge_{i=1 \ldots k} \varphi_{H_i}\},$$

where the φ_{H_i}, $i = 1 \ldots k$, are as defined below. If H_i is a correspondence literal then $\varphi_{H_i} = H_i$. Else, H_i is a tree term and

$$\varphi_{H_i} = \exists v, l \ (\ Vertex(v,l) \wedge \psi_{H_i}(v)\ )\ ,$$

where $\psi_{H_i}(v)$ is a formula that tests if there is an assignment ν for the variables in H_i s.t. $\nu(H_i)$ represents the vertex v, and returns all such possible assignments. The formula basically follows Definition 3.1, and is defined recursively as follows:

$$\begin{aligned}
&\psi_{H_i}(v) = \psi'_{H_i}(v) \ \vee \ (Vertex(v,l) \wedge \exists\ l' \ (\ Vertex(l,l') \wedge \psi'_{H_i}(l)\ )\ ) \\
&\psi'_{\&X\ L\{\&X_1\ L_1\{\ldots\},\ldots,\&X_q\ L_q\{\ldots\}\}}(v) \ = \ Vertex(\&X, L) \wedge v{=}\&X \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ \textstyle\bigwedge_{j=1\ldots q} Child(\&X, \&X_j, j) \wedge \ \psi_{\&X_j\ L_j\{\ldots\}}(\&X_j) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ \neg\ \exists\ v' \ (\ Child(v, v', q+1)\ ) \\
&\psi'_{\&X}(v) = \exists\ l \ (\ Vertex(\&X, l) \wedge v{=}\&X\ )\ .
\end{aligned}$$

The first equation considers the two possible cases, namely (1) v itself is represented by the tree term, or (2) v is a leaf vertex whose label is the identifier of another vertex l represented by the tree term. (Note that we do not have to actually check in the formula that v has no children because, from the definition of a forest, only such vertices can have vertex labels). The second and third equations assert that the vertex represented by H_i has the same identifier as the root of H_i and, if the label and children are explicitly described in H_i, then we further require that the labels

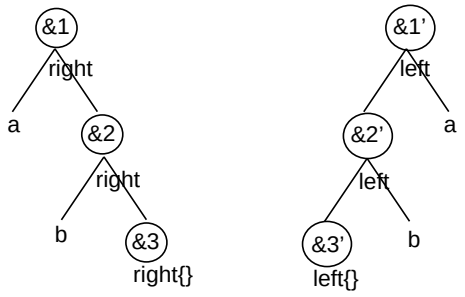

Figure 4: Right- and Left-Deep Trees

are the same and that all the children tree terms respectively represent children of v, and that v has no additional children.

To conclude the proof, consider the case where H contains some constants $c_1, \ldots, c_j$. We can carry the same construction as above and then additionally introduce new variable names $y_1, \ldots, y_j$, and add to the FO formula the requirement $y_1 = c_1 \wedge \ldots \wedge y_j = c_j$. □

The above rule r_{so} is an example of a non-recursive correspondence rule. (We assume that the extension of *concat* is given in I.) To see an example of a recursive rule, we consider the correspondence between "left-deep" and "right-deep" trees. For instance, we would like to derive a correspondence between the right and left deep trees shown in Figure 4. This is achieved using the program $r2l$ which consists of the following rules:

$$r2l: \quad \begin{array}{lcl} R2L(\&U, \&V) & \leftarrow & \&U\ right\{\} \\ & & \&V\ left\{\} \\ \hline & & \&U\ right\{X,\ \&Y\} \\ R2L(\&U, \&V) & \leftarrow & \&V\ left\{\&Z,\ X\} \\ & & R2L(\&Y, \&Z) \end{array}$$

Suppose that we start with $I = \emptyset$, and the forest F shown on Figure 4. Then we derive in order the correspondences $R2L(\&3, \&3')$, $R2L(\&2, \&2')$, $R2L(\&1, \&1')$. The computation is:

$$\begin{array}{lcllcl} T^1_{r2l}(F, \emptyset) & = & F \cup \{\ R2L(\&3, \&3')\ \} & T^2_{r2l}(F, \emptyset) & = & T^1_{r2l}(F, \emptyset) \cup \{\ R2L(\&2, \&2')\ \} \\ T^3_{r2l}(F, \emptyset) & = & T^2_{r2l}(F, \emptyset) \cup \{\ R2L(\&1, \&1')\ \} & r2l(F, \emptyset) & = & T^3_{r2l}(F, \emptyset). \end{array}$$

This kind of deep trees is frequent in data exchange formats and it is important to be able to handle them. However, what we have seen above is not quite powerful enough. It will have to be extended with particular operations on trees and to handle data collections. This is described next.

4 Dealing with Collections

Data collections commonly found in data sources we are interested in are not properly handled by the rules introduced so far. First, observe that, in these rules, tree terms may match vertices with a bounded number of children (the number depends on the term structure). On the other hand, one often encounters data collections which may have an unbounded number of members. Also observe that ordered trees are perfect to represent ordered data collections such as lists or arrays. However, collections such as sets or bags obey certain properties such as insensitivity to order or absence of duplicates. The rules that we developed so far do not take into account such properties. In this section, we address these two issues by extending our framework to incorporate (i) operators on trees and (ii) special collection properties.

4.1 Tree Constructors

We consider two binary operations on trees. The first operation, $cons(T_1, T_2)$, takes two trees T_1, T_2 as input. T_1 is interpreted as an element and T_2 as the collection of the children of its root. The operation adds the element to the collection. The second operator, *merge*, allows to merge two data collections into one. (The *cons* operator can be defined using a merge with a singleton collection.) For example

$$\begin{aligned} mylist\{1,2,3,4\} \equiv \quad & cons(1, mylist\{2,3,4\}) \\ & \equiv merge(mylist\{1\}, mylist\{2,3,4\}) \\ & \equiv merge(mylist\{1,2\}, mylist\{3,4\}). \end{aligned}$$

Note that *mylist* here denotes a vertex label, while the *cons* and *merge* are operators. For brevity, we omitted in the above example the vertex ids. In general, *cons* and *merge* construct a new vertex whose children are determined by the operands of the constructor.

More formally, let T, T', T'' be some trees where the roots of T' and T'' have children $S'_1, ..., S'_n$ and $S''_1, ..., S''_m$ respectively, and the root of T'' has label l. Then

- $\&i\ cons(T, T'')$ is a tree with root id &i, labeled by l, and with children $T, S''_1, ..., S''_n$, in that order, namely $\&i\ cons(T, T'') = \&i\ l\ \{T, S''_1, ..., S''_n\}$.
- $\&i\ merge(T', T'')$ is a tree with root id $\&i$, labeled by l, and with children $S'_1, ..., S'_n, S''_1, ..., S''_m$, in that order, namely $\&i\ merge(T', T'') = \&i\ l\ \{S'_1, ..., S'_n, S''_1, ..., S''_m\}$.

When the id of the constructed vertex is irrelevant, we will use a shorthand notation and omit the $\&i$. The *cons* and *merge* operators provide alternative representations for collections that are essential to describe restructuring. The data trees in the forests we consider are all reduced in the sense that they will not include *cons* or *merge* vertices. But, when using the rules, we are allowed to consider alternative representations of the trees. The root vertices of the trees used as operands for the *cons* and *merge* can be regarded as implicit (the root of the second operand, in the case of *cons*, and the root of both operands, in the case of *merge*). So, for instance the data tree $\&10\ mylist\{\&20, \&30, \&40\}$ can be viewed as $\&10\ cons(\&20, \&v\ mylist\{\ \&30, \&40\})$ where the vertex $\&v$ is implicit and has the structure $mylist\{\&30, \&40\}$. Indeed, we will denote this vertex $\&v$ by $\&mylist(\&10, \&30, \&40)$ to specify that it is a vertex with label *mylist*, that it is a sub-collection of &10, and that it has two children &30, &40. Thus the implicit tree $\&v\ mylist\{\&30, \&40\}$ will be denoted by $\&mylist(\&10, \&30, \&40)\ mylist\ \{\&30, \&40\}$ This motivates the following definition:

Definition 4.1 Given a forest F, a vertex $\&v$ in F with children $\&v_1, ..., \&v_n$ (for $0 \leq n$) and label l, the tree

$$\&l(\&v, \&v_i, \&v_{i+1}, ..., \&v_j)\quad l\quad \{\ \&v_i, \&v_{i+1}..., \&v_j\ \}$$

is called an *implicit tree* of[3] F for each subsequence[4] $\&v_i, \&v_{i+1}..., \&v_j$ of $\&v_1, ..., \&v_n$, and its root, the vertex with id $\&l(\&v, \&v_i, \&v_{i+1}, ..., \&v_j)$ is called an *implicit vertex* of F.

The set of all implicit vertices of F is denoted $impl(F)$. We use $tree(v)$ to denote the implicit tree rooted at an implicit vertex $v \in impl(F)$.

Observe that $vertex(F)$ can be viewed as a subset of $impl(F)$ if $\&v$ is identified to the implicit vertex $\&l(\&v, \&v_1, ..., \&v_n)$ of the definition. Observe also that the cardinality of $impl(F)$ is polynomial in the size of F.

[3] The $\&l(\&v, \&v_i, \&v_{i+1}, ..., \&v_j)$ on the left denotes root of the tree, the l in the middle is the label, and the $\&v_i, \&v_{i+1}..., \&v_j$ between the parentheses on the right are the children of the root.

[4] A *subsequence* $\&v_i, \&v_{i+1}..., \&v_j$ of $\&v_1, ..., \&v_n$ is obtained by removing 0 or more elements from the head and the tail of $\&v_1, ..., \&v_n$.

We can now use *cons* and *merge* in rules. For that, we extend the definition of tree terms and allow them to have the form $\&X\ cons\ (t_1, t_2)$ and $\&X\ merge\ (t_1, t_2)$ where $\&X$ is a vertex variable/constant, and t_1, t_2 are tree terms. As before, to obtain a shorter syntax, we sometimes omit the vertex variables when they are irrelevant for the discussion.

Before defining the semantics for the extended syntax, we give some intuition using an example. The following example uses *cons* to define a correspondence between a list structured as a right-deep tree and a list structured as a tree of depth one. Consider again the right-deep tree of Figure 4:

$$\&1\ right\{a, \&2\ right\{b, \&3\ right\{\}\}\ \}.$$

We would like to obtain a correspondence with the collection:

$$\&1''\ mylist\{a, b\}.$$

Recall that *right* and *mylist* are not keywords but only labels with no particular semantics. On the other hand, *cons* is a keyword with semantics, the cons operation on trees.

The correspondence $TreeList(\&1, \&1'')$ is obtained using the rule:

$$tl: \quad \begin{array}{lcl} TreeList(\&U, \&V) & \leftarrow & \&U\ right\{\} \\ & & \&V\ mylist\{\} \\ \hline TreeList(\&U, \&V) & \leftarrow & \&U\ right\{Z,\ \&X\} \\ & & \&V\ cons(Z,\ \&Y) \\ & & TreeList(\&X, \&Y). \end{array}$$

To define the semantics of such rules, we need to extend the notion of valuation to allow terms containing *cons* and *merge*. The new valuation may now assign implicit vertices to vertex variables. Recall that we were only interested in assignments where the tree terms represented vertices in F. We will now be interested in representing vertices in $impl(F)$, and thus have to extend the notion of *representation* of Definition 3.1 to handle tree terms with *cons* and *merge*. The extended definition is presented below. It basically states the same as before, except that the first part is now extended to deal with the *cons* and *merge* operators.

Definition 4.2 (3.1 revisited) Let F be a data forest, and let t be a grounded tree term. We say that t *implicitly represents* (or *i-represents*, for short) a vertex v in $impl(F)$ if one of the following holds:

1. - t has the form $\&x$ or $\&x\ l\ \{t_1 \ldots, t_n\}$, where $\&x = v$, and for the later case $l = L(v)$ and v has exactly n children $v_1, v_2, \ldots, v_n$ that are respectively represented by $t_1, t_2, \ldots, t_n$. Or,
   - t has the form $\&x\ cons(t_1, t_2)$ (or $\&x\ merge(t_1, t_2)$) for some tree terms t_1, t_2, where $\&x = v$ and there are two vertices $v_1, v_2 \in impl(F)$ s.t. t_1 i-represents v_1, t_2 i-represents v_2, and $tree(v) = \&x\ cons(tree(v_1), tree(v_2))$ (or for *merge*, $tree(v) = \&x\ merge(tree(v_1, tree(v_2)))$).
2. v is a single leaf vertex whose label is the identifier of a vertex v' in $impl(F)$ and 1 above holds for t and v'.

Definition 4.3 Given an instance (F, I) and some correspondence rule r, a *valuation* ν over (F, I) is a mapping over variables in r such that

1. ν maps data variables to $dom(F)$, label variables to $label(F)$, and vertex variables to $impl(F)$.

2. For each term H in the body of r

   (a) H is a correspondence literal and $\nu(H)$ is true in I; or

   (b) H is a tree term and $\nu(H)$ i-represents some vertex in $impl(F)$.

Derivation is defined as before using the refined definition of valuation, and the fixpoint $T^{\omega}_{\mathcal{P}}(F,I)$ is now computed w.r.t such derivations. Observe that $T^{i}_{\mathcal{P}}(F,I)$ may now contain correspondences involving vertices in $impl(F)$ and not only F. Since we are interested only in correspondences between vertices in F, we ultimately ignore all other correspondences. So, $\mathcal{P}(F,I)$ is the restriction of $T^{\omega}_{\mathcal{P}}(F,I)$ to vertices in F. For instance, consider rule tl and

$$F = \{\ \&1\ right\{a, \&2\ right\{b, \&3\ right\{\}\}\}\ \ ,\ \ \&1''\ mylist\{a,b\}\ \}.$$

Then:

$$impl(F) = F \cup \{\&mylist(\&1''), \&mylist(\&1'', a), \&mylist(\&1'', b), \&right(\&1), \&right(\&1, a), etc.\}$$

and

$$\begin{array}{lcl} T^1_{tl}(F,\emptyset) & = & TreeList(\&3, \&mylist(\&1'')) \\ T^2_{tl}(F,\emptyset) & = & T^1_{tl}(F,\emptyset) \cup TreeList(\&2, \&mylist(\&1'', b)) \\ T^{\omega}_{tl}(F,\emptyset) & = & T^2_{tl}(F,\emptyset) \cup TreeList(\&1, \&mylist(\&1'', a, b)) = T^2_{tl}(F,\emptyset) \cup TreeList(\&1, \&1'') \\ tl(F,\emptyset) & = & (F, TreeList(\&1, \&1'')). \end{array}$$

In the sequel, we call the problem of computing $\mathcal{P}(F,I)$, the *matching problem.*

Theorem 4.4 The matching problem is in PTIME even in the presence of *cons* and *merge.*

Proof: The proof is very similar to that of Theorem 3.3, with some small modifications:

1. The relational database is extended to describe also the implicit vertices, and the relationship between them. The relations $Vertex$ and $Child$ now also contain the implicit vertices and information about their labels and children, resp. We also have two additional 3-ary relations $Cons$ and $Merge$ over vertices in $impl(F)$ s.t. $Cons(\&i, \&j, \&k)$ holds iff

   $$tree(\&i) = \&i\ cons(tree(\&j), tree(\&k)).$$

   Similarly $Merge(\&i, \&j, \&k)$ holds iff

   $$tree(\&i) = \&i\ merge(tree(\&j), tree(\&k)).$$

   Note that the relational database thus constructed is larger than the one in the proof of Theorem 3.3, but is still polynomial in the size of the forest F. Also, the number of facts that can be derived is still at most polynomial.

2. The definition of $\psi'_{H_i}(v)$ is refined to handle subterms with *cons* and *merge*:

   $$\begin{array}{lcl} \psi'_{\&X\ cons(\&X_1\ L_1\{...\}, \&X_2\ L_2\{...\})}(v) & = & Cons(\&X, \&X_1, \&X_2)\ \wedge\ v{=}\&X \\ & \wedge & \psi_{\&X_1\ L_1\{...\}}(\&X_1)\ \wedge\ \psi_{\&X_2\ L_2\{...\}}(\&X_2) \\ \\ \psi'_{\&X\ merge(\&X_1\ L_1\{...\}, \&X_2\ L_2\{...\})}(v) & = & Merge(\&X, \&X_1, \&X_2)\ \wedge\ v{=}\&X \\ & \wedge & \psi_{\&X_1\ L_1\{...\}}(\&X_1)\ \wedge\ \psi_{\&X_2\ L_2\{...\}}(\&X_2). \end{array}$$

3. After the fixpoint computation is completed, one needs to remove correspondences that involve implicit vertices. Since the total number of derived facts is at most polynomial in the size of $impl(F)$, this does not affect the polynomial complexity.

□

4.2 Special Properties

Data models of interest include collections with specific properties: e.g., sets that are insensitive to order and do not have duplicates, bags that are insensitive to order. In our context this translates to properties of vertices with particular labels. We consider here two cases, namely insensitivity to order (called bag property), and insensitivity to both order and duplicates (called set property). For instance, we may decide that a particular label, say *mybag* (resp. *myset*) denotes a bag (resp. a set). Then, the system should not distinguish between the representations:

$$cons(a, cons(a, mybag\ \{b\})) \equiv cons(a, cons(b, mybag\ \{a\}))$$
$$cons(a, cons(a, myset\ \{b\})) \equiv cons(a, myset\ \{b\}).$$

Observe that in the above example we used an abbreviated syntax omitting the vertex identifiers, and implicitly assumed that the two a's in the left hand side of the second equation are equivalent. In general, there are many possible notions of tree equivalence that one may use to define what *duplicates*, or *insensitivity to duplicate* mean. For example, one may consider vertex identifiers to be significant, and define two trees to be equivalent iff they are exactly identical. On the other hand, one may chose to ignore the vertex identifiers and require only the general structure of the trees and the labels of vertices in them to be the same. To keep our framework general, we do not restrict the model to use a specific definition of tree equality, but just assume in the following the existence of some equivalence relation $\equiv$ on trees and vertices. Nevertheless, to preserve the *bag* and *set* properties of labels, we require $\equiv$ to have the property that whenever two trees rooted at vertices $\&i, \&j$ are identical up to permutation on the order of children in *bag* and *set* vertices, and elimination of duplicate children (w.r.t $\equiv$) of *set* vertices, then the trees and their roots are respectively equivalent.

So for example, if $\equiv$ ignores vertex identifiers, we have that

$$\&1\ myset\ \{\&11\ a\ \{\}, \&12\ a\ \{\}\} \equiv \&1\ myset\ \{\&11\ a\ \} \equiv \&1\ myset\ \{\&12\ a\ \{\}\}.$$

In the context of set/bag properties, the definition of implicit objects becomes a little bit more intricate: Now we not only need to consider subsequences of vertex children, but also, to capture insensitivity to order, we need to consider permutations on the order of the children.

Definition 4.5 Given a forest F, a vertex $\&v$ in F with children $\&v_1, ..., \&v_n$ (for $0 \leq n$) and label l with set/bag property, for every permutation ρ on $1 \ldots n$, the tree

$$\&l(\&v, \&v_{\rho(i)}, \&v_{\rho(i+1)}, ..., \&v_{\rho(j)})\ \ l\ \ \{\&v_{\rho(i)}, \&v_{\rho(i+1)}, \ldots, \&v_{\rho(j)}\},$$

is an *implicit tree* of F for each subsequence $\&v_{\rho(i)}, \&v_{\rho(i+1)} ..., \&v_{\rho(j)}$ of $\&v_{\rho(1)}, ..., \&v_{\rho(n)}$. Its root, the vertex with id $\&l(\&v, \&v_{\rho(i)}, \&v_{\rho(i+1)}, ..., \&v_{\rho(j)})$ is an *implicit vertex* of F.

The notion of valuation is extended in a straightforward manner to use the above implicit objects and take into consideration tree equivalence due to insensitivity to order and duplicates: First, we extend Definition 4.2, (defining when a tree term t i-represents a vertex $v \in impl(F)$), to apply to vertices v in the extended set of implicit vertices, and use $\equiv$ rather than $=$ to test the equivalence of trees and vertex identifiers. Then, we adjust the definition of valuation in Definition 4.3 to also use the extended set of implicit objects and the refined definition of i-representation.

It is important to observe at this point that the number of implicit objects is now exponential in the size of F. It is still finite since we do not allow an implicit set to have more copies of the same member than the original set.

The next example shows how *cons*, and the set property can be used to define a correspondence between a list and a set containing one copy for each distinct list member. We assume in the example that $\equiv$ is insensitive to vertex identifiers and use an abbreviated syntax omitting vertex identifiers/variables when irrelevant.

$$ls: \quad \begin{array}{lcl} \text{label } myset & : & \text{set} \\ ListSet(\&U, \&V) & \leftarrow & \&U\ mylist\ \{\} \\ & & \&V\ myset\ \{\} \\ \hline & & \&U\ cons(Z,\ \&X) \\ ListSet(\&U, \&V) & \leftarrow & \&V\ cons(Z,\ \&Y) \\ & & ListSet(\&X, \&Y) \end{array}$$

Observe the symmetry of the rules between set and list. The only distinction is in the specification of label *myset*.

Using essentially the same proof as in Theorem 4.4 and a reduction from 3-sat, one can prove:

Theorem 4.6 Ignoring the cost of testing for tree equivalence,

1. in the presence of *cons*, *merge*, and collections that are insensitive to order/duplicates, the matching problem can be solved in EXPTIME in the size of the input data;
2. even only with *cons* and insensitivity to order, the matching problem is NP-hard.

Proof: The EXPTIME complexity bound is obtained using essentially the same proof as in 4.4. The only difference is in the construction of the relational database that is now of size exponential in the size of the forest: The relations $Vertex$ and $Child$ now contain the extended set of implicit vertices. Also, the relations $Cons$ and $Merge$ are now defined w.r.t tree equivalence rather than tree equality: For every $\&i, \&j, \&k$ in the extended set of implicit objects, $Cons(\&i, \&j, \&k)$ holds iff $tree(\&i) \equiv \&i\ cons(tree(\&j), tree(\&k))$, and $Merge(\&i, \&j, \&k)$ holds iff $tree(\&i) \equiv \&i\ merge(tree(\&j), tree(\&k))$. Hence the construction of the relation takes time polynomial in the number of implicit objects and the complexity of testing tree equivalence.

The NP-hardness is proved by reduction from the 3-sat problem, known to be NP-hard. For every 3CNF formula φ we construct an instance (F, I) and a set $\mathcal{P}$ of correspondence rules defining a 1-ary predicate R_φ s.t. $R_\varphi(\&x)$ holds in $\mathcal{P}(F, I)$ for some vertex $\&x$ iff φ is satisfiable. The size, and the construction time, of F, I and $\mathcal{P}$ is polynomial in the size of φ.

Initially, I is empty. For every 3CNF formula φ, the forest F contains four trees. All the labels in the trees have bag property, i.e. denote insensitivity to order.

Let $x_1, \ldots, x_n$ be the free variables in φ. For each variable x_i, we use two new labels t_i and f_i denoting the possible truth and false assignments to the variable. The first tree describes the list of variables and their possible assignments. It is basically a bag of bags, each representing one of the variables x_i and containing the two elements representing its possible assignments, t_i and f_i. In full syntax, the tree has the form :

$$\&1\ mybag\ \{\&11\ mybag\ \{\&111\ t_1\ \{\}, \&112\ f_1\ \{\}\}, \ldots, \&1n\ mybag\ \{\&1n1\ t_n\ \{\}, \&1n2\ f_n\ \{\}\}\},$$

abbreviated as:

$$\&1\ mybag\ \{mybag\ \{t_1, f_1\}, \ldots, mybag\ \{t_n, f_n\}\}.$$

The second tree is basically the "unnesting" of the above tree and describes the bag containing all the variables and their possible assignments. In full syntax, it has the form:

$$\&2\ mybag\ \{\&211\ t_1\ \{\}, \&212\ f_1\ \{\}, \ldots, \&2n1\ t_n\ \{\}, \&2n2\ f_n\ \{\}\},$$

abbreviated as:

$$\&2\ mybag\ \{t_1, f_1, \ldots, t_n, f_n\}.$$

As we shall see below, its main role is to provide the appropriate domain for implicit vertices. The idea is to describe the possible valuations to the variables of φ as sub-bags of the above bag, i.e. as implicit vertices of &2. For similar reasons, we will also need a tree to represent the empty bag:

$$\&3\ mybag\ \{\}.$$

Finally, the last tree describes the formula φ. More precisely, it describes, for each clause in the formula, the assignments for the variables in the clause that can make the clause true. If $\varphi = (v_{i_1^1} \vee v_{i_2^1} \vee v_{i_3^1}) \wedge \ldots \wedge (v_{i_1^k} \vee v_{i_2^k} \vee v_{i_3^k})$, then the tree has the form:

$$\begin{array}{ll} \&4\ mybag\ \{ & \&41\ mybag\ \{\&411\ l_1^1\ \{\}, \&412\ l_2^1\ \{\}, \&413\ l_3^1\ \{\}\}, \\ & \ldots, \\ & \&4k\ mybag\ \{\&4k1\ l_1^k\ \{\}, \&4k2\ l_2^k\ \{\}, \&4k3\ l_3^k\ \{\}\}\}, \end{array}$$

or, in abbreviated syntax

$$\&4\ mybag\ \{mybag\ \{l_1^1, l_2^1, l_3^1\}, \ldots, mybag\ \{l_1^k, l_2^k, l_3^k\}\}.$$

where $l_q^p = t_j$ whenever $v_{i_q^p} = x_j$, and $l_q^p = f_j$ whenever $v_{i_q^p} = \neg x_j$.

We assume below that the tree equivalence relation $\equiv$ ignores vertex identifiers and that two trees are equivalent if they are identical up to renaming of vertex identifiers and reordering of vertex children. Note that, in the general case, such equivalence test entails checking graph isomorphism, and may therefore be expensive. However, in this specific case, since the implicit trees contain no vertex identifiers as leaf values, we are dealing with a restricted case of graph isomorphism over trees (i.e. planar graphs) for which the problem is known to be solvable in PTIME [18].

We are now ready to define the correspondence rules. We first present the rules and then explain what they do. Using the abbreviated syntax, we have:

1. $Partial_val(\&3, \&1) \leftarrow$
2. $Partial_val(\&X, \&Y) \leftarrow Partial_val(\&X', \&Y'),\ \&Y'\ cons(cons(V, \&Z), \&Y),\ \&X\ cons(V, \&X')$
3. $Valuation(\&X) \leftarrow Partial_val(\&X, \&3)$
4. $Ok_so_far(\&V, \&4) \leftarrow Valuation(\&V)$
5. $Ok_so_far(\&V, \&F) \leftarrow Ok_so_far(\&V, \&F'),\ \&F'\ cons(cons(X, \&Z), \&F),\ \&V\ cons(X, \&W)$
6. $R_\varphi(\&V) \leftarrow Ok_so_far(\&V, \&3).$

The first two rules iterate over the bags in &1 and construct partial valuation for the variables. We start from the empty valuation (Rule 1), and then at each iteration pick nondeterministically one new variable, and one of the two possible assignments for the variable, and add it to the partial valuation (Rule 2). When we finish (i.e. we iterated over all the variable and are left with an empty bag), the full valuations are assigned to the predicate $Valuation$ (Rule 3). Now, for each valuation, we iterate over all the clauses in the formula, pick nondeterministically one of the variables in the clause, and test if it is satisfied by the valuation. If it is, then we remove the clause from the formula and continue testing the rest (Rules 4 and 5). Once all the clauses are satisfied (i.e. we iterated over all the clauses and are left with the empty bag), the valuation is declared to be successful and is assigned to R_φ (Rule 6).

$$
\begin{array}{lll}
r_1: & is(\&X_0, \&X_2) & \leftarrow \quad \&X_0\ author\ X_1 \\
 & & \quad\ \&X_2\ author\ \{\ fn\ X_3,\ ln\ X_4\ \} \\
 & & \quad\ concat(X_1, X_3, X_4)
\end{array}
$$

$$
\begin{array}{lll}
r_2: & same_list(\&X_0, \&X_3) & \leftarrow \quad \&X_0\ cons(\&X_1,\ \&X_2) \\
 & & \quad\ \&X_3\ cons(\&X_4,\ \&X_5) \\
 & & \quad\ is(\&X_1, \&X_4) \\
 & & \quad\ same_list(\&X_2, \&X_5)
\end{array}
$$

$$
\begin{array}{lll}
r_3: & same_list(\&X_0, \&X_3) & \leftarrow \quad \&X_0\ \{\} \\
 & & \quad\ \&X_3\ \{\}
\end{array}
$$

$$
\begin{array}{lll}
r_4: & \begin{array}{l} is(\&X_0, \&X_8) \\ is(\&X_2, \&X_{10}) \\ is(\&X_5, \&X_{12}) \end{array} & \leftarrow \quad \begin{array}{l} \&X_0\ merge(\ \{\ \&X_1, \&X_2\ title\ X_3\}, \\ \qquad\qquad merge(\&X_4, \\ \qquad\qquad\qquad article\{\&X_5\ abstract\ X_6, \&X_7\ \}\ )\ ) \\ \&X_8\ reference\ \{\ \&X_9, \\ \qquad\qquad\qquad \&X_{10}\ title\ X_3, \\ \qquad\qquad\qquad \&X_{11}, \\ \qquad\qquad\qquad \&X_{12}\ abstract\ X_6\ \ \} \\ same_list(\&X_4, \&X_{11}) \end{array}
\end{array}
$$

Figure 5: A rule program

It is easy to see that $R_\varphi(\&i)$ holds for a vertex $\&i$ iff the tree rooted at $\&i$ describes a valuation that satisfies φ. More precisely, if $R_\varphi(\&i)$ holds for some implicit object $\&i$, then the tree rooted at $\&i$ has the form $\&i\ mybag\ \{l_1, \ldots, l_n\}$ where l_j is one of t_j, f_j, for all $j = 1 \ldots m$. From the definition of R_φ one case see that the truth assignment ρ defined by

$\rho(x_j) = true$ if $l_j = t_j$

$\rho(x_j) = false$ if $l_j = f_j$

satisfies φ.

On the other hand, it is easy to see that, for every truth assignment ρ that satisfiesφ, R_φ holds for the root of the implicit tree $maybag\ \{l_1, \ldots, l_m\}$, where $l_i = t_i$ whenever $\rho(x_i) = true$, and $l_i = f_i$ if $\rho(x_i) = false$. Or, using full syntax, R_φ holds for the implicit vertex of &2 with id $\&mybag(\&2, \&21i_1, ..., \&2ni_m)$, where $i_j = 1$ whenever $\rho(x_i) = true$, and $i_j = 2$ whenever $\rho(x_i) = false$.

□

Remark 4.7 The complexity here is *data complexity*. This may seem a negative result (that should have been expected because of the matching of commutative collections). But in practice, merging is rarely achieved based on collections. It is most often key-based and, in some rare cases, based on the matching of "small collections", e.g., sets of authors.

To conclude the discussion of correspondence rules, we illustrate the use of *cons* and *merge* with a simple example. Consider the set of rules in Figure 5. The rules specify a correspondence between

articles and object references. Although the correspondence rule r_{so} presented at the beginning of the paper handles articles with exactly three authors, articles/references here deal with arbitrary number of authors. They are required to have the same title and abstract and the same author list (i.e., the authors appear in the same order). The definition uses an auxiliary predicate *same_list*.

The first rule defines correspondence between authors. The second and third rules define an auxiliary correspondence between sequences from both worlds. It is used in Rule 4 that defines correspondence between articles and references. It also defines correspondence between titles and abstracts from both worlds.

5 Data Translation

Correspondence rules are used to derive relationships between vertices. We next consider the problem of translating data. We first state the general translation problem (that is undecidable). We then introduce a decidable subcase that captures the practical applications we are interested in. This is based on *translation rules* obtained by moving tree terms from the body of correspondence rules to the head.

We start with a data forest and a set of correspondence rules. For a particular forest vertex $\&v$ and a correspondence predicate C, we want to know if the forest can be extended in such a way that $\&v$ is in correspondence to some vertex $\&v'$. In some sense, $\&v'$ could be seen as the "translation" of $\&v$. This is what we call the *data translation problem.*

input: an **R**-correspondence (F, I), a set $\mathcal{P}$ of correspondence rules, a vertex $\&v$ of F, and a binary predicate C.

output: an extension F' of F such that $C(\&v, \&v')$ holds in $\mathcal{P}(F', I)$ for some $\&v'$; or *no* if no such extension exists.

For example consider a forest F with the right deep tree $\&1\ \{1, \{2, \{3, \{\}\}\}\}$. Assume we want to translate it into a left deep tree format. Recall that the $r2l$ correspondence rules define correspondences between right deep trees and left deep trees. So, we can give the translation problem the **R**-correspondence (F, I), the root vertex $\&1$, and the correspondence predicate $R2L$. The output will be a forest F' with some vertex $\&v'$ s.t. $R2L(\&1, \&v')$ holds. The tree rooted at $\&v'$ is exactly the left deep tree we are looking for.

Remark 5.1 In the general case: (i) we would like to translate an entire collection of objects; (ii) the correspondence may be a predicate of arbitrary arity. To simplify the presentation, we consider the more restricted problem defined above. The same techniques work for the general case with minor modifications.

It turns out the general problem is undecidable.

Proposition 5.2 The translation problem is undecidable, even in absence of *cons*, *merge*, and labels with set/bag properties.

Proof: The proof works by reduction from the acceptance problem of Turing machines. Let $F = \&1\ l\{\}$ be a forest containing a single vertex labeled by some label l. We show that for every Turing machine TM, one can construct a set of correspondence rules $\mathcal{C}$ defining a binary correspondence predicate C s.t., for every instance $(F', \emptyset)$, $C(\&1, \&i)$ holds in $\mathcal{C}(F', \emptyset)$ for some vertex $\&i$ in F', iff (1) F' contains a vertex with id $\&1$, and (2) the tree rooted at $\&i$ encodes an accepting computation of TM on the empty string.

The above implies that $C(\&1, \&i)$ holds in $\mathcal{C}(F', \emptyset)$ for some vertex $\&i$ and extension F' of F, iff the tree rooted at $\&i$ encodes an accepting computation of TM on the empty string. Thus the data translation has a positive result on input $(F, \emptyset)$, $\mathcal{C}$, $\&1$, C, iff TM accepts the empty string, which is known to be undecidable.

We first explain how a Turing machine computation is encoded by a tree. For brevity we omit vertex identifiers and labels when they are irrelevant. The tree has a structure $\{S_n, \{S_{n-1}, \{\ldots\{S_1\}\}\}\}$, where each S_i is a subtree encoding one step of the computation. S_1 encodes the initial state, S_n encodes the final state, and for every i, S_i and S_{i+1} represent successive steps. The subtrees S_i, encoding the i step of the computation, have the structure $\{Ltape, \{\text{“}q\text{”}, \text{“}a\text{”}\}, Rtape\}$, where q is the current state of the machine, a is the content of the cell where the head of TM stands, and $Ltape$,$Rtape$ are right-deep-trees whose leafs (from top to bottom) represent the contents of the tape to the left and right of the head, resp. We use the symbol “blank” to denote the end of the tape, and use the predicate EQ_{Ltape} (resp. EQ_{Rtape}) to denote trees that represent identical portions of the tape to the left (right) of the head, i.e. $EQ_{Ltape}(\&x, \&y)$ (and resp $EQ_{Rtape}(\&x, \&y)$) holds iff the trees rooted at $\&x, \&y$ are identical up to renaming of vertex identifiers.

The details are omitted. (See Appendix B.) □

Although the problem is undecidable in general, we show next that translation is still possible in many practical cases and can often be performed efficiently. To do this, we impose two restrictions:

1. The first restriction we impose is that we separate data into two categories, input vertices and output vertices[5]. Vertex variables are similarly separated, and the tree terms in rules can either contain only input vertex variables and constants, or only output ones, and are classified as input or output tree terms, respectively. We will assume that the presence of an output vertex depends solely on the presence of some input vertices(s) and possibly some correspondence conditions. This will allow us to focus on essentially one kind of recursion: that is found in the source data structure.

2. The second restriction is more technical and based on a property called *body restriction* that is defined in the sequel. It prevents pathological behavior and mostly prevent correspondences that relate “inside” of tree terms.

We claim that these restrictions typically apply when considering data translation or integration, and in particular we will see that all the examples above have the appropriate properties.

The basic idea is to reuse the correspondence rules and transform them into translation rules by moving output tree terms from the body of rules to their head. For example, consider the $r2l$ correspondence rules. And assume we classify the variables $\&U, \&Y$ as input variables, and $\&U', \&Y'$ as output variables. (In the sequel we use variables with prime to stress the separation between the two worlds.) To translate a right deep tree into a left deep tree, we move the tree terms of the left deep trees (i.e. the output tree terms) to the head of rules, and obtain the following translation rules.

$$r2l': \quad \frac{r: \ \begin{array}{l}\&U'\ left\{\} \\ R2L(\&U, \&U')\end{array} \quad \leftarrow \quad \&U\ right\{\}}{r': \ \begin{array}{l}\&U'\ left\{\&Y',\ X\} \\ R2L(\&U, \&U')\end{array} \quad \leftarrow \quad \begin{array}{l}\&U\ right\{X,\ \&Y\} \\ R2L(\&Y, \&Y').\end{array}}$$

A **translation rule** is a rule where all the tree terms in the body are input terms and all the tree terms in the head are output terms. Of course we need to define the semantics of such rules. We first give some intuition, and then present the formal semantics.

[5]Note that vertices can easily be distinguished using distinct labels.

The output tree terms in the head, and in particular those containing variables that do not appear in the body, are used to create new output trees, or to extend previously constructed ones. (This essentially will do the data translation). We use Skolem functions in the style of [12, 19, 21, 23] to denote new vertex identifiers. Ideally, a vertex id generated for some vertex variable $\&X$ at the head of a rule r could be defined as Skolem function $r_{\&X}$ of the assignments to the variables in the body of the rule. However, this may cause some difficulties: Consider a valuation ν for the body of the second rule above. One may be tempted to use the Skolem term $r'_{\&U'}(\nu(\&U), \nu(X), \nu(\&Y), \nu(\&Y'))$ to denote the new vertex created by the output variable $\&U'$ at the head of the rule. But, since $\&Y'$ is an output variable, $\nu(\&Y')$ may be a (previously) created vertex. A recursive construction of this form may potentially lead to a non-terminating loop of vertex creation. To avoid this type of problems we choose to create objects only as a function of **input** vertices and not of new created output vertices. Thus in the above case the created vertex will be denoted by the Skolem term $r'_{\&U'}(\nu(\&U), \nu(X), \nu(\&Y))$.

The price for this is the following

- We may be excluding some object creation that could be of interest. We are willing to accept this, in particular since we show in the sequel a large class of programs for which this is never the case.

- This may result in inconsistencies (e.g., the same vertex with two distinct values). For example, if the body of the rule r' is satisfied by two valuations ν, ν' differing only on the assignment to $\&Y'$, then $\&U'$ can be declared once as a parent of (exactly) two children $\nu(\&Y')$ and $\nu(X)$ and once as a parent of a different pair $\nu'(\&Y')$ and $\nu'(X)$.

  To solve this we rely on non determinism to choose one value to be assigned to each vertex. Note that data translation inherently needs some form of nondeterminism, for instance when one attempts to construct a list representation from a set.

We will see in the sequel that this leads to an efficient terminating nondeterministic computation. Furthermore, it turns out that for a large class of correspondences, this semantics provides an effective solution to the translation problem.

Before going to the formal definition, there is one more point that needs to be highlighted: As illustrated above, rules may create new vertices and edges, and in particular make existing vertices become children of the new vertices. Note that if a vertex already has another parent, then the result is no longer a tree but a graph. However, observe that this can be easily resolved by using implicit referencing: rather than having multiple edges pointing to a vertex $\&i$, we can introduce for each edge $\&j \rightarrow \&i$ a new leaf vertex $\&i_j$ with label $\&i$, and have $\&j$ point to $\&i_j$ rather than to $\&i$ (i.e. $\&i_j$ serves as an implicit "pointer" to $\imath$). We call this process **graph-to-forest transformation**. We will thus allow the translation rules to create graphs, and will transform them into forests as explained above.

To formally define the semantics of rules we first need to refine the notion of valuation.

Definition 5.3 (*valuation* - revisited) Given an R-correspondence (F, I) where $F = F_{in} \cup F_{out}$ s.t. F_{in} (F_{out}) is an input (output) forest, and some translation rule r, a *valuation* ν over (F, I) is a mapping over the variables in r such that:

1. ν maps data variables to $dom(F)$, label variables to $label(F)$, input vertex variables to $impl(F_{in})$, output vertex variable in the body of r to $impl(F_{out})$, and output vertex variables $\&X$ appearing only in the head of r to the Skolem term $r_{\&X}(\nu(X_1), \ldots, \nu(X_n))$ where the X_i's are the input variables of r.

   We also require that $r_{\&X}(\nu(X_1), \ldots, \nu(X_n))$ does not appear already in F_{out}.

2. For each term H in the body of r

   (a) H is a correspondence literal and $\nu(H)$ is true in I; or

   (b) H is a tree term and $\nu(H)$ i-represents some vertex in $impl(F_{in})$.

3. For each term H in the head of r

   (a) H is a correspondence literal and all the Skolem terms in $\nu(H)$ also appear as vertex id in some tree term $\nu(H')$ in the head; or

   (b) H is a tree term and (1) every Skolem term in $\nu(H)$ has exactly one occurrence as vertex id, and does not appear as vertex id in any of the other tree terms $\nu(H')$ in the head (it is allowed to occur additionally as a label value of leafs), and (2) all the other vertex identifiers (i.e. non Skolem terms) appear only as leafs.

The head of the rule is used to derive correspondences and new output vertices and edges. Condition 3 above guarantees that the correspondences and vertices defined in the head are "legal": Condition 3a requires that correspondences are defined only for "real" vertices; Condition 3b requires that (1) each new vertex is defined exactly once, and (2) that existing vertices can be pointed by new vertices, but their structure is not allowed to be redefined.

Definition 5.4 Given an R-correspondence (F, I) with $F = F_{in} \cup F_{out}$, a translation rule r, and a valuation ν over (F, I), we say that (F', I') is **derived** from (F, I), r, and ν, if

1. $F' = F_{in} \cup F'_{out}$ s.t. F'_{out} was obtained from F_{out} by adding all the vertices and edges of $\nu(H)$ for the tree terms H in the head of r, and then, if needed, transforming the resulting graph into a forest (using the graph-to-forest transformation described above), and

2. I' was obtained from I by adding all the correspondences $\nu(H)$ for the correspondence literals H in the head of r.

Note that, by the definition of valuation, the vertices introduced by distinct tree terms must be disjoint. There are two possible reasons why the graph obtained by adding the vertices and edges in the head tree terms may not be a forest: (i) The output vertex variables that appear in the body of r may have several occurrences as leaves in the head. Hence, the vertices assigned to them may be pointed by several of the newly created vertices, and (ii) terms with *cons* and *merge* create new vertices whose children are, by definition, also children of the operands of the *cons*/*merge*. Since derivations only add new vertices and do not delete existing ones, these children will be pointed by both the operand vertices and by the newly created vertex. Note that, in both cases, the graph we get does not contain cycles, hence the result of a derivation is an acyclic graph.

Case (i) can be easily avoided by strengthening requirement 3b in the definition of valuation and requiring each leaf variable to occur only once. Avoiding (ii) is more difficult since it requires deletion of the merged vertices, and this may cause inconsistency if these vertices are referenced by other vertices (i.e. their vertex id is a label of another vertex). We therefore chose to allow the translation process to create directed graphs, and then transform them into forests using the graph-to-forest transformation, (hence assuring that the resulting forest is consistent).

Definition 5.5 Given a set $\mathcal{P}$ of translation rules, and an instance (F, I), an application of $T_{\mathcal{P}}$ is obtained nondeterministically by choosing a valuation of some rule of $\mathcal{P}$ over (F, I) and computing a derived instance (F', I').

An R-correspondence (F_1, I_1) is in $\mathcal{P}(F, I)$ if it can be generated from (F, I) by applying a sequence of $T_{\mathcal{P}}$ until no application of $T_{\mathcal{P}}$ can extend it further.

Proposition 5.6 For each finite set $\mathcal{P}$ of translation rules and each $\mathbf{R}$-correspondence (F, I), each of the possible sequences of application of $T_{\mathcal{P}}$ converges in a finite number of stages. Furthermore, when no set/bag labels are used each sequence converges in time polynomial in the size of the input, and otherwise in time exponential in the size of the input and the complexity of testing equivalence.

Proof: The number of vertices that can be created (and hence the number of derivations) is bounded by the number of possible Skolem terms and correspondences that can be derived. The number of possible Skolem term is at most is at most $S = R \times V \times |impl(F) \cup dom(F) \cup label(F)|^N$, where R, V, N are the numbers of rules, the number of variables in head of rules, and number of input variables in $\mathcal{P}$, resp. The number of edges being generated in the derivations is bounded by the number of derivations times the number of edges generated by a single rule. The number of such edges depends on the the number of edges described directly in the rule head, plus the number of edges resulting from merging existing vertices (these are edges from a newly constructed vertex to the children of the merged ones). In the worst case the children of the merged vertices include all the vertices in the graph. Hence the number of edges constructed in a single derivation is bounded by the size of the rule's head times S.

The above discussion implies that the size of the output forest is bounded by $G = S + S^2 \times |\mathcal{P}|$

Recall that when no bag/set labels are used, the size of $impl(F)$ is polynomial in the size of F, and is exponential otherwise. Hence both the number of derivations and the size of the output graph are polynomial in the size of F, when no set/bag labels are allowed, and is exponential otherwise.

Now, at each iteration we need to consider the possible valuations for variables in all the rules, and select one (if exists). The set of possible valuations can computed by first computing the possible assignments for the body part of the rule, and then completing the assignment by Skolem functions for the variables in the head, and checking that condition 3 in the definition of valuation holds for the head. This is done as follows: The portion of the valuation for the body of rules can be computed using the FO formula in the proof of Theorem 4.4, for the case where no set/bag labels are allowed, and otherwise using the formula in the proof of Theorem 4.6. The input relations for the formulas are built w.r.t the current $F' = F_{in} \cup F'_{out}$. When no set/bags are allowed, the relations are of size polynomial in F', and so is their construction time. Otherwise the size is exponential, and the construction takes time exponential in the size of F' and the complexity of testing equivalence.

Note that the size of F'_{out} at each stage is no greater than G (G being the bound above for the size of the output forest). Hence, the relations are of size polynomial in F when no set/bag labels are allowed, and exponential otherwise (and similarly for the construction time).

Now, all we need is to complete the valuation using Skolem terms, and check condition 3. This takes, for each valuation, time linear in the size of the head of r and the size of the valuation.

The above implies that the computation converges in time polynomial in the size of the input when no set/bags labels are allowed, and otherwise in time exponential in the size of the input and the complexity of testing equivalence. □

So far, a program computation can be viewed as purely syntactic. We are guaranteed to terminate, but we don't know the semantic properties of the constructed new objects and the derived correspondences.

It turns out that this evaluation of translation rules allows to solve the translation problem for a large class of correspondence rules. (Clearly not all since the problem is unsolvable in general.) In particular it covers all rules we presented in the previous section, and practical cases considered in Section 6.

We next present conditions under which the technique can be used. We are considering correspondences specified with input/output data forests.

Definition 5.7 A correspondence rule r is said to be *body restricted* if in its body (1) all the vertex variables in correspondence literals are leaves of tree terms, and each such variable has at most one occurrence in a correspondence literal, (2) each vertex variable has at most one occurrence in tree terms as a vertex id (it can appear additionally as label of leaf vertices), and (3) each tree term contains only input or only output vertex variables.

Observe that all the correspondence rules used in the examples (rules r_{so}, $r2l$, tl, ls, $r1-r4$) are body restricted. Note that, because we used abbreviated syntax, it may seem in some cases that requirement (2) is violated. For example, in rule r_{so} is seems that the vertex variables $\&X_{16}-\&X_{18}$ appear twice. However, only one of the occurrences (the later one) is as vertex id; the first one denotes a label value! So in this case the abbreviate syntax is misleading; full syntax should be used instead.

An example for rules that are not body restricted are the rules in the proof of the undecidability of the translation problem. The first rule violates restriction (1) since the variable $\&V$ appears in two correspondence literals $Final$ and Run. One may attempt to overcome this by merging the two predicates into one predicate $FinalRun$, and merging the body of rules 2-5 for defining it. However the merged rule will still not be body restricted because the variable $\&V$ will appear in several tree terms, hence (2) will be violated.

The following proposition advocates the use of body restricted rules.

Proposition 5.8 Consider an input/output context. Let $\mathcal{P}$ be a set of body restricted correspondence rules where all the derived correspondence literals relate pairs of input and output vertices. Let (F, I) be an R-correspondence where F is an input data forest, $\&v$ a vertex in F, and C a binary correspondence predicate. Let $\mathcal{P}'$ be the translation rules obtained from $\mathcal{P}$ by moving all output tree terms to the head of rules. Then,

- If the translation problem has a solution on input (F, I) $\mathcal{P}$, $\&v$, C that leaves the input forest unchanged, then each possible computations of $\mathcal{P}'(F, I)$ derives $C(\&v, \&v')$ for some output vertex $\&v'$.
- if some computation of $\mathcal{P}'$ derives $C(\&v, \&v')$ for some vertex $\&v'$, then the forest F' computed by this computation is a correct solution to the translation problem.

The proof of the proposition is long and tedious, and is thus differed to Appendix A.

By Proposition 5.8, to solve the translation problem (with unmodified input) for body restricted rules, we only need to compute nondeterministically one of the possible outputs of $\mathcal{P}$, and test if $C(\&v, \&v')$ holds for some $\&v'$.

6 Conclusion

We presented a specification of the integration of heterogeneous data based on correspondence rules. We showed how a unique specification can serve many purposes (including two-way translation) assuming some reasonable restrictions. We claim that the framework and restrictions are acceptable in practice and validated this by implementing a prototype translation system, called W3TRANS, based on the above ideas. At the heart of the system are the middleware data model to which various data sources are mapped, and the rule language for specifying correspondences and data

translation within the middleware model. To use the system, each data source is expected to provide a mapping to/from the middleware format. The representation of each source inside the middleware is very close to the structure of data in the source, so the implementation of such a mapping is fairly easy. We have experimented with various types of data sources (e.g. HTML, SGML, relational, OODB) and with defining correspondences/translations between them. A detailed description of the system and of the experiments conducted is beyond the scope of this paper. For details see [24].

As assistance to the user, both the source and the target data (in their middleware format) can be displayed on a graphic window in a tree-like representation that is hiding details unnecessary to the restructuring (e.g., tags or parsing information), and starting from that representation, the user can specify correspondences or derive data. We are currently working on further substantiating this by more experimentation.

When applying the work presented here a number of issues arise such as the specification of default values when some information is missing in the translation. A more complex one is the introduction of some simple constraints in the model, e.g., keys.

Another important implementation issue is to choose between keeping one of the representations virtual vs. materializing both. In particular, it is conceivable to apply in this larger setting the optimization techniques developed in a OODB/SGML context for queries [3] and updates [4].

Acknowledgment: We thank Catriel Beeri for his comments on a first draft of the paper. Pini Mogilevsky is thanked for implementing the W3TRANS system.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical report, INRIA and the department of computer science of the Hebrew University of Israel, 1987.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *Int. Journal of Digital Libraries (JODL) 1:1*, pages 54–67, 1997.

[3] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 73–84, 1993.

[4] S. Abiteboul, S. Cluet, and T. Milo. A database interface for files update. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 73–84, San Jose, California, 1995.

[5] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogenous data. In *Proc. of International Conference on Database Theory – ICDT*, pages 351–363, Athens, Greece, 1997.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading-Massachusetts, 1995.

[7] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 159–173, 1989.

[8] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 21–37, 1987.

[9] P. Buneman, S. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 158–169, Zurich, Switzerland, 1995.

[10] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data, May 1996.

[11] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[12] T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, San Jose, California, May 1995.

[13] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. ACM Sigmod, Minneapolis*, 1994.

[14] O. Deux. The story of O_2. *IEEE Trans. on Data and Knowledge Eng.*, 2(1):91–108, March 1990.

[15] J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.

[16] C.F. Goldfarb. *The SGML Handbook*. Calendon Press, Oxford, 1990.

[17] M. Gyssens, J.Paredaens, and D. V. Gucht. A grammar based approach towards unifying hierarchical data models. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1989.

[18] J.E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc 6th ACM Symp. on THeory of Computing*, pages 172–184, 1974.

[19] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object-identifiers. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, Aug 1990.

[20] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects. In *sigmod*, 1989.

[21] M. Kifer, G. Lausen, and Wu James. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, MAY 1995.

[22] G. M. Kuper and M. Y. Vardi. The logical data model. *ACM Trans. on Database Systems*, 18(3):379–413, Sept 1993.

[23] D. Maier. A logic for objects. Technical Report TR CS/E-86-012, Oregon Graduate Center, Nov 1986.

[24] P. Mogilevski. Integration and translation of heterogeneous data. Technical report, M.Sc Thesis, Tel-Aviv University, 1997.

[25] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. Available by anonymous ftp at `db.stanford.edu` as the file `~/pub/papakonstantinou/1995/medmaker.ps`.

[26] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *International Conference on Data Engineering*, 1995.

[27] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Technical report, Stanford University, 1995. Available by anonymous ftp from `db.stanford.edu`.

Appendix A

Proof: (Proposition 5.8) We start from the first claim. To distinguish between input and output vertices/vertex variables, we denote the output ones with dash. Assume that the problem has a positive answer, and let F' be a solution to the translation problem. We show that for every correspondence $R(\&i, \&i')$ derived by $\mathcal{P}(F', I)$ for a vertex $\&i$ in F, it must be the case that every computation sequence of $\mathcal{P}'(F, I)$ also derives some correspondence $R(\&i, \&j')$ for some vertex $\&j'$. This will suffice for proving that all the computations of $\mathcal{P}'(F, I)$ derive $C(\&v, \&v')$ for some output vertex $\&v'$.

The proof works by contradiction. Let n be the first iteration in the computation of $\mathcal{P}(F', I)$ where a correspondence $R(\&i, \&i')$ is derived for a vertex $\&i$ in F, and there is some computation sequence s of $\mathcal{P}'(F, I)$ where no fact of the form $R(\&i, \&j')$ is derived for any vertex $\&j'$. Consider the rule r in $\mathcal{P}$ and the valuation ν used derive $R(\&i, \&i')$. The head of r must contain a correspondence literal $R(\&Z, \&Z')$ s.t. $\nu(\&Z) = \&i$ and $\nu(\&Z') = \&i'$. Let r' be the matching rule in $\mathcal{P}'$ (i.e the rule obtained from r by moving the output tree terms to the head) . We look at the output forest generated at the end of the sequence s, and build an assignment ν' for the variables of r' as follows. ν' is obtained from ν by replacing the assignments to some of the variables in the head of r' in the following way:

1. Each output vertex variables $\&Y'$ that appears also in the body of r' in some correspondence literal $Q(\&X, \&Y')$ is assigned some vertex $\&k'$ s.t. $Q(\nu(\&X), \&k')$ holds. (Note that since r is body restricted there is only one such correspondence literal.) To understand why such an assignment must exist, observe that since ν is a valuation, $Q(\nu(\&X), \nu(\&Y))$ either belongs to I or has been derived in a previous iteration. Since and n is the first iteration in the computation for which the above does not hold, such $\&k'$ must exist.

2. Each output vertex variable $\&Y'$ that appears only in the head of r' is assigned a Skolem function $r'_{\&Y}$ of the assignments to the input variables in the body.

3. Each value/label variable that appears only in the head is assigned some value/label in $dom(F)/lable(F)$, resp. Observe that it must be the case that all the value/label variables appearing in the body are already assigned by ν values/labels in $dom(F)/lable(F)$. This is because ν is assumed to be a legal valuation of r: Clearly ν must assign values in $dom(F)/lable(F)$ to value/label variables appearing in tree terms in the body. As for correspondence literal, since we assumed that the rules derive only correspondences between vertices, for a correspondence involving values/labels to hold it must have held also in the initial I, which is, by definition, restricted to $dom(F) \cup label(F)$.

   Hence, in ν' *all* the value/label variable are mapped to values/labels in $dom(F)/lable(F)$.

First observe that if one of the Skolem terms assigned to the variables already appears in the output forest, then it means that the rule r' was used in the derivation sequence s with some valuation ν'' that agrees with ν w.r.t to the input variables. Hence the correspondence $R(\nu''(\&Z), \nu''(\&Z')) =$

$R(\nu(\&Z), \nu''(\&Z')) = R(\&i, \nu''(\&Z'))$ must have been derived in s, contradicting the assumption that s derives no correspondence of the form $R(\imath, \&j')$.

On the other hand, if non of the Skolem terms already appears in the output forest, then we claim that ν' is a legal valuation, hence can be used to derive more vertices and correspondences, which contradicts the assumption above that the sequence s reached a fixpoint.

To see why ν' is a legal valuation, it suffices test each of the conditions of Definition 5.3: Condition 1 of follows immediately from the way we defined ν'. Condition 2 follows from the fact that (1) r is body restricted, hence each output variable appears in a single correspondence literal, and the way we built ν' assures that each individual predicate is satisfied, and (2) we assumed that F' does not modify the input forest. Hence, for every tree term H, since ν and ν' agree on input variables, $\nu'(H)$ i-represents the the same input vertex that $\nu(H)$ did. Finally, Condition 3 follows from the fact that in body restricted rules (1) all the vertex variables in correspondence literals must be leaves of some tree term, (which guarantees that condition 3a holds), and (2) each vertex variables have at most one occurrence in tree terms as a vertex id, hence condition 3b holds as well.

This concludes the proof of the first claim

To prove the second claim it we show if a computation s of $\mathcal{P}'$ derives a correspondence $R(\&i, \&i')$, then this correspondence also holds in $\mathcal{P}(F', I)$, for $F' = F \cup F_{out}$, where F_{out} is the output forest constructed by s.

This will suffice for proving that whenever s derives $C(\&v, \&v')$ for some vertex $\&v'$, the forest F' is a correct solution to the translation problem.

The proof works again by contradiction. Assume there is some correspondence $R(\&i, \&i')$ derived by the derivation sequence s but is not in $\mathcal{P}(F', I)$, and let n be the first derivation in the sequence driving such a correspondence. Let I' denote the set of correspondences derived so far by the sequence. Since n is the first derivation for which the above happens, we have that $\mathcal{P}(F', I)$ must include all the correspondences in I'. Now, consider the rule r' in $\mathcal{P}'$ and the valuation ν used to derive $R(\&i, \&i')$ in the n derivation step. Let r be the matching rule in $\mathcal{P}$ (i.e the rule from which r' was obtained by moving the output tree terms to the head) . We claim that ν is also a legal valuation for r w.r.t the forest F' and the correspondences derived by $\mathcal{P}(F', I)$, hence an additional correspondence $R(\&i, \&i')$ can be derived, which contradicts the fact that $\mathcal{P}(F', I)$ is by definition a fixpoint of the computation. To prove this we need to show that all the conditions in the definition of legal valuations (Definition 4.3) hold.

1. Clearly, ν maps data variables to $dom(F')$, label variables to $label(F')$, and vertex variables to $impl(F')$ (in fact, all the output vertex variables are mapped to "real" vertices), hence the first condition holds.

2. Since ν is a legal valuation for r', all the correspondence $\nu(H)$ in the body of r' either belong to I or were derived in some previous derivation step, hence belong to I'. Since, as explained above, $I' \subseteq \mathcal{P}(F', I)$, $\nu(H)$ holds in $\mathcal{P}(F', I)$. Thus, first part of the second condition of the definition is satisfied as well.

3. To conclude the proof we need to show that that the second part holds as well, i.e. that for all the tree terms H in the body of r, $\nu(H)$ i-represents some vertex in $impl(F')$.

   Recall that the a derivation step for translation rules consists of two parts. First, for each of the tree term H in the head, all the vertices and edges of $\nu(H)$ are added to the output graph. Then, if needed, the resulting graph is transformed into a forest. Since the derivations never remove vertices, or redefine existing ones, and since the n derivation step added, in the first

step, for each H all the vertices $\nu(H)$, is it clear that if the graph-to-forest transformation would not be applied, $\nu(H)$ would i-represents in the root vertex $\&r$ of $\nu(H)$ (and further more, to show this i-representation we would only need to use item 1 of Definition 4.2).

To see that it also i-represents it after the transformation is applied, observe that the only difference between the graph and its transformed forest is that some of the edges were replaced by implicit pointers, (i.e. rather than having an edge to a given vertex $\&v$, we have an edge to a new leaf vertex with label $\&v$). Note however that the definition of i-representation does not distinguish between actual children of a vertex and implicit ones (i.e. we can use item 2 of Definition 4.2), hence $\nu(H)$ i-represents $\&r$ in the forest as well.

□

Appendix B

Proof: (Proposition 5.2) The correspondence rules are given below. Again, for brevity, we omit in the rules vertex identifiers and labels when they are irrelevant. In fact, the only constant/variable labels that will be used are leaf labels that represent values (in particular, contents of tape cells and states of the Turing Machine) We also omit the brackets {} for vertices with one or zero children. Let q_0 and q_f be the initial and accepting states of the machine, resp.

		%% An accepting computation is a run that %% ends with an accepting state
$C(\&1, \&V)$	$\leftarrow$	$Final(\&V)$ $Run(\&V)$
		%% Final(&V) holds iff the left child of &V %% (i.e. the last step of the computation) %% represents an accepting state
$Final(\&V)$	$\leftarrow$	$\&V\{\{\&X, \{\text{“}q_f\text{”}, Y\}, \&Z\}, \&W\}$
		%% A run is a sequence of steps ending %% with two succesive steps
$Run(\&V)$	$\leftarrow$	$Sequence(\&V)$ $SucStep(\&V)$
		%% each sequence of steps either contains only one %% step (which is the initial state of the %% machine), or is a run followed by %% something (potentially a move)
$Sequence(\&V)$	$\leftarrow$	$\&V\{\{\text{“}blank\text{”}, \{\text{“}q_0\text{”}, \text{“}blank\text{”}\}, \text{“}blank\text{”}\}\}$
$Sequence(\&V)$	$\leftarrow$	$\&V\{\&X, \&Y\}$ $Run(\&Y)$
		%% for each possible move of the TM %% we have two rules. One describes %% the succesive moves when executed %% in the middle of the tape, %% and the second when executed at the %% end of the tape. We give below the rules %% for a right move $\delta(\mathrm{q}, \mathrm{a}) = (\mathrm{p}, \mathrm{b}, \mathrm{R})$.The %% case of left moves is symetric (omitted)
$SucStep(\&V)$	$\leftarrow$	$\&V\{\{\{\text{“}b\text{”}, \&X\}, \{\text{“}p\text{”}, Y\}, \&Z\}, \{\{\&X', \{\text{“}q\text{”}, \text{“}a\text{”}\}, \{Y, \&Z'\}\}, \&W\}\}$ $EQ_{Ltape}(\&X, \&X')$ $EQ_{Rtape}(\&Z, \&Z')$
$SucStep(\&V)$	$\leftarrow$	$\&V\{\{\{\text{“}b\text{”}, \&X\}, \{\text{“}p\text{”}, \text{“}blank\text{”}\}, \&Z\}, \{\{\&X', \{\text{“}q\text{”}, \text{“}a\text{”}\}, \text{“}blank\text{”}\}, \&W\}\}$ $EQ_{Ltape}(\&X, \&X')$
		%% We also have one rule to handle %% the first step of the computation
$SucStep(\&V)$	$\leftarrow$	$\&V\{\{\text{“}blank\text{”}, \{\text{“}q_0\text{”}, \text{“}blank\text{”}\}, \text{“}blank\text{”}\}\}$

%% $\mathrm{EQ_{Ltape}}$(&X, &Y) (resp. $\mathrm{EQ_{Ltape}}$(&X, &Y))
%% holds iff the trees rooted at &X, &Y
%% represent the same left (resp., right) portion of the tape.
%% We give the rules for $\mathrm{EQ_{Ltape}}$. The case
%% of $\mathrm{EQ_{Rtape}}$ is symetric (omitted).

$EQ_{Ltape}(\&X, \&Y) \leftarrow \&X$ "*blank*"
$\&Y$ "*blank*"

$EQ_{Ltape}(\&X, \&Y) \leftarrow \&X\{\&U, X\}$
$\&Y\{\&U', X\}$
$EQ_{Ltape}(\&U, \&U')\}$

□