Mapping Dependence

Ryan Wisnesky

TR-09-09

Computer Science Group
Harvard University
Cambridge, Massachusetts

Mapping Dependence

Ryan Wisnesky

Harvard University
ryan@cs.harvard.edu
September 23, 2009

Abstract. We describe DMSL, a domain specific language for defining schema mappings. Schema mappings are assertions in carefully crafted logics that express constraints between data represented in different formats, including XML and relational schema. DMSL is suitable for representing programs over mappings, which, for instance, occur in dataflow graphs of mappings. DMSL programs of mapping type are statically guaranteed by a qualified type system to denote satisfiable constraints; the principal polymorphic schemas of source and target solution data instances are automatically inferred. DMSL implements a variety of operations over mappings (e.g., composition) by interfacing with IBM's Clio Mapping Engine.

1 Introduction

We describe DMSL, a domain specific language for defining schema mappings. Schema mappings are assertions in carefully crafted logics that express constraints between data represented in different formats, including XML and relational schema. These expressions are often created automatically by a "mapping generator" that uses as input a set of source and target schemas and a set of "correspondences" between source and target schema elements [39,37,5]. Figure 1 shows IBM's Clio mapping tool [27] in action.

In this screenshot, the user has loaded source and target XML schemas and has entered a number of correspondences between atomic-level elements of both schemas, indicated by the blue lines. Clio generates a set of *mapping expressions* from this simple input of schemas and correspondences. The generated mapping expressions capture the meaning of the correspondences as a constraint between source and target data instances conforming to the schema: the generated mapping expressions can be converted into a semantics-preserving query that transforms data from the source schema to the target schema. Queries can be generated in a number of target languages (SQL, XSLT, etc). Figure 2 illustrates Clio's architecture.

Mapping tools are typically used when semantics-preserving data transformation is needed but users cannot or do not want to write queries themselves [39]; for instance, in a business context where non-programmers need to migrate information between departmental databases. Moreover, it can be difficult to manually create semantics preserving queries in the presence of schema integrity

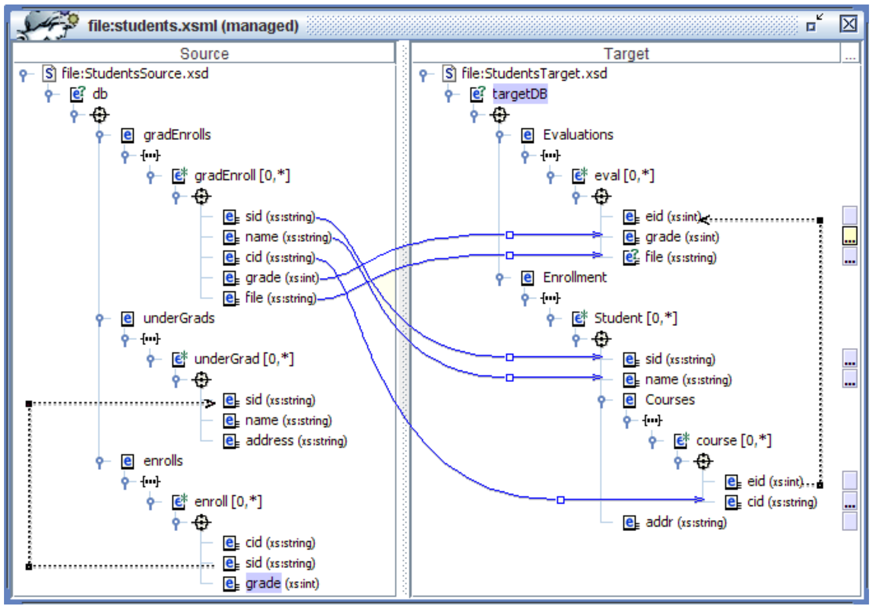

Fig. 1. A schema mapping in Clio

constraints (e.g. foreign keys). In Figure 1, foreign key constraints are indicated by dotted lines.

IBM [27], Microsoft [3], BEA [6], and others are building an ecosystem of tools around schema mappings. Indeed, mappings are used as building blocks for more complex data transformations. Models and semantics of schema mappings for data exchange [15], and operations over mappings [38][16][18] have been extensively studied within the database and information integration community.

1.1 Motivation

Mapping languages in the Clio tradition (e.g., [22]) cannot express mappings that depend on other mappings. That is, they cannot express functions over mappings. Such dependence occurs, for instance, when mappings are used within larger dataflow systems [14], where we may need to express mappings that depend on mappings defined earlier in the flow. Such a scenario is illustrated in Figure 3.

Dependence also occurs in other situations. In semantic adapatation [47], mappings are adapted to changes in schema by composition with a mapping from the old to new schema; an adapted mapping can be represented as a sequence of mapping compositions that depend on the original mapping. When the initial mapping changes, we would like to propagate the changes through the entire sequence to obtain a new, adapted mapping. Thus it can be conve-

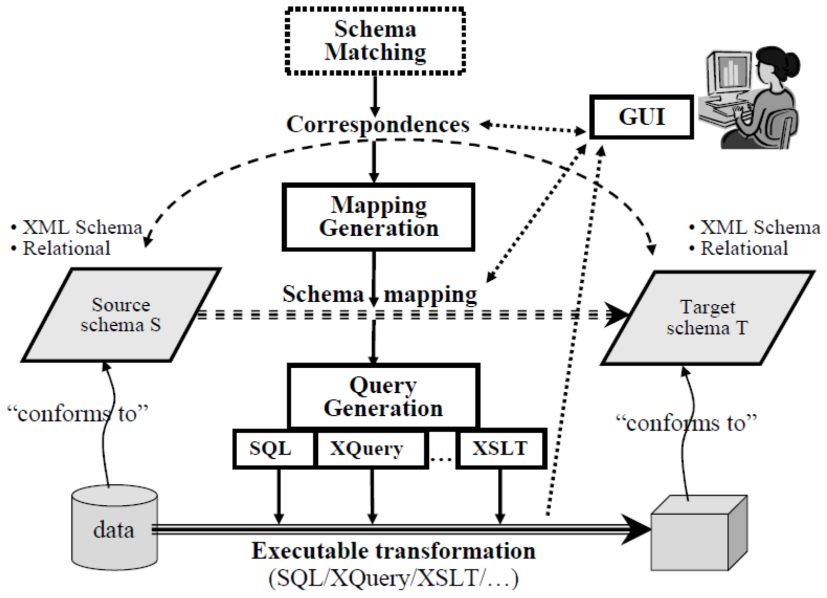

Fig. 2. Clio Architecture

nient to represent the sequence of composed mappings as a function from the initial mapping to the adapted mapping.

Ideally, any mechanism for dependence should rule out invalid dependencies. DMSL uses syntactic substitution to express functions over mappings, and we define a strong static typing discipline to ensure the well-formedness of substitution instances. This discipline ensures that DMSL expressions of mapping type denote satisfiable constraints, and allows DMSL mappings to be polymorphic (usable with many source and target schemas); the language of [22] is monomorphic. Such a discipline also helps alleviate the impedance mismatches [10] involved when integrating mappings with other systems.

Work on typed SQL combinators has helped address similar challenges when exchanging purely relational data using Haskell [35] and C♯ [12]. Our study of a formalism for transforming nested data thus follows the tradition of embedding relational algebra into Haskell [44] and C♯-LINQ [36]. Unlike relational algebra, however, the mapping language contains binding constructs and has a declarative semantics. In addition, users typically interact with mappings graphically as lines/correspondences, as in Figure 1, rather than as text (e.g. SQL); DMSL is thus designed for mapping systems, rather than human programmers. It is an intermediate form usable by mapping engines for representing programs over mappings.

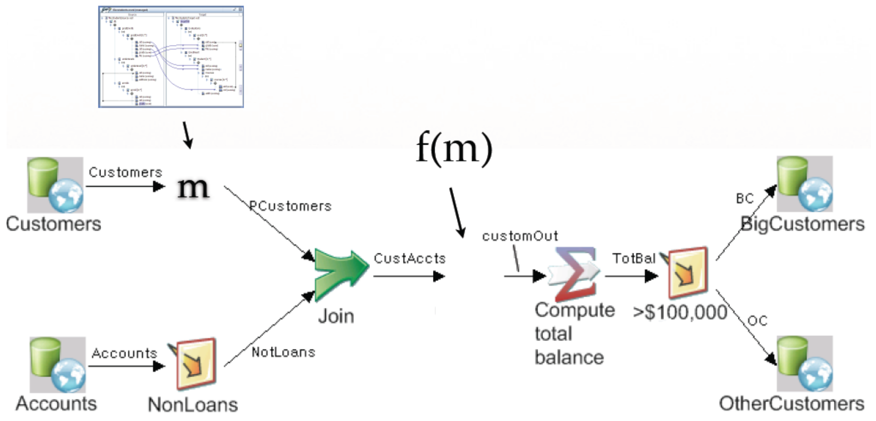

Fig. 3. Mapping Dependence

1.2 Contributions

We present an embedding of the Clio nested mapping language [22] into a system of qualified types [24], exploiting λ as a type-safe mechanism for expressing mapping dependence. Mappings are defined using a basis of primitives, whose semantics are given by mapping transformations implemented in the Clio Mapping Engine. DMSL expressions of mapping type are statically guaranteed to normalize into satisfiable constraints. The principal polymorphic schemas of source and target solution data instances satisfying the constraints are automatically inferred.

2 Overview

2.1 Nested Mappings

Mapping expressions capture the meaning of correspondences/lines between schema as constraints between source and target data instances. We will be using the nested mapping formalism based on [22], which is implemented in Clio. (Other formalisms exist, like [38]). Nested mapping expressions resemble formulae of an enriched set-theory, like

$$\begin{array}{l} \forall student \in src.\mathsf{students},\ \exists employee \in dst.\mathsf{employees} \\ \quad \texttt{s.t.}\ student.\mathsf{fullname} = employee.\mathsf{name} \wedge \\ \qquad \forall s\ \texttt{of}\ \mathsf{teaching}\ \texttt{from}\ student.\mathsf{status}\ \texttt{s.t.}\ s = employee.\mathsf{job} \end{array}$$

but with syntactic restrictions that ensure solution data instances can always be computed. We defer their exact definition to the end of the section.

Mapping expressions are typically given a set theoretic semantics, and in general there may be many or no data instances that satisfy a set of mapping expressions. A precise semantics of canonical solutions is given in [15]; informally, a solution to the above mapping expression is

$$\begin{aligned}(\mathsf{students} : \{(&\mathsf{fullname} : \mathsf{John\ Doe},\\ &\mathsf{status} : \{(\mathsf{teaching} : \mathsf{CS100}),\ (\mathsf{taking} : \mathsf{CS200})\})\})\end{aligned}$$

$$\begin{aligned}(\mathsf{employees} : \{&(\mathsf{name} : \mathsf{John\ Doe},\ \mathsf{job} : \mathsf{CS100},\ \mathsf{id} : 1),\\ &(\mathsf{name} : \mathsf{Jane\ Doe},\ \mathsf{job} : \mathsf{CS101},\ \mathsf{id} : 2)\})\end{aligned}$$

Fields not mentioned in the constraints (like id) may appear in solutions. Not all mapping expressions can be represented as correspondences/lines (e.g. solutions requiring Cartesian product, although all correspondences/lines can be represented as sets of mapping expressions.

2.2 NR Schema

Nested mapping expressions obey an implicit typing discipline, and the shape of canonical data instances can be described by *NR schema*, which consist of records, sets of records, atomic types, and sets of variant (choice) types:

$$\begin{aligned} row &::= (\!|\!) \ \mid\ (\!|\mathcal{L} : schema, row|\!) \\ schema &::= \mathcal{A} \qquad\quad \text{atomic} \\ &\ \mid\ [row] \quad \text{record of} \\ &\ \mid\ \{row\} \quad \text{set of record of} \\ &\ \mid\ \langle row\rangle \quad \text{set of choice of} \end{aligned}$$

Here $\mathcal{A}$ represents atomic base types and $\mathcal{L}$ represents labels. We additionally require that *row*s contain only one instance of any label name, and we equate *rows* that are equivalent up to permutation of record label name and schema pairs. Note that these additional restrictions are not captured in NR schema syntax. NR schema corresponding to the above solution are, for any $a, b \in \mathcal{A}$ and $c \in schema$, using abbreviated notation:

$$[(\!|\mathsf{students} : \{(\!|\mathsf{fullname} : a,\ \mathsf{status} : \langle(\!|\mathsf{teaching} : b,\ \mathsf{taking} : c|\!)\rangle|\!)\}|\!)]$$
$$[(\!|\mathsf{employees} : \{(\!|\mathsf{name} : a,\ \mathsf{job} : b,\ \mathsf{id} : \mathsf{Nat}|\!)\}|\!)]$$

NR schema are expressive enough to capture both relational and XML schemas. Clio, for instance, converts XML schemas into NR schemas internally. In fact, XML schemas, not NR schemas, are displayed in Figure 1.

Foreign key constraints are sometimes considered to be part of a schema definition. For instance, XSD definitions allow key constraints as part of XML schema. We treat such constraints as mappings whose source and target roots coincide, rather than as part of NR schema.

2.3 Introduction to DMSL

In DMSL, the example mapping is written

```
do  src ← root_SRC
    dst ← root_DST
    student ← setGen src.students
    employee ← setGen dst.employees
    eq student.fullname employee.name
    nest (do s ← chcGen_teaching student.status
             eq s employee.job)
```

DMSL supports a monadic style of programming [32], and DMSL mapping expressions often resemble set comprehensions [26]. The exact meaning of this type will be made clear in Section 4, but the inferred type of this DMSL expression is

$$c \setminus \mathsf{teaching}, d \setminus (\mathsf{fullname}, \mathsf{status}), e \setminus \mathsf{students}, f \setminus (\mathsf{name}, \mathsf{job}), g \setminus \mathsf{employees},$$
$$\mathtt{Atomic}(a, b),\ \mathtt{SchemaRow}(c, d, e, f, g) \Rightarrow$$
$$\mathtt{Map}\ (\!|\mathsf{students} : \{(\!|\mathsf{fullname} : a,\ \mathsf{status} : \langle(\!|\mathsf{teaching} : b, c|\!)\rangle, d|\!)\}, e|\!)$$
$$(\!|\mathsf{employees} : \{(\!|\mathsf{name} : a,\ \mathsf{job} : b, f|\!)\}, g|\!)\ ()$$

The three key features of this type are

– a set of *lacks qualifiers*, the infix backslashes, which ensure that labels occur uniquely in records,
– `Atomic` and `SchemaRow` qualifiers, that express constraints on the types of solution data instances, and
– polymorphic row variables, which express that solution instances are free to have extra structure that is not "required" by the constraints

A basic result from our current work on mapping polymorphism is that inhabitants of such types type denote satisfiable constraints (i.e. the mapping has a solution). In this paper we will focus on the design of DMSL itself and only point out connections to mapping semantics.

It is easy to write functions that manipulate mappings. For instance, we can abstract the nesting:

$$\begin{aligned} m\ x = \mathtt{do}\quad & src \leftarrow \mathtt{root}_{\mathsf{SRC}} \\ & dst \leftarrow \mathtt{root}_{\mathsf{DST}} \\ & student \leftarrow \mathtt{setGen}\ src.\mathsf{students} \\ & employee \leftarrow \mathtt{setGen}\ dst.\mathsf{employees} \\ & \mathsf{eq}\ student.\mathsf{fullname}\ employee.\mathsf{name} \\ & \mathsf{nest}\ (x\ student\ employee) \end{aligned}$$

The mapping becomes

$$\begin{aligned} m\ (\lambda student.\ \lambda employee.\ \mathtt{do}\ & s \leftarrow \mathtt{chcGen}_{\mathsf{teaching}}\ student.\mathsf{status} \\ & \mathsf{eq}\ s\ employee.\mathsf{job}) \end{aligned}$$

The explicit passing of mapping expressions is hidden by the monadic style; nevertheless, the type of m expresses that it may only be used to mutate mappings that, for instance, have name labels:

$$\begin{aligned} m :: \ldots \Rightarrow \big(& \mathtt{Var}\ [(\!|\mathsf{fullname} : a, d|\!)] \rightarrow \mathtt{Var}\ [(\!|\mathsf{name} : a, b|\!)] \rightarrow \\ & \mathtt{Map}\ (\!|\mathsf{students} : \{(\!|\mathsf{fullname} : a, c|\!)\}, e|\!)\ (\!|\mathsf{employees} : \{(\!|\mathsf{name} : a, f|\!)\}, g|\!)\big) \ \rightarrow \\ & \mathtt{Map}\ (\!|\mathsf{students} : \{(\!|\mathsf{fullname} : a, c|\!)\}, e|\!)\ (\!|\mathsf{employees} : \{(\!|\mathsf{name} : a, f|\!)\}, g|\!) \end{aligned}$$

The type constructor `Map` is a "mapping building" monad, which we have implemented using the Clio mapping engine. This allows us access to sophisticated operations over mappings, like composition, which we can use alongside DMSL's built in features. For instance, we can define conditional composition:

$$\begin{gathered} f :: \mathtt{SchemaRow}(a, b, c) \Rightarrow \mathsf{Bool} \rightarrow \mathtt{Map}\ a\ b \rightarrow \mathtt{Map}\ b\ c \rightarrow \mathtt{Map}\ b\ c \rightarrow \mathtt{Map}\ a\ c \\ f\ x\ m_1\ m_2\ m_3 = m_1 \circ \mathtt{if}\ x\ \mathtt{then}\ m_2\ \mathtt{else}\ m_3 \end{gathered}$$

2.4 Formal Mapping Expressions

The mapping language we are considering in this paper is based on [22], which is implemented in Clio. Mapping expressions are given by the grammar

$$\begin{aligned} M ::= & \ \forall\ v_1\ B\ P_1, \ldots, v_n\ B\ P_n \\ & \ \exists\ v_{n+1}\ B\ P_{n+1}, \ldots, v_m\ B\ P_m \\ & \ \mathtt{s.t.}\ \ (E_1 \wedge ... \wedge E_j \wedge M_1 \wedge \cdots \wedge M_k) \\ P ::= & \ v \mid \mathsf{Src} \mid \mathsf{Dst} \mid P.\mathcal{L} \\ E ::= & \ P = P \\ B ::= & \ \in \mid\ \mathtt{of}\ \mathcal{L}\ \mathtt{from} \end{aligned}$$

A mapping expression M contains three main components: the $\forall$ clause, the $\exists$ clause, and the `s.t.` clause (sometimes called the `where` clause). The $\forall$ clause contains a list of variable bindings of the form $v_i\ B\ T_i$ called *generators*. Each T_i is a path expression that resolves to a unique set of record or set of choice schema element. B represents binders for set (`in`) and choice (`of`) elimination, where the label for choice indicates which branch is being followed. (Multiple mappings are required to handle all branches of a choice. This is a different mechanism for eliminating choice than is found in most programming languages [41]). Variables are visible to other parts of the mapping expression from the point they are declared. The $\exists$ clause is similar to the $\forall$ clause.

Path expressions P are constructed using variables and the dot operator for projecting into a label. We distinguish schema roots: Src (for the source-side root) and Dst (for the target-side root). We allow generator lists to be empty. The formalism extends easily to handle multiple source and target roots.

The `s.t.` clause is a conjunction of equalities over the variables bound in the $\forall$ and $\exists$ clauses (the $E_1 \wedge \cdots \wedge E_j$ part), plus optional nested mapping expressions (the $M_1 \wedge \cdots \wedge M_k$). The equalities E can be divided into source-side equalities (those that only use variables declared in $\forall$ clauses), target-side equalities (those that only use variables declared in $\exists$ clauses), and source-to-target equalities. Source-side equalities represent join and filtering conditions that must apply to the source data instance. Target-side equalities represent target constraints that hold when the source data is converted into target data; they are usually used to force the correct generation of target surrogate key and foreign key values. Finally, source-to-target equalities are the value correspondences that encode how source atomic values are converted into target values. This mapping language can easily be extended to handle predicates other than equality and to include atomic function constants. These additions are important for closing the mapping expressions under composition; details are discussed in Section 5.3.

A mapping expression M can recursively include zero or more nested mapping expressions $M_1, \ldots, M_k$ in the `s.t.` clause. Variables defined in generators of M can be used in path expressions of any nested mapping. For instance, the `s.t.` clause of the nested mappings have access to all variables declared in M. We do not care about the order of nested mappings or equalities. Finally, we must disallow using a variable bound by a $\forall$ in an $\exists$ generator path, and vice versa, which is not captured by the grammar.

The language of [22] also includes "grouping conditions," which allow the fine tuning of solution data instances. We omit grouping conditions from our mapping language, although it is possible to add them (see Section 5.3).

3 DMSL

DMSL is a modification of the system of qualified types in [24]. We discuss the differences between the two systems in Section 6; in this section, we describe DMSL, an implicitly typed λ calculus:

$$E ::= v \mid c \mid EE \mid \lambda x.E \mid \texttt{let}\ x = E\ \texttt{in}\ E$$

We will define constants c beginning in Section 4. DMSL's type system has kinds

$$
\begin{array}{lll}
\kappa ::= & * & \text{the kind of all types} \\
\mid & \mathit{row} & \text{the kind of all rows} \\
\mid & \kappa \to \kappa & \text{function kinds}
\end{array}
$$

A type expression with *row* kind (a row expression) represents a list of label to type bindings, much like an NR schema *row*. They are "extensible," in the same way, as can be seen from the empty row and row extension type constructors:

$(\!\|\|\!)$	$:: \mathit{row}$	empty row
$(\!\| l : -, - \|\!)$	$:: * \to \mathit{row} \to \mathit{row}$	row extension, for each l
$[]$	$:: \mathit{row} \to *$	record of
$\{\}$	$:: \mathit{row} \to *$	set of record of
$\langle\rangle$	$:: \mathit{row} \to *$	set of choice of
`Map`	$:: \mathit{row} \to \mathit{row} \to *$	mapping
`Var`	$:: * \to *$	mapping variable
$\to$	$:: * \to * \to *$	function space

The row extension type constructor is parameterized by a label; in DMSL, labels are not first class [33]. Recall that we consider row expressions that differ only in the row-extension order of their label and type pairs to be equal. For instance,

$$(\!| l_1 : t_1,\ l_2 : t_2 |\!) = (\!| l_2 : t_2,\ l_1 : t_1 |\!)$$

To support this notion of equality the unification algorithm of the type language must be extended; details are found in [24].

To enforce that rows only contain single occurrences of labels, we will make use of qualified types. Hence DMSL's language of types distinguishes between simple types τ, described above; and qualified types ρ, which make use of predicates π; and type schemes or polymorphic types σ:

$$
\begin{array}{l}
\pi ::= (\sigma :: \mathit{row}) \setminus l \ \mid\ \texttt{Atomic}\ (\sigma :: *)\ \mid \\
\qquad \texttt{SchemaRow}\ (\sigma :: \mathit{row})\ \mid\ \texttt{SchemaType}\ (\sigma :: *) \\
\sigma ::= \rho\ \mid \forall\alpha :: *.\ \sigma\ \mid\ \forall\alpha :: \mathit{row}.\ \sigma \\
\rho ::= \tau\ \mid \pi \Rightarrow \rho
\end{array}
$$

The *lacks* predicate, written $r \setminus l$ for row expression r and label l, indicates a requirement that a row expression not contain a given label. For instance, the type of our path forming primitive (discussed in Section 4.2) is:

$$\cdot_l :: r \setminus l \Rightarrow [(\!| l : a, r |\!)] \to a$$

Here we obey the convention that free type variables are implicitly universally quantified. As with the type constructor $(\!| l : -, - |\!)$, the dot operator $\cdot_l$ is indexed by a label name l because DMSL labels are not first class.

The other predicates, `Atomic`, `SchemaRow`, and `SchemaType`, let us shallowly embed NR schema into types. They capture the more restrictive nature of schema over types; for instance, function types are not allowed in schema. Intuitively, any type in `SchemaType` is expressible as an NR schema, and vice-versa. The formal predicate entailment relation for DMSL is

$$P, \pi \Vdash \pi \qquad P \Vdash (\!|\,|\!) \setminus l \qquad \frac{P \Vdash r \setminus l \qquad l \neq l'}{P \Vdash (\!| l' : \tau, r |\!) \setminus l} \qquad \frac{\tau \in \mathcal{A}}{P \Vdash \texttt{Atomic}\ \tau}$$

$$\frac{P \Vdash \texttt{SchemaRow}\ \tau}{P \Vdash \texttt{SchemaType}\ [\tau]} \qquad \frac{P \Vdash \texttt{SchemaRow}\ \tau}{P \Vdash \texttt{SchemaType}\ \{\tau\}} \qquad \frac{P \Vdash \texttt{SchemaRow}\ \tau}{P \Vdash \texttt{SchemaType}\ \langle\tau\rangle}$$

$$\frac{P \Vdash \texttt{Atomic}\ \tau}{P \Vdash \texttt{SchemaType}\ \tau} \qquad P \Vdash \texttt{SchemaRow}\ (\!|\,|\!)$$

$$\frac{P \Vdash \texttt{SchemaType}\ \tau \qquad P \Vdash \texttt{SchemaRow}\ r}{P \Vdash \texttt{SchemaRow}\ (\!| l : \tau, r |\!)}$$

The typing rules we are using are given in Figure 4.

$$\text{(CONST)}\ \ P \mid A \vdash c : \sigma_c \qquad \text{(VAR)}\ \frac{(x : \sigma) \in A}{P \mid A \vdash x : \sigma} \qquad (\rightarrow E)\ \frac{P \mid A \vdash E : \tau' \rightarrow \tau \qquad P \mid A \vdash F : \tau'}{P \mid A \vdash EF : \tau}$$

$$(\rightarrow I)\ \frac{P \mid A, x : \tau' \vdash E : \tau}{P \mid A \vdash \lambda x.E : \tau' \rightarrow \tau} \qquad (\Rightarrow E)\ \frac{P \mid A \vdash E : \pi \Rightarrow \rho \qquad P \Vdash \pi}{P \mid A \vdash E : \rho} \qquad (\Rightarrow I)\ \frac{P, \pi \mid A \vdash E : \rho}{P \mid A \vdash E : \pi \Rightarrow \rho}$$

$$(\forall E)\ \frac{P \mid A \vdash E : \forall \alpha.\sigma}{P \mid A \vdash E : \sigma(\alpha \mapsto \tau)} \qquad (\forall I)\ \frac{P \mid A \vdash E : \sigma \qquad \alpha \notin fv(A) \cup fv(P)}{P \mid A \vdash E : \forall \alpha.\ \sigma}$$

$$\text{(LET)}\ \frac{P \mid A \vdash E : \sigma \qquad Q \mid A, x : \sigma \vdash F : \tau}{P \cup Q \mid A \vdash \texttt{let}\ x = E\ \texttt{in}\ F : \tau}$$

Fig. 4. The theory of qualified types [24]

A judgment $P \mid A \vdash E : \sigma$ asserts that E has type σ in context A, provided P is derivable. The type inference algorithm described in [24] calculates a principal satisfiable type for an expression under these rules, which we use in our DMSL implementation.

DMSL supports datatype definitions, which we have omitted from this section for simplicity. They can be added in a straightforward way, and we use unit (written ()) in our mapping combinator types. The type constructors in this section (like []) are not regular algebraic datatypes; our implementation treats them specially.

4 Mapping Combinators

In this section we describe our variable and path representation, give types for DMSL's primitive mapping combinators and describe their Clio implementation. More sophisticated operations are discussed in the next section.

4.1 The Mapping Monad

We begin with constants representing schema roots:

$$\begin{aligned}\texttt{root}_{\mathsf{SRC}} &:: \texttt{SchemaRow}\ (s,t) \Rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ [s])\\ \texttt{root}_{\mathsf{DST}} &:: \texttt{SchemaRow}\ (s,t) \Rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ [t])\end{aligned}$$

The type constructor `Map` is parameterized by two row expressions that give the schema of the source and target solution data instances: the solution conforms to $([s],[t])$. The return value of these constants are `Var`s that refer to schema roots. That the same type variable (either s or t) appears in both the return type and in one of the parameterized records is crucial to propagating unification constraints generated by uses of other combinators back into the structure of the entire solution instance.

We are not limited to using a single source and target; rather than having `Map` parameterized by two rows, we can parameterize it by a row of generator root labels and corresponding schema; here,

$$\texttt{Map}'\ (\!|\mathsf{SRC} : s, \mathsf{DST} : t|\!)$$

We will use single source and target for simplicity. The two constants are the base cases of the introduction rules for `Var`.

The mapping monad is implemented as, essentially, a state monad consisting of a native Clio mapping expression and a fresh variable index. The combinators either update the state by rewriting mapping syntax themselves or by invoking the Clio Mapping Engine. DMSL can both normalize expressions of mapping type into conventional Clio mappings, and convert conventional Clio mappings into DMSL.

4.2 Paths

In the nested mapping language, a path is simply a sequence of record projections terminating on a variable. We add two primitives to DMSL:

$$\begin{aligned}\hat{}\ &:: \texttt{Var}\ x \rightarrow x\\ \cdot_l &:: r \setminus l \Rightarrow [(\!|l : a, r|\!)] \rightarrow a\end{aligned}$$

The operational behavior of $._l$ is different in DMSL than in most programming languages: for DMSL, it builds paths; in programming languages, it is used to destruct record values.

DMSL has no introduction rules for $[], \{\}, \langle\rangle$, or any type in the `Atomic` class (that is, no atomic valued constants; this extension is examined in 5.3). In fact, any value belonging to a type in the `SchemaType` class must have the form

$$\hat{v}\ \ldots\ .l_n\ \ldots$$

for some $v :: \texttt{Var}\ x$. This allows us to treat any expression of type t, where t is in `SchemaType`, as a path to a schema element of type t. The implementation builds Clio native path expressions from DMSL terms during normalization using this mechanism.

4.3 Generating Data

We specify the creation of nested sets of data with another primitive:

$$\texttt{setGen} :: \texttt{SchemaRow}(x, s, t) \Rightarrow \{x\} \rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ [x])$$

Canonical solution data instances always nest records inside of sets, so we reflect this in the type of `setGen`.

The implementation of this primitive simply adds a corresponding generator clause to the mapping state: if the `Var` is descended from a source root, we add a $\forall$ clause; otherwise, an $\exists$ clause. However, we must be careful to interpret alternating quantifiers, like

$$\forall x \in p, \exists y \in q, \forall z \in r, \texttt{s.t.}\ \phi(p, q, r)$$

as nesting in order to faithfully embed the mapping language:

$$\forall x \in p, \exists y \in q, \texttt{s.t.}\ \forall z \in r, \phi(p, q, r)$$

In the type of `setGen` it appears that x has no relation to s or t, but they will in fact always contain common subexpressions. This is because `Var`s must descend from uses of `root`, which return `Var`s of type s and t.

NR schema variants/choice types are unlike those in most programming languages. The mapping language's choice eliminator has a single fixed branch, and so our primitive for specifying variant elimination is

$$\texttt{chcGen}_l :: \texttt{SchemaType}\ x, \texttt{SchemaRow}(r, s, t) \Rightarrow \langle (\!| l : x, r |\!) \rangle \rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ x)$$

As with set elimination, this primitive adds a generator clause to the mapping.

4.4 Filtering Data

Mapping expressions allow source and target side filtering, along with source to target equality constraints. It is easy to tell which schema root each `Var` is descended from, so we just have a single primitive which adds atomic equality constraints to the mapping:

$$\texttt{eq} :: \texttt{Atomic}(x, y), \texttt{SchemaRow}(s, t) \Rightarrow x \rightarrow y \rightarrow \texttt{Map}\ s\ t\ ()$$

4.5 Nesting constraints

Nesting is done with a single primitive

$$\texttt{nest} :: \texttt{SchemaRow}(s,t) \Rightarrow \texttt{Map}\ s\ t\ () \rightarrow \texttt{Map}\ s\ t\ ()$$

which updates the mapping state such that the input mapping becomes a nested constraint. The monadic plumbing ensures correct scoping and α-conversion of Vars.

4.6 Summary

The types of the typesafe DMSL mapping primitives are shown in Figure 4.6. Any expression of mapping type built using these primitives is guaranteed to normalize into a satisfiable mapping.

$$\begin{aligned}
\hat{} &:: \texttt{Var}\ x \rightarrow x\\
._l &:: r \setminus l \Rightarrow [(\!|l : a, r|\!)] \rightarrow a\\
\texttt{root}_{\mathsf{SRC}} &:: \texttt{SchemaRow}\ (s,t) \Rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ [s])\\
\texttt{root}_{\mathsf{DST}} &:: \texttt{SchemaRow}\ (s,t) \Rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ [t])\\
\texttt{setGen} &:: \texttt{SchemaRow}(x,s,t) \Rightarrow \{x\} \rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ [x])\\
\texttt{chcGen}_l &:: \texttt{SchemaType}\ x, \texttt{SchemaRow}(r,s,t) \Rightarrow \langle(\!|l : x, r|\!)\rangle \rightarrow \texttt{Map}\ s\ t\ (\texttt{Var}\ x)\\
\texttt{eq} &:: \texttt{Atomic}(x,y), \texttt{SchemaRow}(s,t) \Rightarrow x \rightarrow y \rightarrow \texttt{Map}\ s\ t\ ()\\
\texttt{nest} &:: \texttt{SchemaRow}(s,t) \Rightarrow \texttt{Map}\ s\ t\ () \rightarrow \texttt{Map}\ s\ t\ ()
\end{aligned}$$

Fig. 5. Statically typesafe DMSL primitives

5 Operations on Mappings

In addition to the basic primitives shown above, we have implemented operators for casting/lifting a mapping, and performing other operations on mappings. These operations may not be statically typesafe, but satisfiability is checked during mapping normalization. In other words, with these extended operations, we can only guarantee that DMSL expressions of mapping type that pass the runtime checks are guaranteed to denote satisfiable constraints.

5.1 Lifting

We have implemented an additional set of combinators for "lifting" a mapping expression from one schema into another, when a mapping's schema occurs inside of a larger schema: in Figure 6, the lower (XML) schema is contained inside the upper schema. We use these combinators to evolve DMSL programs though schema changes.

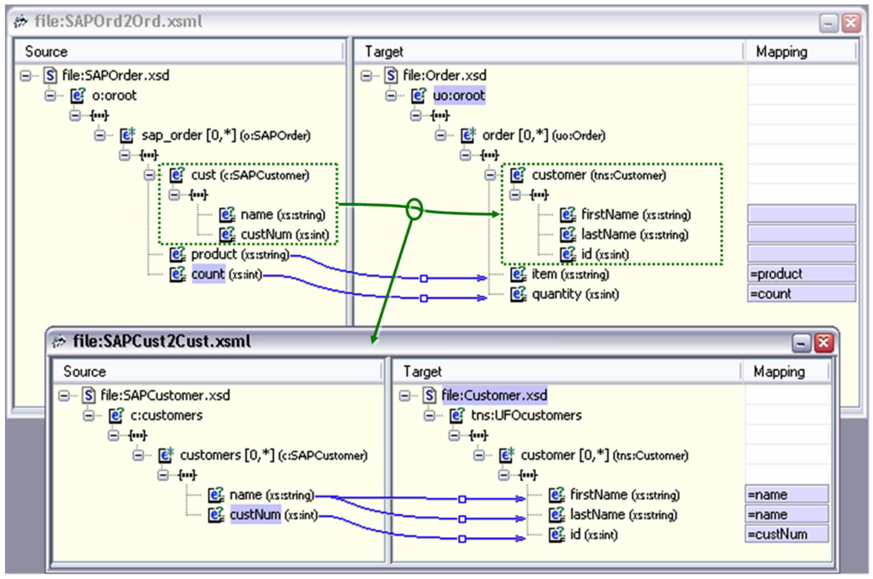

Fig. 6. Schema containment

Lifting Primitives Suppose we wish to reuse a mapping m from some schema s to a particular schema Book:

$$m :: \ldots \Rightarrow \texttt{Map}\ s\ \mathsf{Book}$$
$$m = \forall \ldots \exists v \in dst.\mathsf{author}\ \texttt{s.t.}\ \phi(v)$$

by applying it to a new target schema

$$[(\!|\mathsf{book} : \mathsf{Book}, \mathsf{loanedTo} : \mathsf{String}|\!)]$$

using the straightforward rewrite

$$\forall \ldots \exists v \in dst.\mathsf{book}.\mathsf{author}\ \texttt{s.t.}\ \phi(v)$$

First, we can encode the containment path as

$$\lambda x.\ x.\mathsf{book}$$

Then, we use one of the record lifting combinators

$$\texttt{liftRec}_{\mathsf{DST}} :: \texttt{SchemaRow}(s,t,x) \Rightarrow \texttt{Map}\ s\ t\ () \rightarrow ([x] \rightarrow [t]) \rightarrow \texttt{Map}\ s\ x\ ()$$
$$\texttt{liftRec}_{\mathsf{SRC}} :: \texttt{SchemaRow}(s,t,x) \Rightarrow \texttt{Map}\ s\ t\ () \rightarrow ([x] \rightarrow [s]) \rightarrow \texttt{Map}\ x\ t\ ()$$

The lifting combinators change schema root types without changing the types of other `Var`s, so that the mapping simply becomes

$$\texttt{liftRec}_{\mathsf{DST}}\ m\ (\lambda x.\ x.\mathsf{book})$$

For this path encoding to be correct each path must terminate on the λ-bound argument. The DMSL implementation checks this at runtime.

Likewise, DMSL has lifting primitives for sets of records and variants:

$$\texttt{liftSet}_{\mathsf{DST}} :: \texttt{SchemaRow}(s,t,x) \Rightarrow \texttt{Map}\ s\ t\ () \to ([x] \to \{t\}) \to \texttt{Map}\ s\ x\ (\texttt{Var}\ [t])$$
$$\texttt{liftVar}_{\mathsf{DST}} :: \texttt{SchemaRow}(s,t,x) \Rightarrow \texttt{Map}\ s\ t\ () \to ([x] \to \langle t\rangle) \to \texttt{Map}\ s\ x\ (\texttt{Var}\ [t])$$
$$\texttt{liftSet}_{\mathsf{SRC}} :: \texttt{SchemaRow}(s,t,x) \Rightarrow \texttt{Map}\ s\ t\ () \to ([x] \to \{s\}) \to \texttt{Map}\ x\ t\ (\texttt{Var}\ [s])$$
$$\texttt{liftVar}_{\mathsf{SRC}} :: \texttt{SchemaRow}(s,t,x) \Rightarrow \texttt{Map}\ s\ t\ () \to ([x] \to \langle s\rangle) \to \texttt{Map}\ x\ t\ (\texttt{Var}\ [s])$$

These primitives prefix the mapping state with a new binder. For instance, and roughly speaking, given mapping m, applying $\texttt{liftSet}_{\mathsf{SRC}}$ with path p yields

$$\forall v \in src.p, m(src \mapsto v)$$

These four lifting primitives add a bound variable, in this case v, which they return. Intuitively, this returned variable is the old schema root.

The lifting primitives are complete: if a schema is contained inside of another, then there is some sequence of record projections, set eliminations, and choice eliminations that specifies the containment.

Applications of lifting One reason we are interested in the lifting primitives is because they allow us to evolve DMSL programs though schema changes. In many situations, lifting rewrites are semantics preserving, and so we can "relax" the DMSL typing rules by "type-checking modulo lifting."

Our DMSL implementation provides just such a mode of operation: if an ill-typed DMSL program can be made well-typed by inserting lifting primitives, then such insertions are performed automatically. One particular scenario where such functionality is useful is when part of a DMSL program changes independently from another; for instance, the schema of one departmental database changes and existing mappings need to be updated. This gives DMSL programs a measure of robustness against definition changes, because mappings can be automatically updated.

It is possible to define an analogous notion of "downcasting", or "de-lifting", but we leave this for future work. Likewise, there are other rewrites besides lifting that are "usually semantics preserving." For instance, polymorphism itself is a kind of semantics-preserving re-use where the subtyping relation is used instead of the containment relation. We are currently exploring these ideas.

DMSL also uses lifting to expand the applicability of pre-existing mappings. In this scenario, we would like to map from S to T but cannot create a new mapping from scratch. Such scenarios often occur when we are automating mapping construction. For instance, we may be trying to complete a dataflow graph of mappings based on a user provided skeleton. If a mapping m from S' to T, where S' is contained in S, is registered with the DMSL implementation, then Clio will suggest a lifted version of m as a potential mapping.

5.2 Composition

DMSL includes a composition primitive

$$\circ :: \texttt{SchemaRow}(a, b, c) \Rightarrow \texttt{Map}\ a\ b\ () \rightarrow \texttt{Map}\ b\ c\ () \rightarrow \texttt{Map}\ a\ c\ ()$$

Which is implemented by interfacing with Clio. The composition algorithm itself [16] performs sophisticated rewriting operations which may remove generators. Hence we cannot statically guarantee the well-formedness of mappings defined using composition; instead we must include runtime checks during mapping normalization.

We can add further operations, like inversion [17], in a similar manner.

5.3 Expressive Power of DMSL

DMSL can be extended to permit more expressive mappings at the cost of weakening the semantic guarantees provided by the type system. For instance, naively adding atomic valued constants to DMSL results in typeable mappings that contain unsatisfiable constraints, like $1 = 2$. Enriching the language of types or the sophistication of the type system can alleviate this to some extent, and we are studying more advanced type systems as part of our current work on mapping polymorphism.

We can also add atomic valued function symbols to DMSL. When this is done, DMSL mapping expressions are equivalent to second order tuple generating dependencies, called so-tgds [16]. One simple way to do this is to simply interpret atomic functions as the skolemizations of functions existentially bound at the outermost level of a so-tgd. (Mapping expressions use second-order quantification to capture lookup tables and unique keys; as such the second-orderness of the expression is required in many situations arising from, e.g., target side foreign key constraints.) For purely relational data, so-tgds are closed under composition.

For closure under composition with nested data (which is distinct from the `nest`ing occurring in a mapping expressions), we must add set-valued function symbols to DMSL. As a consequence, we require a check outside the type system to ensure well-formedness of the mapping expressions: we must prevent set-valued functions from being used as inputs to generators (e.g. `setGen` $(f\ x)$ must be disallowed). Having set-valued function symbols allows DMSL to express grouping conditions [22].

6 Implementation in Trex

DMSL is a modification of the system of qualified types in [24], which is implemented as the Trex extension of the Hugs Haskell implementation [1]. Essentially, Trex adds extensible records to Haskell 98. The key differences of DMSL from [24] are

- DMSL has different built-in primitives
- DMSL has additional type constructors
- DMSL has an extended predicate entailment relation
- DMSL has a semantics tailored to mappings and is implemented by integrating with the Clio mapping engine

It is not, to the best of our knowledge, possible to embed DMSL into Trex because of two difficulties:

- The entailment relation that defines schema cannot be expressed without classes over row expressions
- Some DMSL primitives must be parameterized by labels

Trex itself is a compiler extension, and not a pure Haskell 98 library, for similar reasons. Nevertheless, we can get relatively far in a Trex DMSL implementation by just making use of the additional expressiveness Trex provides. Trex writes `Rec` for $[]$, and does not have the type constructors $\{\}$ and $\langle\rangle$, but we can easily add then:

```
data Set a
data Variant a
```

The right hand sides can be empty, because we use them only at the type level for schema-related book-keeping.

We cannot parameterize Trex definitions by labels or use "row classes," but we can write `Schema` (`Rec` a) instead of `SchemaRow` a. Our entailment relation is necessarily incomplete:

```
class Schema a where
class Atomic a where
instance Schema (Rec a) => Schema (Set (Rec a)) where
instance Schema (Rec a) => Schema (Variant (Rec a)) where
```

Still, we can try to implement the mapping monad, say by printing out mapping expression syntax as we normalize expressions of mapping type:

```
type Map a b r = StateT (a, b, Int) IO r
```

Here we including `a` and `b` in the rhs, even though they are not associated with any values (they are "phantom types"), because doing so aids type inference [11] [43]. Variables can be implemented as

```
data Var r = Var Int
```

where the `Int` is a unique variable index, which can plausibly be kept fresh (and in scope/"non-exotic") using monadic plumbing. Thus we add

```
root_SRC :: (Schema (Rec a), Schema b) => Map (Rec a) b (Var (Rec a))
root_SRC = return (Var 0)

root_DST :: (Schema (Rec b), Schema a) => Map a (Rec b) (Var (Rec b))
root_DST = return (Var 1)
```

Without label parameterized definitions, we can try to use Trex's record projection (written $\#l$) as DMSL's dot operator ($.l$). Because $.l$ builds paths and # projects from actual record values, we can't really use #. Hence, we won't really be able to print out the mapping. However, the type of $.l$ and # coincide, so we can still get mappings to typecheck, although we can't actually create them. For instance, this typechecks in Trex:

```
eq :: (Schema (Rec r1, Rec r2, x, y), Atomic a) =>
 (Var x, x -> a) -> (Var y, y -> a) -> Map (Rec r1) (Rec r2) ()
eq = error "unimplementable"

test :: (SchemaType (Rec (nickName :: b | e)),
         SchemaType (Rec (name :: Rec (first :: b | c) | d)),
         c\first, d\name, e\nickName, Atomic b,
         SchemaType (Rec (name :: Rec (first :: b | c) | d),
                     Rec (nickName :: b | e),
                     Rec (name :: Rec (first :: b | c) | d),
                     Rec (nickName :: b | e))) =>
Map (Rec (name :: Rec (first :: b | c) | d))
    (Rec (nickName :: b | e)) ()
test = do s <- root_SRC
          t <- root_DST
          eq (s, \s -> #first (#name s)) (t, \t -> #nickName t)
```

Here we are using a simple λ-encoding of paths, representing a path as a pair of a `Var` y and a function $y \to a$. This encoding is not adequate, but we cannot finish the implementation anyway.

Trex will not actually infer the qualifiers; instead, it issues an error message saying exactly what they are supposed to be. Because our typeclass encoding of the predicate entailment relation is incomplete, Trex will not detect inconsistent qualifiers, and so the mapping itself may not be satisfiable or even well-formed. In addition, types have "redundant" qualifiers.

It may be possible to use other Haskell 98 extensions, like [31], to obtain a complete implementation.

7 Related Work

7.1 Languages for Tree Transformation

The programming languages community has a wealth of knowledge about transformations on tree-like data [21], including bi-directional tree transformations [25], and languages for XML processing [30].

The Harmony data synchronization project [19] and associated lens system [25] tackle the perpetually difficult "view-update problem" [42] for tree data. A system with view-update allows users to create particular views (slices, aggregations, etc) of data and allow updates to the view to propagate into the original data. Lenses shares our approach of creating a DSL containing functional programming primitives, although its expressions are given an operational semantics. The language of [20] has only an informal type system.

The XML processing languages and systems XDuce [29] and Xtatic [23] aim to create general XML processing languages where XML values are first-class. The languages are functional in nature and are have an intuitive semantics for XML processing. They introduce a rich language of types to describe XML values (including regular expressions). The specificity of types for XML, however, leads to restrictions on polymorphism, function types, and type inference that have only recently been addressed [46]. These systems are, in a certain sense, the XML counterparts of LINQ (see below).

7.2 Purely Relational languages

There is a fair amount of work on schema inference for SQL and relational algebra expressions [40][7][45][13]. A common approach is to add relational operations as primitives to a type theory to obtain polymorphism and higher order functions in the same way as DMSL. When this is done with mainstream functional languages the results are highly-reusable languages that suffer very little from the harmful "impedance mismatch" [10] between the programming language and relational data. The details of how polymorphism is achieved in these systems varies widely, although the use of extensible records is a common theme. These ideas have found their way into Haskell [35][44], and into C♯ as Microsoft's LINQ project [36].

7.3 Related type theory

Early work in extensible records uses bounded subtyping [8], but extensible records can also be implemented using qualified types [24]. Labels themselves can be first class [33] or even scoped [34]. Use of qualified types as a type-level deductive system is also well-studied [28]. Qualified types are a good fit for our mapping language, but more expressive type theories can potentially type more sophisticated operations, like pivot, which turns data into schema and so has a dependent type [4]. Likewise, NR schema lack a recursive binding construct but one can be added in a way similar to datalog [9]. The general approach of using functional languages to host domain specific languages is studied in [43] and for databases in particular in [35]; issues with ghost variables (datatypes with type variables on the LHS of a datatype definition that are not used in the RHS) are covered in [11]. Our use of monadic/stateful syntactic plumbing stems from [32].

7.4 Related Mapping Languages

If we have understood mapping in a ground up way, then model management understands mapping systems from the top down. It studies “schema and database transformation capabilities that are independent of a particular data model” [2]. In particular, model management tries to understand how data integrity constraints can be preserved across particular commonly occuring “semantically-meaningful” data transformations, like Match, Compose, Inverse, and Merge. Model structure (the model-management equivalent of an NR schema) is typically given by directed, labelled graphs reminiscent of entity-relationship diagrams. [2] is a categorical characterization of these operators. The model management system Rondo [38] lets users write complex model transformations as simple programs over a basis of primitive operations. The language of Rondo itself, however, is not directly suitable for type-based reuse or schema inference because the type system of its programs is weak.

Work on mappings in the Clio tradition is extensive; see the introduction for references.

8 Conclusion

We have described DMSL, a system of qualified types for expressing programs over mappings. Mappings are a particular kind of data transformation and DMSL programs of mapping type are statically guaranteed by a qualified type system to denote such transformations. We infer the schemas of canonical source and target solution data instances and implement DMSL by interfacing with IBM’s Clio mapping engine. DMSL brings together powerful operations over mappings (like composition) from the mapping engine and combines them with programming constructs for constructing programs over mappings, which occur in schema evolution and in dataflow programming with mappings. DMSL programs can automatically adapt to minor changes in mapping definitions through automatic insertion of type coercions. We believe hybrid systems like DMSL which are part mapping language and part system language will be invaluable as mapping systems begin to scale and integrate with other systems.

References

1. Hugs haskell, http://www.haskell.org/hugs.
2. Suad Alagic and Philip A. Bernstein. A model theory for generic schema management. In *DBPL ’01: Revised Papers from the 8th International Workshop on Database Programming Languages*, pages 228–246, London, UK, 2002. Springer-Verlag.

3. Philip A. Bernstein, Sergey Melnik, and John E. Churchill. Incremental Schema Matching. In *VLDB (demo)*, pages 1167–1170, 2006.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.
5. Angela Bonifati, Elaine Qing Chang, Terence Ho, Laks V. S. Lakshmanan, and Rachel Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In *VLDB(demo)*, pages 1267–1270, 2005.
6. Vinayak Borkar, Michael Carey, Daniel Engovatov, Dmitry Lychagin, Till Westmann, and Warren Wong. XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform. In *ICDE*, pages 1307–1316, 2008.
7. Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Trans. Database Syst.*, 21(1):30–76, 1996.
8. Luca Cardelli. Extensible records in a pure calculus of subtyping. In *In Theoretical Aspects of Object-Oriented Programming*, pages 373–425. MIT Press, 1994.
9. Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *PODS '92: Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 55–66, New York, NY, USA, 1992. ACM.
10. J. Chen and Q. Huang. Eliminating the impedance mismatch between relational systems and object-oriented programming languages, 1995.
11. James Cheney and Ralf Hinze. First-class phantom types. Technical report, 2003.
12. Microsoft Corp. Micorsoft Corp. The LINQ Project, 2005–2006. http://msdn.microsoft.com/netframework/future/linq/.
13. Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. A crash course on database queries. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 143–154, New York, NY, USA, 2007. ACM.
14. S. Dessloch, M. A Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.
15. R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. In *PODS*, pages 90–101, 2003.
16. R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *TODS*, 30(4):994–1055, 2005.
17. Ronald Fagin. Inverting schema mappings. *ACM Trans. Database Syst.*, 32(4):25, 2007.
18. Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Quasi-inverses of schema mappings. In *PODS*, pages 123–132, 2007.
19. J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 2007.
20. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007.
21. J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *Workshop on Programming Language Technologies for XML (PLAN-X), informal proceedings*, January 2007.
22. A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.

23. Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, January 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.
24. Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report Technical report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.
25. Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. A language for bi-directional tree transformations. Technical report, In Workshop on Programming Language Technologies for XML (PLAN-X), 2003.
26. Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.
27. L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *SIGMOD*, pages 805–810, 2005.
28. Thomas Hallgren. Fun with functional dependencies or (draft) types as values in static computations in haskell. In *Proc. of the Joint CS/CE Winter Meeting*, 2001.
29. Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.
30. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
31. Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.
32. John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *In Programming Languages Design and Implementation*, pages 24–35. ACM Press, 1994.
33. Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, December 2004.
34. Daan Leijen. Extensible records with scoped labels. In *Trends in Functional Programming '05*, pages 297–312, 2005.
35. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, volume 35.1, pages 109–122, New York, NY, USA, January 1999. ACM Press.
36. Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, page 706, 2006.
37. S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting Executable Mappings in Model Management. In *SIGMOD*, pages 167–178, 2005.
38. S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, pages 193–204, 2003.
39. Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, 2000.
40. Lajos Nagy and Ryan Stansifer. Polymorphic type inference for the relational algebra in the functional database programming language neon. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 673–678, New York, NY, USA, 2006. ACM.
41. Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
42. Benjamin C. Pierce. Adventures in bi-directional programming, December 2007. FSTTCS invited talk.
43. Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.

44. Alexandra Silva and Joost Visser. Strong types for relational databases. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 25–36, New York, NY, USA, 2006. ACM.
45. Jan van den Bussche and Emmanuel Waller. Type inference in the polymorphic relational algebra. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 80–90, New York, NY, USA, 1999. ACM.
46. Jérôme Vouillon. Polymorphic regular tree types and patterns. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 103–114, New York, NY, USA, 2006. ACM.
47. C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, pages 1006–1017, 2005.