Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 200 (2008) 3–23
www.elsevier.com/locate/entcs

Coupled Transformation of Schemas, Documents, Queries, and Constraints[1]

Joost Visser[2]

Software Improvement Group & CWI
Amsterdam, The Netherlands

Abstract

Coupled transformation occurs when multiple software artifacts must be transformed in such a way that they remain consistent with each other. For instance, when a database schema is adapted in the context of system maintenance, the persistent data residing in the system's database needs to be migrated to conform to the adapted schema. Also, queries embedded in the application code and any declared referential constraints must be adapted to take the schema changes into account. As another example, in XML-to-relational data mapping, a hierarchical XML Schema is mapped to a relational SQL schema with appropriate referential constraints, and the XML documents and queries are converted into relational data and relational queries. The 2LT project is aimed at providing a formal basis for coupled transformation. This formal basis is found in data refinement theory, point-free program calculation, and strategic term rewriting. We formalize the coupled transformation of a data type by an algebra of information-preserving data refinement steps, each witnessed by appropriate data conversion functions. Refinement steps are modeled by so-called two-level rewrite rules on type expressions that synthesize conversion functions between redex and reduct while rewriting. Strategy combinators are used to composed two-level rewrite rules into complete rewrite systems. Point-free program calculation is applied to optimized synthesize conversion function, to migrate queries, and to normalize data type constraints. In this paper, we provide an overview of the challenges met by the 2LT project and we give a sketch of the solutions offered.

Keywords: Coupled transformation, two-level transformation, model transformation, data refinement, strategic term rewriting, format evolution, data mappings, point-free program transformation, query migration, constraint propagation

1 Introduction

In the context of assessing and monitoring scores of industrial software systems [21,26,27,11] we have had the opportunity to obtain an overview over the challenges and problems that beset modern software engineering practise.

Whether looking at administrative and financial transaction systems or at embedded control software, invariably three important sources of complexity can be distinguished. Firstly, the *internal architecture* of these systems is complex in the

[1] This paper is the extended abstract of an invited talk of the same title.

[2] Email: j.visser@sig.nl

doi:10.1016/j.entcs.2008.04.090

sense that they are composed from various components constructed in different technologies. A typical combination is a database programmed in PL/SQL or T-SQL, business logic encoded in Java or C# application code, and a user interface built as ASP or JSP pages. Secondly, the *external architecture* of these systems is complex in the sense that they connect to various other systems and provide interfaces to various types of users. The communication channels to these systems and users ranges from web services and message queues to spool directories and screen scraping. Thirdly, complexity derives from the various layers and perspectives present in the software production process. Apart from source, byte, and binary code, system construction includes artifacts such a configuration files, documentation, requirements, UML models, protocol specifications, generators, and document schemas. Due to these sources of complexity, a software system can be seen as a *network of software artifacts* connected by various kinds of relationships.

During software development, maintenance, and evolution, the artifacts of which a system consists must be enhanced and adapted in such a way that their interrelationships are kept intact, i.e. such that the artifacts remain somehow *consistent* with each other. Typically, a single change request impacts more than a single artifact. For example, an additional entry field in an online form may induce changes in the communication protocol with a back-office system and in the schema, triggers, and stored procedures of the underlying database. Such coordinated changes tend to be labour-intensive and error-prone. Interconnections are easily overlooked and implementation decision must be made for each impacted artifact.

These observations lead us to posit that a large proportion of the costs and failures in software engineering derive from the profoundly *ad-hoc* approach to the preservation of *consistency between software artifacts* as they undergo change.

Lämmel coined the term "coupled software transformation" for the transformation of two or more software artifacts where changes to one artifact necessitate changes to the other(s) in order to maintain global consistency [28]. In spite of the widespread occurrence of coupled transformation in problem domains such as cooperative editing, software modeling, model transformation, and re-/reverse engineering, Lämmel identified an important remaining research challenge in providing a general and comprehensive conceptual framework for coupled transformations [29].

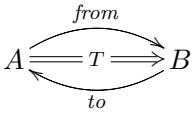
In the 2LT project, we have taken up the challenge of providing such a framework. We have focussed initially on an important instance of coupled transformation that involves a transformation on the level of types, coupled with transformations on the level of values and operations. We have provided a formalisation of such *two-level transformations* and we have constructed tooling to support these transformations [15,18,7,19,1]. In this paper, we provide an overview of this work.

Section 2 provides a detailed description of the challenges involved in formalizing and supporting two-level transformation. In subsequent sections, these challenges are tackled. Section 3 discusses how data refinement theory can be used to model two-level transformations. A strategic term rewriting system is constructed in which type-transformations induce the automatic composition of conversion functions between source and target types. Section 4 shows how program transformation

techniques can be used to transform automatically composed conversion functions as well as other value-level operations. These additional transformations enable optimization of conversion functions and migration of queries. Section 5 discusses how two-level transformation can be made constraint-aware, in the sense that constraints on transformed types can be propagated and introduced during type-level transformation. We discuss related work in Section 6. In Section 7 we summarize the contributions of the 2LT project and we indicate avenues of future work.

2 The challenge

Diagrammatically, a two-level transformation can be depicted as follows:



Thus, at the type level, T transforms a source datatype A into a target datatype B. This type-level transformation is witnessed by conversion functions *to* and *from* between the source and target datatypes. In the sequel, the exact nature of the 'witness' relationship will become clear. We start by considering some examples of two-level transformation scenarios.

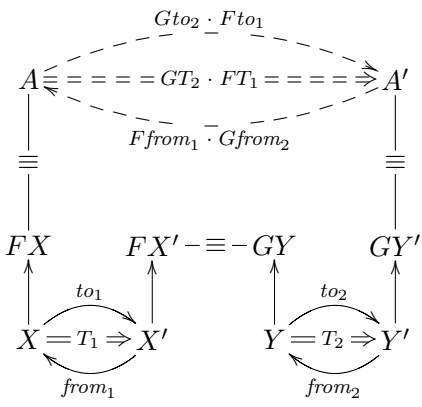
When a database schema is adapted in the context of system maintenance, the persistent data residing in the system's database needs to be migrated to conform to the adapted schema. When the grammar of a programming language is modified, the source code of existing applications and libraries written in that language must be upgraded to the new language version. These scenarios are examples of *format evolution* [30] where a data structure and corresponding data instances are transformed in small, infrequent, steps, interactively driven during system maintenance.

Similar coupled transformation of data types and corresponding data instances are involved in the scenario of *data mapping* [31]. Such mappings generally occur on the boundaries between programming paradigms, where for example object models, relational schemas, and XML schemas need to be mapped onto each other for purposes of interoperability or persistence. Data mappings tend not to be evolutionary, but rather involve fully automatic translation of entire data structures, carried out during system operation.

What these two-level transformation scenarios have in common is that a type-level transformation (of the schema or format) determines value-level transformations (of the documents or data instances). The challenges posed by providing a general framework for these scenarios are explained below.

2.1 Driving value-level composition by type-level composition

The diagram in Figure 1 depicts the composition of two type-safe transformation steps into a more complex transformation. In this diagram, a datatype A is transformed in two steps into a new datatype. A type X occurs nested inside A, where the nesting context is captured by the datatype constructor F, i.e. $A \equiv FX$. For



A	datatype
FX	breakdown of A into F applied to X
X	nested datatype
X'	transformed nested datatype
T_1	transformation of type X into X'
to_1	conversion function $X \to X'$
$from_2$	conversion function $X' \to X$
FX'	result of transforming nested type X
GY	alternative breakdown of FX'
GY'	result of transforming nested type Y
A'	transformed datatype

Fig. 1. The two-level transformations steps T_1 of data type X to data type X' and T_2 of data type Y to Y' are combined into a more complex two-level transformation of A to A'. Two kinds of composition are employed: sequential composition indicated by $\cdot$, and structural composition indicated by the data type constructors F and G and their corresponding map functions of the same name. The challenge is to drive composition at the value level by composition at the type level.

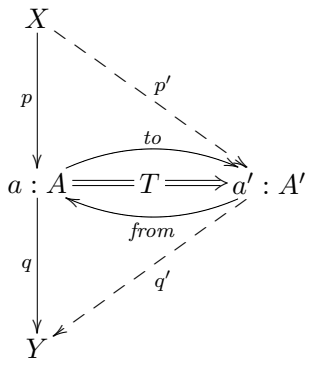
example, if A is the type of lists of X elements, then F would be the list constructor. The map function associated to F is denoted by the same name.

In the first transformation step, the nested type X is transformed by transformation T_1, witnessed by to_1 and $from_1$. To pull the nested transformation to the level of A itself, the map operator associated to F is applied, which results in the transformation FT_1 (where we once more overload the symbol F) which converts FX into the intermediate type FX'. The witnessing conversion functions are lifted to Fto_1 and $Ffrom_1$. This is an example of *structural* composition of two-level transformations.

For the second transformation step, this intermediate type is broken down differently, revealing a nested type Y, i.e. $FX' \equiv GY$. Subsequently, a second transformation T_2 is applied, again lifted to the top level, but now using the G constructor and its associated map operator. Finally, the two conversions are *sequentially* composed. This entails applying function composition to the conversion functions, to obtain $Gto_2 \cdot Fto_1$ and $Ffrom_1 \cdot Gfrom_2$ as witnesses of the overal transformation.

When developing a framework for two-level data transformation, the challenge arises to drive composition at the value level by composition at the type level. In other words, from a compositional specification of the transformation of one type into another, it should be possible to derive compositional specifications of the value-level transformation functions that convert between values of these types. Moreover, the derivation should be *dynamic*, in the sense that the target type of the type-level transformation can not be assumed to be known before hand, but is only arrived at by actually carrying out the transformation. Likewise, the types of the derived conversion functions, as well as their compositional specification are computed dynamically.

This implies a further challenge regarding the degree of type-safety that can be achieved for the various composition operators. As we demonstrated in [15], such dynamic two-level data transformation systems can in fact be developed in a



A	datatype
A'	transformed datatype
a,a'	instances of A and A', resp.
p	producer, generates data of type A
q	query, consumes data of type A
T	transformation of type A into type A'
to	migration function of type $A \to A'$
from	migration function of type $A' \to A$
p'	transformed producer
q'	transformed query

Fig. 2. Coupled transformation of data type A, data instance a, data producer p, and data query q. The challenge is to calculate p' and q' by fusing the compositions $to \circ p$ and $q \circ from$ such that they work on A' directly rather than via A.

type-safe manner by judicious use of dynamic types. Using this approach, value-level transformations are statically checked to be well-typed with respect to the type-level transformations to which they are associated, and well-typed composition of type-level transformation steps induces well-typed compositions of value-level transformation steps. The approach will be reviewed in Section 3.

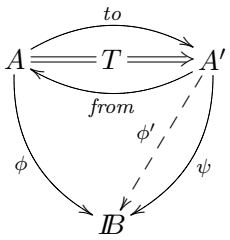
2.2 Fusion and migration of data processing operations

When providing a framework for two-level transformation, two further challenges can be recognized, both related to data processing operations at the value level.

Firstly, the dynamically computed compositions of value-level functions resulting from two-level transformations fulfill the task of converting source values into target values and back. But they do not necessarily perform this task in the best possible way. In particular, these functions may include redundant intermediate steps and may not perform their steps in optimal order. Thus, a further challenge arises to post-process conversion functions after composing them, in such a way that conversion steps are reordered and fused and more optimal conversions are derived.

Secondly, data processing operations may exist on the source type of a two-level transformation that we wish to *migrate* somehow to the target type. This issue of migrating data operations is illustrated in Figure 2. Again, let a type-level transformation T of a source type A into a target type A' be witnessed by associated instance migration functions *to* and *from*. The query q that consumes values of type A and the producer p that generates such values are examples of data processing programs. To obtain queries and producers on the transformed type A', we can simply compose q and p with the migration functions *from* and *to*. This amounts to a *wrapper* approach to program migration where the original type and the original processors are still explicitly present. The challenge, however, is to calculate processors q' and p' from those wrapper compositions in such a way that they no longer involve the original type and processors.

In [18] we demonstrated that both these challenges involving data processing function can be tackled by the use of program transformation techniques. The key



A, A'	datatype and transformed datatype
T	transformation of type A into type A'
to	migration function of type $A \to A'$ (injective)
from	migration function of type $A' \to A$ (surjective)
ϕ, ϕ'	constraint and transformed constraint
ψ	newly introduced constraint

$\phi' = \phi \circ \mathit{from} \wedge \psi$

Fig. 3. Constraint-aware transformation of datatype A with constraint ϕ into datatype A' with constraint ϕ'. The constraint on the target type is the logical conjunction of (i) the constraint on the source type post-composed with the migration function *from*, and (ii) any new constraint ψ introduced by the type--change. When ϕ' is normalized it works on A' directly rather than via A. The challenge is to take into account introduction, propagation, and matching of constraints during transformation at the type-level.

idea is to use fusion or deforestation techniques [46] in order to eliminate intermediate data types. This approach will be reviewed in Section 4.

2.3 Constraint-aware transformation

Generally, schema definitions consist of a structural description augmented with constraints that capture additional semantic restrictions. For example, SQL database schemas and XSD document schemas may declare referential integrity constraints, grammars include operator precedences, VDM specifications contain datatype invariants. When a data schema is transformed, the corresponding constraints must also be adapted.

Figure 3 concisely illustrates the issue of constraint-aware schema transformation. In general, constraints can be represented by boolean-valued functions. Two kinds of contraint-awareness are involved in transformation T from type A to type A'. Firstly, *constraint-propagation* concerns the migration of a constraint on the source type A into a constraint on the target type. This is achieved by composing a constraint ϕ on the source data type with a backward conversion function *from* between target and source type. Secondly, some transformation steps may require the imposition of a new constraint on the target type. Such *constraint-introduction* is achieved by logical conjunction of a new constraint ψ to the propagated constraint. A third form of contraint-awareness, not illustrated in the figure, occurs when a transformation step can be applied only if a certain constraint holds on the input type. In that case, *constraint-matching* is required, and *constraint-discharge* may be appropriate. The challenge is to include constraint-awareness into the framework for two-level transformation.

In [1], we showed that constraint-awareness can be built into our framework for two-level transformation in a straightforward manner. In this approach, constraints are represented in a similar manner as data conversion functions and queries. Unlike these data processing operations, the functions that represent constraints are embedded into representations of types. The approach is explained in more detail in Section 5.

3 Two-level transformation as data refinement

In this section, we explain how data refinement theory, combined with typed strategic term rewriting can be used to provide an initial framework for two-level transformation [15]. This initial framework addresses the first challenge of two-level transformation (formulated in Section 2.1) of driving composition at the value level by composition at the type level.

3.1 Data refinement

At the heart of the 2LT project lies the observation that two-level transformations are in essence data refinements. Data refinement theory provides an algebraic framework for calculating with datatypes [42,38,39,40]. The following inequation captures the essence of refining a datatype A to a datatype B:

$$A \underset{from}{\overset{to}{\leqslant}} B \quad \text{where} \quad \begin{cases} to : A \to B \text{ injective and total} \\ from : B \to A \text{ surjective} \\ from \cdot to = id_A \end{cases}$$

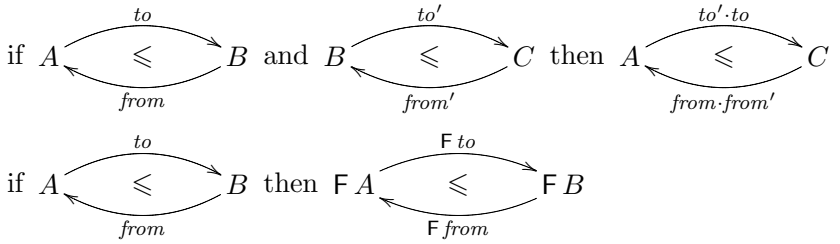
Here, id_A is the identity function on datatype A. Thus, the inequation $A \leqslant B$ expresses that B is a refinement of A, which is witnessed by the conversions functions *to* and *from*. The *to* function is required to be injective and total, while the *from* function must be surjective. (In fact, *to* can be any injective and total relation, not necessarily a function.)

Since the equality of two relations (or functions) is a bi-inclusion we can read the equation $from \cdot to = id_A$ in two directions. In the first direction ($id_A \subseteq from \cdot to$), we read that every inhabitant of datatype A has a representation in datatype B, which means that no information is lost when switching from A to B. In the reverse direction ($from \cdot to \subseteq id_A$), the equation expresses that there is no "confusion" in the transformation process, in the sense that only one inhabitant of the datatype A will be transformed to a given representative in datatype B. Thus, data refinements are not arbitrary transformations on types. They arise from the existence of witnessing functions whose properties preclude data mixup.

When applied left-to-right, an inequation $A \leqslant B$ will preserve or enrich information content, while applied in the right-to-left direction it will preserve or restrict information content. In a situation where B is not only a refinement of A, but also vice versa, we have an isomorphism $A \cong B$. This is a special case of the $\leqslant$-inequation which works in both directions.

On the basis of this formalization of data refinement, an algebraic theory for calculation with datatypes has been constructed [42]. This theory is summarized in Figure 4. We will discuss the various parts of this figure to explain how data refinement theory can be used to provide a formal framework for two-level transformation.

Sequential and structural composition



Hierarchical-relational data mapping

$A^\star \leqslant \mathbb{N} \rightharpoonup A$	List elimination
$2^A \cong A \rightharpoonup 1$	Set elimination
$A? \cong 1 \rightharpoonup A$	Optional elimination
$A + B \leqslant A? \times B?$	Sum elimination
$A \times (B + C) \cong (A \times B) + (A \times C)$	Distribute product over sum
$A \rightharpoonup (B + C) \leqslant (A \rightharpoonup B) \times (A \rightharpoonup C)$	Distribute map over sum
$(B + C) \rightharpoonup A \cong (B \rightharpoonup A) \times (C \rightharpoonup A)$	Distribute map over sum
$A \rightharpoonup (B \times (C \rightharpoonup D)) \leqslant (A \rightharpoonup B) \times (A \times C \rightharpoonup D)$	Flatten nested map
$\mu\mathsf{F} \leqslant (K \rightharpoonup \mathsf{F}\,K) \times K$	Recursion elimination

Format evolution

$A \leqslant A \times B$	Add field	$A^+ \leqslant A^\star$	Allow empty list
$A \leqslant A + B$	Add alternative	$A? \leqslant A^\star$	Allow repetition
$A \leqslant A?$	Make optional	$A \leqslant A^+$	Allow non-empty repetition

Fig. 4. Summary of data refinement theory. For a complete account, the reader is referred to Oliveira [42]. Note that $\cdot \rightharpoonup \cdot$ denotes a simple relation, of which finite maps are a special case.

3.2 Sequential and structural composition laws

The top part of Figure 4 shows data refinement laws for sequential and structural composition.

The law for sequential composition of data refinements expresses that individual two-level transformation steps can be chained by sequentially composing abstraction and representation functions. Such transitivity, together with the fact that any datatype can be transformed to itself (reflexivity, witnessed by identity functions $from = to = id$), means that $\leqslant$ is a preorder.

The law for structural composition of data refinements expresses that two-level transformation steps can be applied, not only at the top-level of a datatype, but also at deeper levels. Such transformations on locally nested datatypes must then be propagated to the global datatype in which they are embedded. For example, a transformation on a local XML element must induce a transformation on the level of a complete XML document. In the formulation of the law, F is a *functor* that models the *context* in which a transformation step is performed. Recall that a functor F from category C to D is a mapping that (i) associates to each object X in C an object $\mathsf{F}X$ in D, and (ii) associates to each morphism $f : X \to Y$ in C a morphism $\mathsf{F}f : \mathsf{F}X \to \mathsf{F}Y$ in D such that identity morphisms and composition of morphisms are preserved. When modeling two-level transformations, the objects X

Primitive combinators	
$nop : Rule$	do nothing
$(\triangleright) : Rule \to Rule \to Rule$	sequential composition
$(\oslash) : Rule \to Rule \to Rule$	alternative composition
$all : Rule \to Rule$	apply to all immediate child types
$one : Rule \to Rule$	apply to exactly one immediate child type
Defined combinators	
$try : Rule \to Rule$	$many : Rule \to Rule$
$try\ r = r \oslash nop$	$many\ r = try\ (r \triangleright many\ r)$
$topdown : Rule \to Rule$	$once : Rule \to Rule$
$topdown\ r = r \triangleright all\ (topdown\ r)$	$once\ r = r \oslash once\ (once\ r)$
$innermost : Rule \to Rule$	
$innermost\ r = all\ (innermost\ r) \triangleright try\ (r \triangleright innermost\ r)$	

Fig. 5. Summary of strategic term rewriting combinators. Only signatures are shown for primitive combinators. The definitions that instantiate these combinators for two-level transformations can be found in [15].

and Y are data types, and the morphisms f are value-level transformations.

Thus, a functor F captures (i) the embedding of local datatypes A or B inside global datatypes, and (ii) the lifting of value-level transformations *to* and *from* on the local datatypes to value-level transformations on the global datatypes, in a way such that the preorder (transitivity and reflexivity) on local datatypes is preserved on the global datatypes. Generally, a functor that mediates between a global datatype and a local datatype is constructed from primitive functors, such as products $A \times B$, sums $A + B$, finite maps $A \rightharpoonup B$, sequences $A^\star$, sets 2^A, etc. By modeling the context of a local datatype by a composition of such functors, the propagation of two-level transformations from local to global datatype can be derived.

3.3 Strategy combinators for two-level transformation

Based on the sequential and structural composition laws of data refinement theory, we have created a suite of combinators that allows the compositional construction of transformation systems from individual transformation steps. In fact, we have defined a new instantiation of a well-known suite of operators that have previously been defined for strategic rewriting of terms [44,45,33,35]. A summary of the combinator suite is presented in Figure 5. The novelty lies in instantiating the combinators for two-level transformation. We will explain in detail how this instantiation can be done.

Firstly, we need to define a datatype to represent types. The instances of this datatype, i.e. the type representations, will be subject to rewriting. Elsewhere [15] we discuss how such representations can be defined by a *generalized algebraic datatype* (GADT), following a well-known technique [43,24]. In Haskell syntax, this GADT is defined as follows:

data *Type a* **where**

$$
\begin{array}{ll}
\mathit{Int} & :: \mathit{Type}\ \mathit{Int} \\
\mathit{String} & :: \mathit{Type}\ \mathit{String} \\
\cdot + \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a + b) \\
\cdot \times \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b) \\
\mathit{List} & :: \mathit{Type}\ a \to \mathit{Type}\ [a] \\
\ldots &
\end{array}
$$

The inhabitants of type $\mathit{Type}\ a$ are representations of type a. For example, if t is of type $\mathit{Type}\ \mathit{Integer}$, then t represents the type $\mathit{Integer}$.

Secondly, we use this type of type representations to define the following type of rewrite rules:

$$\mathit{Rule} = \forall a\ .\ \mathit{Type}\ a \to M\ (\exists b\ .\ a \to b \times b \to a \times \mathit{Type}\ b)$$

Thus, a rewrite rule consumes a representation of type a, and returns a triple, embedded in monad M. The monad is used to represent partiality (success and failure of rewrite rules). The triple contains a representation of type b into which a is refined, as well as the two witnessing functions *to* and *from* that convert between a and b. Thus, a rewrite rule does not simply transform values into other values, as is the case in normal rewrite systems. Rather a rewrite rule is defined as a transformation of one type representation into another, witnessed by value-level transformations. The universal quantifier expresses that rewrite rules are polymorphic in a, i.e. they can be applied to representations of any type. The existential quantifier expresses that the target type of the refinement is computed *dynamically*, i.e. is not known before executing the rule.

Now, the combinator suite of strategic term rewriting combinators can be instantiated for two-level transformations. For example, *nop* is defined as:

$$
\begin{array}{l}
\mathit{nop} : \mathit{Rule} \to \mathit{Rule} \\
\mathit{nop}\ t = \mathit{return}\ (\mathit{id}, \mathit{id}, t)
\end{array}
$$

Here, *return* is the unit function of the monad M. For the other primitive combinators in Figure 5 similar definitions can be given [15]. This means that strategic term rewriting is fully enabled for two-level transformation: one-step two-level transformations can be composed in arbitrary ways into complex two-level rewrite systems that dynamically compute target types, while composing witnessing functions sequentially and structurally as needed.

3.4 Rules for data mapping and format evolution

In [2] we presented a set of two-level transformation rules that can be combined with combinators presented above into a calculator that automatically converts a hierarchic, possibly recursive data structure to a flat, relational representation. These rules are summarized in abbreviated form in the middle part of Figure 4. For

example, the law $A \rightharpoonup (B \times (C \rightharpoonup D)) \leqslant (A \rightharpoonup B) \times (A \times C \rightharpoonup D)$ abbreviates:

$$A \rightharpoonup (B \times (C \rightharpoonup D)) \underset{njoin}{\overset{unnjoin}{\leqslant}} (A \rightharpoonup B) \times (A \times C \rightharpoonup D)$$

This particular law describes the flattening of nested maps into separate maps, where the key of the inner map is extended with the key of the outer map.

Jointly, the rules for hierarchical-relational data mapping are designed for stepwise elimination of sums, sets, optionals, lists, recursion, and such, in favor of finite maps and products. When applied according to an appropriate strategy, they will lead to a normal form that consists of a product of basic types and maps, which is readily translatable to a relational database schema in SQL [2,15]. There are rules for elimination and distribution, and a particularly challenging rule for recursion elimination, which introduces pointers in the locations of recursive occurrences.

While data mappings rely on a automatic and fully systematic strategy for applying individual transformation rules, format evolution assumes more surgical and adhoc modifications. For instance, new requirements might call for the introduction of a new data field, or for the possible omission of a previously mandatory field. The lower part of Figure 4 shows in abbreviated form a set of two-level transformation rules that cater for these scenarios. These rules formalize coupled evolution of XML documents and their DTDs as discussed by Lämmel *et al* [30]. For example, the law $A \leqslant A \times B$ abbreviates:

$$A \underset{\pi_1}{\overset{\lambda x.(x,b)}{\leqslant}} A \times B$$

This law for adding a field assumes that a new value b for that field is somehow supplied. This may be done through a generic default for type B, through interaction with a user or some other oracle, or by querying another part of the data.

4 Transformation of queries and conversions

In this section, we take up the next challenge, formulated in Section 2, of fusing and migrating data processing operations [18]. To this end, we will employ well-known techniques for point-free program transformation.

4.1 Point-free program transformation

In his 1977 Turing Award lecture, Backus advocated a variable-free style of functional programming, on the basis of the ease of formulating and reasoning with algebraic laws over such programs [5]. Others have adopted, complemented, and extended this so-called point-free style of programming [23,16], and a summary is shown in Figure 6. The most fundamental combinators of point-free programming

Primitive combinators	
$id : A \to A$	$(\circ) : (B \to C) \to (A \to B) \to (A \to C)$
$\pi_1 : A \times B \to A$	$(\vartriangle) : (A \to B) \to (A \to C) \to (A \to B \times C)$
$\pi_2 : A \times B \to B$	$(\times) : (A \to B) \to (C \to D) \to (A \times C \to B \times D)$
$\delta : (A \rightharpoonup B) \to Set\ A$	$list : (A \to B) \to ([A] \to [B])$
$\rho : (A \rightharpoonup B) \to Set\ B$	$set : (A \to B) \to (Set\ A \to Set\ B)$
$\bowtie_n : ((A \rightharpoonup B) \times ((A \times C) \rightharpoonup D)) \to (A \rightharpoonup (B \times (C \rightharpoonup D)))$	
$\bowtie_n^{-1} : (A \rightharpoonup (B \times (C \rightharpoonup D))) \to ((A \rightharpoonup B) \times ((A \times C) \rightharpoonup D))$	
Laws	
$f \circ id = f$	$id \circ f = f$ $\quad F\ id = id$
$f \circ (g \circ h) = (f \circ g) \circ h$	$F\ f \circ F\ g = F\ (f \circ g)$
$\pi_1 \circ (f \vartriangle g) = f$	$\pi_2 \circ (f \vartriangle g) = g$
$\pi_1 \vartriangle \pi_2 = id$	$(f \times g) \circ (h \vartriangle i) = (f \circ h) \vartriangle (g \circ i)$
$\pi_1 \circ (f \times g) = f \circ \pi_1$	$\pi_2 \circ (f \times g) = g \circ \pi_2$
$id \times id = id$	$(f \times g) \circ (h \times i) = (f \circ h) \times (g \circ i)$
$list\ id = id$	$list\ f \circ list\ g = list\ (f \circ g)$
$set\ id = id$	$set\ f \circ set\ g = set\ (f \circ g)$

Fig. 6. Summary of point-free program transformation. For a complete account, the reader is referred to Cunha et al. [16].

are function composition and the identity function. Apart from these, every type constructor, such as binary product, disjoint sum, lists comes with its own associated set of operators. The laws for these operators describe properties such as associativity and commutativity, but also expansion and cancelation properties.

4.2 Strategy combinators for point-free program transformation

As for two-level transformations, we can harnas the algebraic laws of point-free program transformation into a strategic term rewriting system. For this purpose, we again instantiate the strategic term rewriting combinators of Figure 5, but using a different type of rewrite rules. The subject of rewriting in these rules are not type representations, but representations of functions. For this representation, we resort again to a GADT:

```
data F f where
  Id    :: F (a → a)
  Comp  :: Type b → F (b → c) → F (a → b) → F (a → c)
  Fst   :: F ((a, b) → a)
  Snd   :: F ((a, b) → b)
  · △ · :: F (a → b) → F (a → c) → F (a → (b, c))
  · × · :: F (a → b) → F (c → d) → F ((a, c) → (b, d))
  ...
```

Thus, an inhabitant of type $\mathsf{F}\ (a \to b)$ is a point-free representation of a function of type $a \to b$. In addition, we define an evaluator for function representations:

$$
\begin{aligned}
&\mathit{eval} :: \mathsf{F}\ a \to a\\
&\mathit{eval}\ \mathit{Id} = \lambda x \to x\\
&\mathit{eval}\ (\mathit{Comp}\ \_\ f\ g) = \lambda x \to (\mathit{eval}\ f)\ (\mathit{eval}\ g\ x)\\
&\mathit{eval}\ \mathit{Fst} = \lambda(x, y) \to x\\
&\mathit{eval}\ \mathit{Snd} = \lambda(x, y) \to y\\
&\mathit{eval}\ (f \vartriangle g) = \lambda x \to (\mathit{eval}\ f\ x, \mathit{eval}\ g\ x)\\
&\mathit{eval}\ (f \times g) = \lambda(x, y) \to (\mathit{eval}\ f\ x, \mathit{eval}\ g\ y)\\
&\ldots
\end{aligned}
$$

With *eval* we can always return from function representations to the represented functions themselves.

Now we can define the rule type for instantiation of the combinator suite for strategic term rewriting of point-free programs:

$$\textbf{type}\ \mathit{Rule} = \forall a\ .\ \mathit{Type}\ a \to \mathsf{F}\ a \to M\ (\mathsf{F}\ a)$$

Here, M is again a monad. Thus, rewrite rules are basically monadic functions on point-free representations, additionally parameterized with a type representation. This additional parameter of rules is used for *type-directed* rewriting, i.e. it allows us to create rewrite rules that decide their applicability on the basis of the type of their input expression.

The actual instantiation of the combinators suite can now be done in a straightforward manner. For example, the *nop* combinator is defined as follows:

$$
\begin{aligned}
&\mathit{nop} :: \mathit{Rule}\\
&\mathit{nop}\ t\ f = \mathit{return}\ f
\end{aligned}
$$

For the definition of the other primitive combinators, we refer elsewhere [18].

4.3 Combining two-level transformation with point-free program transformation

In order to combine the two-level transformation systems of Section 3 with point-free program transformation, we modify the type of two-level rewrite rules:

$$\mathit{Rule} = \forall a\ .\ \mathit{Type}\ a \to M\ (\exists b\ .\ \mathsf{F}\ (a \to b) \times \mathsf{F}\ (b \to a) \times \mathit{Type}\ b)$$

Thus, the conversion functions in the result triplet are replaced by point-free representations of these functions. As a consequence, the conversion functions that are composed during type-transformation can afterwards be subjected to transformation with the transformation system for point-free programs. In particular, a rewriting systems can be applied that applies fusion rules to eliminate intermediate data types. After such optimisation, the *eval* function can be applied to obtain the optimized function itself from its representation.

5 Constraint-aware transformation

In this section, we take up the challenge, formulated in Section 2, of making two-level transformations constraint-aware. This can be done by augmenting type representations with representations of boolean-valued point-free functions that capture constraints [1].

5.1 Data types with constraints

A constraint on a datatype can be modeled as a unary predicate, i.e. a boolean function which distinguishes between legal values and values that violate the constraint. To associate a constraint to a type, we will write it as a subscript:

$$A_\phi \quad \text{where} \quad \phi : A \to \mathbb{B} \text{ total and functional}$$

This notation, as well as some of the results below, originates in [41]. We will write constraints as much as possible as point-free expressions, to enable subsequent calculation with them. For example, the following datatype represents two tables with a foreign key constraint:

$$((A \rightharpoonup B) \times (C \rightharpoonup A \times D))_{(set\ \pi_1)\circ\rho\circ\pi_2 \subseteq \delta\circ\pi_1}$$

Here we use projection functions π_1 and π_2 to select the left or right table, we use δ and ρ to select the domain and range of a map, and *set* f to map a function f over the elements of a set. Note that we use a lifted variant on the set inclusion operator: $\cdot \subseteq \cdot : (A \to Set\ B) \to (A \to Set\ B) \to (A \to \mathbb{B})$.

When a second constraint is added to a constrained datatype, both constraints can be composed with logical conjunction:

$$(A_\phi)_\psi \equiv A_{\phi \wedge \psi} \qquad \text{logical composition}$$

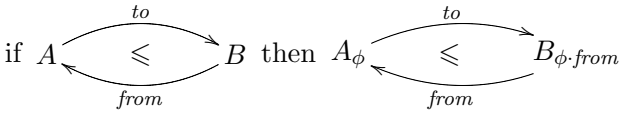
Note that we use a variant of the conjunction operator lifted to point-free predicates: $\cdot \wedge \cdot : (A \to \mathbb{B}) \to (A \to \mathbb{B}) \to (A \to \mathbb{B})$. When a constraint is present on a datatype under a functor, the constraint can be pulled up through the functor (for a categorical proof, see [41]):

$$\mathsf{F}\,(A_\phi) \equiv (\mathsf{F}\,A)_{(\mathsf{F}\,\phi)} \qquad \text{functorial pull}$$

For example, a constraint on the elements of a list can be pulled up to a constraint on the list: $(A_\phi)^\star \equiv (A^\star)_{list\phi}$.

5.2 Introducing, propagating, and eliminating constraints

The laws of the data refinement calculus must be enhanced to deal with constrained datatypes. Firstly, if a constrained datatype is refined with a 'classic' law, i.e. a law that does not involve constraints, the constraint must be properly propagated through the refinement:



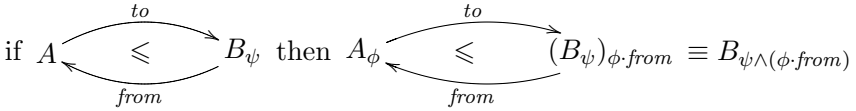
Thus, the constraint of the source datatype is propagated to the target datatype, where it is post-composed with the backward conversion function *from*. Such compositions can give rise to opportunities for point-free program transformation, as we will see further on.

Several refinement laws can be changed from inequations to isomorphisms by adding a constraint to the target type. For example, the laws from Figure 4 for sum elimination, distribution of map over sum in its range, and flattening of nested maps can be enhanced as follows:

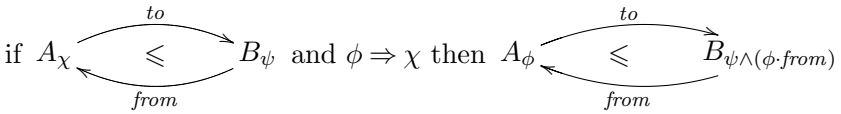
$$A + B \cong A? \times B?_{(\epsilon \circ \pi_1) \oplus (\epsilon \circ \pi_2)}$$
$$A \rightharpoonup (B + C) \cong (A \rightharpoonup 1) \times (A \rightharpoonup B) \times (A \rightharpoonup C)_{(\delta \circ \pi_2 \subseteq \delta \circ \pi_1) \wedge (\delta \circ \pi_3 \subseteq \delta \circ \pi_1)}$$
$$A \rightharpoonup (B \times (C \rightharpoonup D)) \cong (A \rightharpoonup B) \times (A \times C \rightharpoonup D)_{(set\ \pi_1) \circ \delta \circ \pi_2 \subseteq \delta \circ \pi_1}$$

Here, we have used point-free variants of exclusive disjunction ($\oplus$) and a test for emptiness of an optional (ϵ).

When applying a law that introduces a constraint to a datatype that already has a constraint, the new and existing constraints must be combined:



This is the *invariant pulling* theorem of [41]. A more general case arises when not only the target, but also the source is constrained in the law that is applied:



Here we use a point-free variant on logical implication ($\Rightarrow$) to state that the actual constraint ϕ on A must imply the required constraint χ.

Constraints can not only be introduced and propagated. They can also be weakened or even eliminated, by virtue of the following:

$$\text{if } \phi \Rightarrow \psi \text{ then } A_\phi \leqslant A_\psi$$

In the special case that ψ is the constant true predicate, such weakening boils down to elimination of a constraint.

5.3 Representation of constrained types

To represent constrained datatypes, the first GADT above needs to be enhanced with another constructor:

```
data Type t where
  ...
  (·). :: Type a → F (a → B) → Type a
```

Thus, the $(\cdot).$ constructor has as first argument the type that is being constrained, and as second argument the function that represents the constraint. This use of the function representation inside the type representation has as important consequence that the rewriting system for functions is now embedded into the rewrite system for types.

6 Related work

6.1 Software transformation

Lämmel *et al* [30] propose a systematic approach to evolution of XML-based formats, where DTDs are transformed in a step-wise fashion, and migration of corresponding documents can largely be induced from the DTD-level transformations. They discuss properties of transformations and identify categories of transformation steps, such as renaming, introduction and elimination, folding and unfolding, generalization and restriction, enrichment and removal, taking into account many XML-specific issues, but they stop short of formalization and implementation of two-level transformations. In fact, they identify the following 'challenge':

> "We have examined typeful functional XML transformation languages, term rewriting systems, combinator libraries, and logic programming. However, the coupled treatment of DTD transformations and induced XML transformations in a typeful and generic manner, poses a challenge for formal reasoning, type systems, and language design."

We have taken up this challenge by showing that formalization is feasible.

Lämmel *et al* [31] have identified data mappings as a challenging problem in software engineering practice, and data-processing application development in particular. An overview is provided over examples of data mappings and of existing approaches in various paradigms and domains. Some key ingredients are described for an emerging conceptual framework for mapping approaches, and 'cross-paradigm impedance mismatches' are identified as important mapping challenges. According to the authors, better understanding and mastery of mappings is crucial, and they identify the need for "general and scalable *foundations*" for mappings. Our formalization of two-level data transformation provides such foundations.

Cleve *et al* use the term 'co-transformation' for the process of re-engineering three kinds of artifacts simultaneously: a database schema, database contents, and application programs linked to the database [14,13]. They use generative and transformational techniques to transform data manipulation statements of legacy information systems, but is limited to information preserving transformations on procedural statements (basically: insert, delete, update). The approach abstracts over various languages (COBOL, Codasyl, SQL), but falls short of formalization and generalization. Transformations are wrapper based and do not involve fusion.

6.2 Generic functional programming

Type-safe combinators for strategic rewriting were introduced by Lämmel *et al* in [35], after which several simplified and generalized approaches were proposed [34,32,24]. These approaches cover type-preserving transformations (input and output types are the same), and type-unifying ones (all input types mapped to a single output type), but not *type-changing* ones.

Atanassow *et al* show how canonical isomorphisms (corresponding to laws for zeros, units, and associativity) between types can induce the value-level conversion functions [4]. They provide an encoding in the polytypic programming language Generic Haskell involving a universal representation of types, and demonstrate how it can be applied to mappings between XML Schema and Haskell datatypes. Beyond canonical isomorphisms, a few limited forms of refinement are also addressed, but these induce single-directional conversion functions only. A fixed strategy for normalization of types is used to discover isomorphisms and generate their corresponding conversion functions. By contrast, our type-changing two-level transformations encompass a larger class of isomorphism and refinements, and their compositions are not fixed, but definable with two-level strategy combinators. This allows us to address more scenarios such as format evolution, data cleansing, hierarchical-relational mappings, and database re-engineering.

6.3 Bi-directional programming

Foster *et al* tackle the *view-update problem* for databases with *lenses*: combinators for bi-directional programming [22]. Each lens connects a concrete representation C with an abstract view A on it by means of two functions $get : C \rightarrow A$ and $put : A \times C \rightarrow C$. Thus, *get* and *put* are similar to our *from* and *to*, except for *put*'s additional argument of type C. Also, an additional law on these functions guarantees that *put* can be used to reconstruct an updated C from an updated A. A more detailed treatment of bi-directional programming in the light of data refinement and two-level transformation is given by Oliveira [42].

On the level of problem statement, a basic difference exists between lenses and two-level transformations or refinements. In refinement, a (previously unknown) concrete representation is intended to be derived by calculation from an abstract one, while lenses start from a concrete representation on which one or more abstract views are then explicitly defined. This explains why some ingredients of our solution, such as representation of types at the value level, statically unkown types, and combinators for strategic rewriting, are absent in bi-directional programming.

6.4 Program transformation in calculational form

Several systems have been developed for performing program transformation in calculational form using fusion laws. Among these, MAG [37] and Yicho [25] are prominent, but both are targeted towards Haskell programs written in the pointwise style. In order to cope with fusion laws for generic recursion patterns both resort to advanced higher-order matching algorithms. We do not need such techniques

because our recursive functions are limited to very specific patterns, such as maps, for which fusion is easier to encode. A disadvantage of the MAG system is that it uses a fixed strategy to apply the transformation rules, while Yicho provides some basic strategy combinators.

Cunha *et al.* [17] presented a rewriting system for simplifying point-free expressions, which was used to optimize expressions resulting from a program transformation tool that translates pointwise Haskell code into point-free style. The main improvement of our approach is typing: we can use type representations to guide the rewriting process and rewrite rules are guaranteed to be type-safe. In his introductory book to Haskell programming [8], Bird presents a functional calculator that can also be used to simplify point-free expressions. Unfortunately, the expressions are not typed and, likewise to MAG, it uses a fixed rewriting strategy, which makes it difficult to apply in our scenarios.

6.5 Constraint-aware transformation

A large number of approaches has been proposed for mapping XML to relational databases [9,10,3,6], but usually without taking constraints into account. Lee *et al* [36] first addressed the issue of constraint preservation. Their CPI algorithm deals with referential integrity constraints and some cardinality constraints, which are stored in an annotated DTD dependency graph. When the graph is serialized to an SQL schema, various SQL constraints are generated along with the tables. In contrast to our approach, this graph-based algorithm does not deal with arbitrary constraints, it is specific for hierarchical-relational mapping, and it lacks type-safety and formal justification.

A notion of *XML Functional Dependency* (XFD) was introduced by Chen *et al* [12], based on path expressions. Mapping algorithms are provided that propagate XFDs to the target relational schema and exploit XFDs to arrive at a schema with less redundancy. Davidson *et al* [20] present an alternative constraint-preserving approach, also using path expressions. In contrast, our constraints are not restricted to relational integrity constraints. We have expressed constraints as point-free functions, which can be converted automatically to and from structure-shy programs including path expressions [19].

Barbosa *et al* [6] discuss generation of constraints on relational schemas that make XML-relational mappings information preserving, i.e. isomorphic. Non-structural constraints on the initial XML schema are not taken into account. Constraints and conversion functions are expressed in (variations on) Datalog, which can be (manually) rewritten to normal form in a mechanical way.

7 Concluding remarks

The aim of the 2LT project has been to take up the challenges involved in formalizing and supporting two-level transformations. The offered solutions consists of the combination of techniques for data refinement, type strategic term rewriting, point-free program transformation, and advanced functional programming. With

this combination of techniques, we have been able to tackle transformation of data schemas, coupled with transformation of data instances, queries, and constraints.

In order to progress to a more comprehensive solution for coupled transformation, a number of further issues need to be dealt with. We will briefly discuss a number of important ones.

Our approach so far has been limited to a number of fundamental type constructors, sufficient for modeling relational databases and most constructs found in XML schemas. However, a number of further constructs for data type construction would be desirable to include, such as mutual recursive datatype definitions, inheritance, and parametric polymorphism. These enhancements would enlarge the scope of 2LT to data formats such as grammars and object-oriented data models.

At the level of behaviour, the 2LT project has focussed on point-free functional programs as conversion functions, queries, and constraint definitions. For these programs, calculation laws are readily available and highly developed. However, it would be desirable to include other kinds of behavioural descriptions. For example, structure-shy query specifications as found in XPath have been shown to be amenable to calculation by converting them to and from point-free structure-sensitive programs [19]. Also, point-wise functional programs, as well as imperative programs with side effects need to be brought within scope. This would allow the application of the 2LT approach to more general model-transformation problems.

A particularly interesting challenge, would be to extend the 2LT approach to components and services. The challenge here would be to formalize and support the coupled transformation of components such as clients and servers, in such a way that wrapper and glue components can be introduced automatically and to some extent fused into the various components. Techniques that might be employed to meet this challenge include refinement of co-algebras, automata, and other component models. This avenue of elaboration could for example find application in evolution of web services.

Acknowledgement

Many have contributed to the 2LT project, including Tiago Alves, Pablo Berdaguer, Alcino Cunha, Claudia Necco, José Nuno Oliveira, Hugo Pacheco, and Paulo Silva.

References

[1] T. Alves, P. F. Silva, and Joost Visser. Contraint-aware schema transformation. Draft, 2008.

[2] T.L. Alves, P.F. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a VDM-SL to SQL conversion. In J. Fitzgerald, IJ. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *LNCS*, pages 399–414. Springer, 2005.

[3] S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the XML-to-relational mapping problem. In *WIDM '04: Proc. 6th annual ACM Int workshop on Web Information and Data Management*, pages 31–38. ACM Press, 2004.

[4] F. Atanassow and J. Jeuring. Inferring type isomorphisms generically. In *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125, pages 32–53, 2004.

[5] J.W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[6] D. Barbosa, J. Freire, and A.O. Mendelzon. Designing information-preserving mapping schemes for XML. In *VLDB'05: Proc. 31st Int. Conf. Very Large Data Bases*, pages 109–120. VLDB Endowment, 2005.

[7] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data conversion for XML and SQL. In Michael Hanus, editor, *PADL*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007.

[8] R. Bird. *Introduction to Functional Programming using Haskell*. International Series in Computer Science. Prentice Hall, 1998.

[9] P. Bohannon et al. LegoDB: Customizing relational storage for XML documents. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 1091–1094, 2002.

[10] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE '02: Proc. 18th Int. Conf. on Data Engineering*, pages 64–. IEEE Computer Society, 2002.

[11] E. Bouwers and R. Vis. Multidimensional software monitoring applied to erp. In Christos Makris and Joost Visser, editors, *Proceedings of the Second International Workshop on Software Quality and Maintainability*. To appear, 2008.

[12] Y. Chen et al. Constraints preserving schema mapping from XML to relations. In *Proc. 5th Int. Workshop Web and Databases (WebDB)*, pages 7–12, 2002.

[13] A. Cleve and J.-L. Hainaut. Co-transformations in database applications evolution. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*. Springer, 2006. To appear.

[14] A. Cleve, J. Henrard, and J.-L. Hainaut. Co-transformations in information system reengineering. *Electr. Notes Theor. Comput. Sci.*, 137(3):5–15, 2005.

[15] A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In J. Misra et al., editors, *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[16] A. Cunha and J. Sousa Pinto. Point-free program transformation. *Fundam. Inform.*, 66(4):315–352, 2005.

[17] A. Cunha, J. Sousa Pinto, and J. Proença. A framework for point-free program transformation. In A. Butterfield, C. Grelck, and F. Huch, editors, *Implementation and Application of Functional Languages, 17th Int. Workshop, IFL 2005, Revised Selected Papers*, volume 4015 of *LNCS*, pages 1–18. Springer, 2006.

[18] A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. *ENTCS*, 174(1):17–34, 2007. Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006).

[19] A. Cunha and J. Visser. Transformation of structure-shy programs: applied to XPath queries and strategic functions. In G. Ramalingam and Eelco Visser, editors, *PEPM*, pages 11–20. ACM, 2007.

[20] S.B. Davidson et al. Propagating XML constraints to relations. In *Proc. 19th Int. Conf. on Data Engineering*, pages 543–. IEEE Computer Society, 2003.

[21] A van Deursen and T. Kuipers. Source-based software risk assessment. In *ICSM '03: Proc. Int. Conference on Software Maintenance*, page 385, Washington, DC, USA, 2003. IEEE Computer Society.

[22] J.N. Foster et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proc. 32nd ACM symp. on Principles of Programming Languages*, pages 233–246. ACM Press, 2005.

[23] J. Gibbons. Calculating functional programs. In R. Backhouse et al., editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer, 2002.

[24] R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. 8th Int. Symp. on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2006.

[25] Z. Hu, T. Yokoyama, and M. Takeichi. Program optimizations and transformations in calculational form. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*. Springer, 2006.

[26] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In Mario Piattini and Manuel Serrano, editors, *Proceedings of the 1st International Workshop on Software Audit and Metrics, SAM 2004, In conjunction with ICEIS 2004, Porto, Portugal, April 2004*, pages 118–128. INSTICC Press, 2004.

[27] T. Kuipers, J. Visser, and G. de Vries. Monitoring the quality of outsourced software. In J. van Hillegersberg, F. Harmsen, C. Amrit, E. Geisberger, P. Keil, and M. Kuhrmann, editors, *Proceedings of the International Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007)*, Enschede, The Netherlands, 2007. Center for Telematics and Information Technology (CTIT).

[28] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.

[29] R. Lämmel. Transformations everywhere. *Sci. Comput. Program.*, 52:1–8, 2004. Guest editor's introduction to special issue on program transformation.

[30] R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th Int. Conf. on Reverse Engineering for Information Systems*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[31] R. Lämmel and E. Meijer. Mappings make data processing go 'round. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*. Springer, 2006.

[32] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[33] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Available at http://www.cwi.nl/~ralf, October 8 2003.

[34] R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! Technical Report cs.PL/0212048, arXiv, December 2002. An early version was published in the informal preproceedings IFL 2002.

[35] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer, January 2002.

[36] D. Lee and W. W. Chu. Cpi: Constraints-preserving inlining algorithm for mapping xml dtd to relational schema. *Data Knowl. Eng.*, 39(1):3–25, 2001.

[37] O. de Moor and G. Sittampalam. Generic program transformation. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Proc. 3rd Int. Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 116–149. Springer, 1999.

[38] C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[39] J.N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.

[40] J.N. Oliveira. Software reification using the SETS calculus. In T. Denvir et al., editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development*, pages 140–171. Springer, 1992.

[41] J.N. Oliveira. 'Fractal' Types: an Attempt to Generalize Hash Table Calculation. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.

[42] J.N. Oliveira. Data transformation by calculation. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*. Springer, 2008.

[43] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, Univ. of Pennsylvania, July 2004.

[44] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 357–361. Springer, May 2001.

[45] E. Visser and Z. Benaissa. A core language for rewriting. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[46] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. European Symposium on Programming*, volume 300 of *LNCS*, pages 344–358. Springer, 1988.