Type-safe Two-level Data Transformation

Alcino Cunha, José Nuno Oliveira, and Joost Visser

Departamento de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal

Abstract. A *two-level data transformation* consists of a type-level transformation of a data format coupled with value-level transformations of data instances corresponding to that format. Examples of two-level data transformations include XML schema evolution coupled with document migration, and data mappings used for interoperability and persistence.
We provide a formal treatment of two-level data transformations that is *type-safe* in the sense that the well-formedness of the value-level transformations with respect to the type-level transformation is guarded by a strong type system. We rely on various techniques for generic functional programming to implement the formalization in Haskell.
The formalization addresses various two-level transformation scenarios, covering fully automated as well as user-driven transformations, and allowing transformations that are information-preserving or not. In each case, two-level transformations are disciplined by one-step transformation rules and type-level transformations induce value-level transformations. We demonstrate an example hierarchical-relational mapping and subsequent migration of relational data induced by hierarchical format evolution.
Keywords: Two-level transformation, Program calculation, Refinement calculus, Strategic term rewriting, Generalized abstract datatypes, Generic programming, Coupled transformation, Format evolution, Data mappings.

1 Introduction

Changes in data types call for corresponding changes in data values. 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. Or, 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* [12] 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 *data mappings* [13]. 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.

Both format evolution and data mappings are instances of what we call *two-level* transformations, where a type-level transformation (of the data type) determines or constrains value-level transformations (of the data instances).

When developing a two-level data transformation system, a challenge arises regarding the degree of type-safety that can be achieved. Two approaches to deal with this challenge are common: (i) define a universal representation in which any data can be encoded, or (ii) merge the input, output, and intermediate types into a single union type. Transformation steps can then be implemented as type-preserving transformations on either the universal representation or the union type. The first approach is simple, but practically abandons all typing. The second approach maintains a certain degree of typing at the cost of the effort of defining the union type. In either case, defensive programming and extensive testing are required to ensure that the transformation is well-behaved.

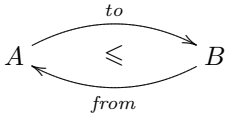
In this paper, we show how two-level data transformation systems can be developed in a type-safe manner. In this approach, value-level transformations are statically checked to be well-typed with respect to the type-level transformations to which they are associated, and well-typed composition of type-level transformation steps induces well-typed compositions of value-level transformation steps. Unlike the mentioned approaches, our solution does not compromise precise typing of intermediate values.

In Section 2 we present a formalization of two-level transformations based on a theory of data refinement. Apart from some general laws for any transformation system, we present two groups of laws that cater for data mapping and format evolution scenarios, respectively. In Section 3, we implement our formalization in the functional programming language Haskell. We rely on various techniques for data-generic functional programming with strong mathematical foundations. In Section 4 we return to the data mapping and format evolution scenarios and demonstrate them by example. Section 5 discusses related work, and Section 6 discusses future extensions and applications.

2 Data refinement calculus

The theory which underlies our approach to two-level transformations finds its roots in a data refinement calculus which originated in [17, 19, 20]. This calculus has been applied to relational database design [21] reverse engineering of legacy databases [18].

Abstraction and representation Two-level transformation steps are modeled by inequations between datatypes and accompanying functions of the following form:



Here, the inequation $A \leqslant B$ models a type-level transformation where datatype A gets transformed into datatype B, and abbreviates the fact that there is an injective, total relation to (the *representation relation*) and a surjective, possibly partial function $from$ (the *abstraction relation*) such that $from \cdot to = id_A$, where id_A is the identity function on datatype A. Though in general to can be a relation, it is usually a function as well,

and functions to and $from$ model the value-level transformations that accompany the type-level transformation.

Since the equality of two relations is a bi-inclusion we have two readings of the above equation: $id_A \subseteq from \cdot to$, which ensures that every inhabitant of datatype A has a representation in datatype B; and $from \cdot to \subseteq id_A$, which prevents "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.

In a situation where the abstraction is also a representation and vice-versa we have an isomorphism $A \cong B$, a special case of $\leqslant$-law which works in both directions.

Thus, type-level transformations are not arbitrary. They arise from the existence of value-level transformations 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.

Below we will present a series of general laws for composition of two-level transformations that form a framework for any two-level transformation system. This framework can be instantiated with sets of problem-specific two-level transformations steps to obtain a two-level transformation system for a specific purpose. We will show sets of rules for data mapping and for format evolution.

Sequential and structural composition laws Individual two-level transformation steps can be chained by sequentially composing abstractions and representations:

$$\text{if } A \underset{from}{\overset{to}{\leqslant}} B \text{ and } B \underset{from'}{\overset{to'}{\leqslant}} C \text{ then } A \underset{from \cdot from'}{\overset{to' \cdot to}{\leqslant}} C$$

Such transitivity, together with the fact that any datatype can be transformed to itself (reflexivity), witnessed by identity value-level transformations ($from = to = id$), means that $\leqslant$ is a preorder.

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. The following law captures such upward propagation:

$$\text{if } A \underset{from}{\overset{to}{\leqslant}} B \text{ then } \mathsf{F}\,A \underset{\mathsf{F}\,from}{\overset{\mathsf{F}\,to}{\leqslant}} \mathsf{F}\,B \tag{1}$$

Here F is a *functor* that models the *context* in which a transformation step is performed. Recall that a functor F from categories 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 and Y are data types, and the morphism f and g 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.

Rules for data mapping and format evolution In [1] we presented a set of two-level transformation rules that can be combined with the general laws presented above into a calculator that automatically converts a hierarchic, possibly recursive data structure to a flat, relational representation. These rules are summarized in Figure 1. They are designed for step-wise 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. 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. Figure 2 shows a set of two-level transformation rules that cater for these scenarios. These rules formalize co-evolution of XML documents and their DTDs as discussed by Lämmel *et al* [12]. Note that the rule for adding a field assumes that a new value x 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.

3 Two-level Transformations in Haskell

Our solution to modeling two-level data transformations in Haskell consists of four components. Firstly, we will define a *datatype* to represent the types that are subject to rewriting. Secondly, we will extend that datatype with a *view* constructor that can encapsulate the result of a type-level rewrite step together with the corresponding value-level functions. Such encapsulation will allow type-changing rewrite steps to masquerade as type-preserving ones. Thirdly, we define combinators that allow us to fuse local, single-step transformations into a single global transformation. Finally, we provide functions to release these transformations out of their type-preserving shell, thus obtaining the corresponding type-changing, bi-directional data migration functions.

We will illustrate the Haskell encoding with this example transformation sequence:

$$(A + B)^\star \quad \leqslant \quad \mathbb{N} \rightharpoonup (A + B) \quad \leqslant \quad (\mathbb{N} \rightharpoonup A) \times (\mathbb{N} \rightharpoonup B)$$

This is a valid sequence according to rules (2) and (5) presented in Figure 1.

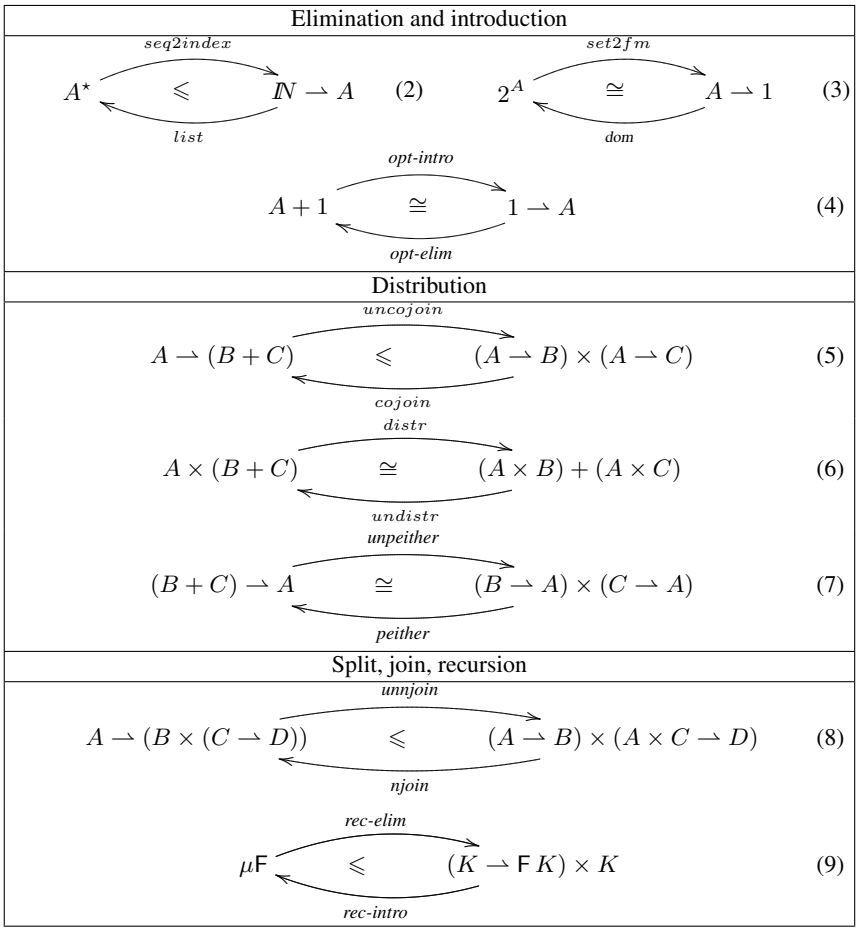


Fig. 1. One-step rules for a two-level transformation system that maps hierarchic, recursive data structures to flat relational mappings. Only the names of type-level functions are given. More details can be found elsewhere [20, 21, 1].

Representation of types Assume that $\mathbb{N}$ will be represented by Haskell type Int, $A \rightharpoonup B$ by the data type $Map\ a\ b$ (finite maps from standard library module $Data.Map$), and $A + B$ by $\mathbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$. We would like now to define a rewriting strategy that converts type $[Either\ a\ b]$ to type $(Map\ Int\ a, Map\ Int\ b)$, building at the same time a function of type $[Either\ a\ b] \to (Map\ Int\ a, Map\ Int\ b)$ to perform the data migration.

Both type-level and value-level components of this transformation will be performed on the Haskell term-level, and to this end we need to represent types by terms. Rather than resorting to an untyped universal representation of types, we define the following *type-safe* representation, adapted from [8]:

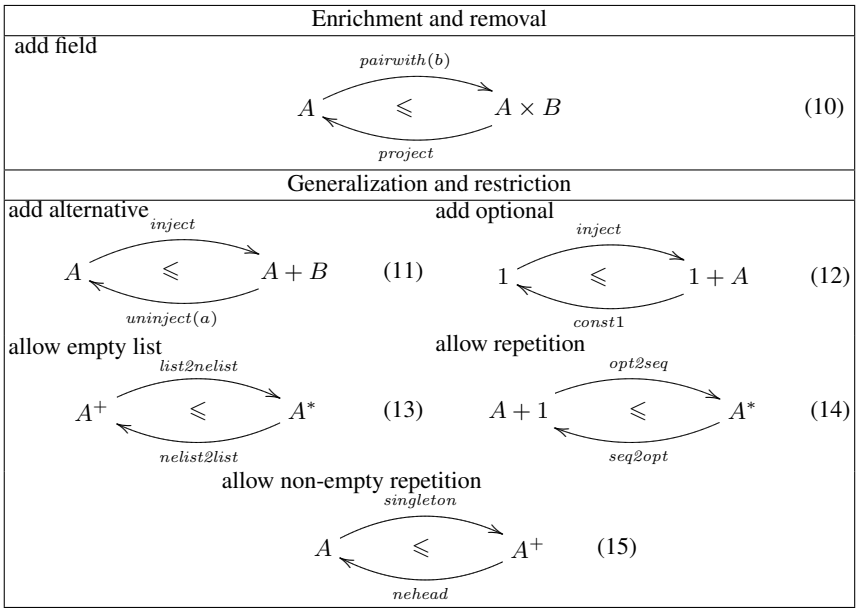


Fig. 2. One-step rules for a two-level transformation system for format evolution. These rules formalize the discussion of XML format evolution of Lämmel *et al* [12].

.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Type}\ a\ \textbf{where}\\
&\quad \mathit{Int} :: \mathit{Type}\ \mathit{Int}\\
&\quad \mathit{String} :: \mathit{Type}\ \mathit{String}\\
&\quad \mathit{One} :: \mathit{Type}\ ()\\
&\quad \mathit{List} :: \mathit{Type}\ a \to \mathit{Type}\ [a]\\
&\quad \mathit{Set} :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Set}\ a)\\
&\quad \mathit{Map} :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (\mathit{Map}\ a\ b)\\
&\quad \mathit{Either} :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (\mathit{Either}\ a\ b)\\
&\quad \mathit{Prod} :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b)\\
&\quad \mathit{Tag} :: \mathit{String} \to \mathit{Type}\ a \to \mathit{Type}\ a
\end{aligned}
$$

This definition ensures that $\mathit{Type}\ a$ can only be inhabited by representations of type a. For example, the pre-defined type Int of integers is represented by the data constructor Int of type $\mathit{Type}\ \mathit{Int}$, and the type $[\mathit{Int}]$ of lists of integers is represented by the value $\mathit{List}\ \mathit{Int}$ of type $\mathit{Type}\ [\mathit{Int}]$. The Tag constructor allows us to tag types with names.

The datatype Type is an example of a *generalized algebraic data type* (GADT) [22], a recent Haskell extension that allows to assign more precise types to data constructors by restricting the variables of the datatype in the constructors' result types. Note also that the argument a of the Type datatype is a so-called phantom type [7], since no value of type a needs to be provided when building a value of type $\mathit{Type}\ a$. Using a phantom type we can represent a type at the term level without building any term of that type.

Going back to our example, our intended transformation must convert type representation $List\ (Either\ a\ b)$ into $Prod\ (Map\ Int\ a)\ (Map\ Int\ b)$.

Encapsulation of type-changing rewrites Whenever single-step rewrite rules are intended to be applied repeatedly and at arbitrary depths inside terms, it is essential that they are type-preserving [3, 16]. Otherwise, ill-typed terms would be created as intermediate or even as final results. But two-level data transformations are *type-changing* in general. To resolve this tension, type-changing transformations will *masquerade* as type-preserving ones.

The solution for masquerading type-changing transformation steps as type-preserving ones is simple, but ingenious. When rewriting a type representation, we do not replace it, but *augment* it with the target type and with a pair of value-level functions that allow conversion between values of the source and target type.

$\mathbf{data}\ Rep\ a\ b = Rep\{\, to :: a \to b, from :: b \to a \,\}$
$\mathbf{data}\ View\ a\ \mathbf{where}$
$\quad View :: Rep\ a\ b \to Type\ b \to View\ (Type\ a)$
$showType :: View\ a \to String$

The $View$ constructor expresses that a type a can be represented as a type b, denoted as $Rep\ a\ b$, if there are functions $to :: a \to b$ and $from :: b \to a$ that allow data conversion between one and the other. Note that only the source type a escapes from the $View$ constructor, while the target type b remains encapsulated — it is implicitly existentially quantified[1]. The function $showType$ just allows us to obtain a string representation of the target type.

Now the type of type-preserving transformation steps is defined as follows[2]:

$\mathbf{type}\ Rule = \forall a.\, Type\ a \to Maybe\ (View\ (Type\ a))$

Note that the explicit quantification of the type variable a will allow us to apply the same rewrite step of type $Rule$ to various different subterms of a given type representation, e.g. to both Int and $String$ in $Prod\ Int\ String$. Thus, when rewriting a type representation we will not change its type, but just signal that it can also be *viewed* as a different type.

We can now start encoding some transformation rules of our data refinement calculus. For instance, given value-level functions (see Figure 1):

$list :: Map\ Int\ a \to [a]$
$seq2index :: [a] \to Map\ Int\ a$
$uncojoin :: Map\ a\ (Either\ b\ c) \to (Map\ a\ b, Map\ a\ c)$
$cojoin :: (Map\ a\ b, Map\ a\ c) \to Map\ a\ (Either\ b\ c)$

the rule (2) that convert a list into a map, and the rule (5) that converts a map of sums into a pair of maps can be defined as follows:

$listmap :: Rule$
$listmap\ (List\ a) = Just\ (View\ rep\ (Map\ Int\ a))$
$\quad \mathbf{where}\ rep = Rep\{\, to = seq2index, from = list \,\}$
$listmap\ \_ = Nothing$

[1] $View$ is somewhat similar to the folklore $\mathbf{data}\ Dynamic = \forall a.Dyn\ a\ (Type\ a)$, which pairs a *value* of an existentially quantified type with its representation.

[2] We model partiality with $\mathbf{data}\ Maybe\ a = Nothing \mid Just\ a$.

$mapsum :: Rule$
$mapsum\ (Map\ a\ (Either\ b\ c)) = Just\ (View\ rep\ (Prod\ (Map\ a\ b)\ (Map\ a\ c)))$
  where $rep = Rep\{\, to = uncojoin, from = cojoin \,\}$
$mapsum\ \_ = Nothing$

The remaining rules of Figure 1 can be implemented in a similar way.

The only rule that poses a significant technical challenge is rule (9) for recursion elimination. We will only present an outline of our solution (more details in [5]), which uses the Haskell class mechanism and monadic programming. Firstly, we represent the fixpoint operator μ as follows:

newtype $Mu\ f = In\{\, out :: f\ (Mu\ f) \,\}$
data $Type\ a$ **where**
  ...
  $Mu :: Dist\ f \Rightarrow (\forall a.Type\ a \rightarrow Type\ (f\ a)) \rightarrow Type\ (Mu\ f)$

Here f is a functor[3], and the class constraint $Dist\ f$ expresses that we require functors to commute with monads. Rule (9) can now be implemented:

type $Table\ f = (Map\ Int\ (f\ Int), Int)$

$fixastable :: Rule$
$fixastable\ (Mu\ f) = Just\ (View\ rep\ (Prod\ (Map\ Int\ (f\ Int))\ Int))$
  where $rep = Rep\{\, to = recelim, from = recintro \,\}$
$fixastable\ \_ = Nothing$

$recelim :: Dist\ f \Rightarrow Mu\ f \rightarrow Table\ f$
$recintro :: Functor\ f \Rightarrow Table\ f \rightarrow Mu\ f$

Internally, *recelim* incrementally builds a table while traversing over a recursive data instance. It uses monadic code to thread the growing table through the recursion pattern.

Strategy combinators for two-level transformation To build a full two-level transformation system, we must be able to apply two-level transformation steps sequentially, alternatively, repetitively, and at arbitrary levels inside type representations. For this we introduce strategy combinators for two-level term rewriting. They are similar to strategy combinators for ordinary single-level term rewriting [16], except that they simultaneously fuse the type-level steps and the value-level steps. As we will see, the joint effect of two-level strategy combinators is to combine the view introduced locally by individual steps into a single view around the root of the representation of the target type.

Let us begin by supplying combinators for identity, sequential composition, and structural composition of pairs of value-level functions:

$idrep :: Rep\ a\ a$
$idrep = Rep\{\, to = id, from = id \,\}$

$comprep :: Rep\ a\ b \rightarrow Rep\ b\ c \rightarrow Rep\ a\ c$
$comprep\ f\ g = Rep\{from = (from\ f).(from\ g), to = (to\ g).(to\ f)\}$

$maprep :: Functor\ f \Rightarrow Rep\ a\ b \rightarrow Rep\ (f\ a)\ (f\ b)$
$maprep\ r = Rep\{\, to = fmap\ (to\ r), from = fmap\ (from\ r) \,\}$

[3] Functors are instances of: **class** $Functor\ f$ **where** $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$.

Using these combinators for pairs of value-level functions, we can define the two-level combinators. Sequential composition is defined as follows[4]:

$$
\begin{aligned}
&(\triangleright) :: \mathit{Rule} \to \mathit{Rule} \to \mathit{Rule}\\
&(f \triangleright g)\ a = \mathbf{do}\ \mathit{View}\ r\ b \leftarrow f\ a\\
&\qquad\qquad\quad\ \ \mathit{View}\ s\ c \leftarrow g\ b\\
&\qquad\qquad\quad\ \ \mathit{return}\ (\mathit{View}\ (\mathit{comprep}\ r\ s)\ c)
\end{aligned}
$$

We further define combinators for left-biased choice ($f \oslash g$ tries f, and if it fails, tries g instead), a "do nothing" combinator, and repetitive application of a rule until it fails[5]:

$$
\begin{aligned}
&(\oslash) :: \mathit{Rule} \to \mathit{Rule} \to \mathit{Rule}\\
&(f \oslash g)\ x = f\ x\ `\mathit{mplus}`\ g\ x\\
&\mathit{nop} :: \mathit{Rule}\\
&\mathit{nop}\ x = \mathit{Just}\ (\mathit{View}\ \mathit{idrep}\ x)\\
&\mathit{many} :: \mathit{Rule} \to \mathit{Rule}\\
&\mathit{many}\ r = (r \triangleright \mathit{many}\ r) \oslash \mathit{nop}
\end{aligned}
$$

These combinators suffice for combining transformations at a single level inside a term.

Two-level combinators that descend into terms are more challenging to define. They rely on the functorial structure of type representations and use *maprep* defined above to push pairs of value-level functions up through functors. An example is the *once* combinator that applies a given rule exactly once somewhere inside a type representation:

$$
\begin{aligned}
&\mathit{once} :: \mathit{Rule} \to \mathit{Rule}\\
&\mathit{once}\ r\ \mathit{Int} = r\ \mathit{Int}\\
&\mathit{once}\ r\ (\mathit{List}\ a) = r\ (\mathit{List}\ a)\ `\mathit{mplus}`\\
&\qquad\qquad\qquad\quad (\mathbf{do}\ \mathit{View}\ s\ b \leftarrow \mathit{once}\ r\ a\\
&\qquad\qquad\qquad\qquad\ \ \mathit{return}\ (\mathit{View}\ (\mathit{maprep}\ s)\ (\mathit{List}\ b)))\\
&\ldots
\end{aligned}
$$

Note that *once* performs a pre-order which stops as soon as its argument rule is applied successfully. Other strategy combinators can be defined similarly.

It is now possible to combine individual two-level transformation rules into the following rewrite system:

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Rule}\\
&\mathit{flatten} = \mathit{many}\ (\mathit{once}\ (\mathit{listmap} \oslash \mathit{mapsum} \oslash \ldots))
\end{aligned}
$$

which can be successfully applied to our running example, as the following interaction with the Haskell interpreter shows:

$$
\begin{aligned}
&> \mathit{flatten}\ (\mathit{List}\ (\mathit{Either}\ \mathit{Int}\ \mathit{Bool}))\\
&\mathit{Just}\ (\mathit{View}\ (\mathit{Rep} <to> <from>)\ (\mathit{Prod}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ (\mathit{Map}\ \mathit{Int}\ \mathit{Bool})))
\end{aligned}
$$

Note that the result shown by the interpreter is a *String* representation of a value of type $\mathit{Maybe}\ (\mathit{View}\ (\mathit{Type}\ (\mathit{List}\ (\mathit{Either}\ \mathit{Int}\ \mathit{Bool}))))$. Placeholders $<to>$ and $<from>$ are shown in place of function objects, which are not printable. Thus, the existentially quantified result type of the transformation is *not* available statically, though its string representation is available dynamically.

---

[4] For composing partial functions we use the monadic **do**-notation, exploiting the fact that *Maybe* is an instance of a Monad [23].

[5] $\mathit{mplus} :: \mathit{Maybe}\ a \to \mathit{Maybe}\ a \to \mathit{Maybe}\ a$ returns the first argument if it is constructed with *Just* or the second argument otherwise.

Unleashing composed data migration functions So far, we have developed techniques to implement rewrite strategies on types, building at the same time functions for data migration between the original and the resulting type. Unfortunately, it is still not possible to use such functions with the machinery developed so far. The problem is that the target type is encapsulated as an existentially quantified type variable inside the $View$ constructor. This was necessary to make the type-changing transformation masquerade as a type-preserving one.

We can access the hidden data migration functions in two ways. If we happen to know what the target type is, we can simply take them out as follows:

$$forth :: View\ (Type\ a) \to Type\ b \to a \to Maybe\ b$$
$$forth\ (View\ rep\ tb')\ tb\ a = \mathbf{do}\ \{Eq \leftarrow teq\ tb\ tb'; return\ (to\ rep\ a)\}$$
$$back :: View\ (Type\ a) \to Type\ b \to b \to Maybe\ a$$
$$back\ (View\ rep\ tb')\ tb\ b = \mathbf{do}\ \{Eq \leftarrow teq\ tb\ tb'; return\ (from\ rep\ b)\}$$

Again, GADTs are of great help in defining a data type that provides evidence to the type-checker that two types are equal (*cf* [22]):

$$\mathbf{data}\ Equal\ a\ b\ \mathbf{where}$$
$$Eq :: Equal\ a\ a$$

Notice that a value Eq of type $Equal\ a\ b$ is a witness that types a and b are indeed equal. A function that provides such a witness based on the structural equality of type representations is then fairly easy to implement.

$$teq :: Type\ a \to Type\ b \to Maybe\ (Equal\ a\ b)$$
$$teq\ Int\ Int = return\ Eq$$
$$teq\ (List\ a)\ (List\ b) = \mathbf{do}\ \{Eq \leftarrow teq\ a\ b; return\ Eq\}$$
$$...$$

In the format evolution scenario, where a transformation is specified manually at system design or maintenance time, the static availability of the target type is realistic.

But in general, and in particular in the data mapping scenario, we should expect the target type to be statically unknown, and only available dynamically. In that case we can access the result type via a *staged* approach. In the first stage, we apply the transformation to obtain its result type dynamically, using $showType$, in the form of its string representation. In the second stage, that string representation is incorporated in our source code, and gets parsed and compiled and becomes statically available after all. Below, we will use such staging in Haskell interpreter sessions.

4 Application Scenarios

We demonstrate two-level transformations with two small, but representative examples.

Evolution of a music album format Suppose rudimentary music album information is kept in XML files that conform to the following XML Schema fragment:

```
<element name="Album" type="AlbumType"/>
<complexType name="AlbumType">
  <attribute name="ASIN" type="string"/>
  <attribute name="Title" type="string"/>
  <attribute name="Artist" type="string"/>
```

```
  <attribute name="Format"><simpleType base="string">
    <enumeration value="LP"/><enumeration value="CD"/>
  </simpleType></attribute>
</complexType>
```

In a first evolution step, we want to allow an additional media type beyond CDs and LPs, namely DVDs. In a second step, we want to add a list of track names to the format.

We can represent the album schema and an example album document as follows:

$$
\begin{aligned}
&albumFormat = Tag\ \texttt{"Album"}\ (\\
&\quad Prod\ (Tag\ \texttt{"ASIN"}\ String)\ (\\
&\quad Prod\ (Tag\ \texttt{"Title"}\ String)\ (\\
&\quad Prod\ (Tag\ \texttt{"Artist"}\ String)\\
&\qquad (\quad Tag\ \texttt{"Format"}\ (Either\ (Tag\ \texttt{"LP"}\ One)\ (Tag\ \texttt{"CD"}\ One))))))\\
&lp = (\texttt{"B000002UB2"}, (\texttt{"Abbey Road"}, (\texttt{"The Beatles"}, Left\ ())))
\end{aligned}
$$

With a generic show function $gshow :: Type\ a \to a \to String$, taking the format as first argument, we can print the album with tag information included:

$$
\begin{aligned}
&> putStrLn\ \$\ gshow\ albumFormat\ lp\\
&Album = (ASIN = \texttt{"B000002UB2"}, (\ Title = \texttt{"Abbey Road"}, (\\
&\qquad\qquad Artist = \texttt{"The Beatles"}, Format = Left\ (LP = ()))))
\end{aligned}
$$

This function also ensures us that lp is actually well-typed with respect to $albumFormat$.

To enable evolution, we define the following additional combinators for adding alternatives, adding fields, and triggering rules inside tagged types:

$$
\begin{aligned}
&addalt :: Type\ b \to Rule\\
&addalt\ b\ a = Just\ (View\ rep\ (Either\ a\ b))\\
&\quad \mathbf{where}\ rep = Rep\{\, to = Left, from = \lambda(Left\ x) \to x\,\}\\
&\mathbf{type}\ Query\ b = \forall a.\, Type\ a \to a \to b\\
&addfield :: Type\ b \to Query\ b \to Rule\\
&addfield\ b\ f\ a = Just\ (View\ rep\ (Prod\ a\ b))\\
&\quad \mathbf{where}\ rep = Rep\{\, to = \lambda y \to (y, f\ a\ y), from = fst\,\}\\
&inside :: String \to Rule \to Rule\\
&inside\ n\ r\ (Tag\ m\ a)\\
&\quad \mid n \equiv m = \mathbf{do}\ \{\ View\ r\ b \leftarrow r\ a; return\ (View\ r\ (Tag\ m\ b))\}\\
&inside\ \_\ \_\ \_ = Nothing
\end{aligned}
$$

Note that the *addalt* combinator inserts and removes *Left* constructors on the data level. The *addfield* combinator takes as additional argument a query that gets applied to the argument of *to* to come up with a value of type b, which gets inserted into the new field.

With these combinators in place, we can specify the desired evolution steps:

$$
\begin{aligned}
&addDvd = once\ (inside\ \texttt{"Format"}\ (addalt\ (Tag\ \texttt{"DVD"}\ One)))\\
&addTracks = once\ (inside\ \texttt{"Album"}\ (addfield\ (List\ (Tag\ \texttt{"Title"}\ String))\ q))\\
&\quad \mathbf{where}\ q :: Query\ [String]\\
&\qquad\qquad q\ (Prod\ (Tag\ \texttt{"ASIN"}\ String)\ \_)\ (asin, \_) = ...\\
&\qquad\qquad q\ \_\ \_ = [\,]
\end{aligned}
$$

The query q uses the album identifier to lookup from another data source, e.g. via a query over the internet[6]. Subsequently, we can run the type-level transformation, and print the result type:

[6] For such a side effect, an impure function is needed.

```
> let (Just vw) = (addTracks ▷ addDvd) albumFormat
> showType vw
Tag "Album" (Prod (Prod (
  Tag "ASIN" String) (Prod (
  Tag "Title" String)(Prod (
  Tag "Artist" String)(
  Tag "Format" (Either (Either (
    Tag "LP" One)(Tag "CD" One)) (Tag "DVD" One))))) (
  List (Tag "Title" String))))
```

The value-level transformation is executed in forward direction as follows:

```
> let targetFormat = Tag "Album" (Prod (Prod (...
> let (Just targetAlbum) = forth vw targetFormat lp
> putStrLn $ gshow targetFormat targetAlbum
Album = ((ASIN = "B000002UB2",( Title = "Abbey Road",(
            Artist = "The Beatles", Format = Left (Left (LP = ()))))),
            [Title = "Come Together",...]))
```

In backward direction, we can recover the original LP:

```
> let (Just originalAlbum) = back vw targetFormat targetAlbum
> lp ≡ originalAlbum
True
```

Any attempt to execute the backward value-level transformation on a DVD, i.e. on an album that uses a newly added alternative, will fail.

Mapping album data to relational tables We pursue our music album example to demonstrate data mappings. In this case, we are interested in mapping the hierarchical album format, which models the XML schema, onto a flat schema, which could be stored in a relational database. This data mapping is performed by the *flatten* transformation defined above, but before applying it, we need to prepare the format in two respects. Firstly, we want the enumeration type for formats to be stored as integers. Secondly, we need to remove the tags from our datatype, since the *flatten* transformation assumes their absence. For brevity we omit the definitions of *enum2int* and *removetags*; they are easy to define.

Our relational mapping for music albums is now defined and applied to both our original and our evolved formats as follows:

```
> let toRDB = once enum2int ▷ removetags ▷ flatten
> let (Just vw0) = toRDB (List albumFormat)
> showType vw0
Map Int (Prod (Prod (Prod String String) String) Int)
> let (Just vw1) = toRDB (List targetFormat)
> showType vw1
Prod (Map Int (Prod (Prod (Prod String String) String) Int)) (
      Map (Prod Int Int) String)
```

Note that we apply the transformations to the type of *lists* of albums – we want to store a collection of them. The original format is mapped to a single table, which maps album numbers to 4-tuples of ASIN, title, name, and an integer that represents the format. The

target format is mapped to *two* tables, where the extra table maps compound keys of album and track numbers to track names.

Let's store our first two albums in relational form:

$> \textbf{let}\ \mathit{dbs0} = \mathit{Map}\ \mathit{Int}\ (\mathit{Prod}\ (\mathit{Prod}\ (\mathit{Prod}\ \mathit{String}\ \mathit{String})\ \mathit{String})\ \mathit{Int})$
$> \textbf{let}\ (\mathit{Just}\ \mathit{db}) = \mathit{forth}\ \mathit{vw0}\ \mathit{dbs0}\ [\mathit{lp}, \mathit{cd}]$
$> \mathit{db}$
$\{0 := (((\texttt{"B000002UB2"}, \texttt{"Abbey Road"}), \texttt{"The Beatles"}), 0),$
$\ 1 := (((\texttt{"B000002HCO"}, \texttt{"Debut"}), \texttt{"Bjork"}), 1)\}$

As expected, two records are produced with different keys. The last 1 indicates that the second album is a CD.

The reverse value-level transformation restores flattened data to hierarchical form. By composing the value-level transformations induced by data mappings with those induced by format evolution, we can migrate from and old database to a new one[7]:

$> \textbf{let}\ (\mathit{Just}\ \mathit{lvw}) = (\mathit{addTracks} \triangleright \mathit{addDvd})\ (\mathit{List}\ \mathit{albumFormat})$
$> \textbf{let}\ \mathit{dbs1} = \mathit{Prod}\ (\mathit{Map}\ ...)\ (\mathit{Map}\ (\mathit{Prod}\ \mathit{Int}\ \mathit{Int})\ \mathit{String})$
$> \textbf{let}\ (\mathit{Just}\ x) = \mathit{back}\ \mathit{vw0}\ \mathit{dbs0}\ \mathit{db}$
$> \textbf{let}\ (\mathit{Just}\ y) = \mathit{forth}\ \mathit{lvw}\ (\mathit{List}\ \mathit{targetFormat})\ x$
$> \textbf{let}\ (\mathit{Just}\ z) = \mathit{forth}\ \mathit{vw1}\ \mathit{dbs1}\ y$
$> z$
$(\{0 := (((\texttt{"B000002UB2"}, \texttt{"Abbey Road"}), \texttt{"The Beatles"}), 0),$
$\ 1 := (((\texttt{"B000002HCO"}, \texttt{"Debut"}), \texttt{"Bjork"}), 1)\},$
$\{(0, 0) := \texttt{"Come Together"}, ...\})$

In this simple example, the migration amounts to adding a single table with track names retrieved from another data source, but in general, the induced value-level data transformations can augment, reorganize, and discard relational data in customizable ways.

5 Related work

Software transformation Lämmel *et al* [12] propose a systematic approach to evolution of XML-based formats, where DTDs are transformed in a well-defined, stepwise 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."

[7] Such compositions of *to* and *from* of different refinements are *representation changers* [9].

We have taken up this challenge by showing that formalization and implementation are feasible. A fully worked out application of our approach in the XML domain can now be attempted.

Lämmel *et al* [13] have identified data mappings as a challenging problem that permeates 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.

Generic functional programming Type-safe combinators for strategic rewriting were introduced by Lämmel *et al* in [16], after which several simplified and generalized approaches were proposed [15, 14, 8]. 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 [2]. 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. Recursive datatypes are not addressed. 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. We stay within Haskell rather than resorting to Generic Haskell, and avoid the use of a universal representation.

Bi-directional programming Foster *et al* tackle the *view-update problem* for databases with *lenses*: combinators for bi-directional programming [6]. 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.

On the level of problem statement, a basic difference exists between lenses and two-level transformations (or data 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 Future work

We have provided a type-safe formalization of two-level data transformations, and we have shown its implementation in Haskell, using various generic programming techniques. We discuss some current limitations and future efforts to remove them.

Co-transformation 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 [4]. Currently, our approach formalizes the use of *wrappers* for this purpose, where the application program gets pre- and post-fixed by induced value-level data migration functions. We intend to extend our approach to formalize induction of actual application program transformations, without resorting to wrappers.

Coupled transformations Lämmel [11, 10] identifies *coupled transformation*, where 'nets' of software artifacts are transformed simultaneously, as an important research challenge. Format evolution, data-mapping, and co-transformations are instances where two or three transformations are coupled. We believe that our formalization provides an important step towards a better grasp of this challenge.

Bi-directional programming Among the outstanding problems in bi-directional programming are decidable type checking and type inference, automatic optimization of bi-directional programs, lens inference from given abstract and concrete formats, and support for proving lens properties. We aim to leverage the techniques we used for two-level transformations for these purposes.

Front-ends Work is underway to develop front-ends that convert between our type-representations and formats such as XML Schemas, SQL database schemas, and nominal user-defined Haskell types.

Acknowledgments Thanks to Bruno Oliveira for inspiring discussions on GADTs. The work reported in this paper was supported by *Fundação para a Ciência e a Tecnologia*, grant number POSI/ICHS/44304/2002.

References

1. 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, *Proc. Int. Symposium of Formal Methods Europe*, volume 3582 of *LNCS*, pages 399–414. Springer, 2005.
2. F. Atanassow and J. Jeuring. Inferring type isomorphisms generically. In D. Kozen, editor, *Proc. 7th Int. Conference on Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 32–53. Springer, 2004.
3. M.v.d Brand, P. Klint, and J. Vinju. Term rewriting with type-safe traversal functions. *ENTCS*, 70(6), 2002.

4. A. Cleve, J. Henrard, and J.-L. Hainaut. Co-transformations in information system reengineering. *ENTCS*, 137(3):5–15, 2005.
5. A. Cunha, J.N. Oliveira, and J.Visser. Type-safe two-level data transformation – with derecursivation and dynamic typing. Technical Report DI-PURe-06.03.01, Univ. Minho, 2006.
6. J.N. Foster et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM Press, 2005.
7. R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave, 2003.
8. R. Hinze, A. Löh, and B. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. 8th Int. Symposium on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.
9. G. Hutton and E. Meijer. Back to Basics: Deriving Representation Changers Functionally. *Journal of Functional Programming*, 6(1):181–188, January 1996.
10. R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.
11. R. Lämmel. Transformations everywhere. *Sci. Comput. Program.*, 52:1–8, 2004. Guest editor's introduction to special issue on program transformation.
12. 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.
13. R. Lämmel and E. Meijer. Mappings make data processing go 'round. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Proc. Int. Summer School on Generative and Transformational Techniques in Software Engineering, Braga, Portugal, July 4–8, 2005*, LNCS. Springer, 2006. To appear.
14. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37. ACM Press, March 2003.
15. 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.
16. R. Lämmel and J. Visser. Typed combinators for generic traversal. In *Proc. Practical Aspects of Declarative Programming*, volume 2257 of *LNCS*, pages 137–154. Springer, January 2002.
17. C. Morgan and P.H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.
18. F.L. Neves, J.C. Silva, and J.N. Oliveira. Converting Informal Meta-data to VDM-SL: A Reverse Calculation Approach . In *VDM in Practice! A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France*, September 1999.
19. J.N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.
20. J.N. Oliveira. Software reification using the SETS calculus. In T. Denvir, C.B. Jones, and R.C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development*, pages 140–171. Springer, 1992.
21. J.N. Oliveira. Calculate databases with 'simplicity', September 2004. Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK.
22. 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.
23. P. Wadler. The essence of functional programming. In *Proc. 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992.