Type-safe Two-level Data Transformation

with derecursivation and dynamic typing

Alcino Cunha, José Nuno Oliveira, and Joost Visser

Techn. Report DI-PURe-06.03.01
2006, March

PURE
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-06.03.01
Type-safe Two-level Data Transformation – with derecursivation and dynamic typing by Alcino Cunha, José Nuno Oliveira, and Joost Visser

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 operationalize 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.

An appendix is included with additional detail on the representation and handling of recursive types, and on the use of dynamic types in value-level function application.

Type-safe Two-level Data Transformation
with derecursivation and dynamic typing

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 operationalize 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. Conceptually, the input type, intermediate types, and target types of a two-level transformation are all distinct. As a consequence, it seems infeasible to assign precise types to the atomic transformation steps that must be repeatedly applied to different data elements at various stages of the transformation. 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.

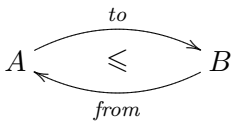
In Section 2 we present a formalization of two-level transformations based on a theory of data refinement. Apart from some general laws for any transformation system, we present two groups of laws that cater for data mapping and format evolution scenarios, respectively. In Section 3, we operationalize 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 [19, 20]. This calculus has been applied to relational database design [22, 23] reverse engineering of legacy databases [18].

2.1 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

$$from \cdot to = id_A \tag{1}$$

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 equation (1): $id_A \subseteq from \cdot to$, 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.

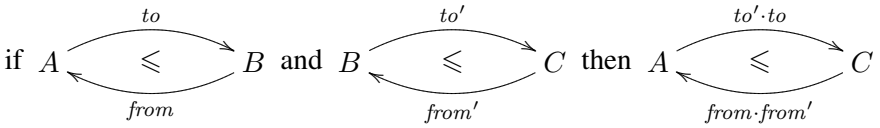
Thus, type-level transformations are not arbitrary. They arise from the existence of value-level transformations whose properties preclude data mixup. When applied left-to-right, an inequation $A \leqslant B$ will preserve or enrich information content, while applied in the right-to-left direction it 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.

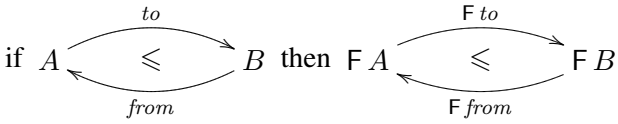
2.2 Sequential and structural composition laws

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



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:



(2)

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.

2.3 Rules for data mapping and format evolution

In [2] 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.

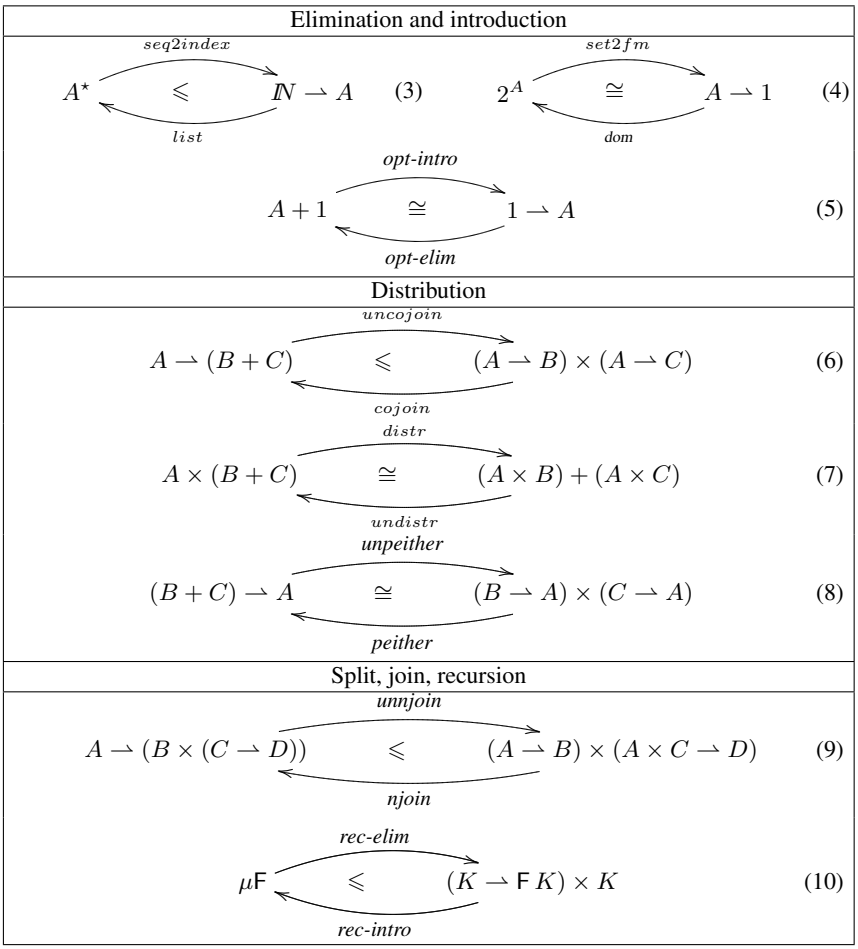


Fig. 1. One-step rules for a two-level transformation system that maps hierarchic, recursive data structures to flat relational mappings. Only the names of type-level functions are given. More details can be found elsewhere [20, 22, 23, 2].

We will illustrate the Haskell encoding with the following 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 (3) and (6) presented in Figure 1.

3.1 Representation of Types

Assume that $\mathbb{N}$ will be represented by Haskell type Int, $A \rightharpoonup B$ by the data type $Map\ a\ b$, and $A + B$ by $\mathbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$. We would

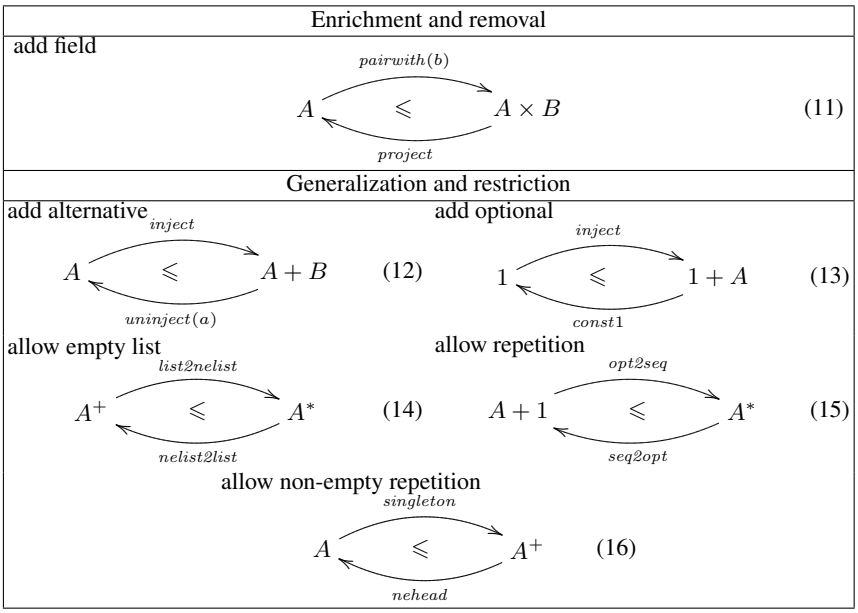


Fig. 2. One-step rules for a two-level transformation system for format evolution. These rules formalize the discussion of XML format evolution of Lämmel *et al* [12].

.

like now to define a rewriting strategy that converts the type $[Either\ a\ b]$ into the 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:

$$
\begin{aligned}
&\mathbf{data}\ Type\ a\ \mathbf{where}\\
&\quad Int :: Type\ Int\\
&\quad Bool :: Type\ Bool\\
&\quad Char :: Type\ Char\\
&\quad String :: Type\ String\\
&\quad One :: Type\ ()\\
&\quad List :: Type\ a \to Type\ [a]\\
&\quad Set :: Type\ a \to Type\ (Set\ a)\\
&\quad Map :: Type\ a \to Type\ b \to Type\ (Map\ a\ b)\\
&\quad Either :: Type\ a \to Type\ b \to Type\ (Either\ a\ b)
\end{aligned}
$$

$Prod :: Type\ a \to Type\ b \to Type\ (a, b)$
$Tag :: String \to Type\ a \to Type\ a$

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

The datatype $Type$, adapted from [9], is an example of a *generalized algebraic data type* (GADT) [24], 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 [8], since no value of type a needs to be provided when building a value of type $Type\ a$. 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, we can now assign a precise type $List\ (Either\ a\ b) \to Prod\ (Map\ Int\ a)\ (Map\ Int\ b)$ to our intended transformation.

3.2 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 [5, 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.

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

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. 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[1]:

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

Note that the explicit quantification of the type variable a will allow us to apply the same argument of type $Rule$ of a given rule combinator 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 the 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 (3) that convert a list into a map, and the rule (6) that converts a map of sums into a pair of maps can be defined as follows:

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

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

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

The only rule that poses a significant technical challenge is rule (16) for recursion elimination. We will only present an outline of our solution, which uses the Haskell class mechanism and monadic programming. Firstly, we represent the fixpoint operator μ as follows:

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

Here f is a functor[2], and the class constraint $Dist\ f$ expresses that we require functors to commute with monads. Rule (16) can now be implemented:

$\mathbf{type}\ Table\ f = (Map\ Int\ (f\ Int), Int)$

$fixastable :: Rule$
$fixastable\ (Mu\ f) = Just\ (View\ rep\ (Prod\ (Map\ Int\ (f\ Int))\ Int))$

[1] We model partiality with $\mathbf{data}\ Maybe\ a = Nothing \mid Just\ a$.
[2] Functors are instances of: $\mathbf{class}\ Functor\ f\ \mathbf{where}\ fmap :: (a \to b) \to f\ a \to f\ b$.

$\quad\textbf{where } rep = Rep\{to = recelim, from = recintro\}$
$fixastable\ \_ = Nothing$

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

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.

3.3 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:

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

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

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

Using these combinators for pairs of value-level functions, we can define the two-level combinators. Sequential composition is defined as follows[3]:

$(\triangleright) :: Rule \rightarrow Rule \rightarrow Rule$
$(f \triangleright g)\ a = \textbf{do}\ View\ r\ b \leftarrow f\ a$
$\qquad\qquad\qquad View\ s\ c \leftarrow g\ b$
$\qquad\qquad\qquad return\ (View\ (comprep\ r\ s)\ c)$

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[4]:

[3] For composing partial functions we use the monadic **do**-notation, exploiting the fact that *Maybe* is an instance of a Monad [25].

[4] $mplus :: Maybe\ a \rightarrow Maybe\ a \rightarrow Maybe\ a$ returns the first argument if it is constructed with *Just* or the second argument otherwise.

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

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:

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

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:

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

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 $<\mathit{to}>$ and $<\mathit{from}>$ are shown in place of function objects, which are not printable. Thus, the existentially qualified result type of the transformation is *not* available statically, though its string representation is available dynamically.

3.4 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

within 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:

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

Again, GADTs are of great help in defining a data type that provides evidence to the type-checker that two types are equal:

$$\begin{aligned}
&\mathbf{data}\ Equal\ a\ b\ \mathbf{where}\\
&\quad Eq :: Equal\ a\ a
\end{aligned}$$

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.

$$\begin{aligned}
&teq :: Type\ a \to Type\ b \to Maybe\ (Equal\ a\ b)\\
&teq\ Int\ Int = return\ Eq\\
&teq\ (List\ a)\ (List\ b) = \mathbf{do}\ Eq \leftarrow teq\ a\ b\\
&\qquad\qquad\qquad\qquad\quad return\ Eq\\
&...
\end{aligned}$$

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

To demonstrate the two-level transformations, we will develop two small, but representative examples.

4.1 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 would like 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:

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

With a generic show function $\mathit{gshow} :: \mathit{Type}\ a \to a \to \mathit{String}$, we can print the album with tag information included:

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

This also ensures us that the album is actually well-typed with respect to the format.

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

$$
\begin{aligned}
&\mathit{addalt} :: \mathit{Type}\ b \to \mathit{Rule}\\
&\mathit{addalt}\ b\ a = \mathit{Just}\ (\mathit{View}\ \mathit{rep}\ (\mathit{Either}\ a\ b))\\
&\quad \mathbf{where}\ \mathit{rep} = \mathit{Rep}\{\,\mathit{to} = \mathit{Left}, \mathit{from} = \lambda(\mathit{Left}\ x) \to x\,\}\\
&\mathbf{type}\ \mathit{Query}\ b = \forall a.\, \mathit{Type}\ a \to a \to b\\
&\mathit{addfield} :: \mathit{Type}\ b \to \mathit{Query}\ b \to \mathit{Rule}\\
&\mathit{addfield}\ b\ f\ a = \mathit{Just}\ (\mathit{View}\ \mathit{rep}\ (\mathit{Prod}\ a\ b))\\
&\quad \mathbf{where}\ \mathit{rep} = \mathit{Rep}\{\,\mathit{to} = \lambda y \to (y, f\ a\ y), \mathit{from} = \mathit{fst}\,\}\\
&\mathit{inside} :: \mathit{String} \to \mathit{Rule} \to \mathit{Rule}
\end{aligned}
$$

```
inside n r (Tag m a)
  | n ≡ m = do { View r b ← r a; return (View r (Tag m b)) }
inside _ _ _ = Nothing
```

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:

```
addDvd = once (inside "Format" (addalt (Tag "DVD" One)))
addTracks = once (inside "Album" (addfield (List (Tag "Title" String)) q))
  where q :: Query [String]
        q (Prod (Tag "ASIN" String) _) (asin, _) = ...
        q _ _ = []
```

The query q uses the album identifier to lookup from another data source, e.g. via a query over the internet[5]. Subsequently, we can run the type-level transformation, and print the result type:

```
> 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.

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

4.2 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:

$> \mathbf{let}\ toRDB = once\ enum2int \triangleright removetags \triangleright flatten$
$> \mathbf{let}\ (Just\ vw0) = toRDB\ (List\ albumFormat)$
$> showType\ vw0$
$Map\ Int\ (Prod\ (Prod\ (Prod\ String\ String)\ String)\ Int)$
$> \mathbf{let}\ (Just\ vw1) = toRDB\ (List\ targetFormat)$
$> showType\ vw1$
$Prod\ (Map\ Int\ (Prod\ (Prod\ (Prod\ String\ String)\ String)\ Int))\ ($
$\quad Map\ (Prod\ Int\ Int)\ String)$

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:

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

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 the 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 an update one.

$> \mathbf{let}\ (Just\ lvw) = (addTracks \triangleright addDvd)\ (List\ albumFormat)$
$> \mathbf{let}\ dbs1 = Prod\ (Map\ ...)\ (Map\ (Prod\ Int\ Int)\ String)$
$> \mathbf{let}\ (Just\ x) = back\ vw0\ dbs0\ db$

$> \textbf{let}\ (Just\ y) = forth\ lvw\ (List\ targetFormat)\ x$
$> \textbf{let}\ (Just\ z) = forth\ vw1\ dbs1\ y$
$> z$
$(\{0 := (((\texttt{"B000002UB2"}, \texttt{"Abbey Road"}), \texttt{"The Beatles"}), 0),$
$\ \ 1 := (((\texttt{"B000002HCO"}, \texttt{"Debut"}), \texttt{"Bjork"}), 1)\},$
$\ \{(0,0) := \texttt{"Come Together"}, ...\})$

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

5 Related work

We will compare our contributions with related work in software transformation and in generic programming.

Software transformation Lämmel *et al* [12] propose a systematic approach to evolution of XML-based formats, where DTDs are transformed in a well-defined, 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 operationalization 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 and operationalization 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, 9]. 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.

Atannassow *et al* show how canonical isomorphisms (corresponding to laws for zeros, units, and associativity) between types can induce the value-level conversion functions [3]. 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 programmable 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.

6 Future work

We have provided a type-safe formalization of two-level data transformations, and we have shown its operationalization 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 [6]. Currently, our can 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.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL '89: Proc. 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–227, New York, NY, USA, 1989. ACM Press.
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. 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.
4. Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02: Proc. 7th ACM SIGPLAN international conference on Functional programming*, pages 157–166, New York, NY, USA, 2002. ACM Press.
5. M.v.d Brand, P. Klint, and J. Vinju. Term rewriting with type-safe traversal functions. In B. Gramlich and S. Lucas, editors, *Proc. 2nd Int. Workshop on Reduction Strategies in Rewriting and Programming*, volume 70 of *ENTCS*. Elsevier, 2002.
6. A. Cleve, J. Henrard, and J.-L. Hainaut. Co-transformations in information system reengineering. *Electr. Notes Theor. Comput. Sci.*, 137(3):5–15, 2005.
7. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 273–279. ACM Press, 1998.
8. R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave, 2003.
9. R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. 8th Int. Symposium on Functional and Logic Programming (FLOPS)*, 2006. To appear.
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 International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, 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-Verlag, 2006. To appear.
14. 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).

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 PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.
17. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proc. 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.
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 Tim Denvir, Cliff B. Jones, and Roger C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).
21. J.N. Oliveira. 'Explosive' Programming Controlled by Calculation . Technical Report UMDITR02/98, DI, University of Minho, September 1998. Presented at AFP'98 (3rd Intern. Summer School on Advanced Functional Programming), Braga, Portugal.
22. J.N. Oliveira. Data processing by calculation, 2001. 108 pages. Lecture Notes for the 6th Estonian Winter School in Computer Science, 4-9 March 2001, Palmse, Estonia.
23. J.N. Oliveira. Calculate databases with 'simplicity', September 2004. Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK.
24. S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. Submitted to PLDI'06, November 2005.
25. P. Wadler. The essence of functional programming. In *Conference record of POPL'92*, pages 1–14. ACM Press, 1992.

A Handling recursion

In Section 3.2 we presented an outline of a solution for representing recursive types and implementing the recursion-elimination rule (16). Here we give the complete solution.

A.1 Representation of recursive types with explicit fixpoints

To represent recursive types, we introduced an explicit fixpoint operator Mu:

$\mathbf{newtype}\ Mu\ f = In\{\ out :: f\ (Mu\ f)\ \}$

Here f is intended to be a functor.

Suppose now we want to express a recursive type such as $\mu X.(A \times X^\star)$. Unfortunately, Haskell does not provide lambda abstraction at the type level. We can form the type expression $(A, [X])$, for some type X, and even $\forall x.(A, [x])$,

but neither expression will do to instantiate f in $Mu\ f$. The solution lies in the introduction of a minimal set of *lifted* functors:

$$\begin{aligned}
&\mathbf{newtype}\ ID\ x &&= Id\{\, unId :: x \,\}\\
&\mathbf{newtype}\ K\ b\ x &&= Const\{\, unConst :: b \,\}\\
&\mathbf{data}\ (g{:}{+}{:}h)\ x &&= Inl\ (g\ x) \mid Inr\ (h\ x)\\
&\mathbf{data}\ (g{:}{*}{:}h)\ x &&= Pair\ (g\ x)\ (h\ x)\\
&\mathbf{newtype}\ (g{:}@{:}h)\ x &&= Comp\{\, unComp :: g\ (h\ x) \,\}
\end{aligned}$$

Each of these is parameterized with a type argument x that stands for a recursive occurrence. The lifted sum, product, and application functors transport this argument to their argument functors. The identity functor has a constructor Id that holds a recursive occurrence. The constant functor has a constructor $Constr$ that ignores x. Now we can express our example functor as $Mu\ ((K\ A){:}{*}{:}([\,]{:}@{:}ID))$, where A is some constant type.

Using these lifted functors, we can create a generalized abstract datatype that represents them at the term-level in a type-safe way:

$$\begin{aligned}
&\mathbf{data}\ Fctr\ f\ \mathbf{where}\\
&\quad ID :: Fctr\ ID\\
&\quad K\ \ :: Type\ a \to Fctr\ (K\ a)\\
&\quad (:{+}:) :: Fctr\ g \to Fctr\ h \to Fctr\ (g{:}{+}{:}h)\\
&\quad (:{*}:) :: Fctr\ g \to Fctr\ h \to Fctr\ (g{:}{*}{:}h)\\
&\quad (:@:) :: Functor\ g \Rightarrow (\forall a.\, Type\ a \to Type\ (g\ a)) \to Fctr\ h \to Fctr\ (g{:}@{:}h)
\end{aligned}$$

And finally, we can extend the type of type representations $Type\ \ a$ with an additional constructor:

$$\begin{aligned}
&\mathbf{data}\ Type\ a\ \mathbf{where}\\
&\quad \ldots\\
&\quad Mu :: Dist\ f \Rightarrow Fctr\ f \to Type\ (Mu\ f)
\end{aligned}$$

The class constraint $Dist\ f$ expresses that we require monads to distribute over functors — below we will explain $Dist$ in more detail. This additional constructor Mu allows us to represent recursive types at the term level in a type-safe manner, just as we were already able to do for non-recursive types. For example, we can ask the Haskell interpreter to infer the type of a term-level recursive type[6]:

$$\begin{aligned}
&>{:}t\ Mu\ ((K\ Int){:}{*}{:}(List{:}@{:}ID))\\
&Mu\ ((K\ Int){:}{*}{:}(List{:}@{:}ID)) :: Type\ (Mu\ ((K\ Int){:}{*}{:}([\,]{:}@{:}ID)))
\end{aligned}$$

The answer tells us that the term indeed represents the intended type.

Finally, we will need a function for applying a functor representation to a type representation, to obtain a type representation in which the corresponding *non-lifted* functor is applied to the represented type:

[6] In the actual output infix type constructors are printed as prefix ones.

```
applyF :: Fctr f → Type a → Maybe (View (Type (f a)))
applyF ID x       = return (View (Rep{to = unId, from = Id}) x)
applyF (K a) x    = return (View (Rep{to = unConst, from = Const}) a)
applyF (g:*:h) x  = do
  (View (Rep g2gx gx2g) gx) ← applyF g x
  (View (Rep h2hx hx2h) hx) ← applyF h x
  let rep = Rep (λ(Pair x y) → (g2gx x, h2hx y))
                (λ(x, y) → Pair (gx2g x) (hx2h y))
  return (View rep (Prod gx hx))
applyF (g:+:h) x = do
  (View (Rep g2gx gx2g) gx) ← applyF g x
  (View (Rep h2hx hx2h) hx) ← applyF h x
  let rep = Rep{to = (g2gx◊h2hx).aux, from = xua.(gx2g◊hx2h)}
  return (View rep (Either gx hx))
    where
      aux (Inl x) = Left x
      aux (Inr x) = Right x
      xua (Left x) = Inl x
      xua (Right x) = Inr x
applyF (g:@:h) x = do
  (View (Rep to from) hx) ← applyF h x
  let rep = Rep (fmap to.unComp) (Comp.fmap from)
  return (View rep (g hx))
(f◊g) (Left x) = Left (f x)
(f◊g) (Right x) = Right (g x)
```

Note that the result is not a representation of $Type\ (f\ a)$ as such, but a *view* on such a type. In this view, the value-level functions are recorded that perform lifting and unlifting of the functors.

We must point out that our representation of recursive types implies an important limitation. Note that the functor f in the type representation $Mu\ f$ is a *function*, on which our rules of type $Rule$ have no effect. The $once$ combinator, for instance, does not descend under Mu. In other words, fixpoint types are *opaque* for two-level transformation, until the moment that they are instantiated with a particular type using $applyF$. In practical terms, this means we need to do de-recursivation before applying other data mapping or format evolution steps, which is actually common practise in data refinement [20]. Lifting this limitation is the subject of ongoing work.

A.2 Recursion patterns

For data types defined with Mu it is possible to give generic definitions for the operators $fold$ and $unfold$, that encode the standard recursion patterns of iteration and co-iteration [17].

$$fold :: Functor\ f \Rightarrow (f\ a \to a) \to Mu\ f \to a$$
$$fold\ g = g.fmap\ (fold\ g).out$$
$$unfold :: Functor\ f \Rightarrow (a \to f\ a) \to a \to Mu\ f$$
$$unfold\ h = In.fmap\ (unfold\ h).h$$

In the $fold$ definition, parameter g is used to combine the result of the recursive invocations in order to compute the output of the function. The dual operator $unfold$ is much less known [7], but is very useful when building values of a recursive type. The parameter function h is applied to the input and, depending on the functor, dictates if the generation of the result stops or proceeds. In the later case it will also output the seeds to be used in the generation of the recursive substructures of the result.

To define the value-level transformation that takes recursive structures as input, and generates a table representation of it, we will want to perform a *stateful* recursive computation. As usual in pure functional programming, we will use a monad to carry the state information. Therefore, we are interested in a special kind of $fold$ that returns a monadic computation instead of a pure value.

Given a monad m and a functor f, if it is possible to distribute one over the other, then it is possible to generically define a recursion operator $mfold$ that given a value of type $Mu\ f$ produces a computation of type $m\ a$.

$$\textbf{class}\ Functor\ f \Rightarrow Dist\ f\ \textbf{where}$$
$$\quad dist :: Monad\ m \Rightarrow f\ (m\ a) \to m\ (f\ a)$$
$$\textbf{instance}\ (Dist\ f, Dist\ g) \Rightarrow Dist\ (f{:}*{:}g)\ \textbf{where}$$
$$\quad dist\ (Pair\ l\ r) = \textbf{do}\ \{x \leftarrow dist\ l; y \leftarrow dist\ r; return\ (Pair\ x\ y)\}$$
$$...$$
$$mfold :: (Monad\ m, Dist\ f) \Rightarrow (f\ a \to m\ a) \to Mu\ f \to m\ a$$
$$mfold\ g = fold\ (\lambda x \to \textbf{do}\ \{y \leftarrow dist\ x; g\ y\})$$

The type class $Dist$ is used to capture the needed distributive property. Note that this class can easily be instantiated for most functors — the instance for products is given as example. The implementation of the monadic $fold$ uses $dist$ to combine all recursive computations into a single computation containing the pure recursive results. This computation is then passed to the parameter g in order to produce the final monadic result.

A.3 Two-level hierarchical-relational mapping

Earlier, we introduced the following type synonym for the target structure of recursion elimination:

$\mathbf{type}\ \mathit{Table}\ f = (\mathit{Map}\ \mathit{Int}\ (f\ \mathit{Int}), \mathit{Int})$

Thus, our implementation of rule (16) refines a type $\mathit{Mu}\ f$ to a type $\mathit{Table}\ f$, which stores a map from pointers to representations of substructures, and a pointer to the root structure.

In the forward-value level function, we will use the folklore state monad, denoted $\mathit{State}\ s$ for a give state s, to carry the table that is being generated during recursion. The following functions on state monads are relevant:

$\mathit{get} :: \mathit{State}\ s\ s$
$\mathit{put} :: s \to \mathit{State}\ s\ ()$
$\mathit{execState} :: \mathit{State}\ s\ a \to s \to s$

Function get reads the current state, and put replaces the state by the given parameter. Given an initial state, $\mathit{execState}$ executes a stateful computation and outputs the final state.

Using the state monad, and the recursion patterns defined earlier, we implement the forward value-level function $\mathit{recelim}$ as follows:

$\mathit{recelim} :: \mathit{Dist}\ f \Rightarrow \mathit{Mu}\ f \to \mathit{Table}\ f$
$\mathit{recelim}\ \mathit{mu} = \mathit{execState}\ (\mathit{mfold}\ \mathit{aux}\ \mathit{mu})\ (\mathit{Map.empty}, -1)$
  $\mathbf{where}$
    $\mathit{aux} :: f\ \mathit{Int} \to \mathit{State}\ (\mathit{Table}\ f)\ \mathit{Int}$
    $\mathit{aux}\ i = \mathbf{do}$
      $(t, k) \leftarrow \mathit{get}$
      $\mathit{put}\ (\mathit{Map.insert}\ (k+1)\ i\ t, k+1)$
      $\mathit{return}\ (k+1)$

The initial state of our computation is a table containing an empty map, and the initial key is set to -1. Every time a new key is needed to store an entry, this component of the state is incremented in order to guarantee key uniqueness. The computation that produces the final table is an mfold which, at each node, performs the following operations: first it reads the state in order to get the current table; then a new entry with the current node is added to the table using a fresh key; finally the fresh key is returned in order to replace this node in its enclosing term.

The converse operation $\mathit{recintro}$ builds a recursive structure from a table[7]. It is defined using unfold:

$\mathit{recintro} :: \mathit{Functor}\ f \Rightarrow \mathit{Table}\ f \to \mathit{Mu}\ f$
$\mathit{recintro} = \mathit{unfold}\ \mathit{aux}$

[7] Notice that this operation may diverge if there are lost keys or circularity.

$$
\begin{array}{l}
\quad \textbf{where} \\
\quad\quad aux :: Functor\ f \Rightarrow Table\ f \rightarrow f\ (Table\ f) \\
\quad\quad aux\ (t,k) = fmap\ (\lambda k \rightarrow (t,k))\ (fromJust\ \$\ Map.lookup\ k\ t)
\end{array}
$$

Thus, in order to rebuild a node of the original recursive type, we first lookup its corresponding value in the table. Function $lookup$ returns a $Maybe$ value, and since we assume that all keys exist, function $fromJust :: Maybe\ a \rightarrow a$ is used to remove the $Just$ constructor. Remember that each original node is stored in the table with its recursive substructures replaced by the keys that point to the respective entries in the table. As such, it suffices to use the same table to progress with the generation of a particular substructure, provided that the original entry key k is first replaced with the key r that points to the entry containing that substructure.

With these value-level functions in place, we can define the rule for recursion elimination itself:

$$
\begin{array}{l}
fixastable :: Rule \\
fixastable\ (Mu\ f) = \textbf{do} \\
\quad (View\ (Rep\ to\ fr)\ fInt) \leftarrow applyF\ f\ Int \\
\quad \textbf{let}\ rep = Rep\ ((Map.map\ to \bowtie id).recelim) \\
\qquad\qquad\qquad\quad (recintro.(Map.map\ fr \bowtie id)) \\
\quad return\ (View\ rep\ (Prod\ (Map\ Int\ (fInt))\ Int)) \\
fixastable\ \_ = Nothing \\
(f \bowtie g)\ (x,y) = (f\ x, g\ y)
\end{array}
$$

Note that the functor f of the recursive structure $Mu\ f$ gets applied to the type Int to obtain the type of values in the pointer-value map.

With the *fixastable* rule, we can augment the $toRDB$ rule, used in the data mapping example of Section 4.2, to handle also recursive types:

$$
\begin{array}{l}
toRDB :: Rule \\
toRDB = many\ (once\ fixastable) \triangleright \\
\qquad\quad many\ (once\ enum2int) \triangleright removetags \triangleright flatten
\end{array}
$$

The $toRDB$ rule will remove fixpoints, then convert enumeration types to integers, remove tags, and apply the remaining data mapping rules, as captured in $flatten$.

A.4 Application scenario: derecursivation of employee hierarchy

Suppose a hierarchy of employees, specifying their names, jobs, and subordination relationships is stored in XML files that conform to the following schema[8]:

[8] This example schema was adapted from the online *.NET Framework Developer's Guide* (http://msdn.microsoft.com/library/).

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Emp" type="EmployeeType" />
  <xs:complexType name="EmployeeType">
    <xs:sequence>
      <xs:element name="Emp" type="EmployeeType" />
    </xs:sequence>
    <xs:attribute name="Job" type="xs:string" />
    <xs:attribute name="FirstName" type="xs:string"/>
    <xs:attribute name="LastName" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Note that this schema is recursive in the complex type `EmployeeType`. We can capture this schema with the following nominal Haskell types:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Emp} = \mathit{Emp}\ \mathit{EmployeeType}\\
&\mathbf{data}\ \mathit{EmployeeType} = \mathit{EmployeeType}\{\\
&\quad \mathit{emp_seq} :: [\mathit{EmployeeType}],\\
&\quad \mathit{job} :: \mathit{String},\\
&\quad \mathit{firstName} :: \mathit{String},\\
&\quad \mathit{lastName} :: \mathit{String}\}
\end{aligned}$$

As an example, we can represent (the tip of) the European Commission hierarchy as a value of type Emp[9]:

$$\begin{aligned}
&\mathit{ec} :: \mathit{Emp}\\
&\mathit{ec} = \mathit{Emp}\ \$\ \mathit{EmployeeType}\{\\
&\quad \mathit{emp_seq} = [\\
&\qquad \mathit{EmployeeType}\{\\
&\qquad\quad \mathit{emp_seq} = [\\
&\qquad\qquad \mathit{EmployeeType}\ [\,]\ \texttt{"Driver"}\ \texttt{"Asdren"}\ \texttt{"Juniku"},\\
&\qquad\qquad \mathit{EmployeeType}\ [\,]\ \texttt{"Head of Cabinet"}\ \texttt{"Ben"}\ \texttt{"Smulders"}],\\
&\qquad\quad \mathit{job} = \texttt{"Competition"},\\
&\qquad\quad \mathit{firstName} = \texttt{"Neelie"},\\
&\qquad\quad \mathit{lastName} = \texttt{"Kroes"}\},\\
&\qquad \mathit{EmployeeType}\ [\,]\ \texttt{"Trade"}\ \texttt{"Peter"}\ \texttt{"Mandelson"}],\\
&\quad \mathit{job} = \texttt{"President"},\\
&\quad \mathit{firstName} = \texttt{"Durao"},\\
&\quad \mathit{lastName} = \texttt{"Barroso"}\}
\end{aligned}$$

In the example we mix Haskell's record constructor syntax (with braces and field names) and its plain constructor syntax.

With our structural representation of types, the same schema is captured as follows:

[9] See http://europa.eu.int/comm/commission_barroso/index_en.htm.

```
emp = Tag "Emp" employeeType
employeeType = Mu employeeTypeF
employeeTypeF =
  ((List.Tag "Emp"):@:ID):*:
  (K $ Tag "job" String):*:
  (K $ Tag "firstName" String):*:
  (K $ Tag "lastName" String)
```

Note that the Mu operator and lifted functors are used to specify the recursive part. A conversion function from the nominal type to the structural type is readily specified, using the *unfold* co-recursion pattern:

```
emp2fix (Emp et) = unfold aux et
where
  aux (EmployeeType s e f l) = Pair (
    Comp (map Id s))(Pair (
    Const e)(Pair (
    Const f)(
    Const l)))
```

By applying the *emp2fix* function to a particular hierarchy such as *ec* it becomes amenable to two-level transformation.

In particular, we can apply the $toRDB$ rule to derive a relational database representation for the European Commission. First, as usual, we perform the type-level transformation:

```
> let (Just vw) = toRDB emp
> putStrLn $ showType vw
(Prod (Prod Int (
            Map Int (Prod (Prod String String) String))) (
            Map (Prod Int Int) Int))
```

Thus, the resulting type is the product of an integer (the key of the root of the hierarchy), a table of employee attributes (mapping an integer key to a 3-tuple of strings), and a table to represent employee subordination relationships (mapping a compound of the boss' key and a subordinate's position number in his/her list of subordinates to that subordinate's key).

The forward value-level transformation can be carried out as follows:

```
> let rdbType = Prod (Prod Int (Map ...)) (Map ...)
> let (Just ecRDB) = forth vw rdbType $ emp2fix ec
> putStrLn $ gshow rdbType ecRDB
((4,{
  0 := (("Driver","Asdren"),"Juniku"),
  1 := (("Head of Cabinet","Ben"),"Smulders"),
  2 := (("Competition","Neelie"),"Kroes"),
```

$3 := (("\texttt{Trade}", "\texttt{Peter}"), "\texttt{Mandelson}"),$
$4 := (("\texttt{President}", "\texttt{Durao}"), "\texttt{Barroso}")\}), \{$
$(2,0) := 0,$
$(2,1) := 1,$
$(4,0) := 2,$
$(4,1) := 3\})$

The backward value-level transformation may be useful, for instance to dynamically generate HTML pages for the European Union portal from a database with employees. It goes as follows:

```
> let (Just ec') = back vw rdbType ecDB
> putStrLn $ gshow emp ec'
Emp = (
  [Emp = (
     [Emp = ([], (
        job = "Driver", (
        firstName = "Asdren",
        lastName = "Juniku"))),
      Emp = ([], (
        job = "Head of Cabinet", (
        firstName = "Ben",
        lastName = "Smulders")))], (
     job = "Competition", (
     firstName = "Neelie",
     lastName = "Kroes"))),
   Emp = ([], (
     job = "Trade", (
     firstName = "Peter",
     lastName = "Mandelson")))], (
  job = "President", (
  firstName = "Durao",
  lastName = "Barroso")))
```

Thus, after being flattened and stored into a relational database, data can be restored into its original hierarchical form.

B Using dynamic types to unleash data migration functions

In Section 3.4 we explained a staged approach to access the data migration functions encapsulated in a view: in a first stage the string representation of the target type is obtained, and in a second stage this representation is incorporated in the source code, and used in combination with functions *forth* and *back*. Though

this staged approach works fine for most scenarios, one may alternatively resort to *dynamic* types to perform type-level and value-level transformation in a single stage. Dynamic types were introduced in statically typed language for precisely such purposes [1].

We will rely on a *type safe* implementation of dynamic types in statically typed languages [4], but strongly simplified by use of GADTs (*cf* [24]). Given that $Type$ already provides type representations it is fairly easy to define a dynamic type:

$\mathbf{data}\ Dynamic\ \mathbf{where}$
$\quad Dyn :: Type\ a \to a \to Dynamic$

Since the type variable a is implicitly existentially quantified in the Dyn constructor, the $Dynamic$ type effectively hides the type of the value it contains. Note that we can turn any generic (type-indexed) function of type $\forall a.\, Type\ a \to a \to b$ into a function on a dynamic value:

$applyDyn :: (\forall a.\, Type\ a \to a \to b) \to Dynamic \to b$
$applyDyn\ f\ (Dyn\ ta\ a) = f\ ta\ a$

Thus, we can process a value encapsulated in Dyn without static availability of its type, by using a function indexed by type.

We can now define variations on the statically typed $forth$ and $back$ functions that employ dynamic types:

$forthDyn :: View\ (Type\ a) \to a \to Maybe\ Dynamic$
$forthDyn\ (View\ rep\ b)\ x = return\ (Dyn\ b\ (to\ rep\ x))$
$backDyn :: View\ (Type\ a) \to Dynamic \to Maybe\ a$
$backDyn\ (View\ rep\ b)\ (Dyn\ c\ x) = \mathbf{do}$
$\quad Eq \leftarrow teq\ b\ c$
$\quad return\ (from\ rep\ x)$

Thus, rather than accepting/returning a value of a given target type b, these functions accept/return that value encapsulated together with its type inside the Dyn constructor.

The net effect of using $forthDyn$ is to postpone the need for knowledge of the target type until *after* application of the value-level transformation function. Subsequently, we have at least two options for further processing the resulting $Dynamic$. One option is to apply a type-indexed function, via $applyDyn$. For instance, we can apply the generic show function:

$> \mathbf{let}\ (Just\ vw) = toRDB\ emp$
$> \mathbf{let}\ (Just\ ecRdbDyn) = forthDyn\ vw\ \$\ emp2fix\ ec$
$> putStrLn\ \$\ applyDyn\ gshow\ ecRdbDyn$
$((4, \{$
$\quad 0 := ((\texttt{"Driver"}, \texttt{"Asdren"}), \texttt{"Juniku"}), ...)\}), \{$
$\quad (2, 0) := 0, ...\})$

The other option is to apply a backward value-level transformation, via $backDyn$:

$> \mathbf{let}\ (Just\ ec'') = backDyn\ vw\ ecRdbDyn$
$> ec'' \equiv emp2fix\ ec$
$True$

Here we applied the backward value-transformation from the same view vw that we used to go forward. In general, any view with the same target type can be used. Assume, for example, that we define a two-level transformation from a format without subordinate relations:

$> \mathbf{let}\ empList = List\ (Prod\ (Tag\ \texttt{"function"}\ String)\ (Prod\ (Tag\ \texttt{"first"}\ String)(Tag\ \texttt{"last"}\ String)))$
$> \mathbf{let}\ (Just\ vw') = (toRDB \triangleright (addfieldl\ Int\ (-1)) \triangleright (addfield\ (Map\ (Prod\ Int\ Int)\ Int)\ Map.empty)\ )\ empList$

Here, $addfieldl$ is a variation on $addfield$ that adds a new field to the left, rather than the right. We can supply this view to $backDyn$ to retrieve values from the database into the list format:

$> \mathbf{let}\ (Just\ el) = backDyn\ vw'\ ecRdbDyn$
$> putStrLn\ \$\ gshow\ empList\ el$
$[(function = \texttt{"Driver"}, (first = \texttt{"Asdren"}, last = \texttt{"Juniku"})),$
$...,$
$(function = \texttt{"President"}, (first = \texttt{"Durao"}, last = \texttt{"Barroso"}))]$

Thus, we can store hierarchies of employees (rich format) into a relational database, and retrieve lists of names and functions (poor format), without any static knowledge of the intermediate database schema.

Note that the rich and the poor format are different in the names of the fields, and in the way products are nested. Since the $toRDB$ transformation removes tags and performs association of products to the left, the composite value-level transformation will match formats modulo names and associativity. This is reminiscent, but potentially more general than canonical isomorphism calculations [3].

Note also that the conversion between the hiearchical format and the list format is a specific case of the type of transformations discussed in [21].