Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 17–34
www.elsevier.com/locate/entcs

Strongly Typed Rewriting For Coupled Software Transformation

Alcino Cunha[1,3] Joost Visser[2,3]

DI-CCTC, Universidade do Minho, Portugal

Abstract

Coupled transformations occur in software evolution when multiple artifacts must be modified in such a way that they remain consistent with each other. An important example involves the coupled transformation of a data type, its instances, and the programs that consume or produce it. Previously, we have provided a formal treatment of transformation of the first two: data types and instances. The treatment involved the construction of type-safe, *type-changing* strategic rewrite systems. In this paper, we extend our treatment to the transformation of corresponding data processing programs.

The key insight underlying the extension is that both data migration functions and data processors can be represented type-safely by a generalized abstract data type (GADT). These representations are then subjected to program calculation rules, harnessed in type-safe, *type-preserving* strategic rewrite systems. For ease of calculation, we use point-free representations and corresponding calculation rules.

Thus, coupled transformations are carried out in two steps. First, a type-changing rewrite system is applied to a source type to obtain a target type together with (representations of) migration functions between source and target. Then, a type-preserving rewrite system is applied to the composition of a migration function and a data processor on the source (or target) type to obtain a data processor on the target (or source) type. All rewrites are type-safe.

Keywords: Program transformation, term rewriting, strategic programming, generalized abstract datatypes, data refinement.

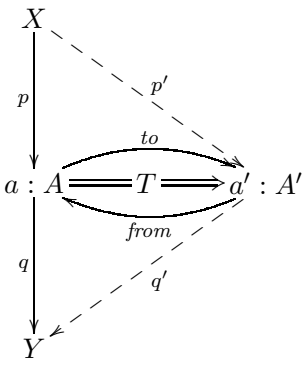
1 Introduction

Coupled transformations occur in software evolution when multiple artifacts must be modified in such a way that they remain consistent with each other. Lämmel [14] identified the category of coupled transformations and discussed their widespread occurrence in problem domains such as cooperative editing, software modeling, model transformation, and re-/reverse engineering.

A particularly challenging instance of coupled transformation involves the joint transformation of a data type, its instances, and the programs that consume or

[1] Email: alcino@di.uminho.pt
[2] Email: joost.visser@di.uminho.pt
[3] Work funded by Fundação para a Ciência e a Tecnologia, POSI/ICHS/44304/2002.

doi:10.1016/j.entcs.2006.10.019



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

Fig. 1. Coupled transformation of data type A, data instance a, data producer p, and data query q. The challenge is to calculate p' and q' by fusing the compositions $to \circ p$ and $q \circ from$ such that they work on A' directly rather than via A.

produce it. In the context of software renovation, this problem occurs for example when the schema of a database needs to be adapted. The adaptation of the database schema must then be coupled with migration of the database instances and with updates of the programs that connect to these database instances.

An important research challenge remains in providing a general and comprehensive conceptual framework for coupled transformations [15]. Previously, we have taken first steps to providing such a framework. In particular, we have provided a formal treatment of two-level data transformations, i.e. coupled transformation of data types and instances [6]. In this paper, we extend our previous treatment to include transformations of data processors.

A global overview of our perspective on the coupled transformation problem is given in Figure 1. A type-level transformation T of a source type A into a target type A' is witnessed by associated instance migration functions *to* and *from*. In Section 5 we recapitulate how such type-level and value-level transformations can be coupled into a two-level data transformation system, modeled as a strongly-typed, type-changing, strategic rewrite system.

The query q that consumes values of type A and the producer p that generates such values are examples of data processing programs. To obtain queries and producers on the transformed type A', we can simply compose q and p with the migration functions *from* and *to*. This amounts to a *wrapper* approach to program migration where the original type and the original processors are still explicitly present. The challenge that we take up in the present paper is to calculate processors q' and p' from those wrapper compositions in such a way that they no longer involve the original type and processors. In Section 4 we explain how such program calculations can be harnessed in an additional type-preserving strategic rewrite system on type-safe representations of the functions (queries, producers, migrations) involved. The key idea is to use fusion or deforestation techniques [24] in order to eliminate the intermediate data type A.

In Section 2 we present a concrete example to motivate our approach to coupled transformation. In Section 6 we show how the type-preserving rewrite system for

program calculation (Section 4), and the type-changing rewrite system for two-level data transformation (Section 5) can be combined to perform coupled transformations. Both rewrite systems employ a strongly-typed representation of types at the value level, recapitulated in Section 3. Section 7 discusses related work and Section 8 concludes.

2 Motivating example: query evolution

Suppose information about music albums is kept in XML files that conform to a document schema captured by the following Haskell types:

$$\begin{aligned}
&\textbf{type } \mathit{Albums} = [\mathit{Album}]\\
&\textbf{type } \mathit{Album} = (\mathit{ASIN}, (\mathit{Title}, \mathit{Artist}))\\
&\textbf{type } \mathit{ASIN} = \mathit{String}\\
&\textbf{type } \mathit{Title} = \mathit{String}\\
&\textbf{type } \mathit{Artist} = \mathit{String}
\end{aligned}$$

A query on this format would be a function such as the following:

$$\begin{aligned}
&\mathit{getArtists} :: \mathit{Albums} \to [\mathit{Artist}]\\
&\mathit{getArtists} = \mathit{List.map}\ (\mathit{snd} \circ \mathit{snd})
\end{aligned}$$

And an instance of a music album collection would look as follows:

$$\begin{aligned}
&\mathit{collection} = [\\
&\quad (\texttt{"B000002UB2"}, (\texttt{"Abbey Road"}, \texttt{"The Beatles"})),\\
&\quad (\texttt{"B000002HCO"}, (\texttt{"Debut"}, \texttt{"Bjork"}))]
\end{aligned}$$

In a realistic situation, there would be many queries and producers on a given format, and large, possibly many instances.

Suppose that the album format is changed into a new format in two steps. First, track information is added to each album. This involves introducing a new field, which holds a nested list of track titles:

$$\begin{aligned}
&\textbf{type } \mathit{Album}' = (\mathit{ASIN}, (\mathit{Title}, (\mathit{Artist}, \mathit{Tracks})))\\
&\textbf{type } \mathit{Tracks} = [\mathit{Title}]
\end{aligned}$$

Subsequently, to store the album collection in a relational database, the format is transformed into a representation based on finite maps:

$$\textbf{type } \mathit{DB} = (\mathit{Map}\ \mathit{Int}\ \mathit{Album}, \mathit{Map}\ (\mathit{Int}, \mathit{Int})\ \mathit{Title})$$

The first map represents a table of albums. The second map is a table with compound key that holds track titles.

The first step is an example of *format evolution*. The second is a hierarchical-relational data mapping. Of course, these two transformation steps invalidate all existing data instances and processors.

In this paper, we will show that the existing data and queries can actually be automatically migrated to the new format by a program transformation coupled to the format transformation. For example, after the first step, the query can be

migrated to:

$$getArtists' = List.map\ (fst \circ snd \circ snd)$$

After the second step, the query is migrated to:

$$getArtistsDB = elems \circ Map.map\ (snd \circ snd) \circ fst$$

Here *elems* is a standard function on maps that returns the range of the map as a list (in ascending order of keys), and *Map.map* applies its argument function to each range element. Note that the migrated queries do not involve the original album type as intermediate format, and are not defined in terms of the original query wrapped by a migration function.

3 Representation of types

In both rewrite systems that we will define, we need access to type-representations on the value-level. The type-preserving rewrite system on point-free expressions needs them for *type-directed* rewrite decisions, while the type-changing rewrite system performs rewrites on types themselves. To ensure type-safety of both rewrite systems, a universal representation of types does not suffice.

Using *generalized algebraic data types* (GADTs) [22], a recent extension to the Haskell type system, it is possible to declare a parameterized data type $Type\ a$ whose inhabitants must be representations of type a [12]:

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

Notice that in this declaration the type a that parameterizes $Type\ a$ is restricted differently in each constructor. This is precisely the difference between a GADT and a regular parameterized data type, where the parameter of the resulting type must always be unrestricted. For example, the type constructor *Int* has type $Type\ Int$ but $List\ Int$ has type $Type\ [Int]$. Thus, the parameter a of the GADT $Type\ a$ allows us to carry around the necessary type-information to ensure type-safety.

$Type\ a$ allows representation of some base types, sums, products, functions, lists, and finite maps – sufficient for the purposes of this paper, but extensible to more types if needed. The *Tag* constructor allows us to tag types with names. For example, the type representation for *Albums* can be defined as follows.

$$albums :: Type\ Albums$$

$albums = List\ (Tag\ \texttt{"Album"}\ (Prod\ String\ (Prod\ String\ String)))$

The use of the *Tag* will become clear later.

Given a ground type a it is possible to use the Haskell type system to infer its representation. We define a class *Typeable a* with a method *typeof* that returns the representation:

class *Typeable a* **where** *typeof* :: *Type a*
instance *Typeable Int* **where** *typeof* = *Int*
instance *Typeable a* ⇒ *Typeable* [*a*] **where** *typeof* = *List typeof*

Trivial instances of *Typeable* must be defined for the remaining types present in *Type*. Now, the following interaction with the interpreter is possible:

> *typeof* :: *Type* [(*Int*, *Char*)]
(*List* (*Prod Int Char*))

This relies on a straightforward instance of the class *Show* for *Type a*.

4 Point-free program calculus

In this section, we explain how type-safe, type-preserving strategic rewrite systems are defined to apply program calculation rules to migration functions and data processors. In particular, we aim to apply fusion laws to simplify wrapped processors of original data into processors that work directly on new data, thus avoiding to build intermediate data structures of the original types. If fusion is successful, substantial gains of efficiency can be achieved.

Our functions will be represented using the so-called point-free style of programming, in which functions are built from simpler ones using a small set of combinators and primitive functions, without mentioning their arguments explicitly. In fact, the *getArtists* query was defined in this style. Point-free combinators satisfy a powerful set of equational laws for reasoning about functional programs by calculation [11], and due to the absence of variables and λ-abstractions, implementing a rewrite system to automate such calculations is straightforward.

Point-free combinators and their laws The most fundamental combinators of point-free programming are function composition and the identity function.

$$(\cdot \circ \cdot)\ :\ (b \to c) \to (a \to b) \to a \to c \qquad \mathsf{id}\ :\ a \to a$$
$$(\cdot \circ \cdot) = \lambda f g x.\ f\ (g\ x) \qquad \mathsf{id} = \lambda x.\ x$$

Together these combinators enjoy the following laws, which are so fundamental that they are usually used implicitly during program calculations.

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \textsf{Comp-Assoc}$$
$$\mathsf{id} \circ f = f \circ \mathsf{id} = f \qquad \textsf{Id-Nat}$$

To handle products we have the split combinator ($\vartriangle$), which pairs the results of

applying two functions to the same value, and the projections:

$$
\begin{array}{l}
(\cdot \vartriangle \cdot) \;:\; (a \to b) \to (a \to c) \to a \to (b \times c) \\
(\cdot \vartriangle \cdot) = \lambda f g x.\ (f\ x, g\ x) \\
\mathsf{fst} \;:\; (a \times b) \to a \qquad \mathsf{snd} \;:\; (a \times b) \to b \\
\mathsf{fst} = \lambda(x, y).\ x \qquad \mathsf{snd} = \lambda(x, y).\ y
\end{array}
$$

Concerning these we have the following laws and a derived product combinator:

$$
\begin{array}{ll}
\mathsf{fst} \vartriangle \mathsf{snd} = \mathsf{id} & \textsf{Prod-Reflex} \\
\mathsf{fst} \circ (f \vartriangle g) = f \quad \wedge \quad \mathsf{snd} \circ (f \vartriangle g) = g & \textsf{Prod-Cancel} \\
(f \vartriangle g) \circ h = f \circ h \vartriangle g \circ h & \textsf{Prod-Fusion} \\
f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd} & \textsf{Prod-Def}
\end{array}
$$

We also have the primitive function $\mathsf{bang} : a \to 1$ that given any value returns the single inhabitant of type 1. It satisfies the following law.

$$f = \mathsf{bang} \quad \text{iff} \quad f : a \to 1 \qquad \textsf{Bang-Eta}$$

Similar combinators and laws exist for sums and functions, but they are not used in this paper. The combinators and laws for lists and maps will be presented later.

Type-safe representation of functions For type-safe representation of point-free functions, we resort again to a GADT:

```
data PF f where
  Id    :: PF (a → a)
  Comp  :: Type b → PF (b → c) → PF (a → b) → PF (a → c)
  Fst   :: PF ((a, b) → a)
  Snd   :: PF ((a, b) → b)
  · △ · :: PF (a → b) → PF (a → c) → PF (a → (b, c))
  · × · :: PF (a → b) → PF (c → d) → PF ((a, c) → (b, d))
  Bang  :: PF (a → One)
  Fun   :: String → (a → b) → PF (a → b)
```

An inhabitant of type $PF\ (a \to b)$ is a point-free representation of a function of type $a \to b$. The presence of the intermediate type representation in the composition constructor will be explained later. The *Fun* constructor allows us to include pointwise functions in point-free expressions without being forced to convert them into point-free shape; it can be used for functions over which no specific reasoning is performed. By using a GADT to represent point-free expressions we gain type-checking for free: it is not possible to write an impossible function like $\mathsf{fst} \circ \mathsf{bang}$ or assigning an incorrect type to an expression.

This abstract syntax of pointfree expressions can be translated into concrete functions by the following evaluation function:

```
eval :: PF a → a
eval Id = λx → x
eval (Comp _ f g) = λx → (eval f) (eval g x)
eval Fst = λ(x, y) → x
```

$$
\begin{aligned}
&\mathit{eval}\ \mathit{Snd} = \lambda(x, y) \to y\\
&\mathit{eval}\ (f \triangle g) = \lambda x \to (\mathit{eval}\ f\ x, \mathit{eval}\ g\ x)\\
&\mathit{eval}\ (f \times g) = \lambda(x, y) \to (\mathit{eval}\ f\ x, \mathit{eval}\ g\ y)\\
&\mathit{eval}\ \mathit{Bang} = \lambda\_ \to ()\\
&\mathit{eval}\ (\mathit{Fun}\ n\ f) = f
\end{aligned}
$$

For example:

$$
\begin{aligned}
&> \textbf{let}\ \mathit{assocr} = (\mathit{Comp}\ (\mathit{Prod}\ \mathit{Int}\ \mathit{String})\ \mathit{Fst}\ \mathit{Fst}) \triangle (\mathit{Snd} \times \mathit{Id})\\
&> \mathit{eval}\ \mathit{assocr}\ ((1, \texttt{"foo"}), \mathit{True})\\
&(1, (\texttt{"foo"}, \mathit{True}))
\end{aligned}
$$

Rewriting point-free expressions In order to simplify point-free expressions we implemented a type-preserving strategic rewrite system, where rules as well as strategies composed from them have the following type:

$$\textbf{type}\ \mathit{Rule} = \forall a\ .\ \mathit{Type}\ a \to \mathit{PF}\ a \to \mathit{RewriteM}\ (\mathit{PF}\ a)$$

Here, *RewriteM* is a monad. Thus, rewrite rules are basically monadic functions on point-free representations, additionally parameterized with a type representation.

The *RewriteM* monad models partiality and is an instance of class *MonadPlus*:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where}\\
&\quad \mathit{mzero} :: m\ a\\
&\quad \mathit{mplus} :: m\ a \to m\ a \to m\ a
\end{aligned}
$$

A well-known instance of the *MonadPlus* class is the type $\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$. Our *RewriteM* monad extends this with the capability to produce a proof trace during rewriting that includes intermediate results and names of applied rules, but this is not relevant for the purposes of this paper.

The additional parameter of rules is used for *type-directed* rewriting. Type-directed rewriting is essential, because some laws need type information: using syntax alone is not possible to decide if the law can be applied. For example, the Bang-Eta law states that any function of type $a \to 1$ can be transformed into bang regardless of its definition. Consider also the Prod-Reflex law when used as an expansion (from right to left). It cannot be applied to just any identity function but only to those of type $a \times b \to a \times b$. The use of such type-directed expansions is becoming extremely relevant; they were fundamental, for example, in establishing the decidability of equality in the simply typed λ-calculus [8,9].

As an example of encoding a type-directed rule, consider Bang-Eta (left to right):

$$
\begin{aligned}
&\mathit{bang\_eta} :: \mathit{Rule}\\
&\mathit{bang\_eta}\ \_\ \mathit{Bang} &&= \mathit{mzero}\\
&\mathit{bang\_eta}\ (\mathit{Func}\ \_\ \mathit{One})\ \_ &&= \mathit{return}\ \mathit{Bang}\\
&\mathit{bang\_eta}\ \_\ \_ &&= \mathit{mzero}
\end{aligned}
$$

In order to avoid non-termination the rule fails (*mzero*) on *Bang* itself. Other expressions of type $a \to ()$ are rewritten to *Bang*. Otherwise the rule fails.

Type-directed rewriting does not require us to annotate all point-free combina-

tors with type representations. In most cases types of subexpressions can be derived from those of the enclosing expressions. Unless a type variable appears existentially quantified in the type of a constructor, as is the case for *Comp*. This explains why we must parameterize *Comp* with a representation of its intermediate type. As a consequence, the user must reason explicitly about types when specifying rules involving composition. An example is Comp-Assoc from left to right:

$$
\begin{aligned}
&comp\_assocr :: Rule\\
&comp\_assocr\ \_\ (Comp\ a\ (Comp\ b\ f\ g)\ h)\\
&\quad = return\ (Comp\ b\ f\ (Comp\ a\ g\ h))\\
&comp\_assocr\ \_\ \_ = mzero
\end{aligned}
$$

Note how the new intermediate types are specified using the input ones.

Dealing with associativity of composition When implementing rewrite systems for point-free expressions special care must be taken about Comp-Assoc. When calculating by hand, this rule is implicitly assumed by omitting parentheses around composition combinators. This strategy has been hard-wired into some rewrite systems, such as Bird's functional calculator [2], by using a composition operator of variable arity and implementing a dedicated pattern matching mechanism modulo Comp-Assoc.

We used a different approach. Before applying any rule concerning products we apply *comp_assocr* exhaustively in order to guarantee that all compositions are associated to the right. Then some completion must be performed on the laws with an outermost composition when the right parameter is not arbitrary. As example consider the encoding of Prod-Cancel:

$$
\begin{aligned}
&prod\_cancel :: Rule\\
&prod\_cancel\ \_\ (Comp\ \_\ Fst\ (f \vartriangle g)) = return\ f\\
&prod\_cancel\ \_\ (Comp\ \_\ Snd\ (f \vartriangle g)) = return\ g\\
&prod\_cancel\ \_\ (Comp\ \_\ Fst\ (Comp\ a\ (f \vartriangle g)\ h)) = return\ (Comp\ a\ f\ h)\\
&prod\_cancel\ \_\ (Comp\ \_\ Snd\ (Comp\ a\ (f \vartriangle g)\ h)) = return\ (Comp\ a\ g\ h)\\
&prod\_cancel\ \_\ \_ = mzero
\end{aligned}
$$

The additional third and fourth equation allow application of this rule at any position in a sequence of right-associated compositions.

Strategy combinators We define the typical strategic rewriting combinators:

$$
\begin{aligned}
&nop :: Rule && many :: Rule \to Rule\\
&nop\ t = return && many\ r = (r \triangleright many\ r) \oslash nop\\
&(\triangleright) :: Rule \to Rule \to Rule && many1 :: Rule \to Rule\\
&(f \triangleright g)\ t\ x = f\ t\ x \gg\!= g\ t && many1\ r = r \triangleright many\ r\\
&(\oslash) :: Rule \to Rule \to Rule && try :: Rule \to Rule\\
&(f \oslash g)\ t\ x = f\ t\ x\ `mplus`\ g\ t\ x && try\ x = x \oslash nop
\end{aligned}
$$

The $\triangleright$ combinator applies two rules in sequence (the second is only applied if the first succeeds). $\oslash$ is a left-biased choice that tries to apply the first rule or, if it fails, it tries the second. The *many* combinator repeatedly tries to apply a rule until it fails. *many1* is similar but must succeed at least once. *try* tries to apply a rule and

returns the original expression if it fails, and *nop* is a rule that always succeeds.

We also define standard traversal combinators, whose definitions we omit:

$$once :: Rule \to Rule \qquad everywhere :: Rule \to Rule$$

These combinators take a single rule as argument and apply it once, resp. everywhere, during a top-down traversal of an expression.

Composing a rewrite system Equipped with basic rules and combinators, it is possible to define rewrite systems, e.g. to simplify expressions involving products:

```
prods :: Rule
prods = many1 (
            many (once comp_assocr) ▷
            once (bang_eta ⊘ prod_cancel ⊘ prod_reflex ⊘ ...))
```

Although simple, this handles an interesting and useful class of simplifications.

For convenience, we define a function to apply a rewrite system to a point-free expression:

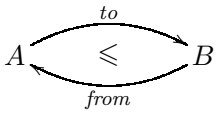
$$rewrite :: Typeable\ a \Rightarrow Rule \to PF\ a \to IO\ (PF\ a)$$

Internally, it handles deriving the type representation of the expression subject to rewriting and the reporting of success and failure.

5 Two-level data transformation

The second component of our solution to coupled transformation are *type-changing* rewrite systems that allow two-level data transformation. We rely on the rewrite systems presented by us in previous work [6] with a single, essential modification: instead of value-level transformations (or migration functions) themselves, we will use their point-free representations, which can then subsequently be subjected to the type-preserving rewrite system of the previous section.

Data refinement calculus Two-level transformation steps are modeled by inequations between datatypes and by 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 $from \circ to = id_A$, where id_A is the identity function on datatype A. Though in general *to* can be a relation, it is usually a function. Thus, type-level transformations arise from the existence of value-level transformations *to* and *from* whose properties preclude data mixup. When applied left-to-right, an inequation $A \leqslant B$ will preserve or enrich information content, while applied right-to-left it will preserve or restrict information content.

Encapsulation of type-changing rewrites The core of our solution to mod-

eling two-level data transformations in Haskell are the following type declarations:

```
data Rep a b = Rep{ to :: PF (a → b), from :: PF (b → a) }
data View a where View :: Rep a b → Type b → View (Type a)
type RULE = ∀a . Type a → Maybe (View (Type a))
```

The *View* constructor expresses that a type a can be represented as a type b, if there are (pointfree expressions of) 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 *RULE* type expresses that, 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 the source and target type.

Strategy combinators for two-level transformation As for type-preserving strategic rewrite systems, we supply combinators for identity, sequential composition, left-biased choice, repetition, and structural composition of two-level transformation rules:

```
nop :: RULE
nop x = Just (View (Rep Id Id) x)
(▷) :: RULE → RULE → RULE
(f ▷ g) a = do View (Rep t1 f1) b ← f a
               View (Rep t2 f2) c ← g b
               return (View (Rep (Comp b t2 t1) (Comp b f1 f2)) c)
(⊘) :: RULE → RULE → RULE
many :: RULE → RULE
once :: RULE → RULE
```

For conciseness, we show definitions of the first two only. These combinators allow us to combine local, single-step transformations into a single global transformation.

Sample two-level rewrite rules Depending on the scenarios to be addressed, the strategy combinators above can be combined with different sets of single-step rewrite rules to obtain appropriate rewrite systems. Elsewhere we provided sets of rules for *format evolution* and for *hierarchical-relational data mappings* [6]. The following format evolution rules are relevant for our example:

```
inside :: String → RULE → RULE
inside n r (Tag m a) | n ≡ m = do
  View rep b ← r a
  return (View rep (Tag m b))
inside _ _ _ = Nothing
type Query b = ∀a . Type a → PF (a → b)
addfield :: Type b → Query b → RULE
addfield b f a = Just (View (Rep (Id△(f a)) Fst) (Prod a b))
assocr :: RULE
assocr (Prod (Prod a b) c) = return $ View rep (Prod a (Prod b c))
```

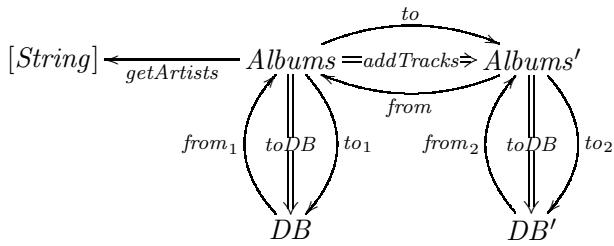


Fig. 2. The *Albums* format is mapped to a relational database both before and after an evolution step that adds track information.

$$\textbf{where } rep = Rep\ ((Comp\ (Prod\ a\ b)\ Fst\ Fst) \triangle (Snd \times Id))\ ((Id \times Fst) \triangle (Comp\ (Prod\ b\ c)\ Snd\ Snd))$$
$$assocr\ \_ = mzero$$

The *inside* combinator applies its argument rule under a given tag. The *addfield* combinator adds a new field, whose value is obtained by querying existing data. The *assocr* rule associates a nested product to the right. Below we will introduce further rules where necessary.

Staged access to data migration functions The data migration functions encapsulated in a *View* can be accessed using a simple *staged* approach, for which we can define the following two functions:

$$showType :: View\ (Type\ a) \to String$$
$$unView :: View\ (Type\ a) \to Type\ b \to Maybe\ (PF\ (a \to b), PF\ (b \to a))$$

Note that the target type representation must be fed to *unView*, but in general this type is only available dynamically, after computing it. Hence the staging. In the first stage, we apply the transformation to obtain b 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 demonstrate the use of this staged approach in interpreter sessions.

Note that this staged approach is not our only alternative for making use of the dynamically computed result type and migration functions. Instead, we can make judicious use of existential and universally quantified types to model dynamic types and dynamically typed values and functions, in a type-safe way [1]. With these, migration functions can be computed and used in a single stage.

6 Coupled transformation

Now we will demonstrate how the rewrite systems introduced in the previous section allow us to address the motivating example of a coupled transformation described in Section 2. An overview of transformations involved is show in Figure 2.

The album example involves lists and finite maps. We need to extend the type of point-free expressions with primitive operations on these data types:

data $PF\ a$ **where**

```
...
Listmap :: PF (a → b) → PF ([a] → [b])
Mapmap :: PF (b → c) → PF (Map a b → Map a c)
Elems :: PF (Map a b → [b])
Fromlist :: PF ([a] → Map Int a)
```

Fromlist builds a map out of a list, by assigning ascending integer keys to its elements. We do not show the straightforward extension of *eval* for these primitives. Of course these functions could be written in terms of more fundamental list and map functions, such as folds, but that would unnecessarily complicate calculations. Getting the right compromise between expressiveness and ease of calculation is one of the most challenging tasks when choosing the set of primitives for a particular application scenario.

Given these primitives, we can write the *getArtists* query in abstract syntax:

```
getArtists :: PF (Albums → [Artist])
getArtists = Listmap (Comp typeof Snd Snd)
```

To allow simplification of function representations involving lists and maps, we must also add new simplification rules to the point-free rewrite system. Some of the relevant laws concerning these primitives are:

listmap $f \circ$ listmap $g =$ listmap $(f \circ g)$	Listmap-Fusion
mapmap $f \circ$ mapmap $g =$ mapmap $(f \circ g)$	Mapmap-Fusion
listmap $f \circ$ elems $=$ elems $\circ$ mapmap f	Elems-Map
elems $\circ$ fromlist $=$ id	Elems-Fromlist

The encoding of these laws as rewrite rules is straightforward. Listmap-Fusion, for example, is encoded as follows (again assuming right-association of compositions):

```
listmap_fusion :: Rule
listmap_fusion _ (Comp (List a) (Listmap f) (Listmap g)) =
  return (Listmap (Comp a f g))
listmap_fusion _ (Comp (List a) (Listmap f) (Comp b (Listmap g) h)) =
  return (Comp b (Listmap (Comp a f g)) h)
listmap_fusion _ _ = mzero
```

Together with *prods* these laws were incorporated in a rewrite system *optimize* that simplifies point-free expressions involving products, lists and finite maps.

Using the combinators defined in the previous section we can define a rule *addTracks* that evolves an *Album* by adding track information.

```
addTracks = once addInside ▷ many (once assocr)
  where addInside = inside "Album" (addfield (List String) getTracks)
        getTracks :: Query [String]
        getTracks (Prod String _)
          = Comp String (Fun "graceNote" graceNote) Fst
```

Notice the use of *inside* to restrict the application of *addfield* to a type represen-

tation tagged with `"Album"`. The *graceNote* function (whose definition is omitted) uses the ASIN field to lookup track titles in some external data source such as the internet. After the field addition, the *assocr* rule is applied exhaustively to bring the resulting nested tuple into the desired right-associated form.

First transformation step With these definitions in place, we can perform migration of the query in accordance with the first transformation step, as show in the following interpreter interaction. First we apply *addTracks* to the type representation of albums to obtain the expected view:

```
> let Just vw1 = addTracks albums
> showType vw1
List (Tag "Album" (
        Prod String (Prod String (Prod String (List String)))))
```

With the new type representation it is now possible to obtain the migration functions, and build the new query by composing *getArtists* with *from*.

```
> let albums' = List (Tag "Album" (Prod ... (List String)))))
> let Just (to, from) = unView vw1 albums'
> let getArtists' = Comp albums getArtists from
> getArtists'
List.map (snd ∘ snd) ∘
List.map fst ∘
List.map ((id × fst)△(snd ∘ snd)) ∘
List.map (id × ((id × fst)△(snd ∘ snd))) ∘ id
```

In this composed query, we can recognize the original query (first line), the removal of track information (second line), and the reversal of right-associating the nested tuple (last two lines).

The simplification of this composed query uses Listmap-Fusion and various laws on products to obtain:

```
> rewrite optimize getArtists'
List.map (fst ∘ snd ∘ snd)
```

This simplified query directly selects the artist field from each album.

Second transformation step The second transformation step involves a strategy called *toDB* for hierarchical-relational mapping [6]. One of its ingredient rules relevant for this example refines lists to maps using the *Fromlist* function:

```
listmap :: Rule
listmap (List a) = Just (View (Rep Fromlist Elems) (Map Int a))
listmap _        = mzero
```

Following similar steps as above we first obtain the type representation of the database refinement of *Albums'*.

```
> let Just vw2 = toDB albums'
> showType vw2
Prod (Map Int (Prod String (Prod String String)))
```

$(Map\ (Prod\ Int\ Int)\ String)$

The migration functions are then used to compose a new query (see Figure 2).

$>$ **let** $db' = Prod\ (Map\ Int\ ...)\ (Map\ (Prod\ Int\ Int)\ String)$
$>$ **let** $Just\ (to_2, from_2) = unView\ vw2\ db'$
$>$ **let** $getArtistsDB = Comp\ albums\ getArtists\ (Comp\ albums'\ from\ from_2)$
$>\ getArtistsDB$
$List.map\ (snd \circ snd) \circ List.map\ fst \circ List.map\ ((id \times fst) \triangle (snd \circ snd)) \circ$
$List.map\ (id \times ((id \times fst) \triangle (snd \circ snd))) \circ id \circ List.map\ id \circ id \circ elems \circ$
$Map.map\ (id \times (id \times (id \times elems))) \circ id \circ id \circ$
$Map.map\ ((fst \circ fst) \triangle (snd \times id)) \circ Map.map\ ((fst \circ fst) \triangle (snd \times id)) \circ$
$id \circ ljoin \circ (id \times njoin) \circ id \circ id \circ id \circ id \circ$
$(Map.map\ ((id \times fst) \triangle (snd \circ snd)) \times id) \circ id$

Due to the complexity of the *toDB* strategy, this query is very long and inefficent. It basically converts the two tables back into a list of albums (using some primitive join operations on maps), and only then it applies the original query. However, after simplification we get the expected result.

$>\ rewrite\ optimize\ getArtistsDB$
$elems \circ Map.map\ (snd \circ snd) \circ fst$

This optimized query ignores the second table, with track information, and obtains the list of artists directly from the first table.

More transformations We have shown how our approach to coupled transformations allows migration of queries on a datatype to queries on a refinement of that datatype. But the rewrite systems that we introduced to support coupled transformations allow us to address many other scenarios. To get a flavour of the possibilities, consider a scenario where *Albums* is mapped to a relational database both before and after the evolution step that adds track information (see Figure 2).

Remember that refinement steps are required to satisfy the equation $from \circ to = id$. In many interesting cases, this law can automatically be *proved* by calculation using our rewrite system. For example, to prove the correctness of the refinement that tranforms *Albums* into DB' one must prove the following equality:

$$from \circ from_2 \circ to_2 \circ to = id$$

Our rewrite system successfully manages to rewrite the left hand side of this equality into id, thus proving this equality.

The composition $to_2 \circ to \circ from_1$ migrates a database without track information, holding a single table of albums, into a database which holds an additional table with track information. If used as such, this migration function first converts the original database into a list of albums, adds track information, and rebuilds the database again with the additional table. This is extremely inefficient because in the new database the first table is just a copy of the existing one. Only the second table with the track information must be created from scratch. By automatic simplification of this composition, a more *direct* migration can be computed, which by-passes the

intermediate list format:

$$(list2map \circ elems)\triangle$$
$$(unnjoin \circ list2map \circ elems \circ Map.map\ (list2map \circ graceNote \circ fst))$$

Note that this expression contains two occurrences of the expression $list2map \circ elems$, that basically rebuilds a map by assigning fresh sequencial keys to its range values. However, if the original database is obtained by migration of a list of albums its keys must already be sequencial and this expression will not create a different map. In fact, each refinement induces the invariant $to \circ from = id$ on the target type, assuming that its data results from migration of the original type. In the case of the first table such invariant is precisely $list2map \circ elems = id$, and thus the above expression could be further simplified into the following expression.

$$id\triangle(unnjoin \circ Map.map\ (list2map \circ graceNote \circ fst))$$

As expected, the database migration boils down to adding a new table (using function *graceNote* and a nested join operation on maps), and leaving the existing table as it is. In essence, we have computed a database migration from the document migration *to*.

7 Related work

Related work concerning coupled transformations that involve format and instance transformation, but not program transformation, is provided in [6] and recapitulated only briefly. We focus on related approaches that (also) take program transformation into account.

Two-level transformation Format evolution and data mappings are important examples of coupled transformation of data format and instances, identified by Lämmel *et al* [16,17]. Our approach to such two-level transformations ([6], Section 5) is based on data refinement theory [20,21].

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 [5,4]. Their approach involves generative and transformational techniques to transform data manipulation statements of legacy information systems, but is limited to information preserving transformations on procedural statements (basically: insert, delete, update). The approach abstracts over various languages (Cobol, Codasyl, Sql), but falls short of formalization and generalization. Transformations are not reported to involve fusion.

Program transformation in calculational form Several systems have been developed for performing program transformation in calculational form using fusion laws. Among these, MAG [19] and Yicho [13] are prominent, but both are targeted towards Haskell programs written in the pointwise style. In order to cope with fusion laws for generic recursion patterns both resort to advanced higher-order matching algorithms. We do not need such techniques because our recursive functions are limited to very specific patterns, such as maps, for which fusion is easier to encode.

A disadvantage of the MAG system is that it uses a fixed strategy to apply the transformation rules, while Yicho provides some basic strategy combinators.

In previous work [7] the first author presented a rewriting system for simplifying point-free expressions, which was used to optimize expressions resulting from a program transformation tool that translates pointwise Haskell code into point-free style. The main improvement of the system presented in Section 4 is typing: we can now use type representations to guide the rewriting process and rewrite rules are guaranteed to be type-safe. In his introductory book to Haskell programming [2], Bird presents a functional calculator that can also be used to simplify point-free expressions. Unfortunately, the expressions are not typed and, likewise to MAG, it uses a fixed rewriting strategy, which makes it difficult to apply in our scenario.

Alternatives to GADTs Our solution relies heavily on GADTs both for type representations and for type-directed and type-safe point-free rewrite rules. Although convenient, GADTs are not essential for these particular tasks. For example, it is possible to encode type representations in Haskell using existential quantification [1,3]. Type-directed rewrite rules could in principle be encoded also with type-classes using the techniques described in [18]. It would be interesting to see whether all ingredients of our solution to coupled rewriting could be realized with similar elegance without resorting to GADTs.

8 Concluding remarks

We have shown that the combination of type-changing rewrite systems for two-level transformation with type-directed, type-preserving rewrite systems for program calculation can capture essential characteristics of coupled transformations. Haskell's type system, including GADTs, have been instrumental in operationalizing such rewrite systems in an elegant and type-safe manner. This approach can been seen as the beginnings of a conceptual framework for coupled transformations [14].

Benefits and limitations Our rewrite systems guarantee type-safety, but for other important formal properties external proofs may be necessary. For instance, the required properties of the *to* functions (total and injective) and *from* functions (surjective), and their composition ($from \circ to = id$) must be respected when defining each type-changing rewrite rule, though proofs of *instantiations* of the latter can be generated by our type-preserving rewrite system. Likewise, each type-preserving rewrite rule on point-free expressions must be proven to be semantics preserving. But, once the properties are proven for individual rules and proven to be preserved by the basic combinators, any system composed from them inherits those properties.

Termination and confluence are also for the responsibility of the programmer. The strategy combinators of both rewrite systems give ample control over which rules are applied, how often, where, and in what order, but reasoning about their combined effect must be done externally.

The inclusion of pointwise functions via the *Fun* constructor must be done with care, lest formal properties of rewrite systems are compromised. Inclusion of the impure *graceNote* function, for example, can be done safely under the assumption that

multiple invocations will produce the same result and have the same termination behaviour. In general, impure functions will not satisfy such assumptions.

Future work As mentioned in Section 6, invariants can be calculated for types targeted by refinement. We intend to explore extension of rule sets with these invariants to drive normalization further. We believe that our techniques for coupled transformations can equally be beneficial in related areas such as bi-directional programming with lenses [10]. In particular, we intend to design rewrite systems for checking and inferring types of bi-directional programs, optimizing them and proving their properties. To support migration of structure-shy queries, we would like to extend our point-free expressions with generic programming primitives, and to extend our type-preserving rewrite system with corresponding laws [23]. Work is underway to apply our rewrite systems to XML and SQL systems.

Implementation All Haskell code shown is part of an implementation that contains a wide range of rules and strategies for coupled transformation. The implementation is available through the authors' homepages under the name 2LT.

References

[1] A.I. Baars and S.D. Swierstra. Typing dynamic typing. In *ICFP '02: Proc. 7th ACM SIGPLAN Int. Conf. Functional programming*, pages 157–166. ACM Press, 2002.

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

[3] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. ACM SIGPLAN 2002 Haskell Workshop*, pages 90–104. ACM Press, 2002.

[4] A. Cleve and J.-L. Hainaut. Co-transformations in database applications evolution. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 399–411. Springer, 2006.

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

[6] A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In *Formal Methods, International Symposium of Formal Methods Europe, 2006, Proceedings*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[7] A. Cunha, J. Sousa Pinto, and J. Proença. A framework for point-free program transformation. In *Selected papers 17th Int. Workshop on Implementation and Application of Functional Languages*, LNCS. Springer, 2006. To appear.

[8] R. di Cosmo and D. Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Proc. of the 20th Int. Conf. on Automata, Languages and Programming (ICALP'93)*, volume 700 of *LNCS*, pages 645–656. Springer, 1993.

[9] D. Dougherty. Some lambda calculi with categorical sums and products. In Claude Kirchner, editor, *Proc. 5th Int. Conf. on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 137–151. Springer, 1993.

[10] J.N. Foster et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, New York, NY, USA, 2005. ACM Press.

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

[12] R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In M. Hagiya and P. Wadler, editors, *Proc. 8th Int. Symp. on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

[13] Z. Hu, T. Yokoyama, and M. Takeichi. Program optimizations and transformations in calculational form. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 136–164. Springer, 2006.

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

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

[16] 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.

[17] R. Lämmel and E. Meijer. Mappings make data processing go 'round. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 165–214. Springer, 2006.

[18] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

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

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

[21] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Asp. Comput.*, 2(1):1–23, 1990.

[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] F. Reig. Generic proofs for combinator-based generic programs. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5, pages 17–32. Intellect, 2006.

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