Universidade do Minho

Relatório de Estágio
da
Licenciatura em Engenharia de Sistemas e Informática

Exploring the 2LT Design Space

Hugo José Pereira Pacheco

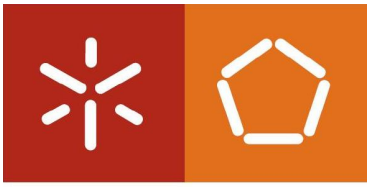
38204

Departamento de Informática da Universidade do Minho

Supervisores:
Alcino Cunha - Dept. de Informática da Universidade do Minho
José Nuno Oliveira - Dept. de Informática da Universidade do Minho

Braga, 2007

About this document...



The present document describes an internship entitled *Exploring the 2LT Design Space*, carried out in the Theory and Formal Methods research group from the Department of Informatics at the University of Minho.
This project is part of a larger research project about Two Level Transformations, and was supervised by Alcino Cunha and José Nuno Oliveira.
Text and layout were written in LaTeX, the document preparation system for the TeX typesetting program.
Images were produced using *Omnigraffle Pro*.

Omnigraffle Pro is a registered trademark from the *The Omni Group*.

Acknowledgments

I must begin by thanking professor José Nuno for the seamlessly passionate classes and earnest discussions that made me embark on the formal methods "boat".

Who also deserve the most acknowledgment are Alcino Cunha, my supervisor, and Joost Visser for the sustained faith in my abilities and all the knowledge they have shared with me in the last two years.

Finally, I dedicate this work to every person who has ever taught me something valuable for life during the long road to become a software engineer.

“There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.“

C. A. R. Hoare

Abstract

Computing is full of situations where one wants to transform a data format into a different format. Transforming a data type implies *coupled transformations* of the data instances conforming to that type. Moreover, to yield model interoperability, *bidirectional transformations* capable of translating between models are required. A formal approach to bidirectional coupled transformation of data is provided by *two-level transformations*, based on calculational data refinement, for transformation of abstract software specifications into concrete low-level programs.

This approach to data calculation has been in use at Minho University, and is the foundation of the "2LT bundle" of tools, including an animated tool for performing two-level transformations, that can be fetched from `http://haskell.di.uminho.pt/svn/wsvn/2lt/`.

This thesis is devoted to the study of possible extensions to the "2LT bundle". The first contribution is the extension of the two-level rewrite system to allow explicitly handling abstraction transformations. This solves an important misbehaved termination problem for partial backward transformations. After, we study in detail the rewriting of point-free expressions over fix-point recursive types. A consequence of this study is the implementation of a *one-level rewrite system* for point-free program calculation from scratch, as an instance of a generic rewrite system that supports native type and functor equality. The last contribution is motivated by the limitations of stateless two-level transformations in the preservation of abstract information for backward value transformations. A library is provided for bidirectional transformations based on the *view-update problem*, that encompass statefull transformations from concrete representations into abstract models.

Key-words: Data calculation, Coupled transformations, Bidirectional transformations, Two-level Transformations, Program calculation

Contents

List of Figures

List of Tables

Glossary

DI Departament of Informatics

LESI Degree on Computer Science and Systems Engineering

UM Universsityof Minho

XML Extensible Markup Language

XPath XML Path Language

XPTO XPath Preprocessor with Type-aware Optimization

SQL Structured Query Language

1LT One-level Transformation

2LT Two-level Transformation

PF Point-free

Happy The Parser Generator for Haskell

Parsec A Monadic Parser Combinator Library for Haskell

GHC Glasgow Haskell Compiler

SyB Scrap Your Boilerplate

AST Abstract Syntax Tree

LL(k) Top-down parser for context-sensitive grammars with k look-ahead

LR(k) Bottom-up parser for context-free grammars with k look-ahead

GUI Graphical User Interface

API Application Programming Interface

VDM Vienna Development Method

TH Template Haskell

Chapter 1

Introduction

1.1 Internship Objectives

Data transformations frequently occur in software engineering, whenever interoperability between different data models is pursued, even though the programmer is not always aware of the techniques they comprise.

With the ever growing list of technological offers, "bridging the gap" between technology layers becomes of most importance to ensure sharing of information among software applications.

Data refinement is deeply concerned with data transformation and expresses the relationship between abstract data specifications and concrete implementations.

A formal approach to data transformation by calculation has been studied at Minho University, based on the theoretical concepts of calculational data refinement and point-free program transformation, and enables the formalization of type-safe two-level transformations.

A two-level transformation consists of a type-level transformation of a data format coupled with value-level transformations of data instances corresponding to that format, where relations between data formats are expressed in a point-free algebraic style.

During the last years, research has been done for the purpose of bringing these concepts into pratical implementations, giving birth to the "2LT bundle", a powerful set of tools available from the UMinho Haskell libraries. These are important for mechanizing repetitive tasks and accelerate the data transformations from demonstration examples to scaled-up real-size case-studies.

However, the separate developing of solutions toward different goals imposed the existence of several implementations with distinct features.

We will describe the different branches of the "2LT bundle", in relation to the features we consider of most importance (Table 1.1): the representation of value transformers as point-free expressions, the support for single-recursive types, the concern for partial backward transformations and the encoding of types using type-safe tags.

	Point-Free	Single Recursion	Partiality	Type-safe Tags
2LT Core [COV06]	×	√	×	×
2LT + 1LT [CV07]	√	×	×	×
XML SQL [BCPV07]	×	√	×	×
XPTO [CV06, FP07]	√	×	*n.c.*[a]	√

[a] Not considered

Table 1.1: Previous research on two-level transformations.

The first implementation of type-safe two-level data transformation is reported in [COV06] (that we fatherly called of 2LT Core), and defines how to develop a rewrite system in Haskell that couples type-

level transformations with well-formed value transformations. It also provides a set of transformations for mapping of recursive hierarchical structures into relational data and subsequent migration of values.

After, the coupled transformation of type and values have been generalized for other data artifacts, namely point-free functions that consume or produce such type [CV07]. The transformation of those functions is accomplished by combining them with the value transformers bounded within the two-level transformation. Besides, value transformers also have to be specified as point-free expressions.

Additionally, this article considers the implementation of a one-level rewrite system simplifying for point-free expressions.

The third branch of the "2LT bundle" was proposed in a paper describing the work on equipping the core rewrite system with front-ends for transformation of XML schemas into SQL databases [BCPV07]. An example of data migration is suggested and special emphasis is placed on the creation and manipulation of SQL references.

Then, the research focused in one-level transformations, with the development of point-free laws for structure-shy to structure-sensitive program transformation [CV06], along with the creation of XML and XPath front-ends for specialization of XPath queries into Haskell point-free functions [FP07].

The main objective of this internship is to discover the current problems and relevant extensions to the 2LT project, towards the development of a unified solution for generic program transformation.

Concerning the major flaws in the previous systems (attempt to Table 1.1), most effort was put on implementing 2LT with partiality and on reasoning about point-free expressions over recursive types.

Allied to the serendipity of great discoveries, the remainder of this report will be engrossed with our experiences on exploring the 2LT design space.

1.2 Structure of the Document

Chapter 2 presents in highly detail previous work on 2LT. It starts by defining concepts and continues with an explanation on the representation of types and functions, highlighting the different types of functions.

In 2LT, the backward transformer is a possibly partial function and, therefore, may not be defined for all the concrete domain. Chapter 3 addresses this issue and provides an elegant solution to avoid runtime errors for undefinable transformations: the maybe monad.

In Chapter 4, we explore possible extensions to the existing 1LT rewrite system in order to support rewriting of expressions over fix-point recursive types. Most of the theory backs up on work from Cunha [Cun05] on point-free program calculation. Three approaches are discussed for an Haskell implementation. However, these suffer from some flaws on the representation of functions over types and functors, surpassed by the implementation of a generic rewrite system with native support for type and functor equality. Chapter 5 presents all the design and implementation decisions in the development of this rewrite system.

An actually very "sexy" theme on data transformation and model interoperability are view-update bidirectional transformations (lenses). In Chapter 6, we compare lenses with two-level transformations and converge the differences into a new rewrite system for view-update bidirectional transformations.

Finally, Chapter 7 begins with an overall review of the main contributions of this internship. The report ends with an enumeration of the open issues, must-do improvements, and possible ideas for future work.

Chapter 2

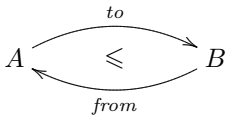
Data Transformations

Data transformations usually encompass conversion from a source data format into a different format. Transforming a data type determines coupled transformations of the data instances conforming to that type, if consistency is to be achieved.

In previous work, we have addressed *two-level transformations* for pairing type-level transformations of data formats with value-level transformations of data instances corresponding to such formats[COV06, BCPV07].

A type-level transformation is coupled with value-level functions *to* and *from*, where the *representation to* is injective and total and the *abstraction from* is surjective and simple, such that:

$$from \circ to \sqsubseteq id_A$$



The ordering of the continuous functions from A to B is determined by the antisymmetric relation:

$$f \sqsubseteq g \Leftrightarrow (f\ x \sqsubseteq g\ x, \forall x \in A)$$

This fact expresses that f is an approximation of g, or in other words, f is less or equally well defined than g if and only if, for every value x in A, $f\ x$ is less or equally well-defined than $g\ x$.

A variety of scenarios can be found for 2LTs. For example, if due to maintenance or re-factoring requirements an XML database schema is adapted, it induces a transformation for XML documents. Like-wise, the evolution of a language's grammar implies an automated conversion of deprecated source code into the new syntax. Both cases are instances of *format evolution by transformation* and contribute to format and document reengineering[LL01].

Similarly, coupled transformations are also involved in *data mappings* [LM06], where distinct data models, generally of different programming paradigms, need to be mapped onto each other for persistence or interoperability reasons. A common example is the mapping between XML tree-structured schemas into SQL relational schemas.

More challenging coupled transformations involve the transformation not only of data instances corresponding to a source data format, but also of other software artifacts, such as programs that consume or produce it[CV07].

Recalling the XML database schema example, the adaptation process would be coupled with the update of XPath queries and programs that generate the database file. New database programs are

calculated by composition with the value-level transformers from the two-level transformation. For simplification purposes, both programs and data transformers will be represented as point-free expressions.

The complex conversion functions derived after combining transformations can be subjected to subsequent simplification using laws for point-free program calculation. This introduces a new concept that we name *one-level transformations*, for simplification of expressions into semantically equivalent expressions. The *simplification by rewriting* of point-free functions is performed through the application of rich a set of algebraic laws for point-free program calculation, strongly based on mathematical equational reasoning[Cun05]. Transformation of point-free programs may follow different approaches for different purposes. In the context of XPath expressions, classified as structure-shy queries, we provide laws for specialization of their generic properties into structure-sensitive point-free functional programs, and *vice versa*[CV06, FP07].

Structure-shy programs are defined generically for different data types, and only specify specific behaviors for a few relevant subtypes. Despite more understandable, concise and reusable, they are less efficient than structure-sensitive programs, that are optimized for structure specific details.

Type-safe rewrite systems have been developed to define and compose data transformations in the functional language Haskell. This chapter is dedicated to explain the representation of types and functions over types that enable the implementation of these rewrite systems.

2.1 Representation of Types

To ensure type-safety in our rewrite systems, a universal type representation of types does not suffice. Both types and values are transformed at the term-level and share the same type context. To achieve this, we will need type-safe representations at the value level, which can be provided by using *generalized algebraic data types* (GADTs), a powerful generalization of Haskell data types [PWW04]. For all parameterized data $Type\ a$, an inhabitant of the following parameterized data type $Type\ a$ is a representation of type a [HLO06].

$$
\begin{aligned}
&\textbf{data}\ Type\ a\ \textbf{where}\\
&\quad One \quad :: Type\ ()\\
&\quad Int \quad :: Type\ Int\\
&\quad Bool \quad :: Type\ Bool\\
&\quad String :: Type\ String\\
&\quad List \quad :: Type\ a \rightarrow Type\ [a]\\
&\quad Prod \quad :: Type\ a \rightarrow Type\ b \rightarrow Type\ (a, b)\\
&\quad Either :: Type\ a \rightarrow Type\ b \rightarrow Type\ (Either\ a\ b)\\
&\quad Func \quad :: Type\ a \rightarrow Type\ b \rightarrow Type\ (a \rightarrow b)\\
&\quad ...
\end{aligned}
$$

Notice that, in this declaration, the type a that parameterizes $Type\ a$ is restricted differently in the result of each constructor. This makes the difference between a GADT and a common Haskell 98 parameterized datatype, where the parameters in the result type must always be unrestricted in all constructors. For a $Type\ a$ definition, the parameter a of each constructor is restricted exactly to the type that the constructor represents, although different constructors can share the same parameter a.

Given a ground type a, it is possible to use the Haskell type system to infer its representation. We can define a class with all representable types.

$$
\begin{aligned}
&\textbf{class}\ Typeable\ a\ \textbf{where}\\
&\quad typeof :: Type\ a
\end{aligned}
$$

Most instances of this class are trivially defined. For example, for integers and functions we have

$$
\begin{aligned}
&\textbf{instance}\ Typeable\ Int\ \textbf{where}\\
&\quad typeof = Int
\end{aligned}
$$

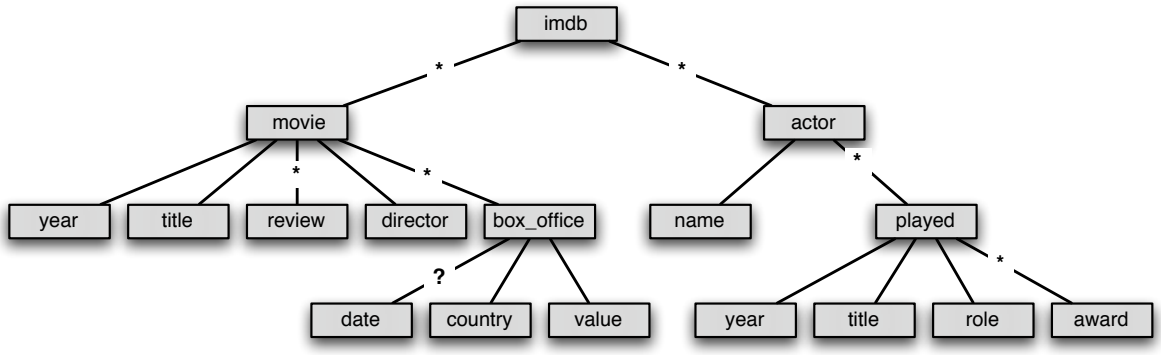


Figure 2.1: A movie database schema, inspired by IMDb (http://www.imdb.com/).

$$\textbf{instance}\ (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \Rightarrow \mathit{Typeable}\ (a \to b)\ \textbf{where}$$
$$\quad \mathit{typeof} = \mathit{Func}\ \mathit{typeof}\ \mathit{typeof}$$

We may define a dynamic type $*$ that hides a type representation inside its definition[LJ05], declared as

$$\textbf{data}\ *\ \textbf{where}\ * :: \mathit{Type}\ a \to a \to *$$

For the type, we also have an equivalent type representation

$$\textbf{data}\ \mathit{Type}\ a\ \textbf{where}$$
$$\quad \ldots$$
$$\quad * :: \mathit{Type}\ a \to \mathit{Type}\ *$$

Type $*$ works as a wrapper and allows the definition of generic functions that can receive arguments of any type. The real type is encapsulated inside $*$ and may be recovered at any time, namely it can be used to guide the definition of a generic function.

Consider the generic function f such that

$$f :: \mathit{Type}\ a \to a \to \mathit{String}$$
$$f\ \mathit{Int}\quad 1\quad = \texttt{"nat1"}$$
$$f\ \mathit{Char}\ \texttt{'a'} = \texttt{"ascii97"}$$

We can apply any generic function f to a dynamically typed value

$$\mathit{applyAny}\ f = \lambda(*\ t\ v) \to f\ t\ v$$

2.1.1 User-defined datatypes

Type allows the definition of basic types, lists, products, sums and functions. In order to represent user-defined datatypes, we define:

$$\textbf{data}\ \mathit{Type}\ a\ \textbf{where}$$
$$\quad \mathit{Data} :: \mathit{String} \to (a \leftrightarrow b) \to \mathit{Type}\ b \to \mathit{Type}\ a$$

New data types can now be easily defined using the *Data* constructor.

You can notice that *Data* requires a value of type $a \leftrightarrow b$, where *Type* b is the encapsulated type and *Type* a the resulting type for the newly defined datatype. *Data* works as a wrapper for a type (similar to

newtype *Imdb* = *Imdb* {*unImdb* :: ([*Movie*], [*Actor*])}
newtype *Movie* = *Movie*{*unMovie* :: (*Title*, (*Year*, ([*Review*], (*Director*, [*BoxOffice*])))))}
newtype *Actor* = *Actor* {*unActor* :: (*Name*, [*Played*])}
...

Here, we represent XML element tags from the schema's type definition from Figure 6.1 with Haskell data types. Each **newtype** defines an XML node, with it's own markup tag. For each node, a function is provided for untagging values of it's type.

Figure 2.2: Haskell datatypes for the schema of Figure 6.1.

an XML node tag), where $\leftrightarrow$ is an embedding-projection pair that converts values from the user-defined type into values of the isomorphic type. The type b is expected to be the sum-of-products representation of the user-defined type a.

Consider the XML schema of Figure 6.1 and the Haskell datatype that represent it on Figure 2.2. The type representation for the IMDb data-type can be defined as follows.

instance *Typeable Imdb* **where**
  typeof = *Data* `"imdb"` (*unImdb* $\leftrightarrow$ *Imdb*) *typeof*

Here, *Typeable* instances are assumed for *Movie* and *Actor*. The instance for *Imdb* is defined on top of instances for it's subtypes.

2.2 Representation of Functions

Analogously to types, we need to represent functions in the same type-safe manner. For this purpose, we resort again to a GADT, allowing function's type-checking for free: impossible or incorrect compositions of functions are checked against Haskell's native type system and rejected. Constructors can be defined for different approaches like point-free, strategic or XPath combinators. Such constructors are explained in [CV06].

Here are some examples of point-free combinators:

data *Type a* **where**
  ...
  PF :: *Type a* $\rightarrow$ *Type* (*PF a*)
data *PF f* **where**
  id :: *PF* $(a \rightarrow a)$
  (.) :: *Type b* $\rightarrow$ *PF* $(b \rightarrow c) \rightarrow$ *PF* $(a \rightarrow b) \rightarrow$ *PF* $(a \rightarrow c)$
  π_1 :: *PF* $((a, b) \rightarrow a)$
  π_2 :: *PF* $((a, b) \rightarrow b)$
  $(\triangle)$ :: *PF* $(a \rightarrow b) \rightarrow$ *PF* $(a \rightarrow c) \rightarrow$ *PF* $(a \rightarrow (b, c))$
  mplus :: *Monoid a* $\rightarrow$ *PF* $((a, a) \rightarrow a)$
  out_a :: *PF* $(a \rightarrow b)$
  mkAny :: *PF* $(a \rightarrow *)$
  fun :: *String* $\rightarrow (a \rightarrow b) \rightarrow$ *PF* $(a \rightarrow b)$
  wrap :: *PF* $(a \rightarrow [a])$
  ...

In the 1LT rewrite system, the specialization of XPath structure-shy queries into point-free functions is encoded via an intermediate transformation into type-unifying generic functions.

Generic combinators as we employ them were introduced with the so-called 'Scrap-your-Boilerplate' approach to generic functional programming[LP03].

There are two classes of generic combinators:

- type-unifying (queries), are defined as generic functions that return a result for a specific type[1];

  type $Q\ r = \forall a\ .\ Type\ a \to a \to r$

- and type-preserving (traversals), that are generic functions that preserve the type of the input[2].

  data T **where** $T :: (\forall a \circ Type\ a \to a \to a) \to T$

We now present the definition for essential generic combinators[3]:

```
data Type a where
  ...
  TP          :: Type T
  TU          :: Type a → Type (Q a)
data PF f where
  ...
  apT_a       :: Type a → PF T → PF (a → a)          -- application
  mkT_a       :: Type a → PF (a → a) → PF T          -- creation
  nop         :: PF T                                -- identity
  ▷           :: PF T → PF T → PF T                  -- sequence
  gmapT       :: PF T → PF T                         -- map over children
  everywhere  :: PF T → PF T                         -- map to every node
  apQ_a       :: Type a → PF (Q r) → PF (a → r)      -- application
  mkQ_a       :: Monoid r → Type a → PF (a → r) → PF (Q r)  -- creation
  ∅           :: Monoid r → PF (Q r)                 -- empty result
  union       :: Monoid r → PF (Q r) → PF (Q r) → PF (Q r)  -- union of results
  gmapQ       :: Monoid r → PF (Q r) → PF (Q r)      -- fold over children
  everything  :: Monoid r → PF (Q r) → PF (Q r)      -- fold over every node
```

TP and *TU* are the type-preserving and type-unifying representations. *Func* and *PF* provide type representations for functions and point-free functions, respectively. *Func* is the previously defined context for functions. The general point-free type context is *PF*.

A nice example of a generic function over point-free expressions is the *eval* function of *PF* expressions:

```
eval :: Type a → PF a → a
eval (PF (Func (Prod a b) a)) π_1          = fst
eval TP                       (everywhere t)   = t ▷ gmapT (everywhere t)
eval (TU a)                   (everything m t) = union m t (gmapQ m (everything m t))
...
```

Also, generic equality for type representations is achieved with *teq*:

```
  -- Data type to express type-equality
data Equal a b where
  Eq :: Equal a a
  -- Type equality generic function
teq :: Type a → Type b → Maybe (Eq a b)
teq One One = Just Eq
teq (List a) (List b) = do
  Eq ← teq a b
  return Eq
```

[1] The result r is assumed to be a monoid, with a *zero* element and associative *plus* operator.
[2] Traversals need to be encoded as a **data**, because T encapsulates the return type a
[3] In our model, a child type corresponds to descending once in the tree-like structure of a type.

```
...
teq _ _ = Nothing
  -- Boolean type equality
teqb :: Type a → Type b → Bool
teqb a b = if (isJust (teq a b)) then True else False
```

XPath combinators are implemented as strategic type-unifying combinators. They were initially defined in terms of strategic combinators by Lämmel [Läm06].

```
data PF f where
  ...
  self     :: PF (Q [*])
  child    :: PF (Q [*])
  desc     :: PF (Q [*])
  descself :: PF (Q [*])
  name     :: String → PF (Q [*])
  /        :: PF (Q [*]) → PF (Q r) → PF (Q r)
```

In XPath, combinators enjoy a very relaxed typing. Selection functions are implemented as aggregation functions for sets of types, but there is no distinction in grouping properties for values of different types. For instance, we may group the results of expressions with different types:

/imdb,1

Additionally, XPath selectors provide an element name based algebra. Thus, selecting a name may involve the selection of different types with the same name.

Hence, sets of values don't have a type distinction, and are represented in our system as $[*]$.

2.2.1 Rewriting Examples

We will now present some examples of expression rewriting with our point-free calculator.

Consider two XPath queries //director and //movie, that collect all non-root elements with names *director* and *movie*, respectively, from an XML document.

First, we convert these expressions into our XPath representation:

```
descself / child / name "director"
descself / child / name "movie"
```

Using the one-level rewrite system for point-free program calculation, the second step is to partially evaluate these structure-shy XPath expressions for the XML schema of Figure 6.1.

The results are two structure-sensitive point-free functions.[4]. The first function collects all *directors* inside *movies* that are themselves under the root element *imdb*:

```
let movies  = π1 ∘ out_Imdb
    director = π1 ∘ π2 ∘ π2 ∘ π2 ∘ out_Movie
in  list (mkAny ∘ director) ∘ movies
```

Analogously, the second function selects all *movies* inside the root element *imdb*:

```
let movies = π1 ∘ out_Imdb
in  list mkAny ∘ movies
```

Remember the need to encapsulate al element results inside the dynamic wrapper $*$, performed by *list mkAny*

[4] We have spitted expressions into different type selectors for improving user readability.

2.3 Representation of Two-level Transformations

Based on the presented representations for types and functions, we can encode a 2LT as a pair of point-free functions that are each other inverse:

$$\textbf{data}\ \mathit{Trans}\ a\ b = \mathit{Trans}\{\mathit{to} :: a \to b, \mathit{from} :: b \to a\}$$

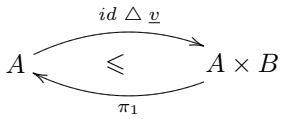
A view encompasses that a type a can be represented by a type b, as witnessed by the value-level functions that convert between these types:

$$\begin{array}{l}\textbf{data}\ \mathit{View}\ a\ \textbf{where}\\ \quad \mathit{View} :: \mathit{Trans}\ a\ b \to \mathit{Type}\ b \to \mathit{View}\ (\mathit{Type}\ a)\end{array}$$

Inside the view, the target type does not escape and is existentially quantified. A transformation rule is defined as the generalization of a view for any input type[5]:

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

Consider an example two-level transformation that adds information to a data model by creating a pair in the target model:



The value-level transformations can be encoded as

$$\mathit{add\_}\pi_2\mathit{\_trans} = \mathit{Trans}\ (\mathit{id} \vartriangle \underline{v})\ \pi_1$$

, and generalized into a rule a combinator that receives the type and value of the added information:

$$\begin{array}{l}\mathit{add\_}\pi_2 :: \mathit{Type}\ b \to b \to \mathit{rule}\\ \mathit{add\_}\pi_2\ b\ v\ a = \mathit{return}\ (\mathit{View}\ \mathit{add\_}\pi_2\mathit{\_trans}\ (\mathit{Prod}\ a\ b))\end{array}$$

2.3.1 Generic Combinators

The 2LT rewrite system relies on smaller type-changing two-level transformation steps that can be combined into powerful rewrite systems, using a set of type-transforming strategic combinators [COV06, BCPV07]. They are similar to the generic combinators presented in Section 2.2 for ordinary functions, 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 list composition of pairs of value-level functions:

$$\begin{array}{l}\quad \text{-- Pair of identity value-level tranformations}\\ \mathit{id\_trans} :: \mathit{Trans}\ a\ a\\ \mathit{id\_trans} = \mathit{Trans}\ \mathit{id}\ \mathit{id}\\ \quad \text{-- Sequential composition of value-level transformations}\\ \mathit{comp\_trans} :: \mathit{Trans}\ a\ b \to \mathit{Trans}\ b\ c \to \mathit{Trans}\ a\ c\\ \mathit{comp\_trans}\ f\ g = \mathit{Trans}\ (\mathit{from}\ f \circ \mathit{from}\ g)\ (\mathit{to}\ g \circ \mathit{to}\ f)\end{array}$$

[5] The Haskell *Maybe* native type designates monads partiality (**data** *Maybe* a = *Nothing* | *Just* a). In our representation, *Maybe* $A \cong A + 1$.

```
list_trans :: Trans a b → Trans [a] [b]
list_trans r = Trans (List.map (to r)) (List.map (from r))
```

Using these combinators for pairs of value-level functions, we can define new two-level combinators:

```
  -- Unit for sequential composition
nop :: Rule
nop t = return (View id_trans t)
  -- Sequential composition.
(▷) :: Rule → Rule → Rule
(f ▷ g) a = do (View r b) ← f a
  (View s c) ← g b
  return (View (comp_trans r s) c)
  -- Repeat until failure, zero or more times
many :: Rule → Rule
many r = (r ▷ many r) ||| nop
```

Note that rule application may fail, since it is partial, what implies the combinator to fail accordingly. Since *Rule* has a monadic codomain, it is easy to encode sequentiation.

The either combinator extends the monadic sum into our model. For two rules *r1* and *r2*, it always applies *r1*, and tries to apply *r2* if *r1* fails:

```
(|||) :: Rule → Rule → Rule
(r1 ||| r2) c = r1 c `mplus` r2 c
```

Well-known generic combinators have also been implemented, such as *one* (applies a rule to one of the children of the a type) and *all* (applies a rule to all of the children of the same generic type). Despite extensive, their definition is very straightforward. For example, for products, *one* applies a rule to any of the children of the products, when *all* applies the same rule to all of the product's children.

Example cases for *one* combinator are:

```
one :: Rule → Rule
one r Int = r Int
one r (List a) = do
  View s b ← r a
  return (View list_trans (List b))
...
```

More challenging combinators are the ones that descend onto the functorial structure of type representations, such as *once* (applies a rule once, at arbitrary depth) and *everywhere* (applies a rule everywhere).

Based on the definition of *one*, we can define *once*:

```
once r :: Rule → Rule
once r = r ||| one (once r)
```

For more detailed information on strategic combinators for two-level transformations, please refer to the previous papers [COV06, CV07, BCPV07].

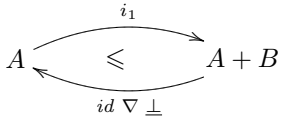
Chapter 3

Two-Level Transformations with Partiality

In the refinement theory, the abstraction relation in a two-level transformation must be surjective and simple, where a simple relation is a possibly partial function.

Although partiality is admitted in the backward value transformation, it is not enforced in the function declaration, what makes valid type refinements to be inconsistent for some values inside it's domain.

A typical example is the *addalt* refinement[1]:



It can be easily shown that this pair of functions establishes a valid refinement:

$$\left[\begin{array}{ll} & \mathit{from} \circ \mathit{to} \\ \Leftrightarrow & \quad \{\, \text{definitions of } \mathit{to} \text{ and } \mathit{from} \,\} \\ & (\mathit{id} \nabla \underline{\perp}) \circ i_1 \\ \Leftrightarrow & \quad \{\, +\text{-Cancel}, \mathit{nat}\text{-Id} \,\} \\ & \mathit{id} \end{array}\right.$$

Despite being a valid refinement, the abstraction function *from* is only well-defined for the range of *to*, defined as the set

$$\{\mathit{to}\ (x) : x\ \in A\}$$

For that reason, *from* is partial for values of the added alternative and may generate runtime errors for undefined cases.

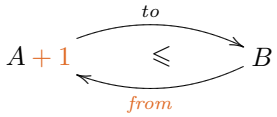
Moreover, throwing of runtime errors aborts the execution of the backward transformation, and compromises the possibility of successful termination. An example will be presented after defining explicit partiality.

3.1 Adding Explicit Partiality

In order to address this issue, we can add explicit partiality to the backward value transformation ($\mathit{from} :: B \to A + 1$). The fundamental of refinements can now be stated as

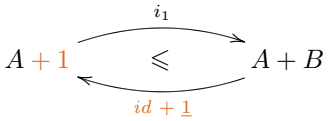
$$\mathit{from} \circ \mathit{to} \ \sqsubseteq\ i_1$$

[1] $\perp$ is the undefined value for any type. $\underline{\perp}$ defines the constant function that always returns $\perp$.



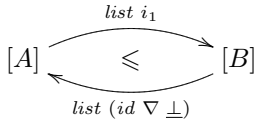
The "partialization" of a refinement consists in adding a fail alternative to its backward function *from*, converting it into a total function. For the cases when the original *from* function is undefined, the new total function returns the added alternative.

Again, for the *addalt* refinement:

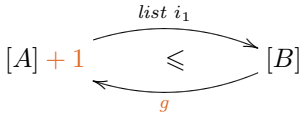


$$\left[\begin{array}{ll} & \mathit{from} \circ \mathit{to} \\ = & \quad \{\, \text{definitions of } \mathit{to} \text{ and } \mathit{from} \,\} \\ & (\mathit{id} \ + \ \underline{1}) \circ i_1 \\ = & \quad \{\, +\text{-Cancel}, \ \mathit{nat}\text{-Id}, \ +\text{-Def} \,\} \\ & i_1 \end{array}\right.$$

Straightaway, we emphasize the benefits of explicit partiality in the sequentiation of transformations. Assume the application of *addalt* inside a list:



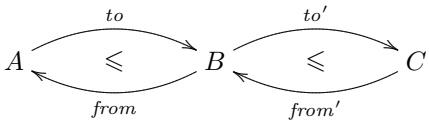
As long as the non-partialized version of *addalt* is considered, in case the backward function of *addalt* $(\mathit{id} \ \nabla \ \underline{\perp})$ fails, *list* $(\mathit{id} \ \nabla \ \underline{\perp})$ will fail accordingly. However, if we consider *addalt* as a partial refinement, the 2LT from $[A]$ to $[B]$ becomes:



Here, explicit partiality in *addalt* allows the backward transformation g to safely recover from the error state, since the partiality of $\mathit{id} \ \nabla \ \underline{\perp}$ is absorbed as the nil value for lists[2]. Intuitively, this could not have been achieved if the abstraction function of *addalt* had previously failed to execute.

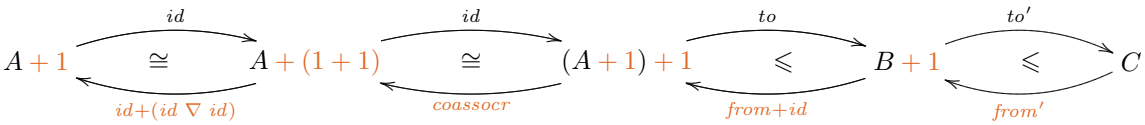
The extra sum implied by partiality in the backward direction will increase the complexity of combining type transformers. It will frequently require the use of associativity , commutativity and distributivity isomorphisms of sums and products [Cun05].

Remember the composition diagram of refinements,



[2]The definition of g is provided furtherly, when studying the partialization of generic combinators for lists.

You can see how the complexity of the backward transformation increases in the partialized version[3]:



3.1.1 Generic Combinators

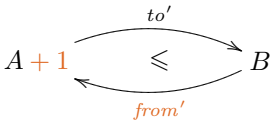
The implemented generic combinators for two-level transformations include top-level combinators that apply an argument rule to the root element of a type (such as composition) and highly recursive combinators that descend onto the functorial structure of type representations.

Two standard non-recursive combinators are *one* (applies argument rule to one of the child types) and *all* (applies argument rule to all child types), that apply transformations to child elements of a type. They are defined generically for any type, but their implementation is specific for each type pattern.

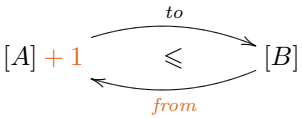
In order to prove that the partialization of these combinators still generates valid refinements, we will need to prove the refinement properties for each of their patterns.

Consider the proof for lists (this proof is valid for both *one* and *all*, since the type representation of $[A]$ has a single child type A).

Assuming that



is a valid refinement r, then



is also a valid refinement *one* r if

$$from \circ to \sqsubseteq i_1$$

The backward function *from* of refinement *one* r can be encoded by mapping $from'$ over $[B]$, resulting a list of partialized A elements. $+\,1$ optionality can be removed from inner list elements by representing them as empty lists (generated by *nil*), imposing the type lifting of $[A]$ to $[[A]]$, furtherly canceled by *concat*. The full definition is as follows:

$$[B] \xrightarrow{list\ from'} [A+1] \xrightarrow{list\ (wrap\ \nabla\ nil)} [[A]] \xrightarrow{concat} [A] \xrightarrow{i_1} [A]+1$$

Following this definition, we can derive the proof for list refinements:

[3] *coassocr* names the right associative property of sums.

$$
\begin{bmatrix}
 & \mathit{from} \circ \mathit{to} \\
= & \quad \{ \text{ definition of } \mathit{to} \text{ and } \mathit{from} \} \\
 & (i_1 \circ \mathit{concat} \circ \mathit{list}\ (\mathit{wrap} \bigtriangledown \mathit{nil})) \circ \mathit{list}\ \mathit{from}' \circ \mathit{list}\ \mathit{to}' \\
= & \quad \{ \mathit{list}\text{-Fusion}; \mathit{from}' \circ \mathit{to}' = i_1 \} \\
 & i_1 \circ \mathit{concat} \circ \mathit{list}\ (\mathit{wrap} \bigtriangledown \mathit{nil}) \circ \mathit{list}\ i_1 \\
= & \quad \{ \mathit{list}\text{-Fusion}; \bigtriangledown\text{-Cancel} \} \\
 & i_1 \circ \mathit{concat} \circ \mathit{list}\ \mathit{wrap} \\
= & \quad \{ \mathit{concat}\text{-Cancel}; \mathit{nat}\text{-Id} \} \\
 & i_1
\end{bmatrix}
$$

Proofs for other supported types are available in Annex B.

3.1.2 Haskell implementation

A partial two-level transformation between two types is encoded in Haskell as[4]

$$\textbf{data}\ \mathit{Trans}\ a\ b = \mathit{Trans}\{\mathit{to} :: \mathit{PF}\ (a \to b), \mathit{from} :: \mathit{PF}\ (b \to \mathit{Either}\ a\ \mathit{One})\}$$

For instance, the value transformations for the *addalt* refinement are encoded as:

$$\mathit{addalt\_trans} = \mathit{Trans}\ i_1\ (\mathit{id}\ +\ \underline{1})$$

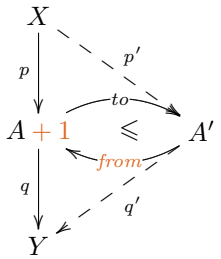
The enconding of views and rules remain the same. Thus, the rule generalization of *addalt* can be defined as:

$$\mathit{addalt} :: \mathit{Type}\ b \to \mathit{Rule}$$
$$\mathit{addalt}\ b\ a = \mathit{Just}\ (\mathit{View}\ \mathit{addalt\_trans}\ (\mathit{Either}\ a\ b))$$

3.2 Coupled Transformation of XPath queries

When migrating a data type, it is common to have some other computational artifacts, such as generators and queries, that need to remain consistent after the evolution. Moreover, the computation of these artifacts results in new artifacts bound to the evolved type.

Consider the following diagram for partialized coupled transformations, where p is a data producer that generates values of type A, and q is a query that consumes values of type A:



Assuming that Y is a monoid, we can derive equivalent functions for A':

$$p' = \mathit{to} \circ p$$
$$q' = (q \bigtriangledown \mathit{zero}) \circ \mathit{from}$$

[4]Note that we are using the point-free representation for value-transformation functions. This will be of most importance for the simplification of these functions.

In both possible scenarios, after the transformation is calculated, we can use the 1LT rewrite system to simplify the composed point-free expressions, removing all possible references to the intermediate type A.

Consequently, it is of great importance to find function rewriting laws capable of canceling the usage of sums and products combinators introduced by the partialization of the transformation.

Examples are the canceling rules for sums left associativity (*coassocl*) and left injections (i_1):

- To associate to the left the left injection of some type is equivalent to injecting that type twice to the left

$$coassocl \circ i_1 = i_1 \circ i_1$$

$$\begin{array}{ccc} A & & A \\ \downarrow{\scriptstyle i_1} & & \downarrow{\scriptstyle i_1} \\ A + (B + C) & = & A + B \\ \downarrow{\scriptstyle coassocl} & & \downarrow{\scriptstyle i_1} \\ (A + B) + C & & (A + B) + C \end{array}$$

- To associate to the left the result of left injecting the right element of a sum is equivalent to left injecting the result of the sum.

$$coassocl \circ (f \ + \ g \circ i_1) = i_1 \circ (f \ + \ g)$$

$$\begin{array}{ccc} A + B & & A + B \\ \downarrow{\scriptstyle f+(g \circ i_1)} & & \downarrow{\scriptstyle f+g} \\ C + (D + E) & = & C + D \\ \downarrow{\scriptstyle coassocl} & & \downarrow{\scriptstyle i_1} \\ (C + D) + E & & (C + D) + E \end{array}$$

More rules may be found in Annex A.

3.2.1 XPath Examples

We will now study a coupled transformation of a schema and a data selector expressed in XPath. For such a scenario, we will:

- refine the schema given a two-level transformation;
- use our one-level rewrite system to specialize the XPath query into a structure-sensitive point-free expression conforming to the schema;
- compose the specialized point-free expression with the point-free abstraction function from the refinement;
- use the one-level rewrite system to simplify the resulting point-free expression and remove references to the original schema.

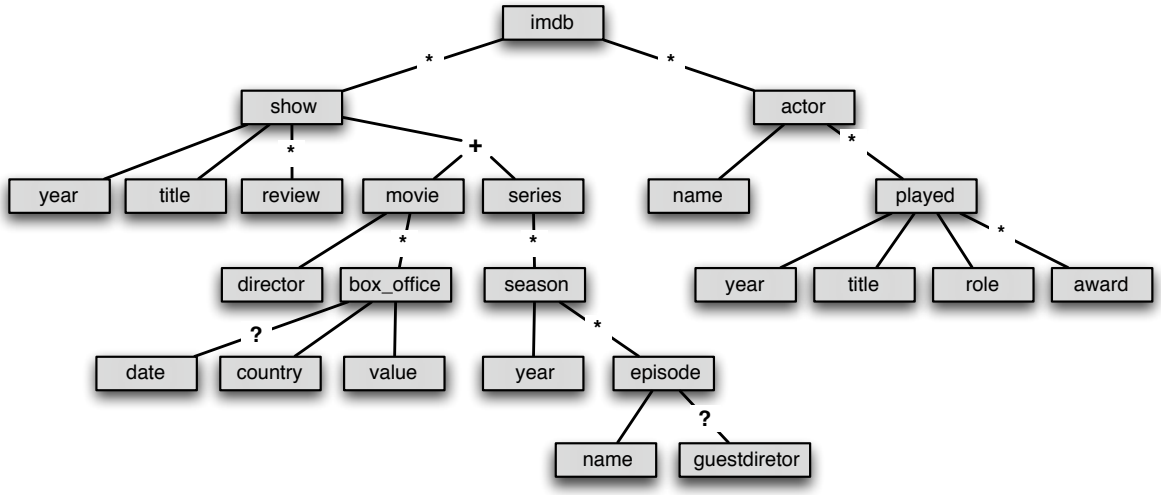


Figure 3.1: An evolution of the original movie database example (Figure 6.1).

Suppose the following transformation for the Imdb schema example presented in Chapter 2 (Figure 6.1): change the type named *movie* to *show*, define *movie* as the product of *director* and *box_offices* and add an alternative *series* to *movie*. The resulting schema is represented in Figure 3.1.

Since we represent schema elements as Haskell native types, new definitions must be encoded for each resulting data type that differs or does not exist in the original format.

For this specific transformation, the structure of *movie* and *imdb* change, and thus require new types[5]:

$$\textbf{newtype}\ Eimdb'1 \quad = Eimdb'1\ \{unEimdb'1 :: ([Eshow], [Eactor])\}$$
$$\textbf{newtype}\ Emovie'1 = Emovie'1\{unEmovie'1 :: (Edirector, [Ebox\_office])\}$$

Also, new *Typeable* instances must be implemented:

$$\textbf{instance}\ Typeable\ Eimdb'1\ \textbf{where}$$
$$typeof = Data\ \texttt{"imdb"}\ \ (unEimdb'1 \leftrightarrow Eimdb'1) \quad typeof$$
$$\textbf{instance}\ Typeable\ Emovie'1\ \textbf{where}$$
$$typeof = Data\ \texttt{"movie"}\ (unEmovie'1 \leftrightarrow Emovie'1)\ typeof$$

Now, we can define the type evolution, remembering that we need to decapsulate the types before applying a transformation[6]:

$$addseries :: Rule$$
$$addseries = atData\ \texttt{"imdb"}\ (unData \triangleright once\ (atData\ \texttt{"movie"}\ (unData \triangleright once\ (when\ isMovie$$
$$(dataMovie'1 \triangleright addalt\ series)) \triangleright dataShow)) \triangleright dataImdb'1)$$
$$\textbf{where}$$
$$isMovie :: Type\ a \rightarrow Bool$$
$$isMovie = teqb\ (typeof :: Type\ (Edirector, [Ebox\_office]))$$
$$series \quad = typeof :: Type\ Eseries$$

The *atData* combinator applies an argument rule according to the name of the root type:

[5] Note that the new declarations, although distinct, will maintain the same type name.

[6] *teqb* is the boolean function for type equality. *when* is a two-level combinator that applies a given transformation only if the boolean predicate succeeds.

$$
\begin{aligned}
&atData :: String \to Rule \to Rule\\
&atData\ s\ r\ x@(Data\ n\ \_\ \_) \mid n \equiv s = r\ x\\
&atData\ s\ r\ \_ = Nothing
\end{aligned}
$$

unData provides data type decapsulation:

$$
\begin{aligned}
&unData :: Rule\\
&unData\ x@(Data\ n\ \_\ t) = Just\ (View\ (Trans\ out_a\ (i_1 \circ in_a))\ t)\\
&unData\ \_\ \_ = Nothing
\end{aligned}
$$

By the other side, *dataShow*, *dataMovie′1* and *dataImdb′1* are default encapsulators for their respective types. Default encapsulators succeed only when applied to the particular structure of the types they encapsulate, what is assured by *teq* type equality. Assume the definition of *dataImdb′1* as an example:

$$
\begin{aligned}
&dataImdb'1 :: Rule\\
&dataImdb'1\ a = \mathbf{do}\\
&\quad Eq \leftarrow teq\ a\ (typeof :: ([Eshow], [Eactor]))\\
&\quad \mathbf{let}\ imdb = typeof :: Type\ Eimdb'1\\
&\quad return\ (View\ (Trans\ in_a\ (i_1 \circ out_a))\ imdb)
\end{aligned}
$$

Collect all directors

Remember the specialization of the XPath expression //director from 2.2.1:

$$
\begin{aligned}
&\mathbf{let}\ movies\ \ = \pi_1 \circ out_{Imdb}\\
&\quad\ \ director = \pi_1 \circ \pi_2 \circ \pi_2 \circ \pi_2 \circ out_{Movie}\\
&\mathbf{in}\ \ list\ (mkAny \circ director) \circ movies
\end{aligned}
$$

Composed with the abstraction function of the evolution and after simplification with 1LT, the query that collects all directors becomes:

$$
\begin{aligned}
&\mathbf{let}\ shows\ \ \ \ \ \ \ \ = \pi_1 \circ out_{Imdb'1}\\
&\quad\ \ movieorseries = \pi_2 \circ \pi_2 \circ \pi_2 \circ out_{Show}\\
&\quad\ \ director\ \ \ \ \ \ \ = \pi_1 \circ out_{Movie'1}\\
&\mathbf{in}\ \ concat \circ list\ ((wrap \circ mkDyn \circ director\ \nabla\ nil) \circ movieorseries) \circ movies
\end{aligned}
$$

Remark that the new structure-sensitive expression has no partiality, despite *from* being partial. This can be explained by evidencing that both director and movie elements, on which the original query depends, remained unchanged after the evolution.

Consequently, the resulting expression is independent from the original schema, in the sense that it needs no knowledge on the structure of original elements to query the evolved schema.

Nevertheless, this property also means that the result of applying the query to the evolved type is equivalent to composing the original query with the abstraction function.

Collect all movies

Recapitulate the specialization of the XPath expression //movie from 2.2.1:

$$
\begin{aligned}
&\mathbf{let}\ movies = \pi_1 \circ out_{Imdb}\\
&\mathbf{in}\ \ list\ mkAny \circ movies
\end{aligned}
$$

Composed with the abstraction function of the evolution, the query that collects all movies becomes:

$$
\begin{aligned}
&\mathbf{let}\ shows\ \ \ \ \ \ \ \ \ \ = \pi_1 \circ out_{Imdb'1}\\
&\quad\ \ year\ \ \ \ \ \ \ \ \ \ \ \ = \pi_1 \circ out_{Show}\\
&\quad\ \ title\ \ \ \ \ \ \ \ \ \ \ \ = \pi_1 \circ \pi_2 \circ out_{Show}
\end{aligned}
$$

$$
\begin{array}{ll}
\quad reviews & = \pi_1 \circ \pi_2 \circ \pi_2 \circ out_{Show} \\
\quad movieorseries & = \pi_2 \circ \pi_2 \circ \pi_2 \circ out_{Show} \\
\quad directororbox\_offices & = out_{Movie'1} \\
\quad originalmovie & = inMovie \\
\textbf{in} \;\; concat \circ list\ ((wrap \circ mkAny \circ originalmovie \bigtriangledown nil) \circ distr \circ (year \bigtriangleup (i_1 \bigtriangledown i_2 \circ \underline{1}) \circ distr \circ (title \bigtriangleup \\
\quad (i_1 \bigtriangledown i_2 \circ \underline{1}) \circ distr \circ (reviews \bigtriangleup (i_1 \circ directororbox\_offices \bigtriangledown i_2 \circ \underline{1}) \circ moviesorseries)))) \circ shows
\end{array}
$$

This time, the transformed expression is far more complex, since the distributivity combinator *distr* could not be cancelled. It happens because, in order to return a *movie* element of the original type *Movie*, the added alternative *series* cannot exist, otherwise a value of type *Movie* fails to be created.

Applying the XPath expression over the evolved schema, a much simplier function results from the specialization:

$$
\begin{array}{ll}
\textbf{let} \; shows & = \pi_1 \circ out_{Imdb'1} \\
\quad movieorseries & = \pi_2 \circ \pi_2 \circ \pi_2 \circ out_{Show} \\
\textbf{in} \;\; concat \circ list\ ((wrap \circ mkAny \bigtriangledown nil) \circ movieorseries) \circ shows
\end{array}
$$

This expression does not depend on the original schema and is straightforward in the selection of XML elements with name *movie*, but of type *Movie′1*.

3.3 Relaxing type-safety

The representation of data types as type-safe GADTs enables type-safety in two-level transformations. However, it represents more difficulty in writing simple transformation rules, since transforming the structure of a datatype requires converting it into a new type constrained by the transformed structure. In other words, we need to "open" the opaque type, apply the desired transformation, and perhaps encapsulate it again inside a new datatype containing the result of the transformation.

This implies that the user must know *a priori* the result type for the transformation, since the new encapsulating data type must be defined in the rewrite code.

Replacing data representations with tags

An alternative is to replace the *Data* encapsulator as a mere string identifier that carries the name of the type, as expressed by the following *Tag* constructor:

$$
\begin{array}{l}
\textbf{data}\ Type\ a\ \textbf{where} \\
\quad ... \\
\quad Tag :: String \rightarrow Type\ a \rightarrow Type\ a
\end{array}
$$

Now, we can recurse inside type representations when defining generic combinators for two-level transformations such as *one* and *all*, what in pratical terms means that strategies can be applied inside types without the need to decapsulate them first.

Remark Note that a tag-based type representation can be pretty-printed into an Haskell data type with the corresponding *Typeable* instances. However, this indirection costs an extra compilation step.

XPath Examples Revisited

Tag-based representations cannot be derived from haskell types using *Typeable* class instances, because they do not hold an embedding-pair projection between the representation and the represented datatype and, thus, have no notion on the latter.

This forces the user to write type representations by hand, since no *Typeable* instance can be inferred:

$$
\begin{array}{ll}
series & = List\ season \\
season & = Prod\ year\ (List\ episode)
\end{array}
$$

```
year      = Tag "year" Int
episode   = Prod name (Either guestdirector One)
name      = Tag "name" String
guestdirector = Tag "guestdirector" String
```

However, the evolution turns out to be much more straightforward, since there is no type encapsulation, at the cost of type-safeness:

```
addseries :: Rule
addseries = once (changeTag "movie" "show") ▷ once (when isMovie (tag "movie'1" ▷ addalt series))
  where
  isMovie :: Type a → Bool
  isMovie (Prod (Tag "director" _ _) (List (Tag "box_office" _ _))) = True
  isMovie _                                                         = False
```

Considering the previous XPath sample expressions, the resulting point-free expressions are equal, except for the fact that no type encapsulators/decapsulators (*in* and *out*) are required.

As a general conclusion, tag representations do not alter the expression complexity or the dependability on the source model, in terms of expression simplification, but greatly improve the usability and efficiency of the transformations used.

As long as we can generate a type-safe representation from a tag-based representation, it it possible to convert between both representations. However, when pursuing the coupled transformation of functions, non-type-safe to type-safe translation implies generating type encapsulators/decapsulators for newly defined types inside related point-free expressions.

3.4 Summary

In this chapter, we have extended the two-level rewrite system in order to add explicit partiality to abstraction functions in refinements.

We have studied the impact of partiality on coupled transformations in terms of expression complexity and dependability on the source data model, for two different user-defined data-type representations (2.1).

Dependability on the source model expresses how dependant is the composed expression on the original format, in order to return values of the type of the elements for which it was originally specialized.

Chapter 4

Type-safe Rewriting with Single Recursion

Previously, the 1LT rewrite system has been introduced for type-preserving transformation of functions over non-recursive type representations. In this chapter, we will dig into extending the type representation to single-recursive types, sustaining the same approach of mapping theoretical concepts into pratical program implementations.

Functors are used in mathematics for expressing mappings between categories. In functional programming, a functor can be expressed as a type constructor.

Most recursive types can de defined as the fix-point of a polynomial functor. These result from the following set of functors and combinators:

$$
\begin{array}{lll}
\textbf{newtype}\ \mathit{Id}\ a & = \mathit{Ident}\ \{\mathit{unIdent} :: a\} & \textbf{deriving}\ \mathit{Eq} \\
\textbf{newtype}\ K\ b\ a & = \mathit{Const}\{\mathit{unConst} :: b\} & \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ (g \oplus h)\ a & = \mathit{Inl}\quad (g\ a) \mid \mathit{Inr}\ (h\ a) & \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ (g \otimes h)\ a & = \mathit{Pair}\ (g\ a)\ (h\ a) & \textbf{deriving}\ \mathit{Eq}
\end{array}
$$

The identity functor Id encapsulates value instances for recursive sub-terms of that same type. The constant functor K encapsulates values of a given type; $\oplus$ and $\otimes$ are liftings of Haskell's native sums and products.

The *base functor* that captures the signature of a regular data type T is denoted F_T. A recursive data type can be defined as the fix-point of its *base functor* F_T. For each polynomial T, there exists a base functor F_T such that $T = \mu F_T$, and inverse functions $in_T : F_T\ T \to T$ and $out_T : F_T\ T \to T$[Rey77].

Recursion patterns can be seen as high-order functions that encapsulate typical forms of recursion. The most famous recursion pattern is iteration, also called *catamorphism*, where constructors for recursive datatypes are repeatedly consumed by arbitrary functions:

$$
\begin{array}{ccc}
\mu F & \xrightarrow{out_{\mu F}} & F\ (\mu F) \\
{\scriptstyle (\!|g|\!)_{\mu F}}\downarrow & & \downarrow{\scriptstyle F\ (\!|g|\!)_{\mu F}} \\
A & \xleftarrow[g]{} & F\ A
\end{array}
$$

Consider the following recursive definitions for Haskell native lists and $\mathbb{N}_0^+$ natural numbers, and their base functors:

$$
\begin{array}{ll}
\textbf{data}\ [a] & = \mathit{Nil} \mid \mathit{Cons}\ a\ [a] \\
F_{[a]} & = K\ () \oplus (K\ a \otimes \mathit{Id})
\end{array}
$$

$$\begin{aligned}&\textbf{data}\ Nat = Zero \mid Succ\ Nat\\ &F_{Nat} \quad\ = K\ () \oplus Id\end{aligned}$$

An example of a *catamorphism* is the *length* function that counts the number of elements in a list:

$$\begin{aligned}&length : [\,a\,] \to Nat\\ &length = (\!|in_{Nat} \circ (id\ +\ \pi_2)|\!)_{[a]}\end{aligned}$$

An *anamorphism* resembles the dual of iteration and, hence, defines the inverse of a catamorphism - instead of consuming recursive types, it produces values of those types:

$$\begin{array}{ccc}\mu F & \xleftarrow{in_{\mu F}} & F\ (\mu F)\\ {\scriptstyle [\![g]\!]_{\mu F}}\uparrow & & \uparrow{\scriptstyle F\ [\![g]\!]_{\mu F}}\\ \mu\ F & \xrightarrow[g]{} & F\ (\mu F)\end{array}$$

length is a classical example of a function that can be encoded both as a catamorphism from lists or as an anamorphism to naturals. Consequently, we may define:

$$\begin{aligned}&length : [\,a\,] \to Nat\\ &length = [\![(id\ +\ \pi_2) \circ out_{[a]}]\!]_{Nat}\end{aligned}$$

More sophisticated recursion patterns are *paramorphisms*, that supply the gene of a *catamorphism* with a recursively computed copy of the input. Note that $(id \triangle id)$ duplicates the substructure of the input:

$$\begin{array}{ccccc}\mu F & \xrightarrow{out_{\mu F}} & F\ (\mu F) & \xrightarrow{F\ (id \triangle id)} & F\ (\mu F \times \mu F)\\ {\scriptstyle (\!|g|\!)_{\mu F}}\downarrow & & & & \downarrow{\scriptstyle F\ ((\!|g|\!)_{\mu F} \times id)}\\ R & \xleftarrow[g]{} & & & F\ (R \times \mu F)\end{array}$$

A standard example of a paramorphism is the factorial function. The recursive definition for the factorial of a nonzero natural number n is given by multiplying n by the factorial of $n-1$[1]:

$$\begin{aligned}&fact : Nat \to Nat\\ &fact = (\!|one \nabla\ mult \circ (id \times succ)|\!)_{Nat}\end{aligned}$$

For more information on recursion pattern and functional laws on them, please refer to [Cun05, MFP91].

4.1 Rewriting SYB Combinators

The 1LT rewrite system provides a rich set of algebraic laws for transformation of point-free and generic program combinators (namely for SyB and XPath).

In these laws, “apply to every node“ generic traversals can be unrolled via sequentiation of “apply to children“ combinators.

[1] *one* returns the identity value of multiplication, where *mult* multiplies two natural numbers.

Unrolling recursion implies specializing the combinator for a specific type, and consists in applying an argument transformation at the top and explicitly invoking its recursive definition (that may itself be unrolled) for all the children.

Consider the following laws for recursion elimination in generic strategies:

$$apT_{\mu F}\ (everywhere\ f) = apT_{\mu F}\ f \circ mapT\ F\ \mu F\ (everywhere\ f) \qquad every\text{-}\textsc{ApplyT}$$
$$apQ_{\mu F}\ (everything\ m\ f) = mplus\ m \circ (apQ_{\mu F}\ f \times mapQ\ F\ m\ \mu F\ (everything\ r\ f)) \qquad every\text{-}\textsc{ApplyQ}$$

Here, $mapT$ and $mapQ$ instantiate "apply to children" combinators with application to a concrete functor and type:

$$apT_{\mu F}\ (gmapT\ f) = mapT\ F\ \mu F\ f \qquad map\text{-}\textsc{ApplyT}$$
$$apQ_{\mu F}\ (gmapQ\ m\ f) = mapQ\ F\ m\ \mu F\ f \qquad map\text{-}\textsc{ApplyQ}$$

Accordingly, they are defined as:

$$\left.\begin{array}{l} mapT\ Id\ a\ f = apT_a\ f \\ mapT\ (K\ b)\ a\ f = apT_b\ f \\ mapT\ (f \oplus g)\ a\ h = mapT\ f\ a\ h\ +\ mapT\ g\ a\ h \\ mapT\ (f \otimes g)\ a\ h = mapT\ f\ a\ h \times mapT\ g\ a\ h \end{array}\right\} \quad mapT\text{-}\textsc{Def}$$

$$\left.\begin{array}{l} mapQ\ Id\ m\ a\ f = apQ_a\ f \\ mapQ\ (K\ b)\ m\ a\ f = apQ_b\ f \\ mapQ\ (f \oplus g)\ m\ a\ h = mapQ\ f\ m\ a\ h \nabla mapQ\ g\ m\ a\ h \\ mapQ\ (f \otimes g)\ m\ a\ h = mplus\ m \circ (mapQ\ f\ m\ a\ h \times mapQ\ g\ m\ a\ h) \end{array}\right\} \quad mapQ\text{-}\textsc{Def}$$

However, unrolling laws were formulated for non-recursive data types and are not suitable for recursive type definitions. Recursive references violate the model of tree-structured data and force recursion elimination laws into infinite loops.

For instance, suppose a type that holds the information of an employee, where the latter has a name and may supervise another employee. The type and functor representations become

$$Employee = Name\ \times\ (Employee + 1)$$

and

$$F_{Employee} = K\ Name \otimes (Id \otimes K\ One)$$

respectively.

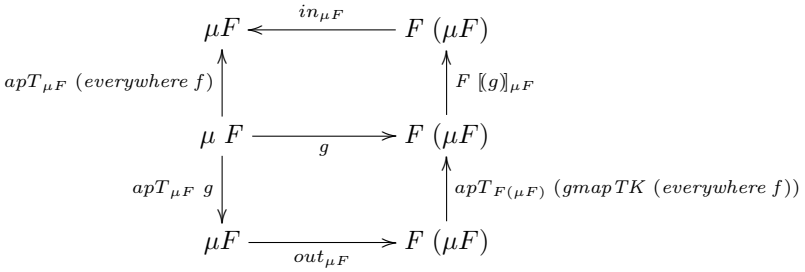
Specialization of a generic traversal for the *Employee* type generates an infinite recursion tree:

$$\left[\begin{array}{ll} & apQ_{Employee}\ (everything\ m\ f) \\ = & \quad \{\ every\text{-}\textsc{ApplyT}\ \} \\ & mplus\ m \circ (apQ_{Employee}\ f \times mapQ\ (K\ Name \otimes (Id \oplus K\ One))\ m\ Employee\ (everything\ m\ f)) \\ = & \quad \{\ mapQ\text{-}\textsc{Def}\ \} \\ & mplus\ m \circ (apQ_{Employee}\ f \times mplus\ m \circ (apQ_{Name}\ (everything\ f) \times \\ & (apQ_{Employee}\ (everything\ f) \nabla (apQ_{One}\ (everything\ f))))) \\ = & \quad \{ \ldots \} \\ & \ldots \end{array}\right.$$

4.1.1 Using recursion patterns

To solve this problem, we will encode traversals using recursion patterns.

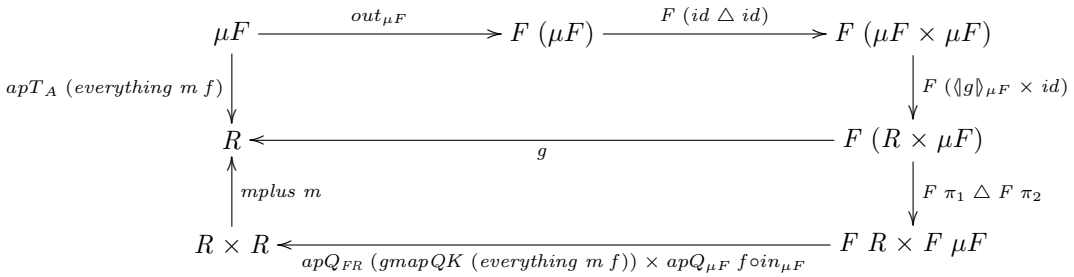
For the encoding of the *everywhere* type-preserving combinator we use an *anamorphism*, that iterates through the type recursive calls in a topdown approach:



The gene of the anamorphism applies the argument transformation f to the input and "opens" its functor definition with $out_{\mu F}$. After, it applies *everywhere* f to all the child types.

It is important to guarantee that *everywhere* f is not applied to recursive sub-terms because it would mean to apply f a second time, since it has already been applied to them by $apT_{\mu F}\ f$. This is attained with the usage of the combinator *gmapTK* in the place of *gmapT*: it denotes a small variant of *gmapT* that does not apply the argument transformations to recursive calls.

The *everything* type-unifying combinator is encoded as a *paramorphism*:



The gene of the paramorphism starts by splitting the functor definitions for the result type R and input type μF. It returns the monoid sum of two values. The first value results from applying *everything* f to all children of the result type, remembering that *gmapQK* mimics the behavior of *gmapTK* for type-unifying transformations.

Secondly, the second value is originated from applying the transformation f to a functor recursive representation of the input type. Remark that the functor encapsulator $in_{\mu F}$ needs to be applied before the actual transformation.

4.2 Implementation in Haskell

In this section, we will discuss techniques and approaches to implement this extension in Haskell, in order to support expression rewriting.

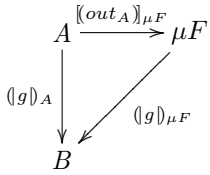
Similarly to types and functions, a GADT representation can be created for functor definitions:

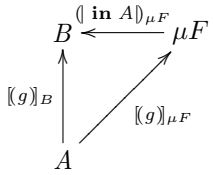
```
data Fctr f where
  Id  :: Fctr Id
  K   :: Type a → Fctr (K a)
  (⊕) :: Fctr g → Fctr h → Fctr (g ⊕ h)
  (⊗) :: Fctr g → Fctr h → Fctr (g ⊗ h)
```

The fix-point operator μF will be implemented as a functor encapsulator:

```
newtype μ f = InMu{outMu :: f (μ f)}
```

Although isomorphic in theory, μF and a datatype with functor f are not the same type in Haskell. Therefore, we need to define purely syntactic translations from types to their μF representations. The following diagrams express how catamorphisms and anamorphisms relate to μF:





Every non-native data type is represented as a recursive data type, even if it does not recurse, helping in reducing the number of combinators and rules implied. Following this minimalist approach, we may redefine the representation type as follows[2]:

```
data Type a where
  Int    :: Type Int
  Bool   :: Type Bool
  Char   :: Type Char
  String :: Type String
  One    :: Type ()
  Set    :: Type a → Type (Set a)
  Map    :: Type a → Type b → Type (Map a b)
  Data   :: FunctorOf f a ⇒ String → Fctr f → Type a
  ApF    :: Fctr f → Type a → Type (f a)
```

The constructor *Data* now typifies some type a by taking a functor representation f and standard conversions between f and some type a. This scheme will be later explained when the *FunctorOf* class is presented.

Type Classes

> *In computer science, a type class is a type system construct that supports ad-hoc polymorphism. This is achieved by adding constraints to type variables in parametrically polymorphic types. Such a constraint typically involves a type class T and a type variable a, and means that a can only be instantiated to a type whose members support the overloaded operations associated with T. Thus, type class constraints implement a form of bounded polymorphism.*
>
> *from Wikipedia*

Type classes were initially conceived for implementing overloading arithmetics and properties over types, such as equality.

Thence, we will recur to type classes for defining properties of type and functor representations.

Recall the *Typeable* class for deriving type representations from a ground type. We can convey this concept to functor representations by declaring a *Fctrable* class that infers a functor representation from an Haskell functor, and instances for our constructs:

```
class Fctrable f where
  fctrof :: Fctr f
instance Fctrable Id where
  fctrof = Id
instance Typeable a ⇒ Fctrable (K a) where
  fctrof = K typeof
...
```

[2] *ApF* expresses a type context for functor application.

Also, when defining an Haskell type for holding name information of another data type

```
data Name a = Name String
  -- Transforms a function from string into a function from Name
applyName :: (String → b) → Name a → b
applyName f (Name x) = f x
```

, a *Nameable* class ensues:

```
class Nameable a where
  nameof :: Name a
```

Although classes allow the implementation of properties on representations, our real goal is to take advantage of their ability to constrain multiple type parameters: it turn possible the mediation among different artifacts and, applied to our specific scenario, it provides the semantics for defining implicit translations between representations.

The first application example is to define the class *FunctorOf* that, given a functor f and a type a, produces two functions in_a and out_a that fold up $f\ a$ values into a values, and vice-versa:

```
class (Functor f, Fctrable f) ⇒ FunctorOf f d | d → f where
  in :: f d → d
  out :: d → f d
```

Remark Note that the functional dependency $d \rightarrow f$ implies that a functor f is unique for each type d.

The *FunctorOf* class provides an elegant solution to avoid the use of μF. When recursing over a type, we may "expose" its functor structure progressively, once for each recursive call, while μF-based folding and unfolding encompasses full translation of values, $[\![out_A]\!]_{\mu F}$ and $(\!|in_B|\!)_{\mu F}$.

Recapitulating the *length* example and maintaining the same encoding for *in* and *out*, the anamorphism and catamorphism correspond exactly to the theoretical definition without the need to lift operators:

$$length\_cata = (\!|in_{Nat} \circ (id\ +\ \pi_2)|\!)_{[a]}$$
$$length\_ana\ = [\![(id\ +\ \pi_2) \circ out_{[a]}]\!]_{Nat}$$

The *Typeable* class backs up on the definition of *Data*, that consumes a string identifier (corresponding to the type name) and a functor representation conformant to the constructed type, given a pair of functions *in* and *out* that are each other's inverse (wrapped within the *FunctorOf* class). Intuitively, a generic *Typeable* instance can be defined for types supporting the *FunctorOf* and *Nameable* classes:

```
instance (Nameable d, FunctorOf f d) ⇒ Typeable d where
  typeof = applyName (λn → Data n fctrof) (nameof :: Name d)
```

This far, we have presented a representation for single-recursive types. The representation of functions is exploited in the next subsections, according to three different approaches.

4.2.1 Functors Approach

As stated before, all types, even if not recursive, are represented as functors, what is equivalent to say that type sums and products are lifted to the functors $\oplus$ and $\otimes$.

For example, if we previously, had some function $f = id\ \nabla\ \pi_1$, where $id :: x \rightarrow x$ and $\pi_1 :: (a, b) \rightarrow a$, its type definition would be $f :: Either\ a\ (a, b) \rightarrow a$.

However, according to our recent instructions to represent Haskell products and sums as lifted functors, we need to redefine the type of f such that $f :: \forall x\ .\ (K\ a \oplus (K\ a \otimes K\ b))\ x \rightarrow (K\ a)\ x$.

Continuing, the lifted version of f can be calculated by lifting its subexpressions[3]: $\underline{f} = id\ \underline{\nabla}\ \underline{\pi_1}$.
Consequently, the patterns for functions over sums and products also have to use explicit functors:

data $PF\ a$ **where**
...
$\underline{\pi_1} :: PF\ ((f \otimes g)\ a \to f\ a)$
$\underline{\pi_2} :: PF\ ((f \otimes g)\ a \to g\ a)$
$\underline{\triangle} :: PF\ (a \to g\ b) \to PF\ (a \to h\ b) \to PF\ (a \to (g \otimes h)\ b)$
$\underline{\times} :: PF\ (f\ a \to h\ b) \to PF\ (g\ a \to i\ b) \to PF\ ((f \otimes g)\ a \to (h \otimes i)\ b)$
$\underline{i_1} :: PF\ (f\ a \to (f \oplus g)\ a)$
$\underline{i_2} :: PF\ (g\ a \to (f \oplus g)\ a)$
$\underline{\nabla} :: PF\ (f\ a \to b) \to PF\ (g\ a \to b) \to PF\ ((f \oplus g)\ a \to b)$
$\underline{+} :: PF\ (f\ a \to h\ b) \to PF\ (g\ a \to i\ b) \to PF\ ((f \oplus g)\ a \to (h \oplus i)\ b)$

After all, it should be possible to maintain the original type definition of f, as long as we want type and functor interoperability.
For that reason, lifting expressions into functors does not suffice: there must exist combinators capable of translating types to equivalent functors, and vice-versa:

data $PF\ a$ **where**
...
$apK :: PF\ (a \to b) \to PF\ (K\ a\ c \to K\ b\ c)$
$mkK :: PF\ (a \to K\ a\ c)$
$unK :: PF\ (K\ a\ c \to a)$
$apId :: PF\ (a \to b) \to PF\ (Id\ a \to Id\ b)$
$mKId :: PF\ (a \to Id\ a)$
$unId :: PF\ (Id\ a \to a)$

Acknowledge f' as an approximation of f that has equal semantics, but follows a functor representation:

$$f' :: (K\ a \oplus (K\ a \otimes K\ b))\ x \to a)$$

Therefore, conversion of $\underline{f}$ into f' implies "de-sugaring" of the lifted functor result $(K\ a)\ x$ into the raw type a. To accomplish that, we simply compose unK with $\underline{f}$:

$$f' = unK \circ \underline{f}$$

Although our implementation already provides enough expressivity to interchange between types and functors, functor application is not as general as we would like to.
Consider the definition of a paramorphism, where the type to which the functor is applied needs to be represented as a functor (the case of sums and products):

$$\langle\!|g|\!\rangle_A : (F\ (B \times A) \to B) \to A \to B$$

Due to this restriction, functor composition is required:

data $(g \odot h)\ a = Comp\ (g\ (h\ a))$ **deriving** Eq
data $Fctr$ **where**
...
$(\odot) :: Fctr\ f \to Fctr\ g \to Fctr\ (f \odot g)$

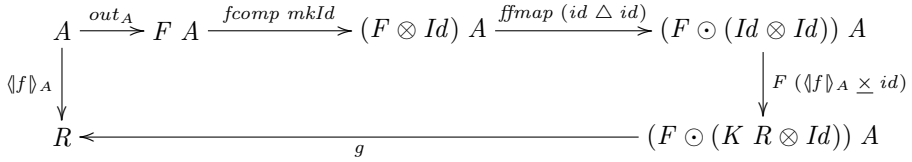
Some combinators for manipulating composition come trivially:

[3]Note that $\underline{id} = id$

data $PF\ a$ **where**
...
$ffmap :: PF\ (f\ a \to g\ b) \to PF\ ((h \odot f)\ a \to (h \odot g)\ b)$
$fcomp :: PF\ (a \to g\ b) \to PF\ (f\ a \to (f \odot g)\ b)$
$compf :: PF\ (g\ b \to a) \to PF\ ((f \odot g)\ b \to f\ a)$

In order to employ functor composition in the paramorhism definition, the product $A \times B$ has to be lifted into a functor $(K\ B \otimes K\ A)\ X$, that can be composed with a functor F, returning $(F \odot (K\ B \otimes K\ A))\ X$. However, B is the recursive substructure passed as argument to the gene of the anamorphism, and therefore is encoded as Id to signal its difference from the result type A.

The following diagram renders a more detailed explanation of functor composition application in paramorphisms:



At last, the concrete specification of a paramorphism is of the form:

data $PF\ a$ **where**
...
$(\!|g|\!)_a :: FunctorOf\ f\ a \Rightarrow PF\ ((f \odot (K\ b \otimes Id))\ a \to b) \to PF\ (a \to b)$

Employees Example

Consider an example of a recursive datatype for employee hierarchies[4]:

newtype $Employee = Employee\ \{unEmployee :: (Maybe\ Employee, Job, FirstName, LastName)\}$
newtype $Job = Job\ \{unJob :: String\}$
newtype $FirstName = FirstName\{unFirstName :: String\}$
newtype $LastName = LastName\ \{unLastName :: String\}$

We can write a type-unifying query to collect all employees (values of type $Employee$):

$apQ_{Employee}\ (everything\ (mkQ_{Employee}\ wrap))$

The result of specializing this query would be:

$(\!|mplus \circ ((mkK \circ unId \mathbin{\underline{\nabla}} mzero) \circ \underline{\pi_1} \circ compf\ unK \circ \underline{\pi_1} \mathbin{\underline{\triangle}} mkK \circ wrap \circ in_{Employee} \circ compf\ unId \circ \underline{\pi_2}) \circ (ffmap\ \pi_1 \mathbin{\underline{\triangle}} ffmap\ \pi_2)|\!)_{[Employee]}$

However useful, the extra point-free combinators increase the syntactic complexity of expressions and force the user to have notion of type and functor conversions: this complexity motivated us to reject this approach.

[4] Adapted from the online *.NET Framework Developer's Guide* (http://msdn.microsoft.com/library/), for representing employee hierarchies.

4.2.2 Combinators Replication Approach

This approach is as simple as duplicating for types and functors all the point-free combinators that involve sums and products.

We will then re-add the *Prod* and *Either* type representations that were removed when defining all types as recursive:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Type}\ a\ \textbf{where} \\
\quad \ldots \\
\quad \mathit{Prod} \quad :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b) \\
\quad \mathit{Either} :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (\mathit{Either}\ a\ b) \\
\textbf{data}\ \mathit{PF}\ a\ \textbf{where} \\
\quad \ldots \\
\quad \pi_1 \quad :: \mathit{PF}\ ((a, b) \to a) \\
\quad \pi_2 \quad :: \mathit{PF}\ ((a, b) \to b) \\
\quad \triangle \quad :: \mathit{PF}\ (a \to b) \to \mathit{PF}\ (a \to c) \to \mathit{PF}\ (a \to (b, c)) \\
\quad \times \quad :: \mathit{PF}\ (a \to c) \to \mathit{PF}\ (b \to d) \to \mathit{PF}\ ((a, b) \to (c, d)) \\
\quad \mathit{prodf} \quad :: \mathit{PF}\ (f\ a \to c) \to \mathit{PF}\ (g\ a \to d) \to \mathit{PF}\ ((f \otimes g)\ a \to (c, d)) \\
\quad \mathit{fprod} \quad :: \mathit{PF}\ (a \to f\ c) \to \mathit{PF}\ (a \to g\ b) \to \mathit{PF}\ ((a, b) \to (f \otimes g)\ c)
\end{array}
$$

Since the number of equivalent combinators increased, this is directly reflected in the number of rule patterns for such combinators, depending on number of these combinators in the pattern expression.

A good example to highlight this behavior is the ×-Absorption rule, that merges application (×) and creation (△) for products. All the valid combinations of semantically identical combinators must be admitted[5]:

$$
\left.
\begin{array}{l}
(f \times g) \circ (h \triangle i) = f \circ h \triangle g \circ i \\
(f \mathbin{\underline{\times}} g) \circ (h \mathbin{\underline{\triangle}} i) = f \circ h \mathbin{\underline{\triangle}} g \circ i \\
(f \ `\mathit{prodf}`\ g) \circ (h \mathbin{\underline{\triangle}} i) = f \circ h \triangle g \circ i \\
(f \ `\mathit{fprod}`\ g) \circ (h \triangle i) = f \circ h \mathbin{\underline{\triangle}} g \circ i
\end{array}
\right\} \quad \times\text{-Absorption}
$$

Employees Example

Following similar steps as above, the specialization of the generic query results in:

$$(\!|\mathit{mplus} \circ ((\mathit{unId} \mathbin{\underline{\nabla}} \mathit{mzero}) \circ \underline{\pi_1} \mathbin{\underline{\times}} \mathit{wrap} \circ \mathit{in}_{\mathit{Employee}}) \circ (\mathit{fmap}\ \pi_1 \mathbin{\underline{\triangle}} \mathit{fmap}\ \pi_2)|\!)_{[\mathit{Employee}]}$$

Although similar in semantics, this representation of point-free expressions is much clearer and close to theory. However, the necessary difference between sums and products expressions on types and functors implies having duplicate combinators for types and functors, as long as *Either a b* is not the same Haskell type as $(K\ a \oplus K\ b)\ x$. This extra cost in user's coding time and number of rewrite laws makes the approach unsuitable for reasonably-sized projects.

4.2.3 Functor-representation Class Approach

In this last approach, the solution is to define class-driven implicit conversion between values of a type representation and a functor that lifts that same representation. Such class will be responsible for the compatibility between types and functors by converting all functors into their equivalent type before applying an expression.

The class name *Rep* stands for representation of functors and, for each functor, there is only one type representation.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Rep}\ f\ a\ b \mid f\ a \to b\ \textbf{where} \\
\quad \mathit{to} \quad :: f\ a \to b
\end{array}
$$

[5] For some binary function $f :: a \to b \to c$, $a\ `f`\ b$ denotes the infix application of $f\ a\ b$.

```
  from     :: b → f a
  apF      :: Fctr f → Type a → Type b
  mapT     :: Fctr f → Type a → PF T → PF (b → b)
  mapTK    :: Fctr f → Type a → PF T → PF (b → b)
  mapQ     :: Fctr f → Monoid r → Type r → Type a → PF (GQ r) → PF (b → r)
  mapQK    :: Fctr f → Monoid r → Type r → Type a → PF (GQ r) → PF (b → r)
data Type a where
  ...
  Data :: (FunctorOf f a, Rep f a b) ⇒ String → Fctr f → Type a
```

Apart from the standard converters *to* and *from*, the class also declares methods that previously existed as point-free combinators or Haskell functions, but must now exist inside the class context, because they return polymorphic representations that change according to the type context (also polymorphic) of the inputs.

The *FunctorOf* declares that f, a and b relate to each other, and that only one b exists for a functor f applied to a type a (functional dependency). Instances allow defining specific behaviors for specific contexts. For instance, the *Id* functor applied to some a is identical to that same type a:

```
instance Rep Id a a where
  to       (Ident x)  = x
  from     x          = Ident x
  apF      Id a       = a
  mapT     Id a  f    = apT_a f
  mapTK    Id a  f    = id
  mapQ     Id m r a f = apQ_a a f
  mapQK    Id m r a f = case teq a r of
                          Just Eq → id
                          otherwise → mzero m
```

If classes were not used, these context-dependant functions could not be defined. For example, if we try to define *apF* without parameterizing it for specific instances

```
apF :: Fctr f → Type a → Type b
apF Id     a = a
apF (K b) a = b
...
```

, the compiler claims that the function's genericalness cannot be satisfied, as long as the unbounded type polymorphism is not supported as the result of functions.

Another important note is that the methods *in* and *out* from the class *FunctorOf* now need to be composed with the *from* and *to* methods from class *Rep*. This happens because point-free expressions are defined for type representations translated from functor representations. Consider an example for Haskell native lists:

```
out_[a]          :: [a] → (K () ⊕ K a ⊗ Id) [a]
out_[a]          = ...
to :: (K () ⊕ K a ⊗ Id) [a] → Either One (a, [a])
to = ...
(to ∘ out_[a]) :: [a] → Either One (a, [a])
> out_[a] [1,2,3]                              = Inl (Pair (Const 1) (Ident [2,3]))
> to      (Inl (Pair (Const 1) (Ident [2,3]))) = Right (1, [2,3])
```

Implementation Issues

Type classes not only permit multiple type-parameters[JJM97], and so define relations on types, but also support functional dependencies[Jon00], that is, the programmer can assert that a given assignment of some subset of the type parameters uniquely determines the remaining type parameters.

However, multiple-parameter type classes do no allow inference of parameters from another parameters through lookup on the existing class instances, because Haskell types do not form a closed world: the programmer might always add new instances for new types.

In our representation, the functors-space is restricted to the small subset of functor representations we have defined, and there is one and only one *Rep* instance for each represented functor. Therefore, we would like the Haskell class system to infere which class instance to use from the functor context, with the guarantee from the functional dependency that there is only one instance for each functor.

Unfortunately, after all the efforts, we were unable to trick Haskell into doing such inference, and could not satisfy functions class contexts (in cases when functors are applied to sums or products), under the compiler error that some instances of class *Rep* are not deducible from the context.

For the simplest example of paramorphism reflexivity:

$$
\begin{aligned}
&\textbf{data } PF\ a \textbf{ where}\\
&\quad \ldots\\
&\quad \langle\!|g|\!\rangle_a^f :: (FunctorOf\ f\ a, Rep\ f\ (b,a)\ c) \Rightarrow Fctr\ f \to PF\ (c \to b) \to PF\ (a \to b)\\
&paraReflex :: Type\ a \to Type\ r \to PF\ (a \to r)\\
&paraReflex\ a@(Data\ \_\ f)\ r = \langle\!|(\circ)\ (apF\ f\ r)\ in_a\ (fmap\ f\ \pi_1)|\!\rangle_a^f
\end{aligned}
$$

From the definition of a, we know that $FunctorOf\ f\ a$ and $\exists\ b\ .\ Rep\ f\ a\ b$. However, the context $\exists\ c\ .\ Rep\ f\ (r,a)\ c$ cannot be inferred from the encoding of *paraReflex*.

4.3 Summary

Either of the proposed approaches proved not to be able to remove language-specific features from the representations or failed to be implemented, due to the difficulty in expressing type and functor equality.

The first approach required many unfriendly functions to put types and functors together in the same representation. The second required repeated sums and products combinators and duplicated laws for each of them, greatly increasing the programmers work load and the inefficiency of the solution. By the other hand, the third approach sounded promising but failed to be implemented successfully in Haskell.

Remark We must not forget that, even if implementable, the type to functor conversions weren't naive and, although implicit inside type classes, would still cost significant time in performance.

Concluding, we still haven't found a good solution for designing one-level rewrite systems with single-recursion. In the next chapter, we will present our prototype of such a rewrite system, with a proper language, but implemented in Haskell, supporting native equivalency between types and functors.

Chapter 5

Developing a Rewrite System from Scratch

5.1 Some notions on Rewrite Systems

A rewrite system is a set of terms, plus rules that specify how one term is replaced with another term. The ability to combine rules into strategies, that can themselves be combined into other strategies, defines the modularity of the rewrite system.

In computer science, rewriting can be seen as the computation involved in transforming programs into another programs. Term rewrite systems have long surpassed their initial mathematical concept and are [Der05]

> *[...] an important part of theoretical computer science. They consist of sequences of discrete transformation steps where one term is replaced with another and have applications in many areas, from functional programming to automatic theorem proving and computer algebra.*

The determinism of a rewrite system is determined by the number of possible substitutions that may transform a specific term.

Consider a rule for natural numbers such that

$$x + x \rightarrow x * 2$$

A term is the unitary element of a rewrite system. It is identified by a symbol and any number of arguments. For this expression, terms are $+(x, y)$, $*(x, y)$ and 2. x is a variable.

The application of a rule is done in three steps: unification, where the left hand side is checked against the input expression; substitution, by assigning values from the expression to the pattern variables; and reduction, the rewriting itself, that replaces references to pattern variables in the right hand side with their actual values.

Unification can be explained by the following example. Suppose the expression:

$$x + x$$

It would unify with

$$1 + 1$$

, but never with

$$1 + 2$$

The definition of context derives from context-sensitive rewriting, which consists in placing rewriting restrictions on arguments of terms, forbidding them to be reduced by any rule.

Context-sensitive rewrite systems allow more control over the rules applied without the need to alter the rewrite strategies, and contexts are usually used as an "hack" to avoid infinite reduction trees. Although the rewrite system we will develop is not intended to be context-sensitive, this concept is an important comparison point for some of the features we aim to define. Quoting Lucas on the properties of context-sensitive rewrite systems [Luc02]:

> *The termination behavior is not only preserved but usually improved and several methods have been developed to formally prove it.*

One of the most important properties of rewriting is confluence, describing that terms can be rewritten in more than one way to yield the same results. It is equivalent to say that, for an input expression, the result must have an unique normal form.

The most well-known rewrite system is Church's lambda-calculus, a formal system designed to study recursive functions. It is universal in the sense that any computable function can be represented as a lambda-expression and evaluated using this formalism.

5.2 Language Definition: properties and features

We chose Haskell as the host language for implementing our generic rewrite system because of features like monads and GADTs, that ease the representation and composition of functions and, more precisely for our project, rewrite rules.

The developed language is intended to be a domain-specific language for designing generic term rewrite systems. In this thesis, we will use it to replace the one-level rewrite system for Haskell point-free expressions, as an alternative to the *embedded* language discussed in Chapter 4.

5.2.1 Terms and Type System

Remembering the composition of point-free expressions

$$(.) :: \mathit{Type}\ a \to \mathit{PF}\ (b \to c) \to \mathit{PF}\ (a \to b) \to \mathit{PF}\ (a \to c)$$

the language, though independant from Haskell, must inherit the notion of type polymorphism for strongly-typed languages. Type-awareness requires the implementation of complex type-inference algorithms in the type-checker [WB89, LO94]. For simplification reasons, the prototype version will discard all the polymorphic information and consider monomorphic types. (.) will then be definable as

$$(.) :: \mathit{PF} \to \mathit{PF} \to \mathit{PF}$$

Remark Note that we are compromising the type-safety of having the type-checker validating the correctness of expressions for free.

Native Types

In the previous attempt to implement a rewrite system for recursive point-free expressions (Chapter 4), the main issue was in handling type and functor equality.

For that reason, the main feature of the rewrite system is to provide an intuitive representation for recursive datatypes and native conversion between the types and functors that represent it. Recursive rules will support automated functor and name inference from a type representation.

We consider three native types: a core representation for types (*Type*), and representations for type functors (*Fctr*) and names (*Name*). In order to import new type representations, the language "embodies" the Glasgow Haskell Compiler's parser for Haskell type declarations.

Because both functor and name representations can be derived from the type representation, we provide two special combinators @ and ! that derive these representations from a type, respectively.

For instance, if t is a type and f an unbounded variable, $t@f$ assigns f with the functor representation for t.

Conversely, if f is assigned to a specific functor expression, $t@f$ only succeeds if f matches some functor f' that is inferred from $t@f'$.

Nevertheless, new types and respective terms may be defined by the user. For the representations of point-free functions we will, in our demonstration rewrite system for one-level transformations, declare a new type PF to hold terms such as $\circ$, the composition of functions presented above.

Associativity

Associativity is a common property of binary operators in rewrite system (remember the algebraic operators for natural numbers). It means that, within an expression containing two or more of the same associative operators in a row, the order of operations can change as long as the sequence of the operands remains unchanged.

Supporting term associativity in the language greatly simplifies the task of the user when defining rule patterns for associative expressions with n-ary arguments (whilst it increases the complexity of pattern unification algorithms).

For example, in a non-associative system, for an associative term $+$, we could have:

$$a + (b + c) \rightarrow d$$
$$(a + b) + c \rightarrow d$$

If associativity is supported, the same semantics can be represented in one single pattern:

$$a + b + c \rightarrow d$$

In our language, a term can be explicitly declared associative by assigning a specific *assoc* tag:

$$+ :: Nat \rightarrow Nat \rightarrow Nat < assoc >$$

However, this feature suffers from some inefficiency issues, discussed as future work (Chapter 7).

5.2.2 Variables and Rules

After terms, we turn to the definition of rules. Rules are the rewriting elements themselves: they express how to transform a term into another.

They can be seen as monadic functions from term to term: the monad denotes partiality in the rewriting. The monadic zero states that a rule has failed to be applied for a certain argument.

Monadic binding and pattern support in the do-notation simplify the implementation of rule sequentiation and optionality.

A rule may contain multiple patterns, that are unified consequently until one successfully succeeds. Each let hand pattern associated with a right hand side expression is called a rule case.

Unlike functional languages, where if a pattern is matched with a term no other pattern is tried even if it fails to return a result, all patterns are tried until a result is returned. Despite being different from context-sensitive rewrite systems, where restrictions can be applied in reduction of terms, the monadic optionality for patterns can be used by explicitly expressing failure behavior for a specific pattern.

A monad also stores function results and side-effect representations, what eases the implementation of rule tracing algorithms.

Even though every term must have a type, the type of rule is monomorphic, what denotes that a rule may contain rewrite cases for expression of different types. In other words, the type of rules simply signals rule definitions, and does not distinguishes a rule on integers from a rule on strings. Rule cases are no more than the combination of different rules.

Additionally to the obligatory term argument, a rule may also admit argument rules. For instance, some rule

$$rule\ r : term\ x\ y \rightarrow r\ x$$

returns the result of applying the argument rule r to the right hand side of its single case.

Like rewrite systems we have developed before, the current one we are developing also provides SyB-like generic combinators for combining rules.

5.2.3 Input and Output

Whenever an user wants to rewrite an expression, he may declare a new variable that applies a specified rule to that expressions

var = *rule exp*

Nevertheless, if we want to output some expression, it has to be expressed explicitly with a *print* command:

print exp
print (*rule exp*)

This requirement is important in the sense that it guarantees the laziness of rewrite system. As in any lazy functional language, our language only computes variables whenever they are referred in another variable or rule or when explicitly required for output.

Haskell integration

In spite of the fact that our generic rewrite system is not an extension to Haskell, its design deeply concerns rewriting of Haskell programs.

Examples of grand integration are: the parsing of Haskell type declarations into internal type representations; the possibility of forwarding properly commented Haskell code to the output; and, most important, syntactic compatibility of term and variable representations, allowing encoded rewrite systems to generate completely valid Haskell modules.

5.3 Implementation

Our rewrite system generator has the following major characteristics:

- It is written purely in Haskell, including the parser. This enables the whole implementation to be based on monads, with clear advantages, as we will furtherly discuss.
- Its design is intended to provide a minimal syntax, generic enough to be independant of the set of terms being rewritten, but providing special features to safely handle native type and functor representations.
- It provides a module system similar to Haskell.
- It provides tracing information for applied rules, for each rewritten expression.

5.3.1 An overview of the compiler

The compiler is structured in a similar approach to the Glasgow Haskell Compiler[JHH+93]. Figure 5.2 renders the main functional blocks of the multi-phased compiler:

1. A context-sensitive, infinite look-ahead parser, written in the Parsec combinator language, defines the lexer. Although combinatorial parsing lacks the excellence in speed of traditional bottom-up parsing, it provides more expressiveness in the grammar definition and infinite look-aheads allow the conscious generation of tokens for further seamless LR parsing. In our case, infinite look-ahead is used wisely and the efficiency of the *lexer* is not compromised. A very resumed and incomplete syntax for our language is described in figure 5.1.

rhs := *moduledecl* | *import*∗ | *sort*∗ | *content* ∗
content := *hsdecl* | *term* | *variable* | *result* | *rule* | *hscode*
term := *name* `':::'` *targs*
targs := *name* ! *name* `'->'` *targs*
variable := *name* `'='` *expr*
result := `'print'` *expr*
rule := *name name* ∗ **case** ∗
case := `':'` *expr* `'->'` *expr* | `':'` *expr* `'->'` `'fail'` | `':'` *name case* ∗
expr := *name expr*∗ | *expr* `'@'` *expr* | *expr* `'!'` *expr* | *case expr*
name := *any string*

The full syntax is available by checking the grammar file in the source modules of our language.

Figure 5.1: *Core* syntax for our generic rewrite system.

Token information is passed to a standard bottom-up LR(1) *parser* written in Happy, an Yacc homologue implemented in Haskell. The produced abstract syntax tree faithfully represents every construct in our language, but differs from a parse tree by omitting syntax that do not alter the semantics of the program.

2. The *renamer* resolves scoping and ambiguous references of term, variable, type or rule names. Scoping also involves full name creation for referenced names: for a variable x that belongs to module m, the full path for that variable's name would be $m \,.\, x$. The definition of a variable y is also extensible to structures "living" inside rule r, such that $m \,.\, r \,.\, y$.

3. The *type checker* annotates the program with type information. The *type inference* algorithm is much simplified by supporting only monomorphic types.

4. The *interpreter* executes the input rewrite system and returns only the AST for the code to be outputted. There is no proper interpretation algorithm as for a normal compiler: variables are loaded into memory when found (variables with the same name are overridden by default). Variable references are then substituted, and rules reduced. In this rudimentary algorithm, no compiler/code optimizations, such as inlining[JM02], are considered, but admitted as future work.

5. The *code generator* does no more than converting the abstract syntax into Haskell's internal representation and printing it into the standard textual representation. This pass also involves generation of tracing files with information on the rules applied to rewrite the initial expressions.

5.3.2 Monads

Monads are a concept that derives from category theory, that functional languages have adapted for forcing non-lazy sequential operations, expressing input/output operations and implement side effects (statefull functions) without introducing language side effects features [Mog89].

Peyton Jones *et al* [JHH+93] have already found it very useful to use monads in the implementation of the Glasgow Haskell Compiler. We will use monads in the encoding of our compiler for similar reasons:

- Monads natively provide sequentiation (that Peyton Jones recalls as plumbing), which makes the code much easier to read and write;
- Along sequentiation, monadic partiality enables implicit error recovery in the compiler;
- Statefull monad transformations enable the storage of tracing information
- For each (partial) rewrite rule, there can always be defined a monad that mimics its semantics. Representing rules as monads provides rule sequentiation ($\gg$) and optionality ($\|$) primitives for free.

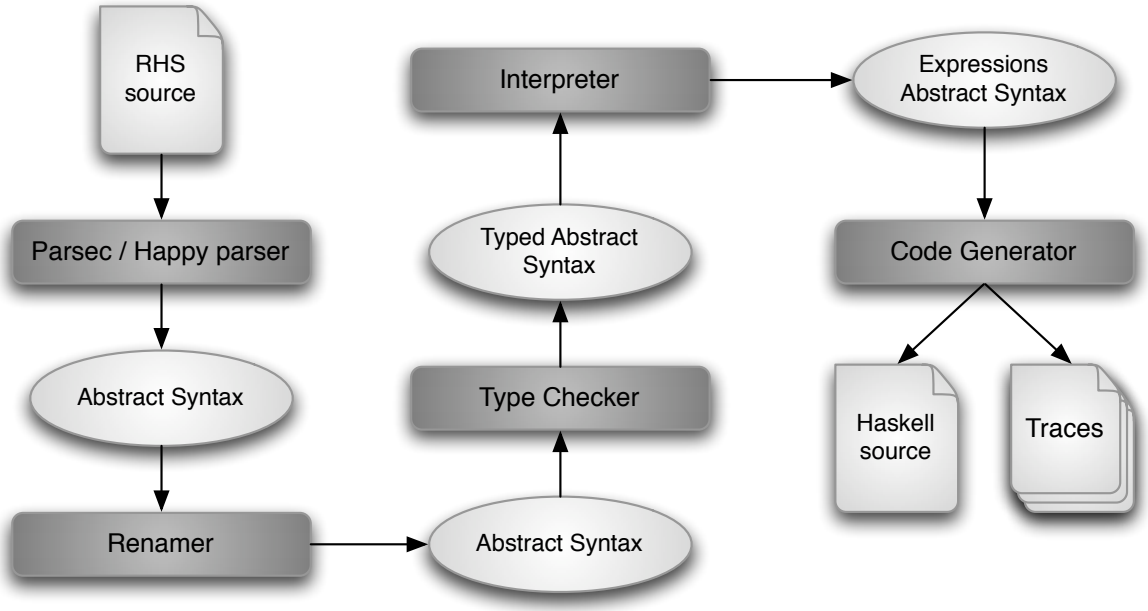


Figure 5.2: An overview of our compiler.

5.4 Libraries for One-level transformations

As mentioned above, the case study that motivated the development of this rewrite system is the simplification of point-free expressions for single-recursive datatypes. Terms include the standard point-free combinators and the SyB and XPath combinators.

Rewrite rules were encoded for classes of expressions, as well as transformation rules between them. A typical motivating example for this one-level rewrite system is the specialization of XPath expressions into SyB type-unifying queries and, again, into point-free functions that are simplified with point-free algebraic laws. The resulting function is conformant to the type against which it was specialized.

In this section, we describe modules for the specification and transformation of point-free expressions.

Finally, we study some examples and compare the results with the XPTO tool [FP07].

5.4.1 Point-free

The representation of terms follows a normal grammar syntax, despite future work contemplates extending it to a GADT-like representation in order to support polymorphic types in term declaration.

As an example, consider the following module that defines some demo point-free combinators and rules for catamorphisms:

```
module PF where

%sorts{PF}      -- type declaration

out     :: PF
(.)     :: PF -> PF -> PF
cata    :: Type -> PF -> PF
fmap    :: Fctr -> PF -> PF

mzero   :: Type -> PF
mplus   :: Type -> PF

nat_Id : id . f -> f
       : f . id -> f
```

```
cata_Def : cata t@f g -> g . fmap_KDef (fmap f (cata t g)) . out t

fmap_KDef : fmap (K a) f -> id
          : fmap (f:+:g) h -> fmap_KDef (fmap f h) -|- fmap_KDef (fmap g h)
          : fmap (f:*:g) h -> fmap_KDef (fmap f h) >< fmap_KDef (fmap g h)
```

In this small example, you may already notice that $t@f$ is deriving functor f from some type t.

More significantly, *cata_Def* invokes *fmap_KDef* in the right-side expression, and shall only succeed if *fmap_KDef* also succeeds. This ability to call rewrite rules inside expressions becomes important to control the application of certain rules, and can be seen as an approximation to context-sensitive rewrite systems [Luc02].

For this specific case, we only want *cata_Def* to be applied if the type it folds over is not recursive, otherwise sequential rule application may succeed in infinite unfolding of *cata*. This is achieved by guaranteeing that *fmap_KDef* fails for the identity functor (implicitly by the omission of that particular case).

However, you may question why would a catamorphism be applied to a non-recursive type. Catamorphisms define iteration over generic types, independently of their structure (recursive or not). Therefore, they can be used to implement generic folding operations over generic types, where simplification into explicit recursive functions is important in the specialization of such generic functions.

This method is largely employed in structure-shy query languages, such as XPath, and will furtherly be studied at the time we discuss rewriting of structure-shy to structure-sensitive expressions.

Monoid terms *mzero* and *mplus* will be useful for grouping multiple values of the same type. This is the case when the resultant expression is the concatenation of specializing the same expression over two distinct types. Such an example is in the further definition of *mapQ_Def* for functor products ($\otimes$).

Terms for monoids are parameterized with a type and defined generically for it, in the assumption that only types that define a monoid may be represented as such.

For example, *mplus* $[a]$ defines the binary concatenation of expressions over lists of a elements, and could be specialized into *listcat* $:: PF$. Conversely, *mplus Int* defines the binary sum of expressions over integers and could be specialized into $+ :: PF$.

5.4.2 Srap Your Boilerplate

For representing SyB combinators, we define new sorts T and Q for type-preserving and type-unifying terms, respectively.

In the GADT representation, the same expressions were represented as $PF\ T$ and $PF\ Q$. However, adapting the rewrite system to monomorphic types encompasses lifting some previously polymorphic type contexts into different types.

Consider the following module for specifying SyB combinators:

```
module SYB where

import PF

%sorts {T,Q}

-- Type-preserving strategy combinators
apT           :: Type -> T -> PF
mkT           :: Type -> PF -> T
gmapT         :: T -> T
gmapTK        :: T -> T
everywhere    :: T -> T
nop           :: T
seq           :: T -> T -> T

mapT          :: Fctr -> Type -> T -> PF
mapTK         :: Fctr -> Type -> T -> PF

-- Type-unifying strategy combinators
apQ           :: Type -> Q -> PF
mkQ           :: Type -> Type -> PF -> Q
gmapQ         :: Type -> Q -> Q
gmapQK        :: Type -> Q -> Q
everything    :: Type -> Q -> Q
emptyQ        :: Type -> Q
```

```
union        :: Type -> Q -> Q -> Q

mapQ         :: Fctr -> Type -> Type -> Q -> PF
mapQK        :: Fctr -> Type -> Type -> Q -> PF

gmapQ_apQ : apQ Any (gmapQ r f) -> fail
          : apQ t@f (gmapQ r g) -> mapQ_Def (mapQ f r t g) . out t
          : apQ t (gmapQ r g) -> mzero r

mapQ_Def : mapQ Id r a f -> apQ a f
         : mapQ (K b) r a f -> apQ b f
         : mapQ (f:+:g) r a h -> mapQ_Def (mapQ f r a h) \/ mapQ_Def (mapQ r g a h)
         : mapQ (f:*:g) r a h -> mplus r . (mapQ_Def (mapQ f r a h) >< mapQ_Def (mapQ g r a h))
```

Now we will study the *gmapQ_apQ* rule. It defines three possible cases.

When the argument type is hidden inside a *Any* encapsulator, we want to leave the expression untouched, since we have defined extra laws that allow the decapsulation of dynamic application. The reserved word *fail* denotes rule instant failure. Similarly to localized rule invocation inside expressions, it can also be seen as an approach to context-sensitive rewrite systems [Luc02].

If the type to which the generic expression is applied is not a basic type and has a functor representation, then the behavior of *gmapQ* can be expressed by specializing the parameterization of *mapQ* with the argument type and functor. Or, the expression is not applicable and the zero monoid is returned.

Placing a restriction to ensure that the argument type is recursive can be compared with context-sensitive rewrite systems, in the sense that we are constraining the application of the rule.

5.4.3 Xpath

As before, XPath combinators are encoded as type-unifying generic queries. You may recall their definition from Chapter 2.

XPath axis indicate navigation direction in the XML tree structure. We now explain the most relevant axis for simplification:

- *self* denotes the current node;
- *child* is the default axis, if none specified, and is equivalent to mapping *self* over the child nodes of the current node (*child_Def*);
- *desc* may be seen as the recursive application of *child* (*des_Def*) and unifies with any node under the current node in the XML tree;
- *descself* is the union of *desc* and *self* (*descself_Def*);
- *name* performs similarly to *self*, but only succeeds if the current node name unifies with its argument name;
- (/) defines XPath sequentiation. It simply passes the result of applying an axis to the following axis. The distributivity of *union* denotes that applying some axis after the union of two other axis is the same as performing the union of the two sequentiation those axis (*union_Dist*).

We can easily define combinators for the XPath base syntax and algebra that translates XPath expressions into generic functions:

```
module Xpath where

import PF
import SYB

%sorts {}

-- XPath combinators
self         :: Q
child        :: Q
desc         :: Q
descself     :: Q
name         :: Name -> Q
(/)          :: Q -> Q -> Q
```

```
qcomp       :: Q -> PF -> Q
(?)         :: Q -> Q -> Q
nonempty    :: Q

xpath_alg : descself_Def | union_Dist | desc_Def | child_Def

descself_Def : descself -> union [Any] self desc
union_Dist : union t f g / h -> union t (f / h) (g / h)
desc_Def : desc -> everything [Any] child
child_Def : child -> gmapQ [Any] self
```

5.4.4 Examples

We will now run some examples and compare the results with the XPTO tool [FP07] for non-recursive XML schemas. The example XPath expressions will be simplified according to the schema of Figure 6.1.

The first example is the XPath expression

self / *child* / *name* `"movie"` / *child* / *name* `"actor"`

that retrieves every *actor* elements that are direct children of *movie* elements under the root element. A demonstration module can be written to express this specialization[1]:

```
module Examples.XPath where

import SYB
import PF
import Prelude
import XPath

%sorts {}

newtype Eimdb = Eimdb{unEimdb :: ([Emovie], [Eactor])}
newtype Emovie = Emovie{unEmovie :: (Etitle, (Eyear, ([Ereview], (Edirector, [Ebox_office]))))}
newtype Etitle = Etitle{unEtitle :: String}
newtype Eyear = Eyear{unEyear :: Int}
newtype Ereview = Ereview{unEreview :: String}
newtype Edirector = Edirector{unEdirector :: String}
newtype Ebox_office = Ebox_office{unEbox_office :: (Either Edate (), (Ecountry, Evalue))}
newtype Edate = Edate{unEdate :: String}
newtype Ecountry = Ecountry{unEcountry :: String}
newtype Evalue = Evalue{unEvalue :: Int}
newtype Eactor = Eactor{unEactor :: (Ename, [Eplayed])}
newtype Ename = Ename{unEname :: String}
newtype Eplayed = Eplayed{unEplayed :: (Etitle, (Eyear, (Erole, [Eaward])))}
newtype Erole = Erole{unErole :: String}
newtype Eaward = Eaward{unEaward :: (Eaward_name, Eresult)}
newtype Eaward_name = Eaward_name{unEaward_name :: String}
newtype Eresult = Eresult{unEresult :: String}

xp1 = self / (child / name Emovie) / (child / name Etitle)
pf1 = optimizexp (apQ Eimdb xp1)
print pf1
```

In both our rewrite system and the XPTO tool, the query is reduced to the void path:

$pf1 = nil$

The second example is similar to the first one, but collects movie titles instead of movie actors:

xp2 = *self* / *child* / *name* `"movie"` / *child* / *name* `"title"`

The resulting point-free expression directly relates to the original query: for each *movie* contained in the first child of the *imdb* root node (a list of movies), it selects the first child, namely the *title* and encapsulates it inside the $*$ dynamic type:

$pf2 = list\ (mkAny\ Etitle \circ \pi_1 \circ out\ Emovie) \circ \pi_1 \circ out\ Eimdb)$

The third query retrieves descendant *result* elements, considering a list of *award* elements as the root node instead of the default root *imdb* element. Note that an *award* element only contains a pair

[1] The Haskell type that represents the schema was generated by the XML Schema interface for the XPTO tool.

of elements *award_name* and *result* as childs. This choice is made for efficiency reasons that we later discuss.

$$xp3 = descself \ / \ name \ \texttt{"result"}$$

The specialized point-free expression intuitively maps a selector of *result* elements (the second child of *award* elements) over the input list:

$$pf3 = list \ (mkAny \ Eresult \circ \pi_2 \circ out \ Eaward)$$

However, remember that the *desc* axis is transformed into a generic combinator that applies the selector to every descending node (*everything*). This recursive pattern is encoded by paramorphism over lists. Since lists are recursive types (remember the functor $K \ () \oplus K \ a \otimes Id$), the *para_Def* rule cannot be applied to the type and it can only be canceled for very specific cases (consult the rules for paramorphisms in Annex A). All these transformation for recursivity elimination are much more expensive than representing lists as non-recursive types, like in the XPTO tool.

Until now, we proved that, following a recursive type representation, we can still obtain the same results as in a non-recursive approach.

As the fourth example, recall the generic query for retrieving information information on employees for a recursive type holding information on them, analyzed in Chapter 4:

$$syb4 = apQ_{Employee} \ (everything \ (mkQ_{Employee} \ wrap))$$

Through specialization, we get:

$$pf4 = (\!|mplus \circ ((id \ \nabla \ nil) \circ \pi_1 \times wrap \circ in_{Employee}) \circ (fmap \ \pi_1 \ \triangle \ fmap \ \pi_2)|\!)_{[Employee]}$$

Compared with the previous attempts, the result is simpler, since the same product and sum functions are defined independently of types and functors, and there is no need to use functor encapsulators/decapsulators. The resultant point-free expression is finally a perfect mapping of the theoretical concepts, since the language does not require any extra syntax to represent the same concepts.

5.5 Tracing

As stated in the language architecture, tracing files are generated for each outputted variable, and provide detailed information of the applied rules and resulting intermediate expressions.

Consider now some simple example for the rewriting of a single point-free expression:

```
module Examples.PF where

import PF
import Prelude

%sorts{}

pf = id . id . id
res = many nat_Id
print res
```

For this example, one single trace file `res.trace` is generated, in the same directory where the module `Examples.PF` is located:

```
-- This is an automatic generated trace file for the variable Examples.PF.res.

res = id . id . id
    = { nat_Id }
      id . id
    = { nat_Id }
      id
```

5.6 Efficiency

In this section, we discuss the results of comparing the developed rewrite system against the original Haskell implementation for non-recursive types. The current rewrite system copies the rewrite strategy from the original implementation, apart from the handling of generic traversals, that recur to recursion patterns rather than specific specialization for lists.

XPath Expression[a]	Old	Current	Root Node
./movie/actor	0.439s	1m19.301s	imdb
./movie/title	0.425s	1m19.497s	imdb
//result	0.51s	57s	[award]

[a] Abbreviated XPath Syntax

Table 5.1: Benchmarking times for XPath expressions over a non-recursive schema.

Functors Approach	Combs. Replication Approach	Current
0.126s	0.050s	1m

Table 5.2: Benchmarking times for a type-unifying expression over a recursive schema.

Benchmark tests were run for all the examples. For testing, GHC version 6.6 with optimization flag -O2 has been used.

By analysis of the results (Tables 5.1 and 5.2), it is possible to conclude that rewriting of the same expressions in our language is much slower than in the Haskell implementation.

We can enumerate two probable causes for such inefficiency:

- as said before, the XPath descendant axis is encoded by a paramorphism, that implies rules for recursivity elimination. This is enforced by representing XPath sets as recursive lists rather than native lists, for which specific rules can be derived. This is evidenced in the third example query, that is straightforward to calculate for a non-recursive type representation, but requires much effort to achieve the same results through paramorphism canceling.

- as long as we have implemented a generic rewrite system, rule and term declarations are loaded into internal non type-safe representations. Implementing a specific rewrite system in Haskell enables type-safe representations that are compiled into the destination program, along with the rewriting rules. These facts allow us to assume that compiling the whole rewrite system enables more optimizations that turn rule pattern matching and application at byte-code level to be more efficient than expressed as Haskell datatypes. This fact can be mostly noticed in the last example for selection of employees information (Table 5.2), where the specific implementations in Haskell for this particular example also performs much faster than our implementation and paramorphism elimination is not employed.

5.7 Summary

In this chapter, we have developed a non-type-safe generic rewrite system for Haskell terms, and explained the advantages of modeling the previous one-level rewrite system for point-free expressions with it, instead of implementing a specific rewrite system in Haskell.

Although it provides a much simpler language for defining and executing generic rewrite systems, with specific features for mediation between types and functors, the most prominent conclusion is that efficiency must be improved in order to apply it to real examples.

Chapter 6

Bidirectional Lenses

5 Computing is full of situations where one wants to transform some structure into another form, maintaining a view that stores forward and backward value-transformations between the data models to yield interoperability. One way to address such bidirectional transformations is via two-level transformations, based on calculational data refinement.

Program refinement is the transformation of an abstract format specification into a concrete low-level program. Another approach to bidirectional data transformation is the well-known *view-update problem*, where a concrete model is abstracted into a view, and where value changes made to the destination format are performed as updates to the original structure, what denotes knowledge on the original values in the backward value transformation.

In relational theory, a database view is a virtual database representation that addresses a portion of the data in the original database, structured in a suitable way for a specific purpose. A user interacts with a view by issuing queries and update requests on views that must be translated into requests on the underlying database. Although mapping of queries does not present particular problems, when translating a view update there is, in general, more than one database update that results in exactly the same changes on the database state as in the original view update. The view-update problem addresses the difficulty of choosing an unique database update for each view update.

Pierce *et al* [FGM+05, Pie06], inspired in similar these concepts from classical relational database theory[BS81] into programming languages, and have implemented a data synchronization tool called Harmony. Harmony builds on bidirectional transformations with view-update named *lenses*.

This chapter explains, in pratical terms, the development of a possible implementation in Haskell for the view-update problem using bidirectional lenses. Special emphasis is placed on the type-safeness and robustness of lens application. Most of the theory is inherited from Pierce *et al* previous work.

6.1 A motivating Example

Suppose the following representation for an address book, as a map of names to tuples of phone numbers and urls[1] [2]:

$$Name \rightharpoonup Phone \times URL$$

Consider that both Name, Phone and URL are uniquely defined Haskell datatypes that represent strings. We can fill some values for this structure:

$$\begin{aligned} c = \{&Name\ Chris \mapsto (Phone\ 888-9999, URL\ http://chris\ .\ org) \\ ,&Name\ Pat \quad \mapsto (Phone\ 333-4444, URL\ http://pat\ .\ com)\} \end{aligned}$$

[1] Example extracted from [FGM+05]

[2] $A \rightharpoonup B$ denotes a map from A to B.

Now, suppose that for synchronization or simplification purposes we would like to create an abstraction for this model where each name is directly associated with a phone number:

$$Name \rightharpoonup Phone$$

$$a = \{Name\ Chris \mapsto 888 - 9999, Name\ Pat \mapsto 333 - 4444\}$$

We can then modify the content of the abstract model as desired. For example, we can drop Chris's contact and add a new contact for Jo.

$$a' = \{Name\ Jo \mapsto 555 - 6666, Name\ Pat \mapsto 333 - 4444\}$$

According to the *view update approach*, we might be able to reconstruct an updated concrete model based on an updated abstract model and the original concrete model.

Applying the backward transformation, we can compute a new concrete model reflecting the changes on the abstract model.

$$\begin{aligned} c' = \{&Name\ Jo \mapsto (Phone\ 555 - 6666, URL\ http://google.com) \\ &, Name\ Pat \mapsto (Phone\ 333 - 4444, URL\ http://pat.com)\} \end{aligned}$$

Note that, in this case, we have to "fill in" default data for Jo's URL, since URL information is lost in the abstraction. The default data is injected by a type-parameterized default generator.

The generic lens that computes the transformation described above is written as

$$map\ (focus\ \texttt{"Phone"}\ defaultURL)$$

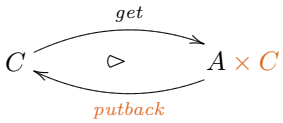
, where

$$defaultURL\ (Data\ \texttt{"URL"}\ (to\ \leftrightarrow\ from)\ String) = from\ \texttt{"http://google.com"}$$

We will later explain better the process of defining default values for lenses.

6.2 Bidirectional Lenses Theory

A lens consists of a type transformation coupled with an *abstraction relation* $get : C \to A$ and a statefull *representation relation* $putback : A \times C \to C$ that pushes the abstract view back into the original concrete model:



Similarly to two-level transformations, compositional creation of lens rewrite systems depends on some properties that preserve the well-behavedness of single and composed lenses. For a lens to be well-behaved, their forward and backward transformations must at least satisfy the properties of *acceptability* and *stability*[AC07], respectively.

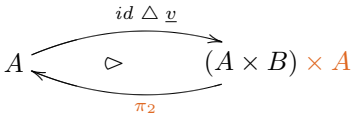
The forward transformation is acceptable if *putback* captures all the information in the abstract view:

$$get \circ putback \sqsubseteq \pi_1$$
$$\forall c\ .\ \in C\ .\ \exists\ a\ \in A\ .\ get\ (putback\ (a, c)) = a$$

The backward transformation is stable if it does not drop concrete informations. Abstracting and immediately putting back a concrete view shall return exactly the same view:

$$putback \circ (get \vartriangle id) \sqsubseteq id$$
$$\forall c \in C \,.\, putback\ (get\ c, c) = c$$

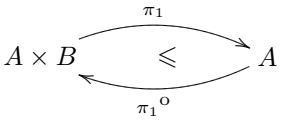
Any valid refinement is not a stable lens (unless it is a bijection).
Consider the attempt to encode the refinement add_π_2 (from Chapter 2, Section 2.3) as a lens:



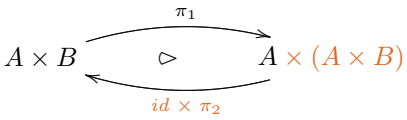
For distinct values $a, a' \in A$ and b *belongs* B, suppose that $\forall x \in A \,.\, get\ (x) = (x, b)$. Through stability:

$$\begin{bmatrix} & get\ (putback\ ((a,b),a')) = (a,b) \\ \Leftrightarrow & \quad \{\ldots\} \\ & get\ (a') = (a,b) \\ \Leftrightarrow & \quad \{\ldots\} \\ & (a',b) \neq (a,b) \end{bmatrix}$$

Similarly, a well-behaved lens is not a valid refinement (unless it is a bijection).
Consider a transformation from $A \times B$ to A, where we drop the second element of the product. If we write a two-level transformation such as



we can easily notice that it is not a valid refinement, because π_1 is not injective.
Thus, we will write a lens $drop\_\pi_2$ to perform the same transformation:

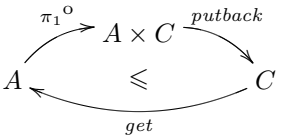


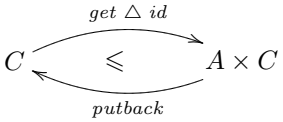
Note that $id \times \pi_2$ prefers the updated abstract value over the original concrete value of A, in order to guarantee lens acceptability. We can now prove that $drop\_\pi_2$ is a well behaved lens:

$$\begin{bmatrix} & get \circ putback \subseteq \pi_1 \\ \Leftrightarrow & \quad \{\text{definition of } get \text{ and } putback \\ & \quad \text{;equality of functions}\} \\ & \pi_1 \circ (id \times \pi_2) = \pi_1 \\ \Leftrightarrow & \quad \{\ nat\text{-FST};\ nat\text{-ID}\ \} \\ & \pi_1 = \pi_1 \end{bmatrix}$$

$$\begin{bmatrix} & putback \circ (get \vartriangle id) \subseteq id \\ \Leftrightarrow & \quad \{\text{definition of } get \text{ and } putback \\ & \quad \text{;equality of functions}\} \\ & (id \times \pi_2) \circ (\pi_1 \vartriangle id) = id \\ \Leftrightarrow & \quad \{\ \times\text{-ABSORPTION};\ nat\text{-ID}\ \} \\ & \pi_1 \vartriangle \pi_2 = id \\ \Leftrightarrow & \quad \{\ \times\text{-REFLEX}\ \} \\ & id \subseteq id \end{bmatrix}$$

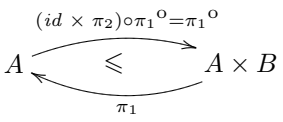
A well-behaved lens can also be expressed in terms of stateless two-level refinements. Oliveira [Oli07] proved that connectivity of the twin transformations

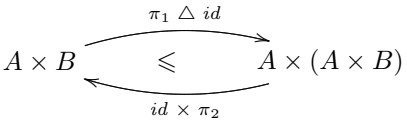




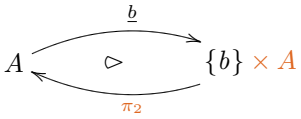
formulates the equations for acceptability and stability of a lens from C to A.

Recapitulating $drop\_\pi_2$, we can define the following pair of two-level transformations[3]:





An example of a lens that not always complies to the well-behaved requirements of lenses is the constant lens *const*, that transforms any type A into a constant value $b \in B$ and ignores the concrete view in the *putback* direction:



For the abstract domain $\{b\}$, *putback* is clearly semi-injective. However, if we consider the *const* lens for an abstract domain larger than b, it is not a well-behaved lens since the subjectivity (and, thus, acceptability) of *get* is violated:

$$get\ (putback\ (b', a)) = get\ a = b\ \wedge\ b\ \neq\ b'$$

Therefore, in our lens library, *const* should only be used for the abstract domain equivalent to the unit type, for that represents the single value 1, otherwise it is unsafe.

This topic will need more research in the near future towards understanding the real implications and applications of representing lenses as data refinements in the original lenses for trees presented by Pierce *et al.*

6.2.1 Handling Partiality

In lens application, cases are considered when no concrete view exists. Therefore, lens functions are considered partial.

In order to achieve totality, the domain of concrete values C is extended with a "missing" value element denoted as Ω, where C_Ω is the set unification $C \bigcup \{\Omega\}$. Ω is particularly of use when defining generic combinators for lens application, and is by convention only used as the second argument of the *putback* function. Thus, we can redefine it into $putback : A \times C_\Omega \to C_\Omega$.

Intuitively, $get\ (a, \Omega)$ represents creating a new concrete view only from the information in the abstract view a.

This approach is similar to ours of adding explicit partiality to two-level transformations, transforming partial backward transformations into entire functions, except for the fact that partiality is propagated from the original concrete model into a possibly undefined updated concrete model. Like-wise, we could define the lens backward transformation as $putback : A \times (C + 1) \to C + 1$, in a semantically equivalent representation.

6.3 Representation of Lenses

After studying the semantic foundations for lens definition, we devote this section to their implementation in Haskell.

As stated before, lenses are implemented as partial functions, which well-behavedness can be constrained to a concrete domain and an abstract codomain. For this reason, lens application outside their

[3] Although we are not presenting formal proof, these specific transformations obey to the two-level properties of value-transformers.

domain may fail. In our library, we define lenses in a type-safe manner, in the sense that lens definition can be constrained to specific types and their validation relies on the type-checker.

The basic definition of a lens is:

$$\textbf{data}\ \mathit{Lens}\ c\ a = \mathit{Lens}\{\mathit{get} :: c \to a, \mathit{putback} :: a \to c \to c\}$$

Therefore, domain validation is always performed at the type level. In Haskell, we can only bind lenses to specific types, but cannot restrict values for such types. For example, we have no way to restrict the *Int* native type to the subset of integers greater than 2. This encoding of lenses does not suffice for conditional lenses, where the codomain is splitted by the conditional predicate. In such cases, lenses will perform arbitrarily or erroneously for abstract values outside their codomain.

Given a *Lens* over values of specific types, we may add type-awareness as a *View* that transforms a *Type c* into a *Type a*.

$$\begin{aligned}&\textbf{data}\ \mathit{View}\ x\ \textbf{where}\\ &\quad \mathit{View} :: \mathit{Lens}\ c\ a \to \mathit{Type}\ a \to \mathit{View}\ (\mathit{Type}\ c)\end{aligned}$$

At last, we can abstract a *View* into a generic *Rule*, partial, that may be applied to any input type and return any type.

$$\textbf{type}\ \mathit{Rule} = \forall c\ .\ \mathit{Type}\ c \to \mathit{Maybe}\ (\mathit{View}\ (\mathit{Type}\ c))$$

6.4 Lens Combinators

6.4.1 Generic Lenses

Here, we describe some basic combinators for lens definition. Generic lenses are the ones that provide generic behavior over types.

The most straightforward is the identity lens. It copies the concrete view in the *get* direction and the abstract view in the *putback* direction.

It is implemented in our system as

$$\begin{aligned}&\mathit{id\_lns} :: \mathit{Lens}\ c\ c\\ &\mathit{id\_lns} = \mathit{Lens}\ g\ p\\ &\quad \textbf{where}\\ &\quad g\ c\ \ \ \ \ = c\\ &\quad p\ (a, c) = a\end{aligned}$$

and easily abstracted into

$$\begin{aligned}&\mathit{nop} :: \mathit{Rule}\\ &\mathit{nop} = \mathit{return} \circ \mathit{View}\ \mathit{id\_lns}\end{aligned}$$

Basic lenses for products are $\pi 1\_lns$ and $\pi 2\_lns$, that select the first and second element of a pair, respectively. In the *get* direction, $\pi 1\_lns$ drops the second element, that is recovered from the concrete view in the *putback* direction:

$$\begin{aligned}&\pi 1\_lns :: \mathit{Lens}\ (a, b)\ a\\ &\pi 1\_lns = \mathit{Lens}\ g\ p\\ &\quad \textbf{where}\\ &\quad g\ (x, y)\ \ \ \ \ \ = y\\ &\quad p\ (x', (x, y)) = (x', y)\end{aligned}$$

Opposite behavior is implemented for $\pi 2\_lns$:

```
π2_lns :: Lens (a, b) b
π2_lns = Lens g p
  where
  g (x, y)       = y
  p (y', (x, y)) = (x, y')
```

Another simple combinator is *const_lns*, that transforms any view into a constant value. In the *get* direction, it always returns the argument value. In the *putback* direction, the combinator makes no inference on the abstract model and returns the original concrete value. However, if we later combine *const_lns*, it will frequently receive undefined concrete values in the *putback* direction, due to removal of information in the abstraction direction. Therefore, as stated before, it will only be used for the *const* () *d* case, where the abstract domain is the unit type.

As presented in [FGM+05], we represent undefined values as Ω, and inject the concrete view with a default view in case it is undefined.

```
const_lns :: a → c → Lens c a
const_lns v d = Lens g p
  where
  g c      = v
  p (a, c) = if (c ≡ Ω) then d else c
```

A very important drawback of this combinator is that it's abstract domain is the single constant value passed as argument to the combinator. This leaves a great "hole" in our model, since we cannot safely constrain the lens to it's codomain, allowing the combinator not to be well-behaved for values others than the constant value.

The *const* generalization is interesting in the sense that we use the class method *typeof* to derive the type for the constant value passed as argument.

```
const :: Typeable a ⇒ a → Default → Rule
const v d t = return (View (const_lns v (d t)) typeof)
```

Default values are represented as generator functions that create a value from a type definition:

```
type Default = ∀a . Type a → a
```

Two example default generators are Ω (creates Ω values) and *empty* (generates empty values, for example, empty lists).

The lens composition combinator • applies one lens *l1* after another lens *l2*. In the *get* direction, it applies the *get* function of *l1* to the result of applying the *get* function of *l2* to the concrete view. In the *putback* direction, it applies the *putback* functions in reverse order. The *putback* function for *l1* is supplied with the intermediate concrete value resulting from abstracting the original concrete value for *l2*.

```
(•) :: Lens a b → Lens c a → Lens c b
(l2 • l1) = Lens g p
  where
  g c      = get l2 (get l1 c)
  p (a, c) = putback l1 (putback l2 (a, get l1 c), c)
```

The ⋙ combinator expresses the generic representation of •:

```
(⋙) :: Rule → Rule → Rule
(r1⋙r2) c = do
  View l1 a1 ← r1 c
  View l2 a2 ← r2 a1
  return (View (l2 • l1) a2)
```

More strategic combinators similar to the ones for two-level transformations have been developed.

6.4.2 Tree combinators

The tree combinators studied in this section are adaptations of tree combinators from [FGM+05] into our model. The main difference is on tree structure representation. As an example, consider the following XML file:

```
<a>
  <b>hello</b>
  <c>world</c>
</a>
```

Pierce *et al* representation is based on a rose tree structure, where all nodes are of the same type. It represents unordered trees, without repeated values, for simplification purposes. Every tree node has a tag and there are indirection nodes such as @*children* for the childs of an element, in a AST-like fashion.

```
{a=
   {@children=
     [{b={@children=[{@pcdata={hello}}]}},
      {c={@children=[{@pcdata={world}}]}}]}}
```

Our solution models trees as n-ary products (*NProd*), and the the child's type and structure is explicit in the structure. Tree nodes only exist for tree elements, and no extra nodes are created. For this reason, trees have the notion of order, and name uniqueness is not enforced by the model. Also, the type representation is independent from the values:

```
type = Data "a" (unA <-> A)
   (NProd
      (Data "b" (unB <-> B) String)
      (Data "c" (unC <-> C) String)
   )

value = A
   (  B "hello"
   ,  C "world"
   )
```

Since tree childs are represented as binary products ($NProd :: Type\ a \rightarrow Type\ b \rightarrow Type\ (a, b)$), this representation adds unnecessary child balancing information. For this reason, we consider differently balanced trees isomorphic in our model. $isomorphic :: Type\ a \rightarrow Rule$ succeeds if the argument type and the concrete view conform to this level of isomorphism. Also, the *one* and *all* combinators recurse over n-ary trees.

For simplification reasons, we will represent some n-ary tree

$$NProd\ (NProd\ a\ b)\ (NProd\ c\ d)$$

as

$$\{\!|\ a, b, c, d\ |\!\}$$

Nevertheless, all tree lenses described in [FGM+05] were derived for trees with string nodes. We keep lenses over string trees for demonstration purposes, but generalize their semantics for generic tree-structured data types.

For this diagram, consider some predicate $px : X \to P_x$ a function from X to a subset $P_x \subseteq X$. $\overline{P_x}$ is the codomain of predicate $\neg p$. The dotted arrows represent splitting due to filtering or concatenating after matching.

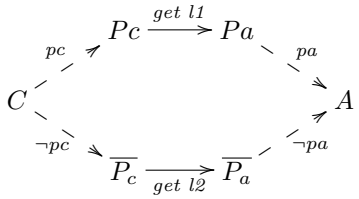


Figure 6.1: The *get* direction of *xfork*.

Hoisting and Plunging

The most basic combinators over trees are *hoist* and *plunge*, that remove and add nodes at the top of trees.

Hoisting removes the top type for trees with a single child, if it matches an input predicate. On our scenario, $hoist :: Filter \to Rule$ can then be applied to *Data* constructors, where *Filter* is a predicate over types:

type $Filter = \forall a\ .\ Type\ a \to Bool$

The string version ($hoist :: String \to Rule$) can be achieved by validating only *Data* elements with name s.

Conversely, the $plunge :: Type\ a \to Rule$ lens is used to deepen a tree by adding an edge at the top. The combinator tries to match the child of the argument type against the current concrete view's type. It is also defined only for *Data* and *List* type nodes, but on the argument type.

Plunging can easily be specialized into a string-based version, by creating a *Data* element with the string name at the top.

Forking

The lens combinator $xfork :: Filter \to Filter \to Rule \to Rule \to Rule$ splits a tree into two parts, according to the names of its immediate children, to which it applies separate lenses. Formally, two predicates *pc* and *pa* and two lenses *l1* and *l2* are used. In the *get* direction, *pc* splits the concrete tree into two parts, where *l1* is applied to the elements conforming to *pc* and *l2* to the leftovers. All the elements in both resulting the abstract trees must obey to *pa* and $\neg pa$, respectively. Conversely, in the *putback* direction, *l1* must map abstract elements from *pa* to concrete elements in *pc*, and *l2* from $\neg pa$ to $\neg pc$.

In our model, predicates are applied at the type-level, and not along value-transformers, since node names and relations are parameterized in the type representations. Forking involves two operations: *filtering*, that corresponds to select the tree children which names match an argument predicate; and *matching*, that denotes structure-preservation after filtering, in the sense that applying a filter does not alter the type structure.

The *fork* combinator is a simpler form of *xfork*, where both predicates are the same:

$fork :: Filter \to Rule \to Rule \to Rule$
$fork\ p\ l1\ l2 = xfork\ p\ p\ l1\ l2$

The *filter′* combinator recurses over n-ary products by invoking the *all* generic combinator. If a tree node matches the argument predicate, it is left unchanged, ohterwise *const* {|} is applied:

```
filter′ :: Filter → Default → Rule
filter′ p d t@(NProd _ _) = all (filter p d) t
filter′ p d t = if (p t) then nop t else const {|} d t
```

{|} represents the empty tree and is encoded in a similar way to the unit type:

```
data {|} = {|}
data Type a where
  ...
  {|} :: Type {|}
```

Remark Inasmuch {|} allows one single value {|}, after applying lens *const* {|} *d*, the abstract values are not subject to change and, therefore, the lens is well-behaved without the need to a default value generator. However, when putting an abstract tree back into a missing concrete value, *d* provides default information that does not appear in the abstract tree but is required in the concrete tree.

Consider that, for example tree {| *a*, *b*, *c* |}, we are filtering elements with name *a*:

```
filter′ (≡ a) (const {|} d) {| a, b, c |}
```

Then, the abstract tree becomes:

```
{| a, {|}, {|} |}
```

Using {|} provides a transformation for filtered out elements, enabling the type-safe implementation of *filter*. Although replacing removed elements by {|} is necessary, these empty trees they are undesired in the target abstract tree. Therefore, we have implemented the lens *remove_{|}s* to accomplish removal of {|} elements:

```
remove_{|} :: Rule
remove_{|} (NProd a {|}) = return (View π2_lns {|})
remove_{|} (NProd {|} b) = return (View π1_lns {|})
remove_{|} t = Nothing

remove_{|}s :: Rule
remove_{|}s = innermost remove_{|}
```

Now we can redefine *filter* as the sequentiation of *remove_{|}s* after the filtering operation itself (*filter′*):

```
filter :: Filter → Default → Rule
filter p d = filter′ p d⋙remove_{|}s
```

For the previous example tree, the abstract tree does not have empty trees for filtered out elements any more:

```
{| a |}
```

Done with filtering, matching is encoded by checking if all child elements conform to a filtering predicate:

```
match :: Filter → Default → Rule
match p d t = do
  View l1 t1 ← filter p d t
  Eq ← teq t t1
  nop t
```

For instance,

$$match\ (\equiv a)\ \{|\ a, b, c\ |\}$$

fails because elements b and c do not obey to the predicate $(\equiv a)$.

In Harmony [FGM+05], *filter* is defined as a derived form of *fork*. This happens because it represents more than pure filtering, since the abstract view is also filtered. In our model, lenses define strict type transformations, and the abstract view structure is only subject to change if the argument lenses modify the type of elements. For such cases, we provide a fork-based implementation of filter:

$$forkfilter :: Filter \rightarrow Default \rightarrow Rule$$
$$forkfilter\ p\ d = fork\ p\ nop\ (const\ \{|\}\ d)$$

The *prune* combinator removes all matches for a specific child name (originally it just removed one, since the model assumed unique node names). This is defined as an opposite *fork* to the *filter* definition:

$$prune :: Filter \rightarrow Default \rightarrow Rule$$
$$prune\ n\ d = fork\_tr\ n\ (const\ ()\ d)\ nop$$

Consequently, this definition is equivalent to defining a negative filtering:

$$prune :: Filter \rightarrow Default \rightarrow Rule$$
$$prune\ n\ d = filter\ (\neg \circ n)\ d$$

In the original definition, since we were pruning a single node, the default value was encapsulated under the same node name. In our encoding, node names are represented as functions and we cannot derive node names from predicate functions. Therefore, no operations are performed over the default value.

We can focus attention on a single child with the *focus* combinator:

$$focus :: Filter \rightarrow Default \rightarrow Rule$$
$$focus\ n\ d = filter\ n\ d \ggg hoist\ n$$

The $hoist\_non - unique$ lens is a version of *hoist* that does not require a child to be unique, but still with an unique name:

$$hoist\_non - unique :: Filter \rightarrow Filter \rightarrow Rule$$
$$hoist\_non - unique\ n\ p = xfork\ n\ p\ (hoist\ n)\ nop$$

The last forking combinator is the rename combinator that renames all occurrences of a name m to n. It has no generic version since it performs explicitly over node names.

$$rename :: String \rightarrow String \rightarrow Rule$$
$$rename\ m\ n = xfork\ (\equiv m)\ (\equiv n)\ (map\ (hoist\ m \ggg plunge\ n))\ nop$$

However, this version requires the name n not to exist in the view. Additionally, *xfork* does not maintain the order of elements.

All these drawbacks can be solved by implementing a version based on *map*:

$$rename :: String \rightarrow String \rightarrow Rule$$
$$rename\ m\ n = map\ ((hoist\ m \ggg plunge\ n)\ |||\ nop)$$

Mapping

The $map :: Rule \rightarrow Rule$ combinator for trees has already been implemented in our solution when defining the *all* generic combinator with recursion for n-ary products.

6.4.3 Conditional combinators

Conditional lens are usually very pratical for the definition of complex lenses, but they are however tricky: the abstract view needs to be guaranteed to be sent to the same sub-lens on the way down as we took on the way up.

As stated before, we have no means of formally guaranteeing this requirement in our system, besides returning runtime errors when the abstract values violate the abstract domain.

There are two types of conditionals: the ones that apply predicates over types (*tcond*); and the ones that perform over concrete values (*ccond*), abstract values (*acond*), or both (*cond*).

tcond is the only type-safe conditional combinator and applies one of the argument rules according to a conditional guard over the concrete type:

$$
\begin{aligned}
&tcond :: Filter \rightarrow Rule \rightarrow Rule \rightarrow Rule \\
&tcond\ f\ r\ s = (\lambda t \rightarrow \mathbf{do}\ \{(guard\ f\ t); r\ t\})\ s
\end{aligned}
$$

For value conditions, we do not present implementations since we cannot guarantee their type-safety.

6.4.4 XPath and Lenses

Recall the type from figure 6.1 that represents an XML Schema. If we would like to select all years inside movies, we would write a lens such as

$$
\begin{aligned}
&lens :: Lens\ Imdb\ Year \\
&lens = list\_lns\ \Omega\ (\pi1\_lns \bullet \pi2\_lns \bullet movie\_lns) \bullet \pi1\_lns \bullet imdb\_lns
\end{aligned}
$$

and apply the *get* function to execute the query over the concrete schema.

Here, *list_lns* defines lens application over lists, and *movie_lns* and *imdb_lns* are default type encapsulators similar to the ones in Chapter 3.

After, we could change all the results to 2000

$$
\begin{aligned}
&years2000 :: [\,Year\,] \rightarrow [\,Year\,] \\
&years2000 = list\ (\lambda(Year\ \_) \rightarrow Year\ 2000)
\end{aligned}
$$

and *putback* the changes into the original XML tree.

However, the created lens is no more than the result of specializing the XPath query //movie/year against the schema into a point-free expression.

Therefore, in the former world of data transformation, lenses can also play a role in the latter case of type data querying, since it is compatible with the evolution through abstraction philosophy of lenses. Additionally, they can also provide reverse transformations for data selectors.

Another curious fact is that, in some cases, it is useful to express lenses in structure-oriented languages such as XPath.

6.5 Supporting Single-Recursion

For this section, recall the functor-based representation of types proposed in Chapter 4.

Since our solution provides distinct type and functor representations, there is the need to define a lens combinators *flns* that unfolds a functor lens into a type lens, making it applicable to unfolded values of recursive types.

Consider functors f and g that uniquely identify types a and b, respectively. If we can define some functor lens $Lens\ (f\ x)\ (g\ x)$ over an arbitrary type x, then *flns* unfolds it into the a type lens $Lens\ a\ b$:

$$
\begin{aligned}
&flns :: (FunctorOf\ f\ a, FunctorOf\ g\ b) \Rightarrow \forall x\ .\ Lens\ (f\ x)\ (g\ x) \rightarrow Lens\ a\ b \\
&flns\ l = Lens\ g\ p \\
&\quad \mathbf{where} \\
&\quad g = nu\ (get\ l) \\
&\quad p = nu\ (putback\ l) \circ fzip
\end{aligned}
$$

In the *putback* direction, updated abstract and original concrete values must be combined into updated concrete values. This merging of recursive types requires fusing their functor definitions into a new functor, that can be produced by an anamorphism. Consider *fzip* the functor zipping task and *ffuse* the operation on *fzip* that calculates the fusion of functors.

However, calculating the fusion of two functors depends on their recursive invocations and requires great reflection on their properties. The fusion of equivalent functor representations is trivial to define. For infinite streams of elements of type a and b:

$$\mathit{fzip}\ (K\ a \otimes \mathit{Id})\ (K\ b \otimes \mathit{Id}) = (K\ a \otimes K\ b) \otimes \mathit{Id}$$

In a more realistic scenario, functors have different recursive structures. For instance, if we are fusing an infinite stream s of values of type a with a finite list l of b elements, the result will be a finitely-sized list with n elements, where n is the the maximum of the sizes of s and l. Also, both s and l may have less elements than the other, what implies optionality in their fused representation. Considering $P\ x = K\ x \otimes K\ ()$, we can write:

$$\mathit{ffuse}\ (K\ a \otimes \mathit{Id})\ (K\ () \oplus (K\ b \otimes \mathit{Id})) = K\ () \oplus ((P\ a \otimes P\ b) \otimes \mathit{Id})$$

The incapacity to generically combine functor representations determines our inability to implement explicitly recursive lenses.

6.6 Summary

At the end of the chapter, we have developed a type-safe Haskell rewrite system for bidirectional lenses, by adapting the implementation structures from the two-level rewrite system.

Our solution mimics the original implementation from [FGM+05], where no theoretical properties of lenses are enforced, but we put particular effort in guaranteeing that our lenses are type-safe, in the sense that they are well-behaved for all their concrete and abstract domains.

The greatest difference is that, similarly to two-level transformations, we define lens transformations for types and couple it with value migrations. In the original implementation, there is no distinction between types and values, where value trees contain an implicit type structure.

In the future, we shall consider providing the current lenses as valid compound pairs of two-level transformations, and developing new lenses based on this concept.

Chapter 7

Conclusions and Future Work

In this report, we have highlighted and presented solutions for some limitations or inabilities related to the 2LT project.

In particular, we have made the following contributions:

1. We have solved untermination of partial backward transformations in two-level transformations, and proved that the refinement properties hold for explicitely partial abstraction functions.

2. We have discussed possible extensions to the one-level rewrite system in order to support recursive types and ended up developing a new generic rewrite system from scratch.

3. We have studied the limitations on two-level transformations and developed a type-safe library for bidirectional transformations with stateful backward transformations (lenses) for non-recursive types.

The source code for the implemented prototypes can be found at the subversion repositories 2LT and RHS under `http://Haskell.di.uminho.pt/svn`.

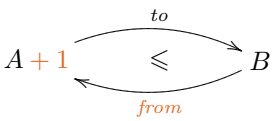
7.1 Future Work

Naturally, there are still some interesting pratical and theoretical issues that we would like to overcome in the future.

Invariants for Two-level Transformations Although explicit partiality ensures more safeness in two-level transformations, we still lack support for type invariants. Preserving invariants is useful because they can be important for the definition of the source format. These may assume different contexts, and two examples are invariants encoded in VDM models or imposed by map primitives (for example, *mapkeys* f requires f to be injective).
An invariant can be assumed as a coreflexive function that filters the valid elements of some type A [Oli04], such that

$$inv_A :: A \to A + 1$$

Considering some partial two-level transformation:



an invariant for B values holds if the original invariant holds for A values converted from those B values. Therefore, it can be calculated through composition of the *from* abstraction with the original invariant over A[1]:

$$inv_B :: B \to B + 1$$
$$inv_B = (\pi_2 \ + \ \underline{1}) \circ distl \circ (coassocr \times id) \circ ((inv_A \ + \ id) \times id) \circ (from \triangle id)$$

Recursivity for One-level Transformations In spite of the fact that we have failed to find an intuitive approach to implement a type-safe Haskell representation for functions over fixpoint recursive types, there are still some "open" options that may empower new developments.
Our current issue is to natively convert functor representations into corresponding type representations, in order to enable the usage of equivalent combinators for functions over the two.
The result type for the function $apF :: Fctr\ f \to Type\ a \to Type\ b$ is dependant on the inputs but can be derived for each specific case.
If we manage to perform this computation dynamically, generating specific Haskell source code, these changes may be propagated to the original program by re-compilation or evaluation of such code at runtime, in a similar form to staged meta-programming.
Existing tools to control the processing of Haskell modules are: 'GHC as a library', that provides an API for importing the GHC as module and manipulate compiler actions; and hs-plugins[PSSC04], for dynamically loading Haskell modules at runtime.

Generic Rewrite System Although it represents just one of this report's research topics, our prototype for a generic rewrite system is the most complex developed tool and still suffers from various limitations.
The most significant one is its poor performance, compared with the Haskell implementation.
A possible research direction to solve this problem is to study the usage of template meta-programming language extensions, such as Template Haskell [SP02]. In order to overcome the limitations of embedding, meta-programming enables compile-time preprocessing of source programs and provides a framework for teaching the compiler about domain-specific optimizations [COST]. An example of the advantages of using TH is described in [Lyn03], where an Haskell program that generates images of the Mandelbrot set is optimized by unrolling fixed depth recursions at compile-time.
Performance could also be improved if we considered the generation of Haskell source code for rewrite rules and declarations and further compilation into specific rewrite systems.

Associativity is not supported in the current implementation, because it compromises even more the efficiency of the rewrite system. It may occur in rule patterns or in the application of the generic combinators *one* and *all*. Suppose an example rule r and an expression e, such that:

$$r : a \circ b \to c$$
$$e = x \circ y \circ z$$

If we are trying to unify e with the pattern of r, two expressions should be considered valid for pattern matching: $r\ (x \circ y) \circ z$ and $x \circ r\ (y \circ z)$. Generally, if a rule pattern is an associative term with arity n, it should match any expression for the same term with n or less arguments.
When applying an associative rule inside an associative expressions, *one* and *all* mean more than simply applying the argument rule to each subexpressions.
Therefore, *one* r will only be guaranteed to succeed if r is applied to every possible partitions that split e in 1 to n pieces, where n is the arity of e.
For the above example, *one* r would be applied until success to any of the expressions resulting from splitting e in one ($r\ (x \circ y \circ z)$), two ($r\ (x \circ y) \circ z$ and $x \circ r\ (y \circ z)$) or three ($r\ x \circ y \circ z$ and $x \circ r\ y \circ z$ and $x \circ y \circ r\ z$) parts.

[1] The diagram of the full derivation of inv_B can be consulted in Annex C.

For *all* r, the same logic can not be used, since application of r to a term with no parameters always succeeds and does not allow this "try until success" algorithm.

Our generic rewrite system receives expressions and rewrite rules and outputs rewritten expressions. However, what happens inside the "black box" is not always intuitive, mostly due to laziness and highly complex rewrite strategies.
Although tracing provides knowledge on the intermediate data structures, it generates great amounts of information and easily becomes unreadable to human perception.
Actually, we provide traces on textual form, for each resultant expression.
In the future, we intend to provide a GUI for trace visualization, taking in account the visualization mechanism studied for Pointless Haskell[Cun05]. The visual tool would read traces from a specific file format and allow the user to navigate through the tracing tree with features to look inside intermediate expressions (think of XML) and see where exactly are rules being applied. Another interesting feature would be to derive tracing trees for subexpressions, exposing the evolution of particular structures.

Additionally, features that we consider as future work are:

- Add support for polymorphic types. In the prototype version, only monoporphic types are supported. This is a great limitation to the type-safety of transformations.
- Add support for two-level transformations. This would imply specific syntax for definition of type transformations and of value transformers.
- Develop macros for helping the development of language plugins, in the form of interfaces for different languages. For example, front-ends for XPath and Pointless Haskell are a priority.
- Support expression equality. For instance, monoid instances could be considered equivalent to terms of specifc types. For lists, $mzero\ [t] \cong nil$ and $mplus\ [t] \cong listcat$.
- Provide advanced features for manipulation of internal representations and execution of Haskell monadic code in the right hand side of rules.
- Add more context-sensitive features, such as prohibiting the application of specific rules inside an expression.
- Allow different levels of precedence for binary combinators.

Lenses In our research, lenses are a fresh topic and deserve more study to understand their relationship with two-level transformations, if it is possible to compose both, and the properties that would guarantee bdirectionality of resultant transformations.
Nevertheless, solving the generic fusion of functors in order to support single recursive type-safe lenses is a priority.

Bibliography

[AC07] Michal Antkiewicz and Krzysztof Czarnecki. Design space of round-trip engineering. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE 2007 Proceedings*, pages 1–12, July 2007.

[BCPV07] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data: Conversion for xml and sql. In *PADL 2007*, pages 290–304. Springer-Verlag, LNCS 4085, February 2007.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[COST] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++.

[COV06] A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[Cun05] Alcino Cunha. *Point-free Program Calculation.* PhD thesis, Department of Informatics, University of Minho, 2005.

[CV06] A. Cunha and J. Visser. Transformation of structure-shy programs: Applied to xpath queries and strategic functions. In *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2006.

[CV07] Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. *Electron. Notes Theor. Comput. Sci.*, 174(1):17–34, 2007.

[Der05] Nachum Dershowitz. Term rewriting systems by "terese" (marc bezem, jan willem klop, and roel de vrijer, eds.), cambridge university press, cambridge tracts in theoretical computer science 55, 2003, hard cover: Isbn 0-521-39115-6, xxii+884 pages. *Theory Pract. Log. Program.*, 5(3):395–399, 2005.

[FGM+05] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 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.

[FP07] Flávio Ferreira and Hugo Pacheco. Xpto - an xpath preprocessor with type-aware optimization. In *CORTA - Compilers, Related Technologies and Applications Workshop*. to appear, February 2007.

[HLO06] R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In M. Hagiya and P. Wadler, editors, *Proc. Functional and Logic Programming, 8th Int. Symp.*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

[JHH+93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 93.

[JJM97] S. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space, 1997.

[JM02] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, 2002.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[Läm06] R. Lämmel. Scrap your boilerplate with XPath-like combinators, 15 July 2006. Draft, 6 pages, Accepted as short paper at POPL 2007.

[LJ05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 204–215, New York, NY, USA, 2005. ACM Press.

[LL01] Ralf Lämmel and Wolfgang 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.

[LM06] R. Lämmel and E. Meijer. Mappings make data processing go 'round — An inter-paradigmatic mapping tutorial. In *Post-proceedings of GTTSE 2005, Generative and Transformation Techniques in Software Engineering, 4–8 July, 2005, Braga, Portugal*, Lecture Notes in Computer Science. Springer-Verlag, 2006. Summer school tutorial, GTTSE 2005, 50 pages, to appear.

[LO94] Konstantin Läfer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[Luc02] Salvador Lucas. Context-sensitive rewriting strategies. *Inf. Comput.*, 178(1):294–343, 2002.

[Lyn03] Ian Lynagh. Unrolling and simplifying expressions with template haskell. May 2003.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[Oli04] J. N. Oliveira. Constrained datatypes, invariants and business rules: a relational approach. In *PUReCafé, DI-UM, 2004.5.20 [talk]*. PURe Project (POSI/CHS/44304/2002), 2004.

[Oli07] J.N. Oliveira. Data transformation by calculation. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE 2007 Proceedings*, pages 139–198, July 2007.

[Pie06] Benjamin C. Pierce. The weird world of bi-directional programming, March 2006. ETAPS invited talk.

[PSSC04] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging haskell in. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 10–21, New York, NY, USA, 2004. ACM Press.

[PWW04] 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.

[Rey77] John C. Reynolds. Semantics of the domain of flow diagrams. *J. ACM*, 24(3):484–503, 1977.

[SP02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM Press.

Appendix A

Point-free laws

We now present some point-free laws required during the implementation of our 1LT rewrite system:

$$out_X \circ in_X = id \qquad \textit{data}\text{-Cancel}$$

$$list\ f \circ concat \circ list\ g = concat \circ list\ (list\ f \circ g) \qquad \textit{concat}\text{-Fusion}$$

$$\left.\begin{array}{l} concat \circ wrap = id \\ concat \circ list\ wrap = id \end{array}\right\} \qquad \textit{concat}\text{-Cancel}$$

$$list\ f \circ (g\ \nabla\ h) = (list\ f \circ g)\ \nabla\ (list\ f \circ h) \qquad \textit{list}\text{-Either}$$

$$\left.\begin{array}{l} list\ f \circ wrap \circ g = wrap \circ f \circ g \\ list\ f \circ mzero\ r = mzero\ r \end{array}\right\} \qquad \textit{list}\text{-Cancel}$$

$$\left.\begin{array}{l} (geq \circ (f\ \triangle\ g) \circ h\ ?\ a : b) \circ c = a \circ x, \textbf{if}\ (h \circ x \equiv g) \\ (geq \circ (f\ \triangle\ g) \circ h\ ?\ a : b) \circ c = a \circ x, \textbf{if}\ (h \circ x \equiv f) \end{array}\right\} \qquad \textit{cond}\text{-Cancel}$$

$$\left.\begin{array}{l} \pi_1 \circ assocl = id \times \pi_1 \\ \pi_2 \circ assocl = \pi_2 \circ \pi_2 \end{array}\right\} \qquad \textit{assocl}\text{-Cancel}$$

$$\left.\begin{array}{l} \pi_1 \circ assocr = \pi_1 \circ \pi_1 \\ \pi_2 \circ assocr = \pi_2 \times id \end{array}\right\} \qquad \textit{assocr}\text{-Cancel}$$

$$\left.\begin{array}{l} coassocl \circ i_1 = i_1 \circ i_1 \\ cossocl \circ i_2 = i_2\ +\ id \\ coassocl \circ i_1 \circ i_2 = i_2 \circ i_1 \\ coassocl \circ i_2 \circ i_2 = i_2 \\ coassocl \circ (f\ +\ g \circ i_1) = i_1 \circ (f\ +\ g) \\ coassocl \circ (f\ +\ g \circ i_2) = (i_1 \circ f)\ +\ g \end{array}\right\} \qquad \textit{coassocl}\text{-Cancel}$$

$$\left.\begin{array}{l} coassocr \circ i_2 = i_2 \circ i_2 \\ cossocr \circ i_1 = id\ +\ i_1 \\ coassocr \circ i_1 \circ i_1 = i_1 \\ coassocr \circ i_2 \circ i_1 = i_1 \circ i_2 \\ coassocr \circ (f \circ i_1\ +\ g) = g\ +\ (i_2 \circ f) \\ coassocr \circ (f \circ i_2\ +\ g) = i_2 \circ (f\ +\ g) \end{array}\right\} \qquad \textit{coassocr}\text{-Cancel}$$

$$\left.\begin{array}{l} \pi_1 \circ swap = \pi_2 \\ \pi_2 \circ swap = \pi_1 \end{array}\right\} \qquad \textit{swap}\text{-Cancel}$$

$$\left.\begin{array}{l} swap \circ swap = id \\ swap \circ (f \times g) = (g \times f) \circ swap \end{array}\right\} \qquad \textit{swap}\text{-Reflex}$$

$$\left.\begin{array}{l} coswap \circ i_1 = i_2 \\ coswap \circ i_2 = i_1 \end{array}\right\} \qquad \textit{coswap}\text{-Cancel}$$

$$\left.\begin{array}{l} coswap \circ coswap = id \\ coswap \circ (f\ +\ g) = (g\ +\ f) \circ coswap \end{array}\right\}$$ *coswap*-Reflex

$$\left.\begin{array}{l} distl \circ ((i_1 \circ f) \triangle g) = i_1 \circ (f \triangle g) \\ distl \circ ((i_1 \circ f) \times g) = i_1 \circ (f \times g) \\ distl \circ ((i_2 \circ f) \triangle g) = i_2 \circ (f \triangle g) \\ distl \circ ((i_2 \circ f) \times g) = i_2 \circ (f \times g) \end{array}\right\}$$ *distl*-lCancel

$$\left.\begin{array}{l} ((f \circ \pi_1) \nabla (g \circ \pi_1)) \circ distl = (f \nabla g) \circ \pi_1 \\ ((f \circ \pi_1)\ +\ (g \circ \pi_1)) \circ distl = (f\ +\ g) \circ \pi_1 \end{array}\right\}$$ *distl*-rCancel

$$\left.\begin{array}{l} distr \circ (f \triangle (i_1 \circ g)) = i_1 \circ (f \triangle g) \\ distr \circ (f \times (i_1 \circ g)) = i_1 \circ (f \times g) \\ distr \circ (f \triangle (i_2 \circ g)) = i_2 \circ (f \triangle g) \\ distr \circ (f \times (i_2 \circ g)) = i_2 \circ (f \times g) \end{array}\right\}$$ *distr*-lCancel

$$\left.\begin{array}{l} ((f \circ \pi_2) \nabla (g \circ \pi_2)) \circ distr = (f \nabla g) \circ \pi_2 \\ ((f \circ \pi_2)\ +\ (g \circ \pi_2)) \circ distr = (f\ +\ g) \circ \pi_2 \end{array}\right\}$$ *distr*-rCancel

$$\left.\begin{array}{l} distl \circ undistl = id \\ undistl \circ distl = id \end{array}\right\}$$ *distl*-Reflex

$$\left.\begin{array}{l} distr \circ undistr = id \\ undistr \circ distr = id \end{array}\right\}$$ *distr*-Reflex

$$\left.\begin{array}{l} ((f \circ \pi_1) \nabla (g \circ mzero\ r)) \circ distl = (f \nabla (g \circ mzero\ r)) \circ \pi_1 \\ ((f \circ mzero\ r) \nabla (g \circ \pi_1)) \circ distl = ((f \circ mzero\ r) \nabla g) \circ \pi_1 \end{array}\right\}$$ *distl*-rCancel

$$\left.\begin{array}{l} ((f \circ \pi_2) \nabla (g \circ mzero\ r)) \circ distr = (f \nabla (g \circ mzero\ r)) \circ \pi_2 \\ ((f \circ mzero\ r) \nabla (g \circ \pi_2)) \circ distr = ((f \circ mzero\ r) \nabla g) \circ \pi_2 \end{array}\right\}$$ *distr*-rCancel

$$\left.\begin{array}{l} f \circ (\!|mzero\ r \nabla g \circ \pi_1|\!)_{\mu F} = (\!|f \circ mzero\ r \nabla f \circ g \circ \pi_1|\!)_{\mu F} \\ list\ f \circ (\!|mzero\ r \nabla mplus\ r \circ (g \times \pi_1)|\!)_{\mu F} = (\!|mzero\ r \nabla mplus\ r \circ (list\ f \circ g \times \pi_1)|\!)_{\mu F} \\ concat \circ (\!|mzero\ r \nabla mplus\ r \circ (g \times \pi_1)|\!)_{\mu F} = (\!|mzero\ r \nabla mplus\ r \circ (concat \circ g \times \pi_1)|\!)_{\mu F} \end{array}\right\}$$ *para*-nilFusion

$$\left.\begin{array}{l} f \circ (\!|mzero\ r \nabla g \circ \pi_1|\!)_{\mu F} = (\!|f \circ mzero\ r \nabla f \circ g \circ \pi_1|\!)_{\mu F} \\ list\ f \circ (\!|mzero\ r \nabla mplus\ r \circ (g \times id)|\!)_{\mu F} = (\!|mzero\ r \nabla mplus\ r \circ (list\ f \circ g \times id)|\!)_{\mu F} \\ concat \circ (\!|mzero\ r \nabla mplus\ r \circ (g \times id)|\!)_{\mu F} = (\!|mzero\ r \nabla mplus\ r \circ (concat \circ g \times id)|\!)_{\mu F} \end{array}\right\}$$ *cata*-nilFusion

$(\!|mzero\ r \nabla f \circ (id \times \pi_1)|\!)_{\mu F} = (\!|mzero\ r \nabla f|\!)_{\mu F}$ *para*-nilCata

$(\!|mzero\ r \nabla mplus\ r \circ (wrap \circ f \times \pi_1)|\!)_{\mu F} = list\ f$ *para*-listCancel

$(\!|mzero\ r \nabla mplus\ r \circ (wrap \circ f \times id)|\!)_{\mu F} = list\ f$ *cata*-listCancel

$forall\ f, g : f, g\ injective\ .\ mapkeys\ f \circ mapkeys\ g = mapkeys\ (f \circ g)$ *mapkeys*-Absorption

$map\ f \circ map\ g = map\ (f \circ g)$ *map*-Absorption

$mapkeys\ f \circ map\ g = map\ g \circ mapkeys\ f$ *map*-Reflex

$$\left.\begin{array}{l} mapkeys\ f \circ empty = empty \\ map\ f \circ empty = empty \end{array}\right\}$$ *map*-Cancel

$$\left.\begin{array}{l} uncojoin \circ map\ i_1 = id \triangle (empty \circ \underline{1}) \\ uncojoin \circ map\ i_2 = (empty \circ \underline{1}) \triangle id \end{array}\right\}$$ *uncojoin*-Cancel

$$\left.\begin{array}{l} unpeither \circ mapkeys\ i_1 = id \triangle (empty \circ \underline{1}) \\ unpeither \circ mapkeys\ i_2 = (empty \circ \underline{1}) \triangle id \end{array}\right\}$$ *unpeither*-Cancel

$$\left.\begin{array}{l} peither \circ ((empty \circ g) \triangle f) = mapkeys\ i_1 \circ f \\ peither \circ (f \triangle (empty \circ g)) = mapkeys\ i_1 \circ f \\ unpeither \circ empty = empty \triangle empty \\ uncojoin \circ empty = empty \triangle empty \\ cojoin \circ ((empty \circ g) \triangle f) = map\ i_2 \\ cojoin \circ (f \triangle (empty \circ g)) = map\ i_1 \end{array}\right\} \quad empty\text{-}\textsc{Absorption}$$

Appendix B

Proofs for partialized refinements

In order to prove that a rule is a refinement we need to, for each pattern, calculate the value-level abstraction function *from* and guarantee that

$$from \circ to \sqsubseteq i_1$$

Refinement properties will be proven for *one* and *all*.

B.1 One

B.1.1 Left Either

Assuming that $A + 1 \underset{from'}{\overset{to'}{\leqslant}} B$, then $(A + C) + 1 \underset{from}{\overset{to}{\leqslant}} B + C$

$$
\begin{array}{l}
B + C \\
\quad \downarrow {\scriptstyle from' + id} \\
(A + 1) + C \\
\quad \downarrow {\scriptstyle coassocr} \\
A + (1 + C) \\
\quad \downarrow {\scriptstyle id + coswap} \\
A + (C + 1) \\
\quad \downarrow {\scriptstyle coassocl} \\
(A + C) + 1
\end{array}
\quad
\left[
\begin{array}{ll}
 & from \circ to \\
= & \quad \{ \text{ definition of } to \text{ and } from \} \\
 & coassocl \circ (id \ + \ coswap) \circ coassocr \circ (from' \ + \ id) \circ (to' \ + \ id) \\
= & \quad \{ +\text{-Absorption}; \ from' \circ to' = i_1 \} \\
 & coassocl \circ (id \ + \ coswap) \circ coassocr \circ (i_1 \ + \ id) \\
= & \quad \{ coassocr\text{-Cancel}; \ +\text{-Absorption} \} \\
 & coassocl \circ (id \circ id \ + \ coswap \circ i_2) \\
= & \quad \{ nat\text{-Id}; \ coswap\text{-Cancel}; \ coassocl\text{-Cancel} \} \\
 & i_1 \circ (id \ + \ d) \\
= & \quad \{ +\text{-}functor\text{-Id}; \ nat\text{-Id} \} \\
 & i_1
\end{array}
\right.
$$

B.1.2 Right Either

Assuming that $A + 1 \underset{from'}{\overset{to'}{\leqslant}} B$, then $(C + A) + 1 \underset{from}{\overset{to}{\leqslant}} C + B$

$$
\begin{array}{l}
C + B \\
\quad\downarrow{\scriptstyle id + from'} \\
C + (A + 1) \\
\quad\downarrow{\scriptstyle coassocl} \\
(C + A) + 1
\end{array}
\left[\begin{array}{ll}
 & from \circ to \\
= & \quad \{ \text{definition of } to \text{ and } from \} \\
 & coassocl \circ (id + from') \circ (id + to') \\
= & \quad \{ +\text{-Absorption}; \ from' \circ to' = i_1; \ nat\text{-Id} \} \\
 & coassocl \circ (id \ + \ i_1) \\
= & \quad \{ nat\text{-Id}; \ coswap\text{-Cancel}; \ coassocl\text{-Cancel} \} \\
 & i_1 \circ (id \ + \ id) \\
= & \quad \{ +\text{-}functor\text{-Id}; \ nat\text{-Id} \} \\
 & i_1
\end{array}\right.
$$

B.1.3 Left Product

Assuming that $A + 1 \underset{from'}{\overset{to'}{\leqslant}} B$, then $(A \times C) + 1 \underset{from}{\overset{to}{\leqslant}} B \times C$

$$
\begin{array}{l}
B \times C \\
\quad\downarrow{\scriptstyle from' \times id} \\
(A + 1) \times C \\
\quad\downarrow{\scriptstyle distl} \\
(A \times C) + (1 \times C) \\
\quad\downarrow{\scriptstyle id + \pi_1} \\
(A \times C) + 1
\end{array}
\left[\begin{array}{ll}
 & from \circ to \\
= & \quad \{ \text{definition of } to \text{ and } from \} \\
 & (id \ + \ \pi_1) \circ distl \circ (from' \times id) \circ (to' \times id) \\
= & \quad \{ \times\text{-Absorption}; \ from' \circ to' = i_1; \ nat\text{-Id} \} \\
 & (id \ + \ \pi_1) \circ distl \circ (i_1 \times id) \\
= & \quad \{ distl\text{-LCancel}; \ nat\text{-Id}; \ \times\text{-}functor\text{-Id} \} \\
 & (id \ + \ \pi_1) \circ i_1 \\
= & \quad \{ +\text{-Cancel} \} \\
 & i_1
\end{array}\right.
$$

B.1.4 Right Product

Assuming that $A + 1 \underset{from'}{\overset{to'}{\leqslant}} B$, then $(C \times A) + 1 \underset{from}{\overset{to}{\leqslant}} C \times B$

$$
\begin{array}{l}
C \times B \\
\quad\downarrow{\scriptstyle id \times from'} \\
C \times (A + 1) \\
\quad\downarrow{\scriptstyle distr} \\
(C \times A) + (C \times 1) \\
\quad\downarrow{\scriptstyle id + \pi_2} \\
(C \times A) + 1
\end{array}
\left[\begin{array}{ll}
 & from \circ to \\
= & \quad \{ \text{definition of } to \text{ and } from \} \\
 & (id \ + \ \pi_2) \circ distr \circ (id \times from') \circ (id \times to') \\
= & \quad \{ \times\text{-Absorption}; \ from' \circ to' = i_1; \ nat\text{-Id} \} \\
 & (id \ + \ \pi_2) \circ distr \circ (id \times i_1) \\
= & \quad \{ distr\text{-LCancel}; \ nat\text{-Id}; \ \times\text{-}functor\text{-Id} \} \\
 & (id \ + \ \pi_2) \circ i_1 \\
= & \quad \{ +\text{-Cancel} \} \\
 & i_1
\end{array}\right.
$$

B.1.5 Map keys

Assuming that $A + 1 \underset{from'}{\overset{to'}{\leqslant}} B$ and $from'$ is injective,

, then $(A \rightharpoonup C) + 1 \;\underset{from}{\overset{to}{\leqslant}}\; B \rightharpoonup C$

$$
\begin{array}{c}
B \rightharpoonup C \\
\downarrow{\scriptstyle mapkeys\ from'} \\
(A+1) \rightharpoonup C \\
\downarrow{\scriptstyle unpeither} \\
(A \rightharpoonup C) \times (1 \rightharpoonup C) \\
\downarrow{\scriptstyle (geq \circ (id \vartriangle (empty \circ \underline{1})) \circ \pi_2)\,?\,i_1 \circ \pi_1 : i_2 \circ \underline{1}} \\
(A \rightharpoonup C) + 1
\end{array}
$$

$$
\begin{array}{ll}
\lceil & from \circ to \\
= & \quad \{ \text{definition of } to \text{ and } from \} \\
& ((geq \circ (id \vartriangle (empty \circ \underline{1})) \circ \pi_2)\,?\,i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ unpeither \circ mapkeys\ from' \circ mapkeys\ to' \\
= & \quad \{ to', from'\ injective \Rightarrow mapkeys\text{-}\textsc{Absorption};\ from' \circ to' = i_1 \} \\
& ((geq \circ (id \vartriangle (empty \circ \underline{1})) \circ \pi_2)\,?\,i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ unpeither \circ mapkeys\ i_1 \\
= & \quad \{ unpeither\text{-}\textsc{Cancel} \} \\
& ((geq \circ (id \vartriangle (empty \circ \underline{1})) \circ \pi_2)\,?\,i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (id \vartriangle (empty \circ \underline{1})) \\
= & \quad \{ cond\text{-}\textsc{Cancel} \} \\
& i_1 \circ \pi_1 \circ (id \vartriangle (empty \circ \underline{1})) \\
= & \quad \{ \times\text{-}\textsc{Cancel};\ nat\text{-}\textsc{Id} \} \\
& i_1 \quad \rfloor
\end{array}
$$

B.1.6 Map values

Assuming that $A + 1 \;\underset{from'}{\overset{to'}{\leqslant}}\; B$, then $(C \rightharpoonup A) + 1 \;\underset{from}{\overset{to}{\leqslant}}\; C \rightharpoonup B$

$$
\begin{array}{c}
C \rightharpoonup B \\
\downarrow{\scriptstyle map\ from'} \\
C \rightharpoonup (A+1) \\
\downarrow{\scriptstyle uncojoin} \\
(C \rightharpoonup A) \times (C \rightharpoonup 1) \\
\downarrow{\scriptstyle (geq \circ (id \vartriangle (empty \circ \underline{1})) \circ \pi_2)\,?\,i_1 \circ \pi_1 : i_2 \circ \underline{1}} \\
(C \rightharpoonup A) + 1
\end{array}
$$

$$
\begin{bracket}
& \mathit{from} \circ \mathit{to} \\
= & \quad \{ \text{definition of } \mathit{to} \text{ and } \mathit{from} \} \\
& ((\mathit{geq} \circ (\mathit{id} \vartriangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ \mathit{uncojoin} \circ \mathit{map}\ \mathit{from}' \circ \mathit{map}\ \mathit{to}' \\
= & \quad \{ \mathit{map}\text{-Absorption}; \mathit{from}' \circ \mathit{to}' = i_1 \} \\
& ((\mathit{geq} \circ (\mathit{id} \vartriangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ \mathit{uncojoin} \circ \mathit{map}\ i_1 \\
= & \quad \{ \mathit{uncojoin}\text{-Cancel} \} \\
& ((\mathit{geq} \circ (\mathit{id} \vartriangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \vartriangle (\mathit{empty} \circ \underline{1})) \\
= & \quad \{ \mathit{cond}\text{-Cancel} \} \\
& i_1 \circ \pi_1 \circ (\mathit{id} \vartriangle (\mathit{empty} \circ \underline{1})) \\
= & \quad \{ \times\text{-Cancel}; \mathit{nat}\text{-Id} \} \\
& i_1
\end{bracket}
$$

B.2 All

B.2.1 Either

Assuming that $A + 1 \underset{\mathit{from}'}{\overset{\mathit{to}'}{\leqslant}} B$ and $C + 1 \underset{\mathit{from}''}{\overset{\mathit{to}''}{\leqslant}} D$

, then $(A + B) + 1 \underset{\mathit{from}}{\overset{\mathit{to}}{\leqslant}} C + D$

$$
\begin{array}{c}
C + D \\
\downarrow{\scriptstyle \mathit{from}' + \mathit{from}''} \\
(A + 1) + (B + 1) \\
\downarrow{\scriptstyle \mathit{coassocl}} \\
((A + 1) + B) + 1 \\
\downarrow{\scriptstyle (\mathit{coswap} + \mathit{id}) + \mathit{id}} \\
((1 + A) + B) + 1 \\
\downarrow{\scriptstyle \mathit{coassocr} + \mathit{id}} \\
(1 + (A + B)) + 1 \\
\downarrow{\scriptstyle \mathit{coswap} + \mathit{id}} \\
((A + B) + 1) + 1 \\
\downarrow{\scriptstyle \mathit{coassocr}} \\
(A + B) + (1 + 1) \\
\downarrow{\scriptstyle \mathit{id} + (\mathit{id} \bigtriangledown \mathit{id})} \\
(A + B) + 1
\end{array}
$$

$$
\begin{bmatrix}
 & \mathit{from} \circ \mathit{to} \\
= & \quad \{ \text{definition of } \mathit{to} \text{ and } \mathit{from} \} \\
 & (\mathit{id} \;+\; (\mathit{id} \;\nabla\; \mathit{id})) \circ \mathit{coassocr} \circ (\mathit{coswap} \;+\; \mathit{id}) \circ (\mathit{coassocr} \;+\; \mathit{id}) \circ ((\mathit{coswap} \;+\; \mathit{id}) \;+\; \mathit{id}) \\
 & \circ \mathit{coassocl} \circ (\mathit{from}' \;+\; \mathit{from}'') \circ (\mathit{to}' \;+\; \mathit{to}'') \\
= & \quad \{ +\text{-Absorption}; \; \mathit{from}' \circ \mathit{to}' = i_1 \} \\
 & (\mathit{id} \;+\; (\mathit{id} \;\nabla\; \mathit{id})) \circ \mathit{coassocr} \circ (\mathit{coswap} \;+\; \mathit{id}) \circ (\mathit{coassocr} \;+\; \mathit{id}) \circ ((\mathit{coswap} \;+\; \mathit{id}) \;+\; \mathit{id}) \\
 & \circ \mathit{coassocl} \circ (i_1 \;+\; i_1) \\
= & \quad \{ \mathit{coassocl}\text{-Cancel} \} \\
 & (\mathit{id} \;+\; (\mathit{id} \;\nabla\; \mathit{id})) \circ \mathit{coassocr} \circ (\mathit{coswap} \;+\; \mathit{id}) \circ (\mathit{coassocr} \;+\; \mathit{id}) \circ ((\mathit{coswap} \;+\; \mathit{id}) \;+\; \mathit{id}) \\
 & \circ i_1 \circ (i_1 \;+\; \mathit{id}) \\
= & \quad \{ +\text{-Cancel}; \; +\text{-Absorption} \} \\
 & (\mathit{id} \;+\; (\mathit{id} \;\nabla\; \mathit{id})) \circ \mathit{coassocr} \circ (\mathit{coswap} \;+\; \mathit{id}) \circ (\mathit{coassocr} \;+\; \mathit{id}) \circ i_1 \circ ((\mathit{coswap} \circ i_1) \;+\; \mathit{id}) \\
= & \quad \{ +\text{-Absorption}; \; +\text{-Cancel} \} \\
 & (\mathit{id} \;+\; (\mathit{id} \;\nabla\; \mathit{id})) \circ \mathit{coassocr} \circ i_1 \circ \mathit{coswap} \circ \mathit{coassocr} \circ (i_2 \;+\; \mathit{id}) \\
= & \quad \{ \mathit{coassocr}\text{-Cancel} \} \\
 & (\mathit{id} \;+\; (\mathit{id} \;\nabla\; \mathit{id})) \circ (\mathit{id} \;+\; i_1) \circ \mathit{coswap} \circ i_2 \circ (\mathit{id} \;+\; \mathit{id}) \\
= & \quad \{ +\text{-Fusion}; \; \mathit{nat}\text{-Id}; \; +\text{-Cancel}; \; \mathit{coswap}\text{-Cancel} \} \\
 & i_1
\end{bmatrix}
$$

B.2.2 Product

Assuming that $A + 1 \underset{\mathit{from}'}{\overset{\mathit{to}'}{\leqslant}} B$ and $C + 1 \underset{\mathit{from}''}{\overset{\mathit{to}''}{\leqslant}} D$

, then $(A \times B) + 1 \underset{\mathit{from}}{\overset{\mathit{to}}{\leqslant}} C \times D$

$$
\begin{array}{c}
C \times D \\
\downarrow \; \mathit{from}' \times \mathit{from}'' \\
(A + 1) \times (B + 1) \\
\downarrow \; \mathit{distl} \\
(A \times (B + 1)) + (1 \times (B + 1)) \\
\downarrow \; \mathit{distr} + \pi_1 \\
((A \times B) + (A \times 1)) + 1 \\
\downarrow \; \mathit{coassocr} \\
(A \times B) + ((A \times 1) + 1) \\
\downarrow \; \mathit{id} + (\pi_2 \;\nabla\; \mathit{id}) \\
(A \times B) + 1
\end{array}
$$

$$
\begin{array}{ll}
\lceil & \mathit{from} \circ \mathit{to} \\
= & \quad \{ \text{definition of } \mathit{to} \text{ and } \mathit{from} \} \\
& (\mathit{id} \ + \ (\pi_2 \nabla \mathit{id})) \circ \mathit{coassocr} \circ (\mathit{distr} \ + \ \pi_1) \circ \mathit{distl} \circ (\mathit{from}' \times \mathit{from}'') \circ (\mathit{to}' \times \mathit{to}'') \\
= & \quad \{ +\text{-Absorption}; \ \mathit{from}' \circ \mathit{to}' = i_1 \} \\
& (\mathit{id} \ + \ (\pi_2 \nabla \mathit{id})) \circ \mathit{coassocr} \circ (\mathit{distr} \ + \ \pi_1) \circ \mathit{distl} \circ (i_1 \times i_1) \\
= & \quad \{ \mathit{distl}\text{-lCancel}; \ +\text{-Cancel} \} \\
& (\mathit{id} \ + \ (\pi_2 \nabla \mathit{id})) \circ \mathit{coassocr} \circ i_1 \circ \mathit{distr} \circ (\mathit{id} \times i_1) \\
= & \quad \{ \mathit{coassocr}\text{-Cancel}; \ \mathit{distr}\text{-lCancel} \} \\
& (\mathit{id} \ + \ (\pi_2 \nabla \mathit{id})) \circ (\mathit{id} \ + \ i_1) \circ i_1 \circ (\mathit{id} \times \mathit{id}) \\
= & \quad \{ \times\mathit{-functor}\text{-Id}; \ +\text{-Absorption}; \ \mathit{nat}\text{-Id}; \ +\text{-Cancel} \} \\
& (\mathit{id} \ + \ \pi_2) \circ i_1 \\
= & \quad \{ +\text{-Cancel} \} \\
& i_1
\end{array}
\rfloor
$$

B.2.3 Map

Assuming that $A + 1 \underset{\mathit{from}'}{\overset{\mathit{to}'}{\leqslant}} B$ and $C + 1 \underset{\mathit{from}''}{\overset{\mathit{to}''}{\leqslant}} D$

and from' is injective, then $(A \rightharpoonup B) + 1 \underset{\mathit{from}}{\overset{\mathit{to}}{\leqslant}} C \rightharpoonup D$

$$
\begin{array}{c}
C \rightharpoonup D \\
\downarrow \ \mathit{mapkeys\ from}' \\
(A + 1) \rightharpoonup D \\
\downarrow \ \mathit{map\ from}'' \\
(A + 1) \rightharpoonup (B + 1) \\
\downarrow \ \mathit{unpeither} \\
(A \rightharpoonup B + 1) \times (1 \rightharpoonup B + 1) \\
\downarrow \ \mathit{uncojoin} \times \mathit{id} \\
((A \rightharpoonup B) \times (A \rightharpoonup 1)) \times (1 \rightharpoonup B + 1) \\
\downarrow \ \mathit{assocr} \\
(A \rightharpoonup B) \times ((A \rightharpoonup 1) \times (1 \rightharpoonup B + 1)) \\
\downarrow \ \mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id}) \\
(A \rightharpoonup B) \times ((A \rightharpoonup B + 1) \times (1 \rightharpoonup B + 1)) \\
\downarrow \ \mathit{id} \times \mathit{peither} \\
(A \rightharpoonup B) \times (A + 1 \rightharpoonup B + 1) \\
\downarrow \ (\mathit{geq} \circ (\mathit{id} \vartriangle (\mathit{empty} \circ \underline{1})) \circ \pi_2)?i_1 \circ \pi_1 : i_2 \circ \underline{1} \\
(A \rightharpoonup B) + 1
\end{array}
$$

$$
\begin{array}{ll}
\lceil & \mathit{from} \circ \mathit{to} \\
= & \quad \{ \text{definition of } \mathit{to} \text{ and } \mathit{from} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id})) \circ \mathit{assocr} \circ \\
& (\mathit{uncojoin} \times \mathit{id}) \circ \mathit{unpeither} \circ \mathit{map}\ \mathit{from}'' \circ \mathit{mapkeys}\ \mathit{from}' \circ \mathit{map}\ \mathit{to}'' \circ \mathit{mapkeys}\ \mathit{to}' \\
= & \quad \{ \mathit{map}\text{-}\textsc{Reflex}; \mathit{from}' \circ \mathit{to}' = i_1; \mathit{from}', \mathit{to}'\ \mathit{injective} \Rightarrow \mathit{mapkeys}\text{-}\textsc{Absorption}; \mathit{map}\text{-}\textsc{Absorption} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id})) \circ \mathit{assocr} \circ \\
& (\mathit{uncojoin} \times \mathit{id}) \circ \mathit{unpeither} \circ \mathit{mapkeys}\ i_1 \circ \mathit{map}\ i_1 \\
= & \quad \{ \mathit{unpeither}\text{-}\textsc{Cancel} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times ((\mathit{geq} \circ (\mathit{id} \triangle \\
& (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id})) \circ \mathit{assocr} \circ (\mathit{uncojoin} \times \\
& \mathit{id}) \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \mathit{map}\ i_1 \\
= & \quad \{ \times\text{-}\textsc{Absorption} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id})) \circ \mathit{assocr} \circ \\
& (\mathit{uncojoin} \triangle (\mathit{empty} \circ \underline{1})) \circ \mathit{map}\ i_1 \\
= & \quad \{ \mathit{uncojoin}\text{-}\textsc{Cancel}; \times\text{-}\textsc{Fusion} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id})) \circ \mathit{assocr} \circ \\
& ((\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \triangle (\mathit{empty} \circ \underline{1})) \\
= & \quad \{ \mathit{assocr}\text{-}\textsc{Def} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \times \mathit{peither}) \circ (\mathit{id} \times (\mathit{map}\ i_2 \times \mathit{id})) \circ (\mathit{id} \triangle \\
& ((\mathit{empty} \circ \underline{1}) \triangle (\mathit{emty} \circ \underline{1}))) \\
= & \quad \{ \times\text{-}\textsc{Absorption}; \mathit{nat}\text{-}\textsc{Id} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \triangle (\mathit{peither} \circ ((\mathit{map}\ i_2 \circ \mathit{empty} \circ \underline{1}) \triangle \\
& (\mathit{empty} \circ \underline{1})))) \\
= & \quad \{ \mathit{empty}\text{-}\textsc{Absorption} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \triangle (\mathit{mapkeys}\ i_1 \circ \mathit{map}\ i_2 \circ \mathit{empty} \circ \underline{1})) \\
= & \quad \{ \mathit{map}\text{-}\textsc{Cancel} \} \\
& ((\mathit{geq} \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \circ \pi_2) \,?\, i_1 \circ \pi_1 : i_2 \circ \underline{1}) \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \\
= & \quad \{ \mathit{cond}\text{-}\textsc{Cancel} \} \\
& i_1 \circ \pi_1 \circ (\mathit{id} \triangle (\mathit{empty} \circ \underline{1})) \\
= & \quad \{ \times\text{-}\textsc{Cancel}; \mathit{nat}\text{-}\textsc{Id} \} \\
\lfloor & \quad i_1
\end{array}
$$

Appendix C

Invariants for Partial Two-level Transformations

Considering a partial two-level transformation from A to B, where $to :: A \to B$ and $from :: B \to A + 1$, and some invariant $inv_A :: A \to A + 1$, then an invariant $inv_B :: B \to B + 1$ can be calculated by:

$$
\begin{array}{c}
B \\
\Big\downarrow{\scriptstyle from \,\triangle\, id} \\
(A + 1) \times B \\
\Big\downarrow{\scriptstyle (inv_A + id) \times id} \\
((A + 1) + 1) \times B \\
\Big\downarrow{\scriptstyle coassocr \times id} \\
(A + (1 + 1)) \times B \\
\Big\downarrow{\scriptstyle distl} \\
(A \times B) + ((1 + 1) \times B) \\
\Big\downarrow{\scriptstyle \pi_2 + \underline{1}} \\
B + 1
\end{array}
$$