Translucent Abstraction: Safe Views through Invertible Programming

Meng Wang and Jeremy Gibbons
Computing Laboratory
Oxford University
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK
{menw,jg}@comlab.ox.ac.uk

Kazutaka Matsuda
JSPS Research Fellow
University of Tokyo
Dept of Mathematical Informatics
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
kztk@ipl.t.u-tokyo.ac.jp

Zhenjiang Hu
GRACE Center
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo 101-8430, Japan
hu@nii.ac.jp

Abstract

Despite the distinctive advantages of pattern matching in program understanding and reasoning, the tight coupling of interface and implementation has hampered its wider acceptance. Since the first proposal of *views* two decades ago, significant effort has been invested in tackling this non-modularity; the obvious target has been to decouple datatype implementations from separate interfaces used for pattern matching. However, having this decoupling to coexist with soundness of reasoning has been a challenge. Inspired by the development of invertible (bidirectional) programming, we propose a design of views based on a right-invertible language. The language is sufficiently expressive to program many of the existing and some novel view applications, with simple and sound reasoning properties: views can be manipulated as if they were datatypes and equivalent programs with respect to reasoning are guaranteed to exhibit identical operational behaviours.

1. Introduction

Algebraic datatypes and pattern matching are probably two of the best loved features of functional programming languages, thanks to their elegant and convenient syntax for both program development and reasoning. Consider a simple example of encoding binary numbers as lists of digits.

$$\textbf{data}\ \mathit{Bin} = \mathit{Zero} \mid \mathit{One}$$
$$\textbf{type}\ \mathit{Num} = [\mathit{Bin}]$$

With pattern matching, we can concisely analyse and bind input values in a function definition. For example, consider a function that normalises binary numbers into their sparse representations (i.e elides leading zeros). In this case, it is advantageous to position the most significant bit as the head of the list.

$$\begin{aligned}&\textbf{type}\ \mathit{MSB} = \mathit{Num}\\ &\mathit{sparse} :: \mathit{MSB} \to \mathit{MSB}\\ &\mathit{sparse}\ [\,] \quad\ = \mathit{error}\ \texttt{"invalid number"}\\ &\mathit{sparse}\ [\mathit{Zero}] \quad = [\mathit{Zero}]\\ &\mathit{sparse}\ (\mathit{One} : \mathit{num}) = \mathit{One} : \mathit{num}\\ &\mathit{sparse}\ (\mathit{Zero} : \mathit{num}) = \mathit{sparse}\ \mathit{num}\end{aligned}$$

An equivalent definition without pattern matching is significantly harder to read.

$$\begin{aligned}&\mathit{sparse} :: \mathit{MSB} \to \mathit{MSB}\\ &\mathit{sparse}\ \mathit{num} = \textbf{if}\ \mathit{nonempty}\ \mathit{num}\ \textbf{then}\\ &\qquad\qquad \textbf{if}\ \mathit{zero}\ (\mathit{head}\ \mathit{num})\\ &\qquad\qquad \textbf{then if}\ \mathit{nonempty}\ (\mathit{tail}\ \mathit{num})\ \textbf{then}\ \mathit{sparse}\ \mathit{num}\\ &\qquad\qquad\qquad\ \textbf{else}\ \mathit{num}\\ &\qquad\qquad \textbf{else}\ \mathit{num}\\ &\qquad \textbf{else}\ \mathit{error}\ \texttt{"invalid number"}\end{aligned}$$

When it comes to equational reasoning, the benefit of pattern matching becomes even more evident. For example, based on the first definition of *sparse*, we can easily conduct the following derivation.

$$\mathit{sparse}\ [\mathit{Zero}, \mathit{One}, \mathit{Zero}] = \mathit{sparse}\ [\mathit{One}, \mathit{Zero}] = [\mathit{One}, \mathit{Zero}]$$

Readers unconvinced of the benefits are encouraged to work out a similar derivation with the non-pattern-matching definition above.

Given the distinct advantages, pattern matching is supported as a standard feature in most modern functional languages. More recently, it has started gaining recognition from the object-oriented community [5, 16, 18] too, although not without opposition. The predominant criticism is the breaking of abstraction and encapsulation, fundamental pillars of modern software engineering. Function definitions are tightly coupled to a particular implementation, in this case binary numbers as lists: changes of the representation (for example, using non-empty lists to prevent invalid numbers) have a global effect.

In addition to the need to support evolution of data representation, it is often beneficial to allow multiple interfaces to the same data. For *sparse*, we chose to position the most significant bit as the head. For the function *incr* that increments a number, a list with the least significant bit as the head is better.

$$\begin{aligned}&\textbf{type}\ \mathit{LSB} = \mathit{Num}\\ &\mathit{incr} :: \mathit{LSB} \to \mathit{LSB}\\ &\mathit{incr}\ [\mathit{One}] \quad\ = [\mathit{Zero}, \mathit{One}]\\ &\mathit{incr}\ (\mathit{Zero} : \mathit{num}) = \mathit{One} : \mathit{num}\\ &\mathit{incr}\ (\mathit{One} : \mathit{num}) = \mathit{One} : (\mathit{incr}\ \mathit{num})\end{aligned}$$

These problems of pattern matching have been recognised for two decades, starting from Wadler's proposal of *views* [24], addressing the conflict between pattern matching and data abstraction; and is still a hot research topic [4, 6, 7, 13, 14, 20–23]. The primary goal of all these works is to provide a pattern matching facility for

abstract types, by which we mean algebraic datatypes with hidden representations. (Do not confuse our concept of abstract types with the well-known notion of *abstract datatypes* [15], which need not have algebraic datatypes as implementations, and are necessarily completely characterised by a fixed set of operations and their derived properties.)

All these existing proposals successfully disconnect pattern matching from datatype implementation, achieving a kind of abstraction. However, they throw the baby out with the bath water: none of these proposals supports local understanding and reasoning. That is to say, one can no longer understand the semantics of *sparse* nor derive any properties of it as we have done, just by looking at its definition. This is clearly unsatisfactory; convenient program understanding and equational reasoning, the main thrust of pattern matching, is lost.

In this paper, we aim to tackle the longstanding problem of equational reasoning with abstract types.

- We propose a variant of the view mechanism that supports sound and easy equational reasoning with abstract types. The same view constructors can be used both in patterns and in expressions, and can be manipulated as if they were datatype constructors. Two expressions proven equal by equational reasoning are guaranteed to exhibit an identical operational behaviour in any programming context.
- We design a small yet powerful combinator-based language that automatically generates a total right-inverse of the function being defined. As a result, designers of views are only required to write and maintain one of the two conversion functions. This language is independently useful in the broader area of bidirectional programming [3, 9, 11, 17, 19].

In the rest of the paper, we firstly review the related literature on views (Section 2) before presenting our proposal (Section 3) to tackle the limitations of existing approaches. We then introduce the right-inverse language (Section 4) on which our design is based, followed by an evaluation of our approach (Section 5). We discuss other related work in the field of invertible (bidirectional) languages (Section 6), before concluding in Section 7.

2. Previous Work

Wadler's views [24] provide different ways of viewing data than their actual implementations. With a pair of conversion functions, data can be converted *to* and *from* a view.

Consider the forward and backward representations of lists:

```
data List a = Nil | Cons a (List a)
view List a = Lin | Snoc (List a) a
  to Nil                 = Lin
  to (Cons x Nil)        = Snoc Nil x
  to (Cons x (Snoc xs y)) = Snoc (Cons x xs) y
  from Lin               = Nil
  from (Snoc Nil x)      = Cons x Nil
  from (Snoc (Cons x xs) y) = Cons x (Snoc xs y)
```

The **view** clause introduces two new constructors, namely *Lin* and *Snoc*, which may appear in both terms and patterns. The first argument to the view construction *Snoc* refers to the datatype *List a*, so a snoclist actually has a conslist as its child. The to and from clauses (highlighted with a special font as to and from throughout the paper) are similar to function definitions. The to clause converts a conslist value to a snoclist value, and is used when *Lin* or *Snoc* appear as the outermost constructor in a pattern on the left-hand side of an equation. Conversely, the from clause converts a snoclist into a conslist, when *Lin* or *Snoc* appear in an expression. Note that we are already making use of views in the definition above; for example, *Snoc* appears on the left-hand side of the third to clause, matching against which will trigger a recursive invocation of to.

Functions can now pattern match on and construct values in a view of the original datatype.

```
last (Snoc xs x) = x
rotLeft (Cons x xs)  = Snoc xs x
rotRight (Snoc xs x) = Cons x xs
rev Nil              = Lin
rev (Cons x xs) = Snoc (rev xs) x
```

Upon invocation, an argument is converted into the view by the to function; after completion of the computation, the result is converted back to the underlying datatype representation.

This semantics can be elaborated by a straightforward translation into ordinary Haskell. First of all, view declarations are translated into data declarations.

```
data Snoc a = Lin | Snoc (List a) a
```

Note that the child of *Snoc* refers to the underlying datatype: view data is typically heterogeneous. Now the only task is to insert the conversion functions into the appropriate places in the program.

```
last xs = case to xs of Snoc xs x → x
rotRight xs = case to xs of Snoc xs x → Cons x xs
rotLeft xs  = case xs of Cons x xs → from (Snoc xs x)
rev xs = case xs of
  Nil          → from Lin
  (Cons x xs)  → from (Snoc (rev xs) x)
```

The design of views aims at a kind of abstraction whereby programmers program to the views, which are decoupled from the actual representations; yet they may expect to be able to reason about their programs correctly, supported by the underlying representations. For example, we can evaluate an expression:

```
    last (Cons 1 (Cons 2 Nil))
≡   { Cons x Nil = Snoc Nil x }
    last (Cons 1 (Snoc Nil 2))
≡   { Cons x (Snoc xs y) = Snoc (Cons x xs) y }
    last (Snoc (Cons 1 Nil) 2)
≡   { definition of last }
    2
```

or calculate with functions:

```
    rotRight (rotLeft (Cons x xs))
≡   { definition of rotLeft }
    rotRight (Snoc xs x)
≡   { definition of rotRight }
    Cons x xs
```

This is a pretty picture, but it is flawed. Wadler expects a view type to be isomorphic to its underlying datatype, and the pair of user-defined total conversions between the values of the two types to be each other's inverses. Unfortunately, neither of these conditions comes easily in practice. Indeed, the majority of views in reality are not isomorphic to their datatypes: views are often abstractions for simplified presentation, or enrichments with derived information. As a matter of fact, out of Wadler's seven examples [24], only two of them faithfully follow the isomorphism requirement between views and datatypes. The conversion functions are defined by pattern matching and are recursive in general. There is no check of their totality. Perhaps more difficult still, the condition that conversions are each other's inverses is very subtle to enforce, even when the view and datatype are isomorphic structures. A diversion from it may result unexpected behaviour of programs making use of views. Consider the join representation of lists.

$$
\begin{array}{l}
\textbf{view}\ \mathit{List}\ a = \mathit{Empty} \mid \mathit{Unit}\ a \mid \mathit{Join}\ (\mathit{List}\ a)\ (\mathit{List}\ a) \\
\quad \mathsf{to}\ \mathit{Nil} = \mathit{Empty} \\
\quad \mathsf{to}\ (\mathit{Cons}\ x\ xs) = \mathit{Join}\ (\mathit{Unit}\ x)\ xs \\
\quad \mathsf{from}\ \mathit{Empty} = \mathit{Nil} \\
\quad \mathsf{from}\ (\mathit{Unit}\ x) = \mathit{Cons}\ x\ \mathit{Nil} \\
\quad \mathsf{from}\ (\mathit{Join}\ \mathit{Nil}\ xs) = \mathsf{from}\ xs \\
\quad \mathsf{from}\ (\mathit{Join}\ (\mathit{Unit}\ x)\ ys) = \mathit{Cons}\ x\ ys \\
\quad \mathsf{from}\ (\mathit{Join}\ (\mathit{Join}\ xs\ ys)\ zs) = \mathsf{from}\ (\mathit{Join}\ xs\ (\mathit{Join}\ ys\ zs))
\end{array}
$$

Surprisingly, evaluating the expression

$$\textbf{let}\ \mathit{Join\ Empty}\ (\mathit{Unit}\ x) = \mathit{Join\ Empty}\ (\mathit{Unit}\ 1)\ \textbf{in}\ x$$

results in a pattern matching error! This seems bizarre on the view level, and can only be explained by investigating the translated code:

$$
\begin{array}{l}
\textbf{let}\ \mathit{Join\ Empty}\ (\mathit{Unit}\ x) = \\
\qquad\qquad \mathsf{to}\ (\mathsf{from}\ (\mathit{Join}\ (\mathsf{from}\ \mathit{Empty})\ (\mathsf{from}\ (\mathit{Unit}\ 1)))) \\
\textbf{in}\ x
\end{array}
$$

The joinlist expression $\mathit{Join\ Empty}\ (\mathit{Unit}\ 1)$ is implicitly converted to the conslist implementation $\mathit{Cons}\ 1\ \mathit{Nil}$; when pattern-matched, it is converted back to a joinlist representation $\mathit{Join}\ (\mathit{Unit}\ 1)\ \mathit{Empty}$, which does not match the pattern.

'Safe' variants of views have been proposed many times [4, 20]. To circumvent the problem of equational reasoning, one typically restricts the use of view constructors to patterns, and does not allow them to appear on the right-hand side of a definition. As a result, expressions like $\mathit{Join\ Empty}\ (\mathit{Unit}\ 1)$ become syntactically invalid. Instead, values are only constructed by 'smart constructors', as in $\mathit{join\ empty}\ (\mathit{unit}\ 1)$. Given the same conversion strategy, this version of the program may produce identical result as the previous one; however, we cannot take for granted equalities such as $\mathit{join\ empty}\ (\mathit{unit}\ 1) \equiv \mathit{Join\ Empty}\ (\mathit{Unit}\ 1)$, which are at the heart of equational reasoning. Instead of the obvious

$$\mathit{Join\ Empty}\ (\mathit{Unit}\ 1) \equiv \mathit{Join\ Empty}\ (\mathit{Unit}\ 1)$$

one has to work on the level of the translated code, and carry out explicitly the non-trivial proof of

$$\mathsf{to}\ (\mathit{join\ empty}\ (\mathit{unit}\ 1)) \equiv \mathit{Join\ Empty}\ (\mathit{Unit}\ 1)$$

In another words, perhaps ironically, safe equational reasoning of views is achieved by prohibiting reasoning with views completely, which defeats the purpose of having pattern matching in the first place. Throughout this paper, we refer to the latter kind of explicit low-level reasoning as on the *datatype level*, in contrast to reasoning on the *view level* which abstracts away the conversion details.

More recently, language designers have started looking into more expressive pattern mechanisms. *Active patterns* [6, 21] and many of their variants [7, 13, 14, 22, 23] go a step further, by proposing languages that embed computational content into pattern constructions. All the above proposals either explicitly recognize the benefit of using constructors in expressions, or use examples that involve construction of view values on the right-hand sides of function definitions. Nevertheless, none of them are able to support pattern constructors in expressions, due to the inability to reason safely. Knowing that there is an absence of good solutions for supporting constructors in expressions, some works focus only on examples that are primarily data consumers. This escape is limited and short lived. For example, in one motivating example in [6], a non-linear pattern can be used to conveniently define the *member* and *delete* functions for an implementation of sets based on the list datatype. However, the same problem of equational reasoning quickly arises on the third function, *insert*, that was defined.

3. Our Approach

Our proposed design of views follows the same spirit as Wadler's, with two important differences: (i) views can be abstractions or enrichments of their underlying datatypes, and (ii) simple and sound equational reasoning on the view level is supported, without the need for any additional proof obligations.

Following the above two principles, our approach differs from Wadler's in a number of design details, which will be illustrated by example.

3.1 First Example: FIFO Queue

Suppose we have an implementation of queue structures.

$$\textbf{type}\ \mathit{Queue}\ a = ([a], [a])$$

The second list of the pair, representing the latter part of a queue, is reversed, so that enqueuing simply prefixes an element onto it.

$$
\begin{array}{l}
\mathit{emptyQ} = ([\,], [\,]) \\
\mathit{enQ}\ a\ (fq, bq) = (fq, a : bq) \\
\mathit{deQ}\ ([\,], bq) \quad = (\mathit{deQ}\ (\mathit{reverse}\ bq, [\,])) \\
\mathit{deQ}\ ((a : q), bq) = (q, bq)
\end{array}
$$

Other than the standard queue operations, there are others that can be conveniently defined based on this representation. For example, reversing a queue is simply a matter of swapping the two lists.

$$\mathit{revQ}\ (fq, bq) = (bq, fq)$$

At the same time, this queue implementation is not ideal for operations such as printing or indexing, and prohibits us from inheriting existing library functions on linear structures. Changing the implementation is not an option, since it incurs the cost of losing all existing code involving the queue structure. Instead, we make list a view of the queue datatype, which doesn't have to be isomorphic to the underlying representation.

$$
\begin{array}{l}
\textbf{view}\ \mathit{Queue}\ a\ @\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\quad \mathsf{to} :: \mathit{Queue}\ a \to \mathit{Queue}\ a\ @\ \mathit{List}\ a \\
\quad \mathsf{to} = \mathit{app} \circ (\mathit{id} \times \mathit{reverse})
\end{array}
$$

Throughout this paper, we use the constructors $(:)$ and $[\,]$ to represent list datatype and *Cons* and *Nil* for the list view. We also use the pseudo-type notation $\mathit{Queue}\ a\ @\ \mathit{List}\ a$ to distinguish a queue value in the *List* view from a queue value in the underlying datatype, denoted by $\mathit{Queue}\ a$. It is worth mentioning that there is not (yet) any static system that checks the annotations; they are added for the sake of presentation clarity. The user-defined function to converts the datatype to the view, as discussed in detail in Section 4. Readers may safely skip them for the time being. As an example, we can now define a prioritisation function on the list view.

$$
\begin{array}{l}
\mathit{prioritise} :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Queue}\ a\ @\ \mathit{List}\ a \to \mathit{Queue}\ a\ @\ \mathit{List}\ a \\
\mathit{prioritise}\ \mathit{Nil} \qquad = \mathit{Nil} \\
\mathit{prioritise}\ (\mathit{Cons}\ x\ xs) = \mathit{insert}\ x\ (\mathit{prioritise}\ xs) \\
\mathit{insert} :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Queue}\ a\ @\ \mathit{List}\ a \to \mathit{Queue}\ a\ @\ \mathit{List}\ a \\
\mathit{insert}\ y\ \mathit{Nil} \qquad = \mathit{Cons}\ y\ \mathit{Nil} \\
\mathit{insert}\ y\ (\mathit{Cons}\ x\ xs) = \textbf{if}\ y < x\ \textbf{then}\ \mathit{Cons}\ y\ (\mathit{Cons}\ x\ xs) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else} \qquad \mathit{Cons}\ x\ (\mathit{insert}\ y\ xs)
\end{array}
$$

Function *prioritise* is essentially a stable sort based on the elements' weights.

In contrast to Wadler's, our views are named, and the recursive elements refer to the view structure instead of the datatype. Thus, the above view definition is translated into

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

instead of

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ ([a], [a])$$

This choice makes the definition of conversions closer to conventional recursively defined functions and more readily lending themselves to the existing studies of program properties. The advantage of our approach is the modular separation of interface (view) from implementation (datatype). In the event where the implementation is changed, no functions defined on the view will be recompiled. Following the same spirit, in our design the underlying datatype is hidden from the view user; if access is really needed, we can supply an identity view to the datatype.

```
view Queue a @ Pair a = Pr ([a],[a])
  to = pr
```

Since this view directly reuses an existing type $([a],[a])$, we mark it with a constructor, just like **newtype** in Haskell. The to function simply does the wrapping through a lower case construction function, discussed in detail in Section 4.

In this setting, changes in datatypes only require the recompilation of the to/from function pair and relinking them to the rest of the program. To avoid namespace pollution, we treat to and from as overloaded. Their behaviour should be clear from the context. When necessary, we use subscripts to make the choice explicit. Without special mention, when used together, we assume to and from are the conversion functions for the same view.

The most fundamental difference between our view to Wadler's is that we don't require user-defined from functions. Instead, they are automatically generated from the definition of to functions, which are the right inverses of the corresponding to functions—that is, $\mathsf{to} \circ \mathsf{from} \equiv id$ for any finitely defined input. The technique used to generate from functions is discussed in Section 4.

Now with the list view, we merrily inherit library functions on lists. For example, we can cut a queue off after a certain number of elements with *take*, update elements in a queue with *map*, remove selectively with *filter*, and so on.

Translating the view declarations and functions defined on views into Haskell follows from a rather straightforward scheme similar to the one in [24]: all patterns are converted into **case** statements that match with views produced by function to; and view constructions on the right-hand side are converted back to the underlying datatypes by from. Effectively, all the translated functions and constructors take datatypes as inputs and produce datatypes as outputs. Views are only constructed as intermediate structures.

First of all, **view** declarations are translated into **data** (or **newtype** if appropriate) declarations.

```
data List a = Nil | Cons a (List a)
newtype Pair a = Pr ([a],[a])
```

The data constructors above construct view values and need to be converted into the underlying datatypes. As a result, we introduce a smart constructor for each of the constructors.

```
cNil :: Queue a
cNil = from Nil
cCons :: a → Queue a → Queue a
cCons a l = from (Cons a (to l))
cPr :: ([a],[a]) → Queue a
cPr p = from (Pr p)
```

Note that since the list value consumed by *Cons* is expected to be in its underlying datatype, we need to convert it into the list view. In *cPr*, a *Pair* is firstly constructed from the input pair before being converted to the underlying datatype. We can now translate functions making use of views.

```
emptyQ :: Queue a
emptyQ = cPr ([],[])
enQ :: a → Queue a → Queue a
enQ a q = case to q of Pr (fq,bq) → cPr (fq,a:bq)
deQ :: Queue a → Queue a
deQ q =
  case to q of Pr ([],bq)    → (deQ (cPr (reverse bq,[])))
               Pr ((a:q),bq) → cPr (q,bq)
revQ :: Queue a → Queue a
revQ q = case to q of Pr (fq,bq) → cPr (bq,fq)
prioritise :: Ord a ⇒ a → Queue a → Queue a
prioritise q =
  case to q of Nil → cNil
               (Cons x xs) → let xs = from xs
                             in insert x (prioritise xs)
insert :: Ord a ⇒ a → Queue a → Queue a
insert y q =
  case to q of
    Nil          → cCons y cNil
    (Cons x xs) → let xs = from xs
                  in if y < x then cCons y (cCons x xs)
                     else          cCons x (insert y xs)
```

Patterns on views are translated to **case** expressions, so that function to can be applied prior to pattern matching on the translated datatypes. For recursive view datatypes, variable bindings to their children are then converted back to the underlying datatypes for use in function body. The view constructors in expressions are converted to construction functions, produceing values in underlying datatypes. Effectively, data flow in the form of underlying datatypes and are only converted to view datatypes for pattern matching.

This straightforward translation aggressively inserts conversion functions, which is suboptimal despite that inlining and fusion may eliminate some of them. An optimised translation may choose to not to insert the redundant conversions in the first place, which we will look into towards the end of this section.

List library functions can be written as if *List a* were a datatype, and specialised to deal with queue values. Since we know that the library only makes use of a single view *List a*, but not *Pair a*, adapting them can be simplified to precomposing with to and postcomposing with from:

```
map :: (a → b) → Queue a → Queue b
map = λf → from ∘ (List.map f) ∘ to
```

where $List.map :: (a \to b) \to List\ a \to List\ b$ is the original non-specialised library function. To avoid ambiguity, in a single scope, a view can be associated with only one underlying datatype.

Equational reasoning can be performed on the view level as if views were datatypes. For example, we are able to conclude the following.

$$revQ \circ revQ \equiv \lambda(Pr\ (fq,bq)) \to Pr\ (fq,bq)$$
$$(deQ \circ (enQ\ a))\ emptyQ \equiv emptyQ$$
$$map\ f \circ map\ g \equiv map\ (f \circ g)$$
$$take\ 1\ (Cons\ 1\ (Cons\ 2\ Nil)) \equiv Cons\ 1\ Nil$$

The derivations of the above equalities are able to take full advantage of pattern matching, which simplifies the calculation. We write down each derivation step for the first equation above as follows; the interested reader is referred to Appendix A for the other three.

$$\begin{array}{ll} & (revQ \circ revQ)\ (Pr\ (fq,bq)) \\ \equiv & \quad \{\text{composition}\} \\ & revQ\ (revQ\ (Pr\ (fq,bq))) \\ \equiv & \quad \{\text{definition of } revQ\} \\ & revQ\ (Pr\ (bq,fq)) \\ \equiv & \quad \{\text{definition of } revQ\} \\ & (fq,bq) \end{array}$$

We can prove the correctness of view level reasoning in our system by reasoning on the datatype level, which is considerably more complicated.

$$
\begin{array}{ll}
 & (revQ \circ revQ)\ (cPr\ (q,q')) \\
\equiv & \quad \{\text{composition}\} \\
 & revQ\ (revQ\ (cPr\ (q,q'))) \\
\equiv & \quad \{\text{definition of } revQ\} \\
 & revQ\ (\textbf{case}\ \mathsf{to}\ (cPr\ (q,q'))\ \textbf{of}\ Pr\ (fq,bq) \quad \rightarrow cPr\ (bq,fq)) \\
\equiv & \quad \{\text{definition of } cPr\} \\
 & revQ\ (\textbf{case}\ \mathsf{to}\ (\mathsf{from}\ (Pr\ (q,q'))))\ \textbf{of}\ Pr\ (fq,bq) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow cPr\ (bq,fq)) \\
\equiv & \quad \{\text{composition}\} \\
 & revQ\ (\textbf{case}\ (\mathsf{to} \circ \mathsf{from})\ (Pr\ (q,q'))\ \textbf{of}\ Pr\ (fq,bq) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow cPr\ (bq,fq)) \\
\equiv & \quad \{\mathsf{to} \circ \mathsf{from} \equiv id\} \\
 & revQ\ (\textbf{case}\ Pr\ (q,q')\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (bq,fq)) \\
\equiv & \quad \{\textbf{case}\} \\
 & revQ\ (cPr\ (q',q)) \\
\equiv & \quad \{\text{definition of } revQ\} \\
 & \textbf{case}\ \mathsf{to}\ (cPr\ (q',q))\ \textbf{of}\ Pr\ (fq,bq) \ \rightarrow cPr\ (bq,fq) \\
\equiv & \quad \{\text{definition of } cPr\} \\
 & \textbf{case}\ \mathsf{to}\ (\mathsf{from}\ (Pr\ (q',q)))\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (bq,fq) \\
\equiv & \quad \{\text{composition}\} \\
 & \textbf{case}\ (\mathsf{to} \circ \mathsf{from})\ (Pr\ (q',q))\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (bq,fq) \\
\equiv & \quad \{\mathsf{to} \circ \mathsf{from} \equiv id\} \\
 & \textbf{case}\ Pr\ (q',q)\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (bq,fq) \\
\equiv & \quad \{\textbf{case}\} \\
 & cPr\ (q,q')
\end{array}
$$

The soundness of view level reasoning holds in general.

THEOREM 1 (Soundness of Reasoning). *Given two view level expressions ev and ev′ and their translations on datatype level ed and ed′. If ev ≡ ev′ is derivable through equational reasoning, then ed ≡ ed′ holds.*

In an evaluation of the translated code, a to-call is always preceded by a from-call because data only flow in the underlying datatypes: any view value is firstly converted into the underlying datatype before being converted to a view value for pattern matching. Consider an expression on view level of the form $f\ v$ where f pattern match on a certain view and v is a value. On datatype level, the evaluation of $f'\ v'$, where f' and v' are the translations of f and v respectively, always involves applying function from to the view value v and then applying function to prior to the application of f. If the equational reasoning of $f\ v$ on view level ever proceeds, it necessarily implies that v is in the same view as f's pattern, which give rise to $\mathsf{to} \circ \mathsf{from} \equiv id$. A very similar argument applies to view constructors. As a result, for each successful step of view level equational reasoning, the execution of to/from pair on the datatype level is equal to the identity and can be eliminated, which validate the correctness of the reasoning on view level.

In all other works on abstract datatypes where there is no automatic guarantee of the right-inverse property $\mathsf{to} \circ \mathsf{from} \equiv id$, the required derivation blows up even more than the explicit proof above, both in size and intricacy. This ability to support elegant reasoning is one of the main thrusts of our proposal. It is clearly important in promoting program comprehension.

We shall admit up front that, although sound, our system is not complete for reasoning on the view level: such reasoning does not transcend from one view to another in general. For example, we cannot derive $([1,2],[]) \equiv Cons\ 1\ (Cons\ 2\ Nil)$ or $filter\ p \circ revQ \equiv revQ \circ filter\ p$. This complies with the expected reasoning behaviour; as on the view level, these two equations obviously do not hold. Very similar to the case of abstract datatypes, when used together, *filter p* now takes the result of *revQ* as a black box and the behaviour of it can only be understood by considering explicit to and from functions, as proposed in [4]. We will discuss reasoning power in more detail in Section 5.

As mentioned briefly before, this naive translation tends to insert an excess of conversions: in particular an argument to a recursive call is converted into the underlying datatype and then immediately converted back to the same view by the recursive call. This is wasted effort, given that $\mathsf{to} \circ \mathsf{from} \equiv id$. A simple but very useful optimisation is to lift the conversion out of the recursion, so that it is done only once for the function. For example, similar to our treatment of library functions for lists, an optimised *enQ* translation is to simply precompose to and postcompose from with the original definition.

$$
\begin{array}{l}
deQ = \mathsf{from} \circ deQ' \circ \mathsf{to} \\
deQ'\ ([\,],bq) \quad = (deQ'\ (reverse\ bq,[\,])) \\
deQ'\ ((a:q),bq) = (q,bq)
\end{array}
$$

Similarly for value constructions

$$x = \mathsf{from}\ (Cons\ 1\ (Cons\ 2\ Nil))$$

We can easily prove the equivalence of the translations by unfolding the recursive call and removing the intermediate conversions using $\mathsf{to} \circ \mathsf{from} \equiv id$.

This optimisation is applicable when the consumption of the result of a recursive call is in the same view as the result. Most practical functions fall into this category, including those involving more than one view in their definitions. For example, in the *rev* function we saw in Section 1, the consumption of the recursive call by constructor *Snoc* is in the same view produced by *rev*.

3.2 Second Example: Binary Numbers

Another advantage of our choice of naming views is the possibility of having 'semantic' views that are not distinguishable by structure. Recall the binary example given in the introduction, where both views (most significant bit first and least significant bit first) share the same structure and differ only in their semantics. In our system, this can be implemented as follows:

$$
\begin{array}{l}
\textbf{view}\ Num\ @\ MSB = M\ [Bin] \\
\quad \mathsf{to} = m \\
\textbf{view}\ Num\ @\ LSB = L\ [Bin] \\
\quad \mathsf{to} = l \circ reverse
\end{array}
$$

We omit the obvious translations for space reasons.

3.3 Third Example: Sized Trees

Views are not restricted to be an abstraction of the underlying datatype; additional information (usually derived from the datatype) can be added to a view to make subsequent computation easier. Consider an internally labelled binary tree.

$$\textbf{data}\ Tree\ a = Leaf\ a \mid Fork\ (Tree\ a)\ (Tree\ a)$$

We can enrich the structure by adding size information.

$$
\begin{array}{l}
\textbf{view}\ Tree\ a\ @\ STree\ a = SLeaf\ a \\
\qquad\qquad\qquad\qquad\qquad \mid\ SFork\ Int\ (STree\ a)\ (STree\ a) \\
\quad \mathsf{to} :: Tree\ a \rightarrow Tree\ a\ @\ STree\ a \\
\quad \mathsf{to} = fold\ (sleaf \triangledown (sfork \circ dupF\ getsize))
\end{array}
$$

Now, retrieving the size of a tree becomes trivial.

$$
\begin{array}{l}
size :: Tree\ a\ @\ STree\ a \rightarrow Int \\
size\ (SLeaf\ \_) \quad = 1 \\
size\ (SFork\ s\ \_\ \_) = s
\end{array}
$$

More interestingly, we can prune a search using the size information.

```
index :: Int → Tree a @ STree a → a
index i (SLeaf a) | i == 0      = a
                  | otherwise   = error "out of bound"
index i (SFork s t1 t2) | i ⩾ s       = error "out of bound"
                        | i ⩾ s1      = index (i − s1) t2
                        | otherwise   = index i t1
  where s1 = size t1
```

The effort of annotating trees with sizes pays off when we need to perform repeated indexing of the same tree.

```
trail :: Int → Tree Int @ STree Int → Int
trail i t | i == 0      = 0
          | otherwise   = trail (index i t) t
```

Function *trail* jumps to the next position in a tree based on the current leaf value and terminates successfully when the value is 0. The translation of the above code follows the same technique discussed before.

```
size = size' ∘ to
size' (SLeaf _)      = 1
size' (SFork s _ _)  = s
index i = index' i ∘ to
index' i (SLeaf a) | i == 0      = a
                   | otherwise   = error "out of bound"
index' i (SFork s t1 t2) | i ⩾ s       = error "out of bound"
                         | i ⩾ s1      = index' (i − s1) t2
                         | otherwise   = index' i t1
  where s1 = size' t1
trail i = trail' i ∘ to
trail' i t | i == 0          = 0
           | otherwise       = trail' (index' i t) t
```

From this translation, we can see clearly that, by the fusion law $\mathsf{from} \circ f \circ \mathsf{to} \circ \mathsf{from} \circ g \circ \mathsf{to} \equiv \mathsf{from} \circ f \circ g \circ \mathsf{to}$, the conversions can be pushed outside a chain of function compositions, as long as just a single view is being used. As a result, instead of trying to fuse step by step translated code with all conversions inserted, a compiler can perform a straightforward view usage check and immediately remove many intermediate conversions.

It is interesting to note that all the functions we have in the *STree* view are consumers, which do not use any of the view constructors in expressions. This is no coincidence: having derived information provides convenience in consuming the data; but less so if it has to be maintained explicitly. Indeed, direct construction of sized trees are cumbersome and error-prone. In our design, a much better way is to construct a binary tree in a different view and having the viewing mechanism to derive the size automatically.

Since there is no view value constructions in expressions, a from function is actually not needed here. Indeed, the semantic connection between a tree and its size makes it tricky to derive a total right-inverse. As a result, *dupF*, used in the definition of to, is not included in the strictly total language RINV that will be introduced shortly. Instead, we postpone a discussion of it to Section 5.3 where non-surjective to functions are looked into.

4. The Right-Inverse Language

In this section, we show the right-inverse language in which the to functions are defined. Note that in this section, we are working on the datatype level: the to functions simply map one datatype to another, and views play no part.

The syntax of the language is as below. (Non-terminals are indicated in small capitals.)

```
Language     RINV  ::= CSTR | PRIM | COMB
Constructors CSTR  ::= nil | cons | snoc
Primitives   PRIM  ::= app | id | assocr | assocl | swap
Combinators  COMB  ::= RINV ∘ RINV | fold RINV |
                       RINV ▽ RINV | RINV × RINV
```

The language is similar in flavour to the *pointfree* style of programming [2]. There is an extensible set of constructor functions that grows with the introduction of new datatypes. We use lowercase constructor functions as the uncurried versions of constructors. In addition to the left-biased list constructor *cons*, we find its right-biased counterpart *snoc* particularly useful; it can be defined in Haskell as

$$snoc = \lambda(x, xs) \to append\ (xs\ (Cons\ x\ Nil))$$

The set of primitive functions is also extensible. It defines the basic non-terminal building blocks of the language. Any surjective functions can be defined as primitives in RINV; we present a small but representative collection. As we will show in the sequel, with just the handful of primitives shown above we can define many interesting examples of views. In RINV, all primitive functions are uncurried; this fits better with the invertible framework, where a clear distinction between input and output is required. Functions *swap*, *assocl*, and *assocr* distribute the components of an input pair. Function *id* is the identity operation, while function *app* is the uncurried append function on lists.

Combinator *fold f* is the unique homomorphism from the (implicit) initial algebra of a datatype to algebra f; we do not explicitly mention the datatype itself, as it is understood from context. Combinator $\circ$ is standard function composition. Combinator $\triangledown$ joins two functions, dispatching according to the result of matching on a sum; combinator $\times$ is the cartesian product of two functions. They are defined as follows:

$$[\![f \triangledown g]\!] = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{Inl\ x \to f\ x; Inr\ x \to g\ x\}$$
$$[\![f \times g]\!] = \lambda(x, y) \to (f\ x, g\ y)$$

In combination with *swap*, *assocl* and *assocr*, $\times$ is able to define all functions that rearrange the components of a pair, while $\triangledown$ is useful in constructing the algebra for a *fold*. We don't include $\triangle$, the dual of $\triangledown$, in RINV because of surjectivity, as will be explained shortly. The combinators in RINV preserve totality: given total primitive functions, all functions in the language are total.

The language RINV is right-invertible: given pre-defined right-inverses for the primitives, every function has a right-inverse by construction. As a result, a function $f :: s \leftrightarrows t$ in RINV actually represents a pair of functions: the forwards function $[\![f]\!] :: s \to t$ and its right-inverse $[\![f]\!]^\circ :: t \to s$. However, for convenience when clear from context, we don't distinguish f and its forwards function $[\![f]\!]$. Note that RINV is not closed under inversion: $[\![f]\!]^\circ$ may not itself be in RINV with $[\![[\![f]\!]^\circ]\!]^\circ$ defined.

The generated right-inverses are expected to be total, with the exception of those of constructors, because the latter are not case-exhaustive. For this reason, there are a couple of straightforward restrictions on composing constructors with the combinators $\circ$ and $\triangledown$ in RINV, as we will see later when the combinators are discussed in more detail. Another note of caution concerns the termination of the right-inverses of recursive functions constructed by *fold* (which are constructed by *unfold*). We ensure well-founded recursion in the *unfold*; practically, this reduces to a simple position-wise check of descending values in recursive calls. The details can be found in Section 4.3.3.

With the language RINV, we can state the following property.

THEOREM 2. *Given a non-constructor function f in* RINV*, for any finitely defined well-typed input x,* $([\![f]\!] \circ [\![f]\!]^\circ)\ x \equiv x$.

The correctness of this theorem should become evident by the end of this section, as we discuss in detail the various components of RINV and their properties. Throughout this paper, unless otherwise mentioned, we always assume finitely defined values.

4.1 The Constructors

The semantics of the constructor functions are simple: they follow directly from the corresponding constructors introduced by datatype declarations, except for being uncurried. For example,

$$\begin{aligned}&[\![nil]\!] &&= \lambda() \to Nil\\ &[\![cons]\!] &&= \lambda(x,xs) \to Cons\ x\ xs\end{aligned}$$

Constructor *Snoc* is not primitive in Haskell, but can be encoded:

$$[\![snoc]\!] = \lambda(xs,x) \to append\ xs\ (Cons\ x\ Nil)$$

Inverses of the primitive constructor functions are obtained simply by swapping the right- and left-hand sides of the definitions. For example, we have

$$\begin{aligned}&[\![nil]\!]^\circ &&= \lambda Nil \to ()\\ &[\![cons]\!]^\circ &&= \lambda(Cons\ x\ xs) \to (x,xs)\end{aligned}$$

They are effectively partial 'guard' functions, succeeding when the input value matches the pattern. The right-inverse of *snoc* is

$$[\![snoc]\!]^\circ = \lambda xs \to (init\ xs, last\ xs)$$

The inverses of constructor functions are generally not case-exhaustive. For example, $[\![cons]\!]^\circ$ only accepts non-empty lists, while $[\![nil]\!]^\circ$ only accepts the empty list. This is the only case in RINV in which the right-inverses are not total. As a result, in contrast to primitive functions, constructor functions cannot be composed arbitrarily, as we will see shortly.

4.2 The Primitive Functions

The primitive functions and their right-inverses are hard-wired in RINV, but are subject to extension. It is worth mentioning that surjectivity of the primitives is a necessary condition of the totality of their inverses. The function *id* is the identity function; functions *assocr*, *assocl* and *swap* manipulate pair data.

$$\begin{aligned}&assocr :: ((a,b),c) \leftrightarrows (a,(b,c))\\ &assocl :: (a,(b,c)) \leftrightarrows ((a,b),c)\\ &swap :: (a,b) \leftrightarrows (b,a)\end{aligned}$$

As mentioned above, together with the combinator $\times$, these are sufficient to define all functions on pairs. For example,

$$\begin{aligned}&subr :: (b,(a,c)) \leftrightarrows (a,(b,c))\\ &subr = assocr \circ (swap \times id) \circ assocl\\ &trans :: ((a,b_1),(b_2,c)) \leftrightarrows ((a,b_2),(b_1,c))\\ &trans = assocl \circ (id \times subr) \circ assocr\end{aligned}$$

Function *app* is the uncurried append function, which is not injective. The ability to admit non-injective functions is one of the most important distinctions between RINV and other invertible languages [11, 19].

There are many possible right-inverses for *app*. We pick one:

$$[\![app]\!]^\circ = \lambda xs \to splitAt\ ((length\ xs + 1)\ `div`\ 2)\ xs$$

We require the right-inverses of all primitive functions to be *affine*: they should not duplicate input data values (values of non-container types) in the output. That is to say, the right-inverses may rearrange input data values, or create structures to hold them, but not invent or duplicate them. (In fact, they will not discard data either.) This condition is not new in the invertible (bidirectional) programming literature [17]; it makes sense, since such right-inverses only create data in the datatypes (sources), which is neither visible in the views nor usable by programmers. This restriction is also related to well-founded recursion in the right-inverse of *fold*s, as we will see later.

4.3 The Combinators

The combinators in RINV are familiar to Haskell programmers too.

4.3.1 Sum and Product

Combinators $\times$ and $\triangledown$ compose functions in parallel. The former applies a pair of functions component-wise to its input, and is defined as

$$\begin{aligned}&(\times) :: (a \leftrightarrows b) \to (c \leftrightarrows d) \to ((a,c) \leftrightarrows (b,d))\\ &[\![f \times g]\!] = \lambda(x,y) \to ([\![f]\!]\ x, [\![g]\!]\ y)\end{aligned}$$

Its right-inverse is:

$$[\![f \times g]\!]^\circ = \lambda(x,y) \to ([\![f]\!]^\circ\ x, [\![g]\!]^\circ\ y)$$

It is well known that $\times$ can be defined in term of a more primitive combinator $\triangle$, which executes both of its input functions on a single datum:

$$\begin{aligned}&(\triangle) :: (a \leftrightarrows b) \to (a \leftrightarrows c) \to (a \leftrightarrows (b,c))\\ &[\![f \triangle g]\!] = \lambda x \to ([\![f]\!]\ x, [\![g]\!]\ x)\end{aligned}$$

However, in the backwards direction, $[\![f]\!]^\circ\ x$ and $[\![g]\!]^\circ\ y$ must converge, which is difficult to enforce. Indeed, functions constructed by $\triangle$ are generally not surjective, and so do not have total right-inverses (see Section 5.3); For this reason, we exclude $\triangle$ from RINV.

The combinator $\triangledown$ consumes an element of a sum type.

$$\begin{aligned}&(\triangledown) :: (a \leftrightarrows c) \to (b \leftrightarrows c) \to (Sum\ a\ b \leftrightarrows c)\\ &[\![f \triangledown g]\!] = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{Inl\ x \to [\![f]\!]\ x\,; Inr\ x \to [\![g]\!]\ x\}\end{aligned}$$

In the backwards direction, if both f and g are total, it doesn't matter which branch is chosen. However, the use of constructor functions deserves some attention. In order to guarantee the totality of $[\![f \triangledown g]\!]^\circ$, functions f and g must be jointly surjective (that is, the union of their ranges should cover the whole of type c). As a result, in the event that $[\![f]\!]^\circ$ fails on certain inputs, $[\![g]\!]^\circ$ should be applicable. To model this failure handling, we lift functions in RINV into the *Maybe* monad, and handle a failure with the first function by invoking the second.

$$[\![f \triangledown g]\!]^\circ = \lambda x \to ([\![f]\!]^\circ\ x)\ \ `mplus`\ \ ([\![g]\!]^\circ\ x)$$

Note that for the sake of presentation, in the sequel of the paper, we still use the non-monadic types for $f \triangledown g$, with the understanding that all functions in RINV are lifted to the *Maybe* monad in the implementation.

In general, it is not an easy task to check (joint) surjectivity of functions. However, in RINV, this is straightforward. The only possibility that $f \triangledown g$ is not jointly surjective is that both f and g are constructor functions; in this case, it is clear that we need the complete set of constructors to satisfy the condition of joint surjectivity.

4.3.2 Composition

Combinator $\circ$ sequentially composes two functions:

$$[\![f \circ g]\!] = [\![f]\!] \circ [\![g]\!]$$

Its inverse is the reverse composition of the inverses of the two functions.

$$[\![f \circ g]\!]^\circ = [\![g]\!]^\circ \circ [\![f]\!]^\circ$$

The more intricate part is to analyse the surjectivity of the composition (and hence the totality of its inverse). It is clear that if one of the functions in a chain of compositions is not surjective, the composed function may also be non-surjective. However, there is no

easy way of determining the range of such a composition if the non-surjective function is not the leftmost one in the chain, which makes it unsuitable for constructing jointly surjective functions through combinator $\triangledown$ as discussed above. Therefore, in RINV, we disallow constructor functions on the right of a composition.

4.3.3 Fold

With the ground prepared, we are now ready to discuss recursive combinators. We define

$$\begin{aligned}[\![\mathit{fold}\, f]\!] &= \mathit{fold}_X\, [\![f]\!] \\ [\![\mathit{fold}\, f]\!]^\circ &= \mathit{unfold}_X\, [\![f]\!]^\circ\end{aligned}$$

In what follows, we call the f in $\mathit{fold}\, f$ the 'body' of the fold. The forward semantics of $\mathit{fold}\, f$ is defined in term of a definition of fold in Haskell for a certain datatype X, whereas the right-inverse semantics of it is defined by a corresponding definition of unfold in Haskell.

Note that *unfold* is not in RINV, but is used to define right-inverses. In this paper, we overload *fold*, *unfold* when the datatype is understood. Intuitively, *fold* disassembles a structure and replaces the constructors with applications of the body, effectively collapsing the structure. Function *unfold* on the other hand, takes a seed, splitting it with the body into building blocks of a structure and new seeds, which are themselves recursively unfolded. In short, *fold* collapses a structure whereas *unfold* grows one. A function that grows a structure by *unfold* and then consumes it by *fold* is called a hylomorphism; in this paper, we are interested in hylomorphisms that are the identity.

When an algebraic datatype X is given, Haskell definitions of fold_X and unfold_X can be generated. For example, consider the datatype of lists:

$$\begin{aligned}&\mathit{fold}_L :: (\mathit{Sum}\ ()\ (a,b) \to b) \to (\mathit{List}\ a \to b) \\ &\mathit{fold}_L\, f = \lambda xs \to \mathbf{case}\ xs\ \mathbf{of} \\ &\qquad\qquad \mathit{Nil} \qquad\quad \to f\ (\mathit{Inl}\ ()) \\ &\qquad\qquad (\mathit{Cons}\ x\ xs) \to f\ (\mathit{Inr}\ (x, (\mathit{fold}_L\, f\ xs))) \\ &\mathit{unfold}_L :: (b \to \mathit{Sum}\ ()\ (a,b)) \to (b \to \mathit{List}\ a) \\ &\mathit{unfold}_L\, f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of}\ \mathit{Inl}\ () \qquad \to \mathit{Nil} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Inr}\ (a,b) \to \mathit{Cons}\ a\ (\mathit{unfold}_L\, f\ b)\end{aligned}$$

It is worth mentioning that our definition of fold_L and unfold_L is slightly different from the standard Haskell library functions *foldr* and *unfoldr*, as we use one body on the sum type to better reflect the duality between the two operations. In RINV, we require the body f of *fold* to map the *Inl* branch to a value constructed by a non-recursive constructor of the result type. Practically, this is a reasonable restriction: most uses of *fold* follow this pattern.

Another example is leaf-labelled binary trees.

$$\begin{aligned}&\mathbf{data}\ \mathit{LTree}\ a = \mathit{Leaf}\ a \\ &\qquad\qquad\quad \mid \mathit{Fork}\ (\mathit{LTree}\ a, \mathit{LTree}\ a) \\ &\mathit{fold}_T :: (\mathit{Sum}\ a\ (b,b) \to b) \to \mathit{LTree}\ a \to b \\ &\mathit{fold}_T\, f = \lambda t \to \mathbf{case}\ t\ \mathbf{of} \\ &\quad \mathit{Leaf}\ a \qquad \to f\ (\mathit{Inl}\ a) \\ &\quad \mathit{Fork}\ (t_1,t_2) \to f\ (\mathit{Inr}\ (\mathit{fold}_T\, f\ t_1, \mathit{fold}_T\, f\ t_2)) \\ &\mathit{unfold}_T :: (a \to \mathit{Sum}\ a\ (b,b)) \to b \to \mathit{LTree}\ a \\ &\mathit{unfold}_T\, f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of} \\ &\qquad\qquad \mathit{Inl}\ a \qquad \to \mathit{Leaf}\ a \\ &\qquad\qquad \mathit{Inr}\ (b,b') \to \mathit{Fork}\ (\mathit{unfold}_T\, f\ b, \mathit{unfold}_T\, f\ b')\end{aligned}$$

We use *unfold* to construct the right-inverse of *fold*. From [8], we have the following lemma.

LEMMA 1. $\mathit{fold}\ [\![f]\!] \circ \mathit{unfold}\ [\![f]\!]^\circ \sqsubseteq \mathit{id}$.

Since both *fold* and *unfold* are case-exhaustive when their bodies are case-exhaustive, the only reason for not having an equality in the conclusion above is that *unfold* is potentially non-terminating: when a body does not split a seed into 'smaller' ones, unfolding it creates an infinite structure. This behaviour is useful in certain applications, but it is undesirable here.

The termination of a function constructed by *unfold* can be guaranteed by careful design of its body. A body of type $b \to \mathit{Sum}\ c\ (\ldots, b, \ldots)$ can be seen as a combination of two partially defined functions $f :: b \to c$ and $g :: b \to (\ldots, b, \ldots)$, where f (called the *termination function*) is the terminating condition of *unfold*, and g (called the *splitting function*) splits out new seeds for recursive calls and possibly other data for the structure construction. From the discussion of inverting $\triangledown$, we know that g is only attempted after f fails. Intuitively, function f is a partially defined function that stops unfolding when its pattern matches. Theoretically, this termination pattern could be any valid one. However, practically, as mentioned above, we restrict it to a non-recursive case of type b. The splitting function g should decrease seeds according to some well-founded ordering (that is, there should be no infinite descending chain of seeds). Since we are only dealing with structured data, it suffices to use the total preorder given by the number of payload data items (i.e values of non-container types) carried by structural values.

DEFINITION 1 (Structure Size Ordering). *Given two values v_1 and v_2, we define a total ordering $v_1 \precsim v_2$ iff size $v_1 \leqslant$ size v_2.*

The function *size* is defined over all values as follows:

$$\begin{aligned}\mathit{size}\ (s_1,s_2) &= \mathit{size}\ s_1 + \mathit{size}\ s_2 \\ \mathit{size}\ (C\ s_1 \ldots s_n) &= \mathit{size}\ s_1 + \ldots + \mathit{size}\ s_n \\ \mathit{size}\ t &= 1\end{aligned}$$

The size of a pair or a container-type data constructor is the sum of its components' sizes. Anything else, namely a non-container-type value, has size 1. Basically, function *size* counts the number of non-container-type values in a structured value.

With this ordering, we have, for example, $\mathit{Nil} \precsim (\mathit{Cons}\ 1\ \mathit{Nil}) \precsim (\mathit{Fork}\ (\mathit{Leaf}\ \texttt{'a'})\ (\mathit{Fork}\ (\mathit{Leaf}\ \texttt{'b'})\ (\mathit{Leaf}\ \texttt{'a'})))$. Note that the exact data carried by and the types of the structures are not important in the ordering.

Now, having the splitting functions produce smaller seeds is a sufficient condition for termination of functions constructed by *unfold*.

The down-side of this technique is that the programmer must check this condition every time *fold* is used. A much better solution would be to make this property compositional, reducing the termination check to the primitive functions only. The difficulty is that since a primitive function can be used to construct bodies for different *unfold*s, it becomes unclear which parts of the output are the new seeds and should be decreasing. As an example, consider the function $[\![\mathit{app}]\!]^\circ$.

$$[\![\mathit{app}]\!]^\circ = \lambda xs \to \mathit{splitAt}\ ((\mathit{length}\ xs + 1)\ `\mathit{div}`\ 2)\ xs$$

For example, applying it to a singleton list gives $[\![\mathit{app}]\!]^\circ\ (\mathit{Cons}\ 1\ \mathit{Nil}) \equiv (\mathit{Cons}\ 1\ \mathit{Nil}, \mathit{Nil})$. The second component of the output is strictly decreasing, whereas the first one is not. (It is worth to reiterate here that we don't have to consider the case $[\![\mathit{app}]\!]^\circ\ \mathit{Nil}$ because non-recursive cases are never handled by the splitting function.) In this case, it is important to know which component is used as the new seed. In the case of

$$\begin{aligned}&\mathit{flatten} :: \mathit{List}\ (\mathit{List}\ a) \leftrightarrows \mathit{List}\ a \\ &\mathit{flatten} = \mathit{fold}\ (\mathit{nil} \triangledown \mathit{app})\end{aligned}$$

where the second component is the new seed, $[\![\mathit{app}]\!]^\circ$ works fine. However, the same function is not suitable for $\mathit{flattenSnoc} :: (\mathit{Snoc}\ (\mathit{List}\ a)) \leftrightarrows \mathit{List}\ a$, since *unfold* for snoclists uses the first component as the new seed. As a result, assuming the knowledge

of which output component of a splitting function is decreasing, it has to be matched with the seed-generation pattern of the *unfold*, which determines the generation of new seeds, when used. We call a body *compatible* if its decreasing output components include the newly generated seeds for the *unfold*. In the above example, $[\![app]\!]^\circ$ is compatible with respect to *flatten* but not to *flatternSnoc*.

The information about decreasing outputs is preserved through chains of composed functions, because all right-inverse functions of RINV are affine (that is, there is no creation or duplication of data c.f Section 4.2). Any output component originated solely from a decreasing output component is decreasing.

LEMMA 2. *Given a compatible body f, unfold f is terminating.*

Together with Lemma 1, we conclude the following.

LEMMA 3. *Given f in* RINV, $(fold\ [\![f]\!]) \circ (unfold\ [\![f]\!]^\circ) = id$.

4.4 Programming in RINV

With the knowledge of RINV, we are now ready to study the conversion functions defined in Section 3, which we skiied through.

To start with, let's firstly look at a very useful derived combinator *map* that can be defined in term of *fold*. For example, *map* on list, map_L, is defined as follows.

$$map_L :: (a \leftrightarrows b) \to (List\ a \leftrightarrows List\ b)$$
$$map_L\ f = fold_L \circ (nil \triangledown (cons \circ (f \times id)))$$

Function $map_L\ f$ applies body f uniformly to all the elements of a list without modifying the list structure. Since *nil* and *snoc* forms a complete set of constructors for list. We know they are joint surjective. It is straightforward to know that $([\![f]\!]^\circ \times [\![id]\!]^\circ) \circ [\![cons]\!]^\circ$ is compatible to the *unfold* of list since the right-inverses of constructors are decreasing on each components of the output and this property is preserved by composition to affine functions (all right-inverses in RINV are affine) to the left.

Similarly, *map* on leaf-labeled tree, map_T, is defined as the following.

$$map_T :: (a \leftrightarrows b) \to (Tree\ a \leftrightarrows Tree\ b)$$
$$map_T\ f = fold_T \circ ((leaf \circ f) \triangledown fork)$$

The right-inverses of maps can be easily defined as

$$[\![map\ f]\!]^\circ = map_X\ [\![f]\!]^\circ$$

with respect to certain datatype X.

A function *reverse* is used in the example of binary number encoding, which can be defined as a fold on list.

$$reverse = fold\ (nil \triangledown snoc)$$

On the forward direction, a list is taken apart and the first element is appended to the rear of the output list by *snoc*. This process is terminated by reaching an empty list and an empty list is returned as the result. Since *nil* and *snoc* forms a complete set of constructors for list, which are joint surjective. It is trivial to know that $[\![snoc]\!]^\circ$ is compatible to the *unfold* of list since the right-inverses of constructors are decreasing on each components of the output. On the backward direction,

$$[\![reverse]\!]^\circ = unfold\ [\![nil \triangledown snoc]\!]^\circ$$

Function $[\![snoc]\!]^\circ$ extracts the last element in a list and is added to the front of the result list by *unfold*, which terminates when $[\![nil]\!]^\circ$ can be successfully applied (i.e when the input is the empty list).

The *toSnoc* function of Wadler's example of backward representation of lists can be defined in a very similar manner. Given

$$\textbf{data}\ SnocList\ a = Lin \mid Snoc\ (SnocList\ a)\ a$$
$$[\![lin]\!]\ = \lambda() \to Lin$$
$$[\![lin]\!]^\circ = \lambda Lin \to ()$$
$$[\![snoc'\,]\!]\ = \lambda(xs, x) \to Snoc\ xs\ x$$
$$[\![snoc'\,]\!]^\circ = \lambda(Snoc\ xs\ x) \to (xs, x)$$

we have

$$toSnoc = fold\ (lin \triangledown snoc')$$

Constructor function $snoc'$ constructs a proper snoclist instead of appending an element to the end and *Lin* is used to replace *Nil*.

Function *reverse* is used to construct the to function for the list view of our queue structure.

$$\mathsf{to} :: Queue\ a \to List\ a$$
$$\mathsf{to} = app \circ (id \times reverse)$$

Function to preprocesses a queue by reversing the latter half before appending the two halves into a list. For example, we have:

$$\mathsf{to}\ ([1,2],[3,4,5,6,7]) = [1,2,7,6,5,4,3]$$

(For the sake of presentation, we overload Haskell list notations for *List a*.) The companion from function is

$$\mathsf{from} :: List\ a \to Queue\ a$$
$$\mathsf{from} = [\![app \circ (id \times reverse)]\!]^\circ = ([\![id]\!]^\circ \times [\![reverse]\!]^\circ) \circ [\![app]\!]^\circ$$

In the from direction, a list is firstly split into two and functions $[\![id]\!]^\circ$ and $[\![reverse]\!]^\circ$ applied to their respective parts. For example, we have

$$\mathsf{from}\ ([1,2,7,6,5,4,3]) \equiv ([1,2,7,6],[3,4,5])$$

Since from is a right-inverse of to, we have

$$\begin{aligned}\mathsf{to}\ (\mathsf{from}\ ([1,2,7,6,5,4,3])) &\equiv \mathsf{to}\ ([1,2,7,6],[3,4,5]) \\ &\equiv [1,2,7,6,5,4,3]\end{aligned}$$

Our last example is the traversal of node-labelled binary trees.

$$\textbf{data}\ BinTree\ a = BLeaf \mid BNode\ a\ (BinTree\ a, BinTree\ a)$$

The fold/unfold functions for binary trees are as follows.

$$fold_B :: (Sum\ ()\ (a,(b,b)) \to b) \to (BinTree\ b \to b)$$
$$fold_B\ f = \lambda x \to \textbf{case}\ x\ \textbf{of}$$
$$\quad BLeaf \to f\ (Inl\ ())$$
$$\quad BNode\ a\ (l,r) \to f\ (Inr\ (a, (fold_B\ f\ l, fold_B\ f\ r)))$$
$$unfold_B :: (b \to Sum\ ()\ (a,(b,b))) \to (b \to BinTree\ b)$$
$$unfold_B\ f =$$
$$\quad \lambda x \to \textbf{case}\ f\ x\ \textbf{of}$$
$$\quad\quad Inl\ () \to BLeaf$$
$$\quad\quad Inr\ (a,(l,r)) \to BNode\ a\ (unfold_B\ f\ l, unfold_B\ f\ r)$$

Using the $fold_B$ combinator, the pre-order traversal of a binary tree can be defined as follows.

$$preOrd = fold_B\ (nil \triangledown (cons \circ (id \times app)))$$

Similarly for the post-order traversal, we have

$$postOrd = fold_B\ (nil \triangledown (snoc \circ swap \circ (id \times app)))$$

In the forwards direction, $fold_B$ adds the node value either at the front or the end of the concatenation of its two subtrees' traversals. In the backwards direction, a node value is extracted from the input list, and the rest of the list is divided and grown into individual trees. For termination, we can easily conclude that $[\![cons]\!]^\circ$ is decreasing at each of its output components, and $[\![id \times app]\!]^\circ$ preserves this property. Thus, $[\![cons \circ (id \times app)]\!]^\circ$ is compatible with the *unfold* of *BinTree a*. A very similar argument applies to $[\![snoc \circ swap \circ (id \times app)]\!]^\circ$.

5. Evaluation

Up to now, we have shown our proposed design of a view system and the right-inverse language RINV, on which it is based. We have

implemented our proposal as a preprocessor for GHC; it translates source syntax with views into native Haskell code.The conversion functions are defined in RINV, which is embedded as a Haskell library. Our system can be invoked directly from GHC with a command such as

```
ShellPrompt> ghc -F -pgmF viewpp YourCodeWithView.hs
```

where `viewpp` is the name of the preprocessor. The implementation is available from `http://www.ipl.t.u-tokyo.ac.jp/~kztk/viewpp/`, together with sample programs.

At the moment, we are still in the process of implementing the termination/surjectivity check for RINV and more optimisations for the translation. Nevertheless, in order to have a more meaningful evaluation of the design, we assume optimised translation in Section 5.2.

The distinctive advantage of our proposal is the support for sound reasoning on the view level. As one usually expects, additional properties may come with certain trade-offs. In this section, we evaluate the performance of our system in various aspects, and make comparison with other related works, in particular Wadler's views, which the closest piece of related work.

5.1 Reasoning Power

Our view system is the only one that reconciles sound equational reasoning with view pattern matching. The limitation of our system is also clear: we don't achieve completeness. View level reasoning does not work between views.

Despite not being able to claim any soundness guarantee, Wadler's proposal is known for supporting reasoning in the presence of views. It is nevertheless interesting to compare its reasoning expressiveness with that of ours, to see how much the pursuit of soundness has costed us.

In Wadler's proposal, the user-defined conversions bridge across different views. Given the heterogeneous view types, the one-step conversions are expected to specify isomorphism of structures. Taking backward lists as an example, $Cons\ x\ (Snoc\ xs\ y)$ is equivalent to $Snoc\ (Cons\ x\ xs)\ y$ and $Cons\ x\ Nil$ is equivalent to $Snoc\ Nil\ x$. This kind of equation is useful in calculating in a bottom-up manner when a fixed value is given. For example, it is possible (though awkward) to derive $Cons\ 1\ (Cons\ 2\ Nil) \equiv Snoc\ (Snoc\ Lin\ 1)\ 2$ as follows, which cannot be done in our approach.

$$
\begin{array}{ll}
 & Cons\ 1\ (Cons\ 2\ Nil) \\
\equiv & \quad \{\, Nil \equiv Lin \,\} \\
 & Cons\ 1\ (Cons\ 2\ Lin) \\
\equiv & \quad \{\, Cons\ x\ Lin \equiv Snoc\ Nil\ x \,\} \\
 & Cons\ 1\ (Snoc\ Nil\ 2) \\
\equiv & \quad \{\, Cons\ x\ (Snoc\ xs\ y) \equiv Snoc\ (Cons\ x\ xs)\ y \,\} \\
 & Snoc\ (Cons\ 1\ Nil)\ 2 \\
\equiv & \quad \{\, Nil \equiv Lin \,\} \\
 & Snoc\ (Cons\ 1\ Lin)\ 2 \\
\equiv & \quad \{\, Cons\ x\ Lin \equiv Snoc\ Nil\ x \,\} \\
 & Snoc\ (Snoc\ Nil\ 1)\ 2 \\
\equiv & \quad \{\, Nil \equiv Lin \,\} \\
 & Snoc\ (Snoc\ Lin\ 1)\ 2
\end{array}
$$

However, this style of reasoning is of limited interest, because proofs are usually conducted by top-down inductive means. As Wadler [24] says, "of course, the main value of equational reasoning is not in calculating values but in performing proofs." For example, the explicit conversions are not useful in proving $rotLeft \circ rotRight \equiv id$. (Wadler's claim [24] that this is provable is a mistake. What was proved was that $rotRight \circ rotLeft \equiv id$, which did not use any of the conversions.) This is due to the fact that in Wadler's setting, inputs are always constructed in the underlying datatype instead of in a view. Given $rotLeft\ (rotRight\ (Cons\ x\ xs))$ there is no way of converting $Cons\ x\ xs$ to a *Snoc* view, which would allow the calculation of *rotRight* to proceed.

This problem is universal in Wadler's approach. Consider another example in [24], which views a list of pairs as a pair of lists (also known as the *Zip* view). The conversions are

$$
\begin{array}{ll}
Nil & = Zip\ (Nil, Nil) \\
Cons\ (a,b)\ (Zip\ (as,bs)) & = Zip\ (Cons\ a\ as, Cons\ b\ bs)
\end{array}
$$

Given a simple function that swaps the two elements in the pairs:

$$f\ (Zip\ xs\ ys) = Zip\ ys\ xs$$

we cannot state anything about $f \circ f$, for the same reason: the conversions do not help us to proceed from $f\ (f\ (Cons\ (a,b)\ xs))$.

In our case, one cannot derive $rotLeft\ (rotRight\ (Cons\ x\ xs)) \equiv Cons\ x\ xs$ either. However, since both conslists and snoclists are views with equal status, we can prove properties of functions in their respective views. For example, we can prove an equally useful property $rotLeft\ (rotRight\ (Snoc\ xs\ x)) \equiv Snoc\ xs\ x$; this gives us a partially defined identity function. Given an input in the *Snoc* view, it is the identity. Otherwise, we know $rotLeft \circ rotRight$ and $\lambda(Snoc\ xs\ x) \to Snoc\ xs\ x$ have the same behaviour; and they can be substituted for each other by a compiler. However, our views do not offer advantages in understanding this programmer behaviour; calculations have to be conducted considering explicit to and from conversions, just like the case of abstract datatypes [4].

In conclusion, the use of explicit axioms for equational reasoning in Wadler's views supports fiddling of values into their intermediate view representations, which is useful in reasoning about computations. However, for more important uses of equational reasoning, namely performing proofs, our approach allows more properties to be expressed.

5.1.1 Parametricity of Views

Inspired by and based on parametricity of datatypes [25], we can conclude an interesting parametricity result of views.

THEOREM 3. *Given two parametric views V a and W a with their corresponding map functions* map_V *and* map_W*, let f be a function pattern matching on view V a and let w be a value in view W a. Then we have* $f\ (map_V\ g\ w) = f\ (map_W\ g\ w)$ *for any g.*

This theorem basically states that given a value constructed in one view but consumed in a different view, using a map function on either view before the consumption gives identical results. A proof can be done on the translated program, making use of parametricity. Assuming that f is translated into $\mathsf{from}_V \circ f' \circ \mathsf{to}_V$, map_V is translated into $\mathsf{from}_V \circ map'_V \circ \mathsf{to}_V$, and map_W is translated into $\mathsf{from}_W \circ map'_W \circ \mathsf{to}_W$, we have:

$$
\begin{array}{ll}
 & (\mathsf{from}_V \circ f' \circ \mathsf{to}_V)\ ((\mathsf{from}_V \circ map'_V\ g \circ \mathsf{to}_V)\ (\mathsf{from}_W\ w)) \\
\equiv & \quad \{\, \text{composition} \,\} \\
 & (\mathsf{from}_V \circ f' \circ \mathsf{to}_V)\ ((\mathsf{from}_V \circ map'_V\ g \circ (\mathsf{to}_V \circ \mathsf{from}_W))\ w) \\
\equiv & \quad \{\, \text{parametricity} \,\} \\
 & (\mathsf{from}_V \circ f' \circ \mathsf{to}_V)\ ((\mathsf{from}_V \circ (\mathsf{to}_V \circ \mathsf{from}_W) \circ map'_W\ g)\ w) \\
\equiv & \quad \{\, \text{composition} \,\} \\
 & (\mathsf{from}_V \circ f' \circ \mathsf{to}_V \circ \mathsf{from}_V \circ \mathsf{to}_V \circ \mathsf{from}_W \circ map'_W\ g)\ w \\
\equiv & \quad \{\, \text{composition} \,\} \\
 & (\mathsf{from}_V \circ f' \circ (\mathsf{to}_V \circ \mathsf{from}_V) \circ \mathsf{to}_V \circ \mathsf{from}_W \circ map'_W\ g)\ w \\
\equiv & \quad \{\, \mathsf{to}_V \circ \mathsf{from}_V \equiv id \,\} \\
 & (\mathsf{from}_V \circ f' \circ \mathsf{to}_V \circ \mathsf{from}_W \circ map'_W\ g)\ w \\
\equiv & \quad \{\, \text{composition} \,\} \\
 & (\mathsf{from}_V \circ f' \circ \mathsf{to}_V)\ (\mathsf{from}_W\ (map'_W\ g\ w))
\end{array}
$$

Note that this theorem does not hold if f pattern matches on the view in which w is constructed. However, this is of less practical interest anyway, since trying to use a different map introduces extra conversions.

5.2 Efficiency

The design of views aims at supporting modular programming with reasoning in mind. With improved modularity, it should be expected that operations defined on high-level data interfaces necessarily incur run-time overhead. In this case, the price to pay is the conversions from datatypes to views. Haskell's lazy semantics helps in certain cases, and fusion optimisations eliminate many intermediate conversions. Compared to Wadler's design, our approach is superior when there is no frequent change of view during an evaluation, which we believe covers the majority of programs. Consider nested pattern matching as a case study, using the function that sums elements of a list of integers pair-wise [22]. We program in the *Snoc* view for illustrative purposes.

$$\begin{array}{ll} \mathit{pairSum}\ (\mathit{Snoc}\ (\mathit{Snoc}\ \mathit{ys}\ y)\ x) & = \mathit{Snoc}\ (\mathit{pairSum}\ \mathit{ys})\ (x+y) \\ \mathit{pairSum}\ (\mathit{Snoc}\ \mathit{Lin}\ x) & = \mathit{Snoc}\ \mathit{Lin}\ x \\ \mathit{pairSum}\ \mathit{Lin} & = \mathit{Lin} \end{array}$$

Assume the underlying datatype is conslists; since only one view is used in the above code, our technique only requires the conversion into the *Snoc* view once, whereas in Wadler's case, nested pattern matching means repeated conversions as shown in the following.

$$\begin{array}{l} \mathit{pairSum}\ l = \textbf{case}\ \mathsf{to}\ l\ \textbf{of}\ \ \mathit{Snoc}\ \mathit{xs}\ x \rightarrow \\ \qquad\qquad \textbf{case}\ \mathsf{to}\ \mathit{xs}\ \textbf{of}\ (\mathit{Snoc}\ \mathit{ys}\ y) \rightarrow \\ \qquad\qquad\quad \mathsf{from}\ (\mathit{Snoc}\ (\mathit{pairSum}\ \mathit{ys})\ (x+y)) \\ \qquad\qquad\quad\ \ldots \end{array}$$

Moreover, due to the fact that homogeneous snoclists do not exist in Wadler's setting, the intermediate conversions in a recursion cannot be removed. As a result, linear time conversions are performed on every iteration, which gives quadratic performance for the program above.

In our case, since the consumptions of view values by *Snoc* are in the same view as the productions of view values by *pairSum*, the optimised translation produces the following program.

$$\mathit{pairSum} = \mathsf{from} \circ \mathit{pairSum}' \circ \mathsf{to}$$

where $\mathit{pairSum}'$ is the original *pairSum*. In this version, there are no double conversions for the nested pattern, and the intermediate conversions inside the recursion are removed, producing a linear-time program.

On the other hand, heterogeneous view structures are beneficial in some scenarios.

$$f\ (\mathit{Snoc}\ (\mathit{Cons}\ x\ \mathit{xs})\ y) = \ldots$$

Given that the nested pattern above is identical to the snoclist heterogeneous construction pattern, Wadler's design fits just as well. However, we believe that in practical programming, the former is far more common than the latter.

Perhaps a bigger controversy in the design of view mechanisms is the datatype-like syntax. We think this is positive, since it preserves the elegant syntax of Haskell. On the other hand, there is the concern that this similar look and feel may cause programmers to overlook the possibility of non-constant run-time cost of pattern matching on views. This is certainly a valid concern. However, at the same time, it is worth pointing out that programmers do have some warning. First of all, views are clearly declared by separate keywords. Secondly, when a calculation is performed to try to understand the behaviour of programs, such as $(\lambda(\mathit{Snoc}\ \mathit{xs}\ x) \rightarrow \mathit{Snoc}\ \mathit{xs}\ x)\ (\mathit{Cons}\ 1\ \mathit{Nil})$, there is a clear signal that some sort of conversion is going on to make this possible.

5.3 Expressiveness of View Definition

In our system, the set of definable views is determined by the existence of to functions in RINV that map to them. RINV is designed to be extensible: new primitives (and even new combinators) can be added to the language if needed. The real limitation of RINV we face here is that all functions must be surjective, in order to ensure totality of the right-inverses.

This is certainly desirable if the functions are used as conversions for views. At the same time, we inherit the weakness of algebraic datatypes: they perform poorly in expressing semantic invariants on data representations. An algebraic datatype often represents a larger set of values than is used in a program. With a more elaborate type system or some other language extensions (such as contracts), more refined specifications may be placed on datatypes. Despite advances in technology, most mainstream programming languages both in the functional and other paradigms have very limited support for such a feature. It is generally accepted that programmers are entrusted with the responsibility not to create values outside certain specified boundaries. In the study of invertible (bidirectional) languages [9], it is common practice to admit a weaker version of totality, namely totality with respect to the actual range of conversion functions.

In the case of views, the valid view values are bound by the actual range of the user-defined to function; invertibility is not guaranteed for view values outside this range. In the current proposal, the to functions in RINV are always surjective, which rules out some useful programs. An example already mentioned is the combinator $\triangle$. Since $f \triangle g$ is generally not surjective, it doesn't have a total right-inverse, despite the fact that we can easily guard against inconsistent input in the reverse direction as follows.

$$\begin{array}{l} [\![f \triangle g]\!]^\circ = \lambda(a,b) \rightarrow \textbf{if}\ x == y\ \textbf{then}\ x\ \textbf{else}\ \mathit{error}\ \texttt{"violation"} \\ \quad \textbf{where}\ x = [\![f]\!]^\circ\ a\,; y = [\![g]\!]^\circ\ b \end{array}$$

The primary use of $\triangle$ is to define the function *dupF* that duplicates a value and transforms one copy by the input function.

$$\begin{array}{l} \mathit{dupF} :: \mathit{Eq}\ a \Rightarrow (a \rightarrow b) \rightarrow (a \leftrightarrows (b,a)) \\ \mathit{dupF}\ f = f \triangle \mathit{id} \end{array}$$

As we have seen in the sized tree example (Section 3.3), function *dupF* is particularly useful when we would like to preprocess an input in some way while keeping the original.

Another very useful function is *unzip*, which can be defined as a fold.

$$\mathit{unzip} = \mathit{fold}_L\ ((\mathit{nil} \triangle \mathit{nil}) \triangledown ((\mathit{cons} \times \mathit{cons}) \circ \mathit{trans}))$$

This definition will be rejected in RINV, since $\mathit{cons} \times \mathit{cons}$ and $\mathit{nil} \triangle \mathit{nil}$ are not jointly surjective. Indeed, *unzip* only produces pairs of lists of equal length. This is also the very reason that we exclude *unfold* as a combinator in RINV, as it in general only constructs structures of a particular shape, decided by the splitting strategy of its operation.

If a view value outside the range is constructed, the integrity of view level equational reasoning is corrupted. It remains an open question whether we should allow programmers to take some reasonable responsibilities or should insist on holding control.

6. Invertible (Bidirectional) Languages

The language RINV presented in Section 4 is directly inspired by the rich literature of invertible and bidirectional languages [1, 3, 9, 11, 17, 19]. Invertible languages [11, 19] aim at automatically deriving inverses of functions. The language proposed in [19] is based on the pointfree style of programming, and has a similar look and feel to RINV. Due to the strong condition of existence of inverses, functions in invertible languages are generally required to be injective; totality of the inverses is not discussed. Alimarine *et al.* [1] express bidirectional combinators as arrows [12]; they also discuss combinators to obtain right inverses, but the totality of the result is not guaranteed in their framework.

Bidirectional languages [3,9,11,17] have a more flexible framework than that of invertible languages. Typically, a function $get :: s \to a$ maps a *source* value of type s into an *abstract* value of type a; a backward function $put :: (s, a) \to s$ takes a possibly updated abstract value and the original source to produce a new source that 'properly' reflects the updates. The additional information available to the backward function relaxes the injectivity requirement in invertible languages, and allows more programs to be defined.

Despite the rich results in bidirectional transformations, the technique does not apply directly to our application: views always start from the abstract level and don't have an 'original' source value. For example, if we construct a queue value in the list view *Cons* 1 *Nil*, a *put* function doesn't apply because there is no concept of the 'original source' queue value.

It is interesting to observe that in the bidirectional framework, it is often difficult or impossible to merge the inputs to the *put* functions, namely the original source and the updated abstract value, to produce an updated source. As a result, the idea of a function $create :: a \to s$ has emerged [3, 9] to tackle the problem of missing or update-invalidated original sources. This *create* function is exactly the right-inverse in RINV. However, prior studies of *create* functions focus on creating missing data fields through user-defined default values; on the other hand, RINV focuses on the construction of structures and assumes no data-loosing on the forward direction.

Total backward transformations has been attempted before [3, 9]. The main difference from our work is the use of *semantic* types [10] instead of the more common *syntactic* types. As a result, their types are closer to specifications of the functions defined, which allows the development of a concept of totality of backward transformations with respect to the precise ranges of the forward transformations. This is certainly useful (c.f. Section 5.3), but precludes checking the types with traditional type-checking mechanisms, such as the Haskell type checker.

Another notable bidirectional system [17] is based on the constant complement approach. The idea is to keep histories (complements) of the computation in the forward direction, and to use this information to reconstruct properly updated sources. The system described in that paper is capable of handling user-defined algebraic datatypes and automatically generating *put* functions. Due to the fact that the complements must be kept constant at all times, not all updates are permissible on views. In order to have a decidable check on permissible updates, the bidirectional language has to be restricted. It will be interesting to see whether *create* functions (right-inverses) constructed in RINV can be used to supplement *put* functions to create total backward transformations that work on all updates and are freed from many of these restrictions.

7. Conclusion

We tackle the reputation of views as being unsound and the long-standing problem of equational reasoning with views in the broader literature. We see our work as a conservative (sound and backwards-compatible) and serious (not compromising any modularity benefit of views) step towards supporting pattern matching with abstract types. The view system we present is built on an extensible combinator-based invertible language that derives a total right-inverse for each of the functions defined. At this stage, we do not admit any pattern extensions (such as non-linear patterns) that are not defined in Haskell; however, there is no technical reason why our approach cannot be used to support pattern extensions as proposed in many pattern languages. Nevertheless, we are likely to stumble across the lack of contract checking for algebraic datatypes; it is definitely a promising direction to investigate the application of stronger invariants to views.

References

[1] A. Alimarine, S. Smetsers, A. van Weelden, M. van Eekelen, and R. Plasmeijer. There and back again: Arrows for invertible programming. In *Haskell Workshop*, 2005.

[2] R. S. Bird. A calculus of functions for program derivation. In *Research Topics in Functional Programming*, pages 287–307. Addison-Wesley, 1990.

[3] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *POPL*, Jan. 2008.

[4] F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *J. Funct. Program.*, 3(2):171–190, 1993.

[5] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, 2007.

[6] M. Erwig. Active patterns. In *IFL*, 1996.

[7] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, 2000.

[8] M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, Netherlands, 1991.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[10] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

[11] Z. Hu, S. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM*, 2004.

[12] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[13] C. B. Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6), 2004.

[14] D. Licata and S. Peyton Jones. View patterns: lightweight views for Haskell, 2007. Haskell Café mailing list.

[15] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.

[16] J. Liu and A. C. Myers. JMatch: Iterable abstract pattern matching for Java. In *PADL*, 2003.

[17] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP*, 2007.

[18] P. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *CC*, 2003.

[19] S. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bidirectional updating. In *APLAS*, 2004.

[20] C. Okasaki. Views for Standard ML. In *ACM Workshop on ML*, 1998.

[21] P. Palao Gostanza, R. Peña, and M. Núñez. A new look at pattern matching in abstract data types. In *ICFP*, pages 110–121, 1996.

[22] D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP*, 2007.

[23] M. Tullsen. First class patterns. In *PADL*, 2000.

[24] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL*, 1987.

[25] P. Wadler. Theorems for free! In *FPCA*, 1989.

A. Additional Examples of Reasoning

A.1 Enqueuing then Dequeuing

First we consider the property

$(deQ \circ enQ\ a)\ emptyQ \equiv emptyQ$

Reasoning on the view level:

$$
\begin{array}{ll}
 & (deQ \circ (enQ\ a))\ emptyQ \\
\equiv & \quad \{\text{ composition }\} \\
 & deQ\ (enQ\ a\ emptyQ) \\
\equiv & \quad \{\ emptyQ \text{ definition }\} \\
 & deQ\ (enQ\ a\ (Pr\ ([\,],[\,]))) \\
\equiv & \quad \{\ enQ \text{ definition }\} \\
 & deQ\ (Pr\ ([\,],[a])) \\
\equiv & \quad \{\ deQ \text{ definition }\} \\
 & (deQ\ (Pr\ (reverse\ [a],[\,])) \\
\equiv & \quad \{\ reverse \text{ definition }\} \\
 & (deQ\ (Pr\ ([a],[\,]))) \\
\equiv & \quad \{\ deQ \text{ definition }\} \\
 & cPr\ ([\,],[\,]) \\
\equiv & \quad \{\ emptyQ \text{ definition }\} \\
 & emptyQ
\end{array}
$$

Reasoning on the datatype level:

$$
\begin{array}{ll}
 & (deQ \circ (enQ\ a))\ emptyQ \\
\equiv & \quad \{\text{ composition }\} \\
 & deQ\ (enQ\ a\ emptyQ) \\
\equiv & \quad \{\ enQ \text{ definition }\} \\
 & deQ\ (\textbf{case}\ to\ emptyQ\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (fq,a:bq)) \\
\equiv & \quad \{\ emptyQ \text{ definition }\} \\
 & deQ\ (\textbf{case}\ to\ (cPr\ ([\,],[\,]))\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (fq,a:bq)) \\
\equiv & \quad \{\ cPr \text{ definition }\} \\
 & deQ\ (\textbf{case}\ (to\ (from\ (Pr\ ([\,],[\,]))))\ \textbf{of} \\
 & \quad Pr\ (fq,bq) \quad \rightarrow cPr\ (fq,a:bq)) \\
\equiv & \quad \{\ to \circ from \equiv id\ \} \\
 & deQ\ (\textbf{case}\ (Pr\ ([\,],[\,]))\ \textbf{of}\ Pr\ (fq,bq) \rightarrow cPr\ (fq,a:bq)) \\
\equiv & \quad \{\ \textbf{case} \text{ statement }\} \\
 & deQ\ (cPr\ ([\,],[a])) \\
\equiv & \quad \{\ deQ \text{ definition }\} \\
 & \textbf{case}\ to\ (cPr\ ([\,],[a]))\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\ cPr \text{ definition }\} \\
 & \textbf{case}\ to\ (from\ (Pr\ ([\,],[a])))\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\text{ composition }\} \\
 & \textbf{case}\ (to \circ from)\ (Pr\ ([\,],[a]))\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\ to \circ from \equiv id\ \} \\
 & \textbf{case}\ Pr\ ([\,],[a])\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\ \textbf{case} \text{ statement }\} \\
 & deQ\ (cPr\ ([a],[\,])) \\
\equiv & \quad \{\ \text{deQ definition }\} \\
 & \textbf{case}\ to\ (cPr\ ([a],[\,]))\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\ cPr \text{ definition }\} \\
 & \textbf{case}\ to\ (from\ (Pr\ ([a],[\,])))\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\text{ composition }\} \\
 & \textbf{case}\ (to \circ from)\ (Pr\ ([a],[\,]))\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\ to \circ from \equiv id\ \} \\
 & \textbf{case}\ Pr\ ([a],[\,])\ \textbf{of} \\
 & \quad Pr\ ([\,],bq) \quad \rightarrow deQ\ (cPr\ (reverse\ bq,[\,])) \\
 & \quad Pr\ ((a:q),bq) \rightarrow cPr\ (q,bq) \\
\equiv & \quad \{\ \textbf{case} \text{ statement }\} \\
 & cPr\ ([\,],[\,]) \\
\equiv & \quad \{\ emptyQ \text{ definition }\} \\
 & emptyQ
\end{array}
$$

A.2 Map Fusion

For map fusion

$map\ f \circ map\ g \equiv map\ (f \circ g)$

reasoning on the view level just uses map fusion on the view as if it were a datatype. For reasoning on the datatype level:

$$
\begin{array}{ll}
 & (from \circ map\ f \circ to) \circ (from \circ map\ g \circ to) \\
\equiv & \quad \{\ to \circ from \equiv id\ \} \\
 & from \circ map\ f \circ map\ g \circ to \\
\equiv & \quad \{\text{ map fusion }\} \\
 & from \circ (map\ (f \circ g)) \circ to
\end{array}
$$

A.3 Take

Now we consider the property

$take\ 1\ (Cons\ 1\ (Cons\ 2\ Nil)) \equiv Cons\ 1\ Nil$

Reasoning on the view level:

$$
\begin{array}{ll}
 & take\ 1\ (Cons\ 1\ (Cons\ 2\ Nil)) \\
\equiv & \quad \{\ take \text{ definition }\} \\
 & Cons\ 1\ Nil
\end{array}
$$

Reasoning on the datatype level:

$$
\begin{array}{ll}
 & (from \circ take\ 1 \circ to)\ (cons\ 1\ (cons\ 2\ nil)) \\
\equiv & \quad \{\ nil \text{ definition }\} \\
 & (from \circ take\ 1 \circ to)\ (cons\ 1\ (cons\ 2\ (from\ Nil))) \\
\equiv & \quad \{\ cons \text{ definition }\} \\
 & (from \circ take\ 1 \circ to) \\
 & \quad (from\ (Cons\ 1\ (to\ (from\ (Cons\ 2\ (to\ (from\ Nil))))))) \\
\equiv & \quad \{\text{ associativity }\} \\
 & (from \circ take\ 1 \circ to \circ from) \\
 & \quad (Cons\ 1\ ((to \circ from)\ (Cons\ 2\ ((to \circ from)\ Nil)))) \\
\equiv & \quad \{\ to \circ from \equiv id\ \} \\
 & (from \circ take\ 1)\ (Cons\ 1\ (Cons\ 2\ Nil)) \\
\equiv & \quad \{\text{ composition }\} \\
 & from\ (take\ 1\ (Cons\ 1\ (Cons\ 2\ Nil))) \\
\equiv & \quad \{\ take \text{ definition }\} \\
 & from\ (Cons\ 1\ Nil)
\end{array}
$$