Gradual Refinement

Blending Pattern Matching with Data Abstraction

Meng Wang[1], Jeremy Gibbons[1], Kazutaka Matsuda[2], and Zhenjiang Hu[3]

[1] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{menw,jg}@comlab.ox.ac.uk
[2] Graduate School of Information Sciences, Tohoku University
Aramaki aza Aoba 6-3-09, Aoba-ku, Sendai-city, Miyagi-pref. 980-8579, Japan
kztk@kb.ecei.tohoku.ac.jp
[3] GRACE Center, National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
hu@nii.ac.jp

Abstract. Pattern matching is advantageous for understanding and reasoning about function definitions, but it tends to tightly couple the interface and implementation of a datatype. Significant effort has been invested in tackling this loss of modularity; however, decoupling patterns from concrete representations while maintaining soundness of reasoning has been a challenge. Inspired by the development of invertible programming, we propose an approach to abstract datatypes based on a right-invertible language RINV—every function has a right (or pre-) inverse. We show how this new design is able to permit a smooth incremental transition from programs with algebraic datatypes and pattern matching, to ones with proper encapsulation (implemented as abstract datatypes), while maintaining simple and sound reasoning.

1 Introduction

1.1 Program Development

Suppose that you are developing a program involving some data structure. You don't yet know which operations you will need on the data structure, or what efficiency constraints you will impose on those operations. Instead, you want to prototype the program, and conduct some initial experiments on the prototype; on the basis of the results from those experiments, you will decide whether a naive representation of the data structure suffices, or whether you need to choose a more sophisticated implementation. In the latter case, you do not want to have to conduct major surgery on your prototype in order to refactor it to use a different representation.

The traditional solution to this problem is to use data abstraction: identify (or evolve) an interface for the abstract datatype, program to that interface, and allow the implementation to vary without perturbing the program. However, that

requires you to prepare in advance for the possible change of representation: it doesn't provide a smooth revision path if you didn't have the foresight to introduce the interface in the first place, but used a bare algebraic datatype as the representation.

Moreover, choosing a naive representation in terms of an algebraic datatype has considerable attractions. Programs that manipulate the data can be defined using *pattern matching* over the constructors of the datatype, rather than having to use 'observer' operations on a data abstraction. This leads to a concise and elegant programming style, which being based on equations is especially convenient for reasoning about program behaviour [42].

1.2 Pattern Matching

As a simple example, consider encoding binary numbers as lists of bits, most significant first:

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

Functions are typically defined by pattern matching. Consider normalizing binary numbers by eliding leading zeroes.

$$\begin{array}{lll}\mathit{normal} :: \mathit{Num} \to \mathit{Num} & & \\ \mathit{normal}\ [\,] & = [\,] & \text{-- clause (1)}\\ \mathit{normal}\ (\mathit{One} : \mathit{num}) & = \mathit{One} : \mathit{num} & \text{-- clause (2)}\\ \mathit{normal}\ (\mathit{Zero} : \mathit{num}) & = \mathit{normal}\ \mathit{num} & \text{-- clause (3)}\end{array}$$

The definition forms a collection of equations, which give a straightforward explanation of the operational behaviour of the function:

$$\begin{array}{ll} & \mathit{normal}\ [\mathit{Zero}, \mathit{One}, \mathit{Zero}]\\ \equiv & \quad \{\,\text{clause (3)}\,\}\\ & \mathit{normal}\ [\mathit{One}, \mathit{Zero}]\\ \equiv & \quad \{\,\text{clause (2)}\,\}\\ & [\mathit{One}, \mathit{Zero}]\end{array}$$

They are also convenient for calculation; for example, here is one case of an inductive proof that *normal* is idempotent:

$$\begin{array}{ll} & \mathit{normal}\ (\mathit{normal}\ (\mathit{Zero} : \mathit{num}))\\ \equiv & \quad \{\,\text{clause (3)}\,\}\\ & \mathit{normal}\ (\mathit{normal}\ \mathit{num})\\ \equiv & \quad \{\,\text{inductive hypothesis}\,\}\\ & \mathit{normal}\ \mathit{num}\\ \equiv & \quad \{\,\text{clause (3)}\,\}\\ & \mathit{normal}\ (\mathit{Zero} : \mathit{num})\end{array}$$

An equivalent definition without using pattern matching is harder to read:

$$
\begin{aligned}
&\mathit{normal} :: \mathit{Num} \to \mathit{Num} \\
&\mathit{normal}\ \mathit{num} = \mathbf{if}\ \mathit{null}\ \mathit{num} \vee \mathit{one}\ (\mathit{head}\ \mathit{num})\ \mathbf{then}\ \mathit{num} \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{normal}\ (\mathit{tail}\ \mathit{num})
\end{aligned}
$$

It is also much less convenient for calculating with.

Pattern matching has accordingly been supported as a standard feature in most modern functional languages, since its introduction in Hope [7]. More recently, it has started gaining recognition from the object-oriented community [9, 24, 28] too. Unfortunately, the appeal of pattern matching wanes when we need to change the implementation of a data structure: function definitions are tightly coupled to a particular representation, and a change of representation has a far-reaching effect. As a result, it has been observed that the wide spread of pattern matching "leads to a discontinuity in programming: programmers initially use pattern matching heavily, and are then forced to abandon the technique in order to regain abstraction over representations" [39].

1.3 Our Contribution

In this work, then, we strive to address the tension between the convenience of pattern matching and the flexibility of data abstraction by proposing a mechanism to allow programs written with pattern matching to be refactored smoothly and incrementally into ones with abstract datatypes (ADTs) [23], without losing the benefits of simple equational reasoning. In particular, we:

- propose the use of definitions with pattern matching as constructive specifications of ADTs;
- devise an equational reasoning framework for both the primitives of and the user-defined operations on ADTs;
- identify necessary and sufficient conditions for correctness of such equational reasoning;
- design a right-invertible language RINV that guarantees these conditions by construction.

For the sake of demonstration, we explain our proposal using Haskell; but any language providing algebraic datatypes would work just as well.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to ADT specification methods. Section 3 presents our proposed design for pattern matching with ADTs, and Section 4 provides a formal definition of the right-invertible language RINV on which our design is based. We then evaluate the performance and explore alternative points in the design space of our system (Section 5), before discussing related work (Section 6) and concluding (Section 7).

2 ADTs and their Specification

By definition, an ADT is characterized not by its representation or implementation, but by its interface: a fixed set of primitive operations, together with a

specification of their semantics. Different styles of specification possess different strengths and weaknesses, which makes them more or less suitable as refactoring targets from programs defined with algebraic datatypes and pattern matching. In this section, we briefly discuss two popular ways of defining the semantics, namely axiomatic and constructive.

An *axiomatic specification* is implicit: the behaviour of the operations is defined by relating them to each other by a collection of equations. For example, consider operations suitable for queue structures:

$$
\begin{array}{l}
\textbf{adt}\ \mathit{Queue}\ a \\
\quad \begin{array}{ll}
\mathit{emptyQ} & :: \mathit{Queue}\ a \\
\mathit{enQ} & :: a \to \mathit{Queue}\ a \to \mathit{Queue}\ a \\
\mathit{deQ} & :: \mathit{Queue}\ a \to \mathit{Queue}\ a \\
\mathit{first} & :: \mathit{Queue}\ a \to a \\
\mathit{isEmpty} & :: \mathit{Queue}\ a \to \mathit{Bool}
\end{array}
\end{array}
$$

The following axioms are sufficient to specify the semantics:

$$
\begin{array}{lll}
\mathit{deQ}\ (\mathit{enQ}\ a\ \mathit{emptyQ}) & \equiv \mathit{emptyQ} & \\
\mathit{deQ}\ (\mathit{enQ}\ a\ q) & \equiv \mathit{enQ}\ a\ (\mathit{deQ}\ q) & \Leftarrow \mathit{isEmpty}\ q \equiv \mathit{False} \\
\mathit{first}\ (\mathit{enQ}\ a\ \mathit{emptyQ}) & \equiv a & \\
\mathit{first}\ (\mathit{enQ}\ a\ q) & \equiv \mathit{first}\ q & \Leftarrow \mathit{isEmpty}\ q \equiv \mathit{False} \\
\mathit{isEmpty}\ \mathit{emptyQ} & \equiv \mathit{True} & \\
\mathit{isEmpty}\ (\mathit{enQ}\ a\ q) & \equiv \mathit{False} &
\end{array}
$$

(All the free variables above are assumed to be universally quantified over well-defined terms, and equality takes precedence over implication.) If the ADT is implemented in a 'faithful' manner [40], the above specification is all that is known to users, and any properties of programs using the datatype should be derived only from these axioms. The axiomatic approach avoids suggesting any particular representation, and so provides a high degree of abstraction. On the other hand, axiomatic specifications are not easy to construct.

As an alternative, a *constructive specification* explicitly defines the semantics of operations by expressing them in terms of an underlying model. For example, the queue ADT can be related to the familiar list model:

$$
\begin{array}{ll}
\mathit{emptyQ} & = \mathit{emptylist} \\
\mathit{isEmpty} & = \mathit{isNull} \\
\mathit{deQ} & = \mathit{tail} \\
\mathit{enQ}\ a\ q & = \mathit{append}\ q\ (\mathit{wrap}\ a) \\
\mathit{first} & = \mathit{head}
\end{array}
$$

(where *wrap* turns an element into a singleton list). The list model can be seen as another ADT that is sufficiently powerful to simulate the queue ADT. Apparently, the constructive approach makes the specifications easier to write and to understand. The underlying model can be further instantiated into a representation as an algebraic datatype (also known as a *typical object* in the literature [22]):

$$\textbf{data}\ \mathit{Queue}\ a = \mathit{None} \mid \mathit{More}\ a\ (\mathit{Queue}\ a)$$

which results in the following specifications:

$$\begin{array}{ll}
\mathit{emptyQ} & = \mathit{None} \\
\mathit{first}\ (\mathit{More}\ a\ q) & = a \\
\mathit{isEmpty}\ \mathit{None} & = \mathit{True} \\
\mathit{isEmpty}\ x & = \mathit{False} \\
\mathit{enQ}\ a\ \mathit{None} & = \mathit{More}\ a\ \mathit{None} \\
\mathit{enQ}\ a\ (\mathit{More}\ x\ q) & = \mathit{More}\ x\ (\mathit{enQ}\ a\ q) \\
\mathit{deQ}\ (\mathit{More}\ a\ q) & = q
\end{array}$$

It it worth emphasising that the datatype acts only as a model of the ADT: it may suggest but it does not imply a particular implementation. We also note that this constructive approach does not cover all ADTs: for example, unordered sets cannot be fully modelled by an algebraic datatype.

Whether the behaviour of an ADT is specified axiomatically or constructively, the specification can be used in reasoning about programs that make use of the ADT. In particular, one can use the model underlying a constructive specification to infer properties of an implementation; for example, we can easily recover the axiom $\mathit{deQ}\ (\mathit{enQ}\ a\ q) \equiv \mathit{enQ}\ a\ (\mathit{deQ}\ q) \Leftarrow \mathit{isEmpty}\ q \equiv \mathit{False}$ with the following derivation.

$$\begin{array}{ll}
 & \mathit{deQ}\ (\mathit{enQ}\ a\ (\mathit{More}\ b\ q)) \\
\equiv & \quad \{\ \mathit{enQ}\ \} \\
 & \mathit{deQ}\ (\mathit{More}\ b\ (\mathit{enQ}\ a\ q)) \\
\equiv & \quad \{\ \mathit{deQ}\ \} \\
 & \mathit{enQ}\ a\ q \\
\equiv & \quad \{\ \mathit{deQ}\ \} \\
 & \mathit{enQ}\ a\ (\mathit{deQ}\ (\mathit{More}\ b\ q))
\end{array}$$

3 Reasoning with Constructive Specifications

Our purpose is to allow incremental refactoring of a program, replacing an algebraic datatype used with pattern matching by a more sophisticated ADT implementation.

As described in Section 2, a typical approach to specifying an ADT is to use an algebraic datatype as its model and the definitions by pattern matching for constructive specifications of the operations. In moving from an initial program explicitly depending on an algebraic datatype to a refactored one using an ADT, one will generally have to reimplement some of the existing functions as primitive operations of the new ADT, and rewrite the remaining functions in terms of these primitives.

There are two problems with this process. Firstly, it is a 'big bang' refactoring: all uses of the original algebraic datatype have to be changed at once, even though

some of the old definitions may not gain from the refactoring. Secondly, it loses the benefits of pattern matching for the functions that have to be redefined in terms of the ADT primitives: it is no longer so convenient for reasoning.

In this section, we propose a framework free from the above pitfalls: refactoring can be done selectively; and at any point in the process, executability and reasoning are fully supported. We look into the details of the design by means of examples.

3.1 A First Example: FIFO Queue

The queue ADT we have seen is defined via the following specification.

$$
\begin{array}{l}
\textbf{adt}\ \mathit{Queue}\ a = \mathit{None} \mid \mathit{More}\ a\ (\mathit{Queue}\ a) \\
\quad \begin{array}{ll}
\mathit{emptyQ} & = \mathit{None} \\
\mathit{first}\ (\mathit{More}\ a\ q) & = a \\
\mathit{isEmpty}\ \mathit{None} & = \mathit{True} \\
\mathit{isEmpty}\ x & = \mathit{False} \\
\mathit{enQ}\ a\ \mathit{None} & = \mathit{More}\ a\ \mathit{None} \\
\mathit{enQ}\ a\ (\mathit{More}\ x\ q) & = \mathit{More}\ x\ (\mathit{enQ}\ a\ q) \\
\mathit{deQ}\ (\mathit{More}\ a\ q) & = q
\end{array}
\end{array}
$$

This looks similar to an algebraic datatype declaration, but the right-hand side of the definition introduces a model, instead of an implementation, of the ADT. The primitive operations of the ADT are specified in term of this model; despite having a previous life as an executable function, each specification now serves only to express semantics, and is to be replaced by a corresponding ADT implementation at run-time.

As an example, the *enQ* declaration should now be interpreted as a specification:

$$
\begin{array}{lll}
\mathit{enQ}\ a\ q & \equiv \mathit{More}\ a\ \mathit{None} & \Leftarrow q \equiv \mathit{None} \\
\mathit{first}\ (\mathit{enQ}\ a\ q) & \equiv \mathit{first}\ q \wedge \mathit{deQ}\ (\mathit{enQ}\ a\ q) \equiv \mathit{enQ}\ a\ (\mathit{deQ}\ q) & \Leftarrow q \not\equiv \mathit{None}
\end{array}
$$

rather than as a concrete implementation.

Now let's consider a possible concrete representation 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. The primitive operations can be implemented as follows:

$$
\begin{array}{ll}
\mathit{emptyQ\_} & = ([\,], [\,]) \\
\mathit{first\_}\ ([\,], bq) & = \mathit{last}\ bq \\
\mathit{first\_}\ ((a : fq), bq) & = a \\
\mathit{isEmpty\_}\ ([\,], [\,]) & = \mathit{True} \\
\mathit{isEmpty\_}\ q & = \mathit{False}
\end{array}
$$

$$
\begin{array}{ll}
enQ\_\ a\ (fq, bq) & = (fq, a : bq) \\
deQ\_\ ([\,], bq)) & = deQ\_\ (reverse\ bq, [\,]) \\
deQ\_\ (a : fq, bq) & = (fq, bq)
\end{array}
$$

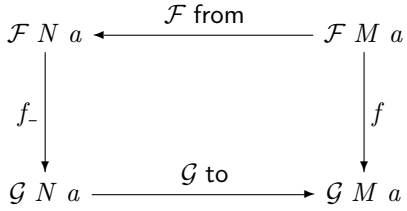
(We use the naming convention of adding an underscore "_" to an operation's name to distinguish its implementation from its specification.) The implementations are what really execute when ADTs are used. Since at the moment, we only reimplemented the primitive operations, queues can still be constructed using the model. During execution, any value constructed in the model is firstly converted to a value in the implementation before being passed to the implemented operations; and the resulting output is converted back to the model. For example, given the program $enQ\ 1\ None$, what really executes is $(\mathsf{to} \circ enQ\ 1 \circ \mathsf{from})\ None$, where the two functions to and from convert the abstract representation of a queue into the model and back again. For the case of queue ADT, the to function can be defined as follows:

$$
\begin{array}{ll}
\mathsf{to}\ ([\,], [\,]) & = None \\
\mathsf{to}\ ([\,], q) & = \mathsf{to}\ (reverse\ q, [\,]) \\
\mathsf{to}\ (x : xs, q) & = More\ x\ (\mathsf{to}\ (xs, q))
\end{array}
$$

The from function should be the right inverse of to—that is, $\mathsf{to} \circ \mathsf{from} \equiv id$.

The operations will not always have types as simple as $Queue\ a \to Queue\ a$, like $enQ\ 1$ does. Suppose we have polymorphic model datatype $M\ a$ and abstract implementation $N\ a$, and polymorphic conversion functions $\mathsf{to} :: N\ a \to M\ a$ and $\mathsf{from} :: M\ a \to N\ a$. In the general case, a model function will take not just a single value in the model (of type $M\ a$), but some combination of model values and other arguments. We capture this in terms of an operation $\mathcal{F}$ on polymorphic datatypes M. Similarly, the operation will return a different combination $\mathcal{G}$ of model values and other results.

Technically, if polymorphic datatypes are represented as functors, then $\mathcal{F}, \mathcal{G}$ are functors on the functor category (what Martin *et al.* [26] call 'higher-order functors', or 'hofunctors' for short), so that $\mathcal{F}\ M$ and $\mathcal{G}\ M$ are themselves functors. Then the model function f will have type $\mathcal{F}\ M\ a \to \mathcal{G}\ M\ a$, and the corresponding implementation function $f\_ :: \mathcal{F}\ N\ a \to \mathcal{G}\ N\ a$ should satisfy the correctness condition $\mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv f$, as shown in the following commuting diagram.



Given the right-inverse property, we can simplify the proof obligation

$$\mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv f$$

to the *promotion* condition [3]

$$\mathcal{G}\ \mathsf{to} \circ f\_ \equiv f \circ \mathcal{F}\ \mathsf{to}$$

which does not involve the function from, as the following lemma demonstrates.

Lemma 1 (Round trip). *Given* $\mathcal{G}$ *to* $\circ f\_ \equiv f \circ \mathcal{F}$ *to, we have* $f \equiv \mathcal{G}$ *to* $\circ f\_ \circ \mathcal{F}$ *from.*

Proof.

$$\begin{array}{ll} & \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\ \equiv & \quad \{\ \mathcal{G}\ \mathsf{to} \circ f\_ \equiv f \circ \mathcal{F}\ \mathsf{to}\ \} \\ & f \circ \mathcal{F}\ \mathsf{to} \circ \mathcal{F}\ \mathsf{from} \\ \equiv & \quad \{\ \mathcal{F} \text{ is a hofunctor}\ \} \\ & f \circ \mathcal{F}\ (\mathsf{to} \circ \mathsf{from}) \\ \equiv & \quad \{\ \mathsf{to} \circ \mathsf{from} \equiv id\ \} \\ & f \circ \mathcal{F}\ id \\ \equiv & \quad \{\ \mathcal{F} \text{ is a hofunctor}\ \} \\ & f \end{array}$$

□

For example, for the operation $first :: Queue\ a \to a$ of the queue ADT, $\mathcal{F}$ is the identity hofunctor, matching the source type $Queue\ a$, and $\mathcal{G}$ is the constantly-identity hofunctor ($\mathcal{G}\ F = Id$), matching the target type a. The operation must satisfy the following promotion condition:

$$first\_ \equiv first \circ \mathsf{to}$$

The promotion equations for the rest of the operations are listed below.

$$\begin{array}{ll} \mathsf{to} \circ deQ\_ & \equiv deQ \circ \mathsf{to} \\ \mathsf{to} \circ enQ\_\ a & \equiv enQ\ a \circ \mathsf{to} \\ isEmpty\_ & \equiv isEmpty \circ \mathsf{to} \\ \mathsf{to}\ emptyQ\_ & \equiv emptyQ \end{array}$$

The proofs of such equations follow by standard equational reasoning.

The astute reader may have noticed that we have avoided explicitly defining the from function. This is because validating user-defined to and from with traditional methods requires additional machinery not available within most mainstream languages, such as Haskell or ML. Instead, we explore a correctness-by-construction technique: in the next section, we will present a combinator-based language RINV implemented as a library in Haskell, in which every definable function gets a right inverse for free. That is to say, the ADT implementer writes only to, in RINV, and the corresponding from function is automatically generated. For the sake of completeness, in the case of the queue ADT presented above, a possible definition in RINV reads:

$$\mathsf{to} = fold\ (none \triangledown more) \circ app \circ (id \times reverse)$$

However, the details of the language are completely orthogonal to the discussion in this section, and can be safely ignored for the time being.

In summary, other than the conventional expectation that implementations of ADTs are certified against the specifications, the only additional requirement on the implementer is the definition of the `to` function. Once this is done, an ADT is sealed off; users of the ADT only interact with the model. They can expect to reason about their programs faithfully using properties of the model; for example, exactly the same reasoning on the model as shown at the end of Section 2 allows us to conclude that

$$deQ\ (enQ\ a\ q) \equiv enQ\ a\ (deQ\ q) \Leftarrow isEmpty\ q \equiv False$$

As mentioned before, a programmer using the ADT now has the choice of keeping the original definitions using pattern matching against the model, or of refactoring them into the traditional style using only the primitive operations of the ADT, or of having a mixture of the two. For example, it is very convenient to define a map function in terms of the list-like model:

$$\begin{aligned}
&mapQ1 :: (a \to b) \to Queue\ a \to Queue\ b\\
&mapQ1\ f\ None \qquad\quad = None\\
&mapQ1\ f\ (More\ a\ q) = More\ (f\ a)\ (mapQ1\ f\ q)
\end{aligned}$$

or a prioritisation function, which is essentially a stable sort based on element weight:

$$\begin{aligned}
&prioritise :: Ord\ a \Rightarrow Queue\ a \to Queue\ a\\
&prioritise\ None \qquad\quad = None\\
&prioritise\ (More\ x\ xs) = insert\ x\ (prioritise\ xs)\\
&\quad \textbf{where}\ insert\ y\ None \qquad\quad = More\ y\ None\\
&\qquad\quad insert\ y\ (More\ x\ xs) = \textbf{if}\ y \leqslant x\ \textbf{then}\ More\ y\ (More\ x\ xs)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\qquad\quad More\ x\ (insert\ y\ xs)
\end{aligned}$$

A definition using only the primitive operations is likely to be more clumsy:

$$\begin{aligned}
&mapQ2 :: (a \to b) \to Queue\ a \to Queue\ b\\
&mapQ2\ f\ q = mapQ2acc\ f\ q\ emptyQ\\
&\quad \textbf{where}\ mapQ2acc\ f\ q\ accq =\\
&\qquad \textbf{if}\ isEmpty\ q\ \textbf{then}\ accq\\
&\qquad \textbf{else}\ mapQ2acc\ f\ (deQ\ q)\ (enQ\ (f\ (first\ q))\ accq)
\end{aligned}$$

Nevertheless, since the non-primitive functions and the primitive operation specifications are based on the same model, we can prove the equivalence of the two versions through equational reasoning.

Definitions with pattern matching are almost always more elegant [42]. However, from time to time, we may want to use the primitive operations for the sake of efficiency, or for reuse of legacy libraries. For example, consider a circular queue that is read for a certain amount of time, say repeatedly playing a piece

of music. Using the primitive *enQ* operation allows us to take advantage of its constant time performance.

```
play1 :: Time → Queue (IO ()) → IO ()
play1 0 q       = first q
play1 (n + 1) q = do hd
                     play1 n (enQ hd tl)
  where hd = first q
        tl = deQ q
```

The two styles of programming can be mixed:

```
play2 0       (More a q) = a
play2 (n + 1) (More a q) = do a
                              play2 n (enQ a q)
```

Equational reasoning interacts with both styles in the obvious way.

3.2 Translation into Haskell

The semantics of non-primitive functions on ADTs can be elaborated by a mechanical translation into ordinary Haskell, following a rather straightforward scheme: each use of a primitive function is replaced with its implementations, precomposed with to and postcomposed with from (subject to the appropriate hofunctors).

First of all, **adt** declarations are translated into **data** declarations.

```
data Queue' a = None | More a (Queue' a)
```

Functions that are written using pattern matching against the model now work with the new datatype. The primitive operations that are defined on the actual implementation require their inputs to be converted from the model before consumption, and the outputs converted back to the model. Effectively, all the translated functions and constructors have the model datatypes as source and target types; the implementations remain only as intermediate structures. As an example, *play2* is translated into the following.

```
play2' 0       (More a q) = a
play2' (n + 1) (More a q) = do a
                               play2' n ((to ∘ enQ_ a ∘ from) q)
```

Given the round trip law (Lemma 1), it is easy to conclude that *play2'* is equivalent to *play2*, in the sense that exactly the same output is produced for each input.

Theorem 2. *The translation into Haskell is semantics-preserving.*

Proof. Follows directly from Lemma 1. □

3.3 Optimization

Up to now, we have achieved sound equational reasoning for ADTs with little additional burden for the ADT implementer. As a result, program construction can benefit from pattern matching and straightforward proofs of correctness. The run-time performance of non-primitive functions making use only of pattern matching can be understood by considering the models as datatypes; however, when primitive operations are called, additional conversion overhead will occur. This performance loss is to be expected for definitions such as *play2*, where an obvious switch from pattern matching to primitive operations is inevitable. However, it may be surprising that *play1*, which only involves primitive operations, is not faster. The translated code is the following.

$$
\begin{aligned}
&\mathit{play1'}\ 0\ q &&= (\mathsf{to} \circ \mathit{first}\_ \circ \mathsf{from})\ q \\
&\mathit{play1'}\ (n+1)\ q &&= \mathbf{do}\ \mathit{hd} \\
&&&\qquad \mathit{play1'}\ n\ ((\mathsf{to} \circ \mathit{enQ}\_\ \mathit{hd} \circ \mathsf{from})\ \mathit{tl}) \\
&\quad \mathbf{where}\ \mathit{hd} &&= (\mathsf{to} \circ \mathit{first}\_ \circ \mathsf{from})\ q \\
&\qquad\quad \mathit{tl} &&= (\mathsf{to} \circ \mathit{deQ}\_ \circ \mathsf{from})\ q
\end{aligned}
$$

There are conversions everywhere in the program. It will be disastrous if all of them have to be executed. Since there is no pattern matching involved, we can try to remove the conversions through fusion. Indeed, the correctness of such fusion follows from the promotion condition. Let's take an expression fragment from the above definition for demonstration. Consider

$$(\mathsf{to} \circ \mathit{enQ}\_\ \mathit{hd} \circ \mathsf{from})\ ((\mathsf{to} \circ \mathit{deQ}\_ \circ \mathsf{from})\ q)$$

Our target is to fuse the intermediate conversions to produce

$$(\mathsf{to} \circ \mathit{enQ}\_\ \mathit{hd} \circ \mathit{deQ}\_ \circ \mathsf{from})\ q$$

This would clearly follow $\mathsf{from} \circ \mathsf{to} \equiv \mathit{id}$, but this is not a property that we guarantee—for good reason, since requiring it in addition to the existing right inverse property $\mathsf{to} \circ \mathsf{from} \equiv \mathit{id}$ demands isomorphic implementations and models, which is too restrictive to be practically useful. Instead, using the promotion condition, we can prove a weaker property that is sufficient for fusion.

Theorem 3 (Fusion Soundness). *Given operation specifications* $f :: \mathcal{F}\ M\ a \to \mathcal{G}\ M\ a$ *and* $g :: \mathcal{G}\ M\ a \to \mathcal{H}\ M\ a$*, and their implementations* $f\_ :: \mathcal{F}\ N\ a \to \mathcal{G}\ N\ a$ *and* $g\_ :: \mathcal{G}\ N\ a \to \mathcal{H}\ N\ a$*, we have* $\mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{H}\ \mathsf{to} \circ g\_ \circ f\_ \circ \mathcal{F}\ \mathsf{from}$.

Proof.

$$
\begin{aligned}
& \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv\ & \quad \{\ \text{promotion: } \mathcal{H}\ \mathsf{to} \circ g\_ \equiv g \circ \mathcal{G}\ \mathsf{to}\ \} \\
& g \circ \mathcal{G}\ \mathsf{to} \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv\ & \quad \{\ \mathcal{G} \text{ is a hofunctor; } \mathsf{to} \circ \mathsf{from} \equiv \mathit{id}\ \}
\end{aligned}
$$

$$
\begin{array}{ll}
 & g \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \quad \{\, \text{promotion: } \mathcal{H}\ \mathsf{to} \circ g\_ \equiv g \circ \mathcal{G}\ \mathsf{to} \,\} \\
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ f\_ \circ \mathcal{F}\ \mathsf{from}
\end{array}
$$

□

Basically, this theorem states that although the input to $g\_$ may differ from the output of $f\_$, due to the $\mathsf{from} \circ \mathsf{to}$ conversions, nevertheless the post-conversion of $g\_$'s output brings possibly different results into the same value in the model.

It is now clear that when pattern matching is not used, *strength reduction* [29] is able to lift the conversion out of the recursion, so that it is done only once. The translation of *play1* can be optimized into the following, which is free from any overhead.

$$
\begin{array}{l}
play1'\ n = \mathsf{to} \circ play1''\ n \circ \mathsf{from} \\
play1''\ 0\ q \qquad\quad = first\ q \\
play1''\ (n+1)\ q = \mathbf{do}\ hd \\
\qquad\qquad\qquad\qquad\quad play1''\ n\ (enQ\ hd\ tl) \\
\quad \mathbf{where}\ hd = first\ q \\
\qquad\qquad tl = deQ\ q
\end{array}
$$

3.4 More Examples

Join Lists. As an alternative to the biased linear list structure, the *join* representation of lists has been proposed for program elegance [4, 27], efficiency [37], and more recently, parallelism [38]. It can be defined as:

$$\mathbf{data}\ List\ a = Empty \mid Unit\ a \mid Join\ (List\ a)\ (List\ a)$$

As a simple example, a constant-time append function (in constrast to the linear-time left-biased-list counterpart) can be defined with this representation.

$$append\ l1\ l2 = Join\ l1\ l2$$

At the same time, we don't want to give up on the familiar notion of *Nil* and *Cons*. Instead, they can serve as a model of the join representation.

$$
\begin{array}{l}
\mathbf{adt}\ List\ a = Nil \mid Cons\ a\ (List\ a) \\
\quad append\ Nil\ ys \qquad\quad = ys \\
\quad append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys)
\end{array}
$$

We can now inherit the rich body of function definitions on lists. For example,

$$head\ (Cons\ x\ xs) = x$$

Binary Numbers. In the introduction, we showed a representation of binary numbers as lists of digits with the most significant bit first (MSB). This representation is intuitive, and offers good support for most operations; however, for incrementing a number, having the least significant bit (LSB) first is better.

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

Effectively, in order to use the above definition with any other operations, we only need to reverse the MSB representation, and a type synonym for *Num* can be used as documentation of this intention. However, the synonyms are of no help to the compiler in guaranteeing correct usage, because they are simply two different names for the same type. At the same time, defining the two representations as completely different types is very cumbersome. With our proposal, we can hide one representation in an ADT, which effectively eliminates any possibility of misuse.

$$
\begin{aligned}
&\textbf{adt } \mathit{Num} = [\mathit{Bin}]\\
&\quad \mathit{incr} = \mathit{reverse} \circ \mathit{incr}' \circ \mathit{reverse}
\end{aligned}
$$

In contrast to other examples, there is no new pattern interface other than the list constructors. However, a programmer using the ADT now only deals with a single representation of binary numbers.

4 The Right-Invertible Language 'RINV'

Our design of ADTs discussed previously relies on the existence of a right inverse, from, of the user-defined conversion function to. This can be guaranteed by writing to in a right-invertible language that automatically generates a right inverse for each function constructed. In this section, we introduce such a language.

The language RINV is defined as a combinator library in Haskell, the syntax of which is as below. (Non-terminals are indicated in small capitals.)

Language	RINV	::= CSTR | PRIM | COMB
Constructors	CSTR	::= *nil* | *cons* | *snoc* | *wrap* | ...
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 [5], but with the additional feature that a right inverse is automatically generated for each function that is constructed. As a result, a definition $f :: s \leftrightarrows t$ in RINV actually represents a pair of functions (hence the notation $\leftrightarrows$): the forward function $[\![f]\!] :: s \to t$, and its right inverse $[\![f]\!]^\circ :: t \to s$, which together satisfy

$[\![f]\!] \circ [\![f]\!]^\circ \equiv id$. For convenience when clear from context, we don't distinguish between f and its forward function $[\![f]\!]$.

The generated right inverses are intended to be total, so the forward functions have to be surjective; this property holds of the primitive functions (except for individual constructors of a multi-variant algebraic datatype) and is preserved by the combinators.

There is an extensible set of primitive functions defining the basic non-terminal building blocks of the language. Any surjective function could be made a primitive in RINV. All primitive functions are uncurried; this fits better with the invertible framework, where a clear distinction between input and output is required. For the sake of demonstration, we present a small but representative collection of primitive functions above: *swap*, *assocl*, and *assocr* rearrange the components of an input pair; *id* is the identity operation; *app* is the uncurried append function on lists. As we will show, with just these few we can define many interesting functions.

The set of constructor functions is also extensible, via new datatypes. We use lowercase names for the uncurried versions of constructors. In addition to the left-biased list constructor *cons* that comes with the usual datatype declaration, we also include its right-biased counterpart *snoc*, which adds an element at the end; it can be defined in Haskell as

$$snoc = \lambda(x, xs) \to xs +\!\!+ [x]$$

Another additional constructor for lists is *wrap*, which creates a singleton list.

$$wrap\ x = [x]$$

This ability to admit functions that do not directly arise from a datatype declaration as constructors is crucial for the expressiveness of RINV, which otherwise would be rigidly surjective. Although this might seem ad hoc, it is by no means arbitrary. One should only use functions that truly model a different representation of the datatype. For example, *snoc* and *nil* form the familar backward representation of lists, while *wrap*, *nil* and the primitive function *app* correspond to the join list representation found in Section 3.4.

Since constructor functions are exceptions to the surjectivity rule, lone constructors must be combined with other functions by the 'junc' combinator $\triangledown$, which dispatches to one of two functions according to the result of matching on a sum. When one of the operands of $\triangledown$ is surjective, or the two operands cover both constructors of a two-variant datatype, the result is surjective. For example, $nil \ \triangledown \ cons$ and $nil \ \triangledown \ id$ are both surjective, but $cons \ \triangledown \ snoc$ is not. Since $\triangledown$ can be nested, this result extends to datatypes with more than two constructors. Constructor functions can be composed with other functions as well, using the standard function composition combinator $\circ$, but only to the left: once a non-surjective function appears in a chain of compositions other than in the leftmost position, it is difficult to analyse the exact range of the composition, and the check for surjectivity ceases to be syntactic.

Other than the two already mentioned combinators, $\times$ is the cartesian product of two functions, and *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. Fold is the only combinator in RINV that is recursive. 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.

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

Theorem 4 (Right invertibility). *Given a function f in* RINV*, for finitely defined input x,* $(\llbracket f \rrbracket \circ \llbracket f \rrbracket^{\circ})\ x \equiv x$.

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

4.1 The Primitive Functions

The function *id* is the identity; functions *assocr*, *assocl* and *swap* manipulate pairs.

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

Together with the combinators $\times$ and $\circ$, these are sufficient to define many interesting functions on pairs. For example,

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

Function *app* is the uncurried append function, which is not injective. The admission of non-injective functions is one of the most important distinctions between RINV and other invertible languages [31], allowing us to break away from the isomorphism restriction. There are many possible right inverses for *app*, of which we pick one:

$$\llbracket \mathit{app} \rrbracket^{\circ} = \lambda \mathit{xs} \rightarrow \mathit{splitAt}\ ((\mathit{length}\ \mathit{xs} + 1)\ `\mathit{div}`\ 2)\ \mathit{xs}$$

4.2 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,

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

Constructors *snoc* and *wrap* are not primitive on left-biased lists, but can be encoded:

$$\begin{aligned}[\![\mathit{snoc}]\!] &= \lambda(\mathit{xs}, x) \to \mathit{xs} +\!\!+ [x] \\ [\![\mathit{wrap}]\!] &= \lambda x \to [x]\end{aligned}$$

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

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

They are effectively partial 'guard' functions, succeeding when the input value matches the pattern. The right inverses of *snoc* and *wrap* are

$$\begin{aligned}[\![\mathit{snoc}]\!]^\circ\,[x] &= ([\,], x) \\ [\![\mathit{snoc}]\!]^\circ\,(x : \mathit{xs}) &= \mathbf{let}\ (\mathit{ys}, y) = [\![\mathit{snoc}]\!]^\circ\ \mathit{xs}\ \mathbf{in}\ (x : \mathit{ys}, y) \\ [\![\mathit{wrap}]\!]^\circ\,[x] &= x\end{aligned}$$

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

4.3 The Combinators

The combinators in RINV are familiar to functional programmers.

Composition, Sum and Product. Combinator $\circ$ sequentially composes two functions:

$$\begin{aligned}[\![f \circ g]\!] &= [\![f]\!] \circ [\![g]\!] \\ [\![f \circ g]\!]^\circ &= [\![g]\!]^\circ \circ [\![f]\!]^\circ\end{aligned}$$

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

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

$$(\times) :: (a \leftrightarrows b) \to (c \leftrightarrows d) \to ((a, c) \leftrightarrows (b, d))$$
$$[\![f \times g]\!] = \lambda(w, x) \to ([\![f]\!]\ w, [\![g]\!]\ x)$$
$$[\![f \times g]\!]^\circ = \lambda(y, z) \to ([\![f]\!]^\circ\ y, [\![g]\!]^\circ\ z)$$

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:

$$(\triangle) :: (a \leftrightarrows b) \to (a \leftrightarrows c) \to (a \leftrightarrows (b, c))$$
$$[\![f \triangle g]\!] = \lambda x \to ([\![f]\!]\ x, [\![g]\!]\ x)$$

However, in the backward direction, $[\![f]\!]^\circ\ x$ and $[\![g]\!]^\circ\ y$ would have to converge, which is difficult to enforce statically. Indeed, functions constructed with $\triangle$ are generally not surjective, and so do not have total right inverses; for this reason, we exclude $\triangle$ from RINV.

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

$$\textbf{data}\ \mathit{Sum}\ a\ b = \mathit{Inl}\ a \mid \mathit{Inr}\ b$$
$$(\triangledown) :: (a \leftrightarrows c) \to (b \leftrightarrows c) \to (\mathit{Sum}\ a\ b \leftrightarrows c)$$
$$[\![f \triangledown g]\!] = \lambda x \to \textbf{case}\ x\ \textbf{of}\ \{\, \mathit{Inl}\ a \to [\![f]\!]\ a\ ;\ \mathit{Inr}\ b \to [\![g]\!]\ b \,\}$$

In the backward direction, if both f and g are surjective, it doesn't matter which branch is chosen. However, the use of constructor functions deserves some attention, since they are not surjective in isolation. As a result, in the event that $[\![f]\!]^\circ$ fails on certain inputs, $[\![g]\!]^\circ$ should be applied. To model this failure handling, we lift functions in RINV into the *Maybe* monad (allowing an extra possibility for the return value), and handle a failure in the first function by invoking the second.

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

This shallow backtracking is sufficient because the guards of conditionals are only pattern matching outcomes, which are completely decided at each level. For brevity, 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 test is made relatively straightforward, since the only possible cause for $f \triangledown g$ not to be jointly surjective is that both f and g use constructor functions; in this case, it is clear that we need the complete set of constructors to satisfy the condition of joint surjectivity. We demonstrate this check with examples towards the end of this section.

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 the $\triangledown$ combinator as discussed above. Therefore, in RINV, we disallow compositions involving constructor functions on the right of a composition.

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

$$[\![fold\ f]\!] = fold_X\ [\![f]\!]$$
$$[\![fold\ f]\!]^\circ = unfold_X\ [\![f]\!]^\circ$$

The forward semantics of *fold f* is defined in terms of the standard $fold_X$ for a datatype X, and the backward semantics is defined by a corresponding $unfold_X$. In what follows, we call the f in *fold f* the 'body' of the fold. Note that *unfold* is not in RINV, but is used to define right inverses. In this paper, we overload *fold* and *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.

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:

$$\begin{aligned}
&fold_L :: (Sum\ ()\ (a, b) \to b) \to (List\ a \to b)\\
&fold_L\ f = \lambda xs \to \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\qquad [\,] \qquad\ \to f\ (Inl\ ())\\
&\qquad\qquad (x : xs) \to f\ (Inr\ (x, (fold_L\ f\ xs)))\\
&unfold_L :: (b \to Sum\ ()\ (a, b)) \to (b \to List\ a)\\
&unfold_L\ f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of}\ Inl\ () \qquad \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\qquad Inr\ (a, b) \to a : (unfold_L\ f\ b)
\end{aligned}$$

Another example is leaf-labelled binary trees. Note that the constructor *Fork* is uncurried to fit better into the RINV framework.

$$\begin{aligned}
&\mathbf{data}\ LTree\ a = Leaf\ a \mid Fork\ (LTree\ a, LTree\ a)\\
&fold_T :: (Sum\ a\ (b, b) \to b) \to LTree\ a \to b\\
&fold_T\ f = \lambda t \to \mathbf{case}\ t\ \mathbf{of}\\
&\quad Leaf\ a \qquad\ \to f\ (Inl\ a)\\
&\quad Fork\ (t_1, t_2) \to f\ (Inr\ (fold_T\ f\ t_1, fold_T\ f\ t_2))\\
&unfold_T :: (a \to Sum\ a\ (b, b)) \to b \to LTree\ a\\
&unfold_T\ f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of}\\
&\qquad\qquad Inl\ a \to Leaf\ a\\
&\qquad\qquad Inr\ (b_1, b_2) \to Fork\ (unfold_T\ f\ b_1, unfold_T\ f\ b_2)
\end{aligned}$$

We use *unfold* to construct the right inverse of *fold*. From [12], we have the following lemma.

Lemma 5. $fold\ [\![f]\!] \circ unfold\ [\![f]\!]^\circ \sqsubseteq id$.

Since both *fold* and *unfold* are case-exhaustive when their bodies are case-exhaustive, the only reason for not having an equality in the lemma above is

that *unfold* is potentially non-terminating: when a body does not split a seed into 'smaller' seeds, unfolding a seed creates an infinite structure. It is well known that a function constructed by *unfold* terminates if the seed transformation is *well-founded* (that is, there should be no infinite descending chain of seeds). Static termination checkers exist in the literature [20, 36] and are orthogonal to the discussion here.

4.4 Programming in RINV

With the knowledge of RINV, we are now ready to look into the kinds of function we can define with it.

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

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

Function $map_L\ f$ applies argument f uniformly to all the elements of a list, without modifying the list structure. Since *nil* and *cons* form a complete set of constructors for lists, we know they are jointly surjective.

Similarly, *map* on leaf-labeled trees, map_T, is defined as follows.

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

The function *reverse* on lists can be defined as a fold:

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

In 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 terminates on reaching an empty list, when an empty list is returned as the result. Function $[\![snoc]\!]^\circ$ extracts the last element in a list and adds it 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). Since *nil* and *snoc* form a complete set of constructors for lists, they are jointly surjective.

Function *reverse* is also used to construct the *apprev* function that reverses a list and appends it.

$$apprev :: ([a], [a]) \to [a]$$
$$apprev = app \circ (id \times reverse)$$

Function *apprev* reverses the second list before concatenating the two. For example, we have:

$$apprev\ ([1,2],[3,4,5,6,7]) = [1,2,7,6,5,4,3]$$

The companion $apprev^\circ$ function is

$$\begin{aligned}&apprev^\circ :: [a] \to ([a],[a])\\&apprev^\circ = [\![app \circ (id \times reverse)]\!]^\circ\end{aligned}$$

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

$$\begin{aligned}apprev\ (apprev^\circ\ ([1,2,7,6,5,4,3])) &\equiv apprev\ ([1,2,7,6],[3,4,5])\\&\equiv [1,2,7,6,5,4,3]\end{aligned}$$

On the other hand,

$$\begin{aligned}apprev^\circ\ (apprev\ ([1,2],[3,4,5,6,7])) &\equiv apprev^\circ\ ([1,2,7,6,5,4,3])\\&\equiv ([1,2,7,6],[3,4,5])\end{aligned}$$

It is clear from above that $apprev^\circ$ is not a left inverse of $apprev$, and it is not intended to be a term in the language RINV.

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

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

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

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

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

$$\begin{aligned}preOrd\ &= fold_B\ (nil \triangledown (cons \circ (id \times app)))\\postOrd &= fold_B\ (nil \triangledown (snoc \circ swap \circ (id \times app)))\end{aligned}$$

In the forward direction, $fold_B$ adds the node value at one end of the concatenation of the two subtrees' traversals. In the backward direction, a node value is extracted from the input list, and the rest of the list is divided and grown into individual trees.

As a final remark to readers familiar with pointfree programming in Haskell, the primitive function *app* can be defined as a fold:

$$app = uncurry\ (flip\ (foldr\ (:)))$$

which effectively partially applies *foldr* and awaits an input as the base case. This idiom of taking an extra argument to form the base case is difficult to realize when the fold body is constructed independently, as it is in RINV. For some cases, it even threatens the existence of right inverses as unfolds. For example, the following function in Haskell

$$
\begin{aligned}
&f :: a \to [\mathit{LTree}\ a] \to \mathit{LTree}\ a \\
&f\ a = \mathit{foldr}\ \mathit{Fork}\ (\mathit{Leaf}\ a)
\end{aligned}
$$

does not have a right inverse as an unfold. In RINV, we rule out definitions of this kind, and treat *app* as a primitive.

5 Discussion

5.1 The Dual Story

In this paper, we have picked the to function to be provided by ADT implementers; the design of RINV and the subsequent discussion of ADTs is based on this decision. However, this choice is by no means absolute. One can well imagine ADT implementers coming up with from functions first, and a left-invertible language generating the corresponding to functions; this would give the same invertibility property $\mathsf{to} \circ \mathsf{from} \equiv id$. But the implementer is now expected to prove a different promotion condition, $f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f$, adapted to involve only from. Nevertheless, the crucial round-trip law and fusion law that form the foundation of the translation and optimization are still derivable; for round-trip, we have

$$
\begin{aligned}
& \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \quad \{\, f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f \,\} \\
& \mathcal{G}\ \mathsf{to} \circ \mathcal{G}\ \mathsf{from} \circ f \\
\equiv & \quad \{\, \mathcal{G} \text{ is a hofunctor; } \mathsf{to} \circ \mathsf{from} \equiv id \,\} \\
& f
\end{aligned}
$$

and for fusion:

$$
\begin{aligned}
& \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \quad \{\, f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f \,\} \\
& \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ \mathcal{G}\ \mathsf{from} \circ f \\
\equiv & \quad \{\, \mathcal{G} \text{ is a hofunctor; } \mathsf{to} \circ \mathsf{from} \equiv id \,\} \\
& \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ f \\
\equiv & \quad \{\, f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f \,\} \\
& \mathcal{H}\ \mathsf{to} \circ g\_ \circ f\_ \circ \mathcal{F}\ \mathsf{from}
\end{aligned}
$$

5.2 Reasoning about Efficiency

A controversy in any design that embeds implicit computations into pattern matching is the datatype-like notation. We think this feature is positive, since it preserves the elegant syntax of pattern matching and offers backward compatibility, an important property for incremental refactoring. 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 models. This is certainly a valid concern. As we have seen in Section 3.3, such conversions only occur when primitive operations and pattern matching interact. If this occurs in a recursion, run-time complexity could be affected. However, it is clear that this inefficiency can be eliminated by not mixing primitive operations and pattern matching in recursions.

5.3 Nested and Overlapping Patterns

Two well-regarded features of pattern matching are the scalability with respect to nesting and the sharing between overlapping patterns. For example, consider a function that sums elements of a list pair-wise:

$$\begin{aligned} &\mathit{pairSum}\ \mathit{Nil} &&= \mathit{Nil} \\ &\mathit{pairSum}\ (\mathit{Cons}\ x\ \mathit{Nil}) &&= \mathit{Cons}\ x\ \mathit{Nil} \\ &\mathit{pairSum}\ (\mathit{Cons}\ x\ (\mathit{Cons}\ y\ \mathit{ys})) &&= \mathit{Cons}\ (x + y)\ (\mathit{pairSum}\ \mathit{ys}) \end{aligned}$$

Nested patterns allow simultaneous matching and variable binding to patterns below top level (such as y above), in contrast to the sequential checking of expressions as guards. There is often a degree of sharing between patterns; for example, input to *pairSum* that, when evaluated, fails to match the first pattern does not need to be evaluated again for subsequent clauses. This is even more important for pattern matching on models where non-constant computation (i.e., the `to` function) may be needed. Our proposal supports both features nicely: nested patterns are written exactly the same way as with datatypes; and execution of `to` functions is done prior to pattern matching and is shared among all the patterns.

5.4 RINV Expressiveness

In our system, the set of definable models 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 existence of the right inverses: valid model values are bounded by the actual range of the user-defined `to` function; invertibility is not guaranteed for model values outside this range.

Totality of `from` is certainly desirable if it is used for model conversion, since failures will not be observable through reasoning. 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 $\triangle$ which executes both of its input functions, and is defined as

$$(\triangle) :: (a \to b) \to (a \to c) \to a \to (b, c)$$
$$(f \triangle g) = \lambda x \to (f\ x, g\ x)$$

Since $f \triangle g$ is generally not surjective, it doesn't have a right inverse, despite the fact that we can easily guard against inconsistent input in the reverse direction as follows.

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

Definitions like the one above are known as *weak right inverses* [30].

Another 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, as determined by the splitting strategy of its body.

If a model value outside the range is constructed, the integrity of model level equational reasoning may be corrupted. On the other hand, it is valid to argue that the same invariant assumed for the original datatype prior to the refactoring applies to the model too. For example, consider a program that requires balanced binary trees. A to function that only produces balanced binary trees is safe if the invariant is correctly preserved in the original program. It remains an open question whether we should allow programmers to take some reasonable responsibilities, or should insist on enforcing control through the language.

6 Related Work

Efforts to combine data abstraction and pattern matching started two decades ago with Wadler's *views* proposal [43]; and it is still a hot research topic [8, 10, 11, 19, 21, 33–35, 39, 41].

Wadler's views 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:

$$\textbf{data } \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$
$$\textbf{view } \mathit{List}\ a = \mathit{Lin} \mid \mathit{Snoc}\ (\mathit{List}\ a)\ a$$
$$\mathsf{to}\ \mathit{Nil} = \mathit{Lin}$$
$$\mathsf{to}\ (\mathit{Cons}\ x\ \mathit{Nil}) = \mathit{Snoc}\ \mathit{Nil}\ x$$
$$\mathsf{to}\ (\mathit{Cons}\ x\ (\mathit{Snoc}\ xs\ y)) = \mathit{Snoc}\ (\mathit{Cons}\ x\ xs)\ y$$

$$
\begin{aligned}
&\mathsf{from}\ \mathit{Lin} &&= \mathit{Nil}\\
&\mathsf{from}\ (\mathit{Snoc}\ \mathit{Nil}\ x) &&= \mathit{Cons}\ x\ \mathit{Nil}\\
&\mathsf{from}\ (\mathit{Snoc}\ (\mathit{Cons}\ x\ xs)\ y) &&= \mathit{Cons}\ x\ (\mathit{Snoc}\ xs\ y)
\end{aligned}
$$

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 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 either the datatype or one of its views.

$$\mathit{last}\ (\mathit{Snoc}\ xs\ x) = x$$

$$
\begin{aligned}
&\mathit{rotLeft}\ (\mathit{Cons}\ x\ xs) &&= \mathit{Snoc}\ xs\ x\\
&\mathit{rotRight}\ (\mathit{Snoc}\ xs\ x) &&= \mathit{Cons}\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rev}\ \mathit{Nil} &&= \mathit{Lin}\\
&\mathit{rev}\ (\mathit{Cons}\ x\ xs) &&= \mathit{Snoc}\ (\mathit{rev}\ xs)\ x
\end{aligned}
$$

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.

Just as with our proposal, this semantics can be elaborated by a straightforward translation into ordinary Haskell. First of all, view declarations are translated into data declarations.

$$\mathbf{data}\ \mathit{Snoc}\ a = \mathit{Lin} \mid \mathit{Snoc}\ (\mathit{List}\ a)\ a$$

Note that the child of *Snoc* refers to the underlying datatype: view data is typically hybrid (in contrast to our approach). Now the only task is to insert the conversion functions at appropriate places in the program.

$$\mathit{last}\ xs = \mathbf{case}\ \mathsf{to}\ xs\ \mathbf{of}\ \mathit{Snoc}\ xs\ x \to x$$

$$
\begin{aligned}
&\mathit{rotLeft}\ xs &&= \mathbf{case}\ xs\ \mathbf{of}\ \mathit{Cons}\ x\ xs \to \mathsf{from}\ (\mathit{Snoc}\ xs\ x)\\
&\mathit{rotRight}\ xs &&= \mathbf{case}\ \mathsf{to}\ xs\ \mathbf{of}\ \mathit{Snoc}\ xs\ x \to \mathit{Cons}\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rev}\ xs = \mathbf{case}\ xs\ \mathbf{of}\\
&\quad \mathit{Nil} &&\to \mathsf{from}\ \mathit{Lin}\\
&\quad (\mathit{Cons}\ x\ xs) &&\to \mathsf{from}\ (\mathit{Snoc}\ (\mathit{rev}\ xs)\ x)
\end{aligned}
$$

In contrast to our approach, Wadler exposes both a datatype and its views to programmers. To support reasoning across the different representations, the conversion clauses are used as axioms.

For example, we can evaluate an expression:

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

or calculate with functions:

$$
\begin{array}{ll}
 & \mathit{rotRight}\ (\mathit{rotLeft}\ (\mathit{Cons}\ x\ xs)) \\
\equiv & \quad \{\ \mathit{rotLeft}\ \} \\
 & \mathit{rotRight}\ (\mathit{Snoc}\ xs\ x) \\
\equiv & \quad \{\ \mathit{rotRight}\ \} \\
 & \mathit{Cons}\ x\ xs
\end{array}
$$

However, this style of reasoning is limited in expressiveness. For example, there is no way to calculate with $\mathit{rotLeft} \circ \mathit{rotRight}$, because in Wadler's setting, inputs are always constructed in the underlying datatype: in $(\mathit{rotLeft} \circ \mathit{rotRight})\ (\mathit{Cons}\ x\ xs)$, there is no way of converting $\mathit{Cons}\ x\ xs$ to a *Snoc* view, which would allow the calculation of *rotRight* to proceed. (The claim in the original paper [43] that $\mathit{rotLeft} \circ \mathit{rotRight} \equiv \mathit{id}$ is provable is incorrect; what is actually provided is a proof that $\mathit{rotRight} \circ \mathit{rotLeft} \equiv \mathit{id}$.)

A perhaps more noticeable weakness of views is the use of user-defined conversions as axioms. It is expected for a view type to be isomorphic to a subset of its underlying datatype, and for the pair of conversions between the values of the two types to be each other's full inverses. This is certainly restrictive; and Wadler didn't suggest any way to enforce such an invertibility condition. As pointed out by Wadler himself [43], and followed up by several others [8,34], this assumption is risky, and may lead to nasty surprises that threaten soundness of reasoning.

Inspired by Wadler's proposal, our work ties up the loose ends of views by hiding the underlying datatype as an ADT, and using only the view (our model) for pattern matching. The implementations of primitive operations of the ADT can be proven correct through comparison against the constructive specification, at no additional cost to ADT implementers. The language RINV for defining conversions guarantees right invertibility, a weaker condition that lifts the isomorphism restriction on abstract representations and models. In contrast to views, our system does not cater for multiple views of the same ADT, because given no explicit axioms connecting them, it is difficult to reason across views.

'Safe' variants of views have been proposed before [8,34]. 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 *Snoc Lin* 1 become syntactically invalid. Instead, values are only constructed by 'smart constructors', as in *snoc lin* 1.

In this setting, equational reasoning has to be conducted on the source level with explict applications of to. A major motivation for such a design is to admit views and sources with conversion functions that do not satisfy the invertibility property. In another words, let *Constr* and *constr* be a constructor and its corresponding smart constructor; in general, we have $Constr\ x \not\equiv constr\ x$. This appears to hinder program comprehension, since the very purpose of the convention that the name of a smart constructor differs only by case from its 'dumb' analogue is to suggest the equivalence of the two.

More recently, language designers have started looking into more expressive pattern mechanisms. *Active patterns* [10, 35] and many of their variants [11, 19, 21, 35, 39, 41] go a step further, by embedding computational content into pattern constructions. All the above proposals either explicitly recognise 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 work focusses mainly on examples that are primarily data consumers, an escape that is expected to be limited and short-lived. Another common pitfall of active patterns is the difficulty in supporting nested and overlapping patterns, as discussed in Section 5.3, because each active pattern is computed and matched independently.

In particular, equational reasoning with ADTs is one of the central themes in two notable proposals [8, 35]. These proposals demonstrate the possibility of reasoning about programs containing safe views or active patterns, through the axiomatic specifications of ADTs. In particular, it is observed in [8] that an algebraic datatype called the 'associated free type' (model in our case) may serve as the interface of an ADT. Through the view mechanism, an associated free type can differ in structure from the abstract representation of the ADT. In contrast to ours, their proposal is not able to separate functions over an ADT into primitive and non-primitive, an essential feature for incremental refactoring; nor to recognize the value of right-invertibility of to as the key to sound reasoning with models.

The language RINV owes its origins to the rich literature on *invertible* programming [2, 32], a programming paradigm where programs can be executed both forwards and backwards. Mu *et al.* concentrate their effort on designing a language that provides only injective functions. The resulting language *Inv* is a combinator library that syntactically rules out any non-injective functions. The most novel operator of *Inv* is *dup* f, which duplicates the input and applies f to one copy. In the backward direction, the two copies of the duplicated input are checked for consistency before being restored. It is shown that *Inv* is practically useful for maintaining consistency of structured data related by some transformations [17, 31]. Invertible arrows [2] extend the arrow framework [18] (a generalization of monads) with a combinator that encodes pairs of functions being each other's inverses. In an aside in [2], it is recognized that when full in-

vertibility is not achievable (due to the non-isomorphic nature of the two sides), biased (either left- or right-) invertibility is nevertheless approximation.

Right inverses have been studied as a component of the much more elaborate bidirectional programming framework of *lenses* [6,13–15] targeting view-updates of XML databases; in this context, right inverses are known as 'create' functions. Based on record types, the combinators of lenses have little similarity to those of RINV. A distinctive feature of the lenses framework is the use of *semantic* types [16] to give precise bounds to the ranges of forward functions (thus the domains of backward functions). As a result, surjectivity now concerns the relationships of domains/ranges of lenses connected by a combinator, instead of being a property between a function and its target datatype.

7 Conclusion

Algebraic datatypes and pattern matching offer great promise to programmers seeking simple and elegant programming, but the promise turns sour when modular changes are demanded. Our work tackles this long-standing problem by proposing a framework for refactoring programs written with pattern matching into ones with ADTs: programmers are able to selectively reimplement original function definitions into primitive operations of the ADT, and either rewrite the rest in terms of the primitive ones, or simply leave them unchanged. This migration is completely incremental: executability and proofs through equational reasoning are preserved at all times during the process.

At the heart of our proposal is a novel design of ADTs enriched with algebraic models for backwards-compatible pattern matching. The model has the same interface as the datatype that is being replaced; and the original definitions of the selected primitive operations are turned into constructive specifications, through which equational reasoning connects the primitive operations and the rest of the functions. The soundness of such reasoning is established by the right-inverse property of the conversion pairs that bridge the model and the abstract representation.

We have implemented the language RINV as a combinator library in Haskell, and a naïve translation of ADTs into Haskell immediately follows. However, it remains to investigate how the fusion theory developed in the paper can be applied in practice. There is literature on fusing embedding-projection pairs (conceptually similar to our to and from functions) [1,25]. However the treatment of recursive functions using a fixpoint combinator in [1] is not applicable directly, because the 'deep' embedding in our approach (not producing hybrid data) does not allow conversions to be isolated into a non-recursive part. Instead, we plan to employ an analysis of mixed uses of patterns and primitive operations; and use strength reduction to remove the conversions, as outlined in Section 3.3.

At this stage, our focus is on supporting refactoring of programs written with datatypes and pattern matching, which automatically excludes some ADTs, such as unordered sets, that cannot be fully modelled by algebraic datatypes. We leave

it as future work to investigate the applicability of our proposal in a more general setting.

Acknowledgements

We are grateful to Ralf Hinze for his valuable comments on an early draft of the paper; the binary number example is due to him. This work is supported by the EPSRC grant *Generic and Indexed Programming* (EP/E02128X), and was partly conducted during the first author's internship at National Institute of Informatics, Japan. Part of the work of the third author was done while the author was at University of Tokyo as JSPS Research Fellow supported by Grant-in-Aid for JSPS Fellows $20 \cdot 9584$.

References

1. A. Alimarine and S. Smetsers. Optimizing generic functions. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.
2. A. Alimarine, S. Smetsers, A. van Weelden, M. van Eekelen, and R. Plasmeijer. There and back again: Arrows for invertible programming. In *Haskell Workshop*, pages 86–97, New York, NY, USA, 2005. ACM.
3. R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.
4. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
5. R. S. Bird. A calculus of functions for program derivation. In *Research Topics in Functional Programming*, pages 287–307. Addison-Wesley, 1990.
6. A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages*, New York, NY, USA, Jan. 2008. ACM.
7. R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. In *Lisp and Functional Programming*, pages 136–143. ACM, 1980.
8. F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.
9. B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*. Springer, 2007.
10. M. Erwig. Active patterns. In *8th Int. Workshop on Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 1996.
11. M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, New York, NY, USA, 2000. ACM.
12. M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, Netherlands, Jan. 1991.

13. 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 Transactions on Programming Languages and Systems*, 29(3), May 2007. Preliminary version in POPL '05.
14. J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 60–74, Washington, DC, USA, 2009. IEEE Computer Society.
15. J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *International Conference on Functional Programming*, pages 383–396, New York, NY, USA, 2008. ACM.
16. A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), 2008.
17. Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Partial Evaluation and Program Manipulation*, pages 178–189, New York, NY, USA, 2004. ACM.
18. J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.
19. C. B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26(6), 2004.
20. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Principles of Programming Languages*, pages 81–92, New York, NY, USA, 2001. ACM.
21. D. Licata and S. Peyton Jones. View patterns: lightweight views for Haskell. `http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns`, 2007.
22. B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA, USA, 2000.
23. B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.
24. J. Liu and A. C. Myers. JMatch: Iterable abstract pattern matching for Java. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, London, UK, 2003. Springer.
25. J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 33–42, New York, NY, USA, 2010. ACM.
26. C. Martin, J. Gibbons, and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004.
27. L. G. L. T. Meertens. Algorithmics: Towards programming as a mathematical activity. In *CWI Symposium on Mathematics and Computer Science*, number 1 in CWI-Monographs, pages 289–344. North–Holland, 1986.
28. P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *12th Conference on Compiler Construction, Warsaw (Poland), volume 2622 of LNCS*, pages 61–76. Springer, 2003.
29. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.
30. K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–155, New York, NY, USA, 2007. ACM.

31. S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2004.
32. S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 289–313. Springer, 2004.
33. P. Nogueira and J. J. Moreno-Navarro. Bialgebra views: A way for polytypic programming to cohabit with data abstraction. In *Workshop on Generic Programming*, pages 61–73, New York, NY, USA, 2008. ACM.
34. C. Okasaki. Views for Standard ML. In *ACM Workshop on ML*, 1998.
35. P. Palao Gostanza, R. Peña, and M. Núñez. A new look at pattern matching in abstract data types. In *International Conference on Functional Programming*, pages 110–121, New York, NY, USA, 1996. ACM.
36. D. Sereni. Termination analysis and call graph construction for higher-order functional programs. In N. Ramsey, editor, *International Conference on Functional Programming*, pages 71–84. ACM Press, 2007.
37. M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software: Practice and Experience*, 12(11), 1982.
38. G. L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *International Conference on Functional Programming*, pages 1–2, New York, NY, USA, 2009. ACM.
39. D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *International Conference on Functional Programming*, pages 29–40, New York, NY, USA, 2007. ACM.
40. S. Thompson. Lawful functions and program verification in Miranda. *Science of Computer Programming*, 13(2-3):181–218, 1990.
41. M. Tullsen. First class patterns. In *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*. Springer, 2000.
42. P. Wadler. A critique of Abelson and Sussman: Why calculating is better than scheming. *ACM SIGPLAN Notices*, 22(3):83–94, 1987.
43. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313, New York, NY, USA, 1987. ACM.