Refactoring Pattern Matching

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

[a] *Computer Science and Engineering, Chalmers University of Technology*
412 96 Göteborg, Sweden
[b] *Department of Computer Science, Oxford University*
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
[c] *Graduate School of Information Sciences, Tohoku University*
Aramaki aza Aoba 6-3-09, Aoba-ku, Sendai-city, Miyagi-pref. 980-8579, Japan
[d] *GRACE Center, National Institute of Informatics*
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract

Defining functions by pattern matching over the arguments is advantageous for understanding and reasoning, but it tends to expose the 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 program refactoring 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, while maintaining simple and sound reasoning.

Keywords: functional programming, refactoring, pattern matching, invertible programming, abstract datatypes, fusion

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, nor 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, program to that interface, and allow the implementation of the interface 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 [**?**].

1.2. Pattern Matching

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

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

The above declarations introduce a new datatype (*Bin*) for binary bits, and define *Num* as a type synonym for lists of bits. Throughout this paper, we will use the syntax and standard prelude functions of Haskell [**?**] for illustration, although any language providing algebraic datatypes would work just as well; however, in contrast to Haskell, we assume a semantics based on sets and total functions rather than on complete partial orders.

Consider this function for normalizing binary numbers by eliding leading zeroes, defined by pattern matching.

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

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

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

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

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

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

$$
\begin{array}{l}
\mathit{normalize} :: \mathit{Num} \to \mathit{Num} \\
\mathit{normalize}\ \mathit{num} = \textbf{if}\ \mathit{null}\ \mathit{num} \vee \mathit{one}\ (\mathit{head}\ \mathit{num})\ \textbf{then} \\
\qquad\qquad\qquad\qquad \mathit{num} \\
\qquad\qquad\qquad\ \ \textbf{else} \\
\qquad\qquad\qquad\qquad \mathit{normalize}\ (\mathit{tail}\ \mathit{num})
\end{array}
$$

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 [**?**]. More recently, it has started gaining recognition from the object-oriented community 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 use 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*" [**?**].

In this stand-off between clarity and abstraction, functional languages usually lean towards the former while object-oriented languages prefer the latter. Can we hope to achieve the best of both worlds? With current technology, encapsulating datatypes in a functional language hinders pattern matching. Ad hoc approaches such as allowing user-defined computations to be embedded in the pattern matching and data construction processes threaten soundness of reasoning. For example, we rely on reasoning to perform known-case elimination that reduces expression **case** (*Cons x xs*) **of** (*Cons a as*) $\rightarrow$ *a* to *x* based on the knowledge that the computation constructing (*Cons x xs*) is reversed by matching the pattern *Cons a as*. If user-defined computations are embedded in the pattern matching process such as in Wadler's *view* proposal [**?**], we no longer have the guarantee that the above mentioned round-trip property still holds, which invalidates reasoning of this kind. To recover soundness, a more disciplined framework is needed.

In an object-oriented setting, there is a concept of *typical object* [**?**], which is a simple and general data structure (often expressed in a way similar to an algebraic datatype) used as a model of the underlying more complicated and user-defined structure. For example, various kinds of linear structure can be modelled as a list datatype. The user-defined structures are used for execution while the typical object caters for specification. The function that maps an underlying structure to its typical object is called an *abstraction function*. An implementation using the underlying structure can be proven correct, with respect to its specification in terms of the typical object [**?**], by reasoning about the abstraction function. However, reasoning in this manner does not help with incrementally refactoring a program already written with bare algebraic datatypes.

1.3. Our Contribution

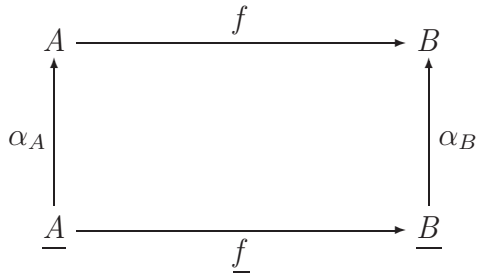
In this work, then, we strive to address the tension between the convenience of pattern matching and the flexibility of encapsulated data abstractions by proposing a mechanism to allow programs written with pattern matching to be refactored smoothly and incrementally into ones with encapsulation, without losing the benefits of simple equational reasoning. In particular, we propose a framework for incrementally refactoring and identify necessary and sufficient conditions for correctness of such refactoring. As part of the framework, we sketch a domain-specific language RINV that helps to guarantee these conditions by construction. We have implemented the

proposed system to demonstrate its feasibility. A prototype of our proposal can be found online [**?**] and is described in detail in the appendix.

The rest of the paper is structured as follows. Section **??** gives a brief introduction to data abstraction. Section **??** presents our proposed design for incremental refactoring of program with pattern matching, and Section **??** 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 **??**), and apply our results to other problems in addition to refactoring (Section **??**), before discussing related work (Section **??**) and concluding (Section **??**).

2. Data Abstraction

Choosing the right data structure is key to achieving an efficient program; data abstraction allows us to defer the choice of representation until after the uses of the data are fully understood. The idea is to firstly program with 'abstract' data, which is then replaced by a more efficient 'concrete' representation. Operations on abstract data are reimplemented on the concrete representation, with the original abstract operations serving as specifications. Milner [**?**] observed that the transition from abstract to concrete representation can be proven correct by showing the following square commutes,



This proof obligation is also known as the *promotion* condition [**?**]

$$\alpha_B \circ \underline{f} \equiv f \circ \alpha_A$$

Throughout this paper, we use $\equiv$ for equality between expressions, whereas we use $=$ for definitions. As a notational convention, we write $\underline{X}$ for the concrete counterpart of an abstract representation X. In the above, the α_X

family of functions are *abstraction* functions [**?**] that map the concrete representations to the abstract ones; we will simply write α if its type is clear from the context. In the above diagram, A and B are abstract representations, whereas $\underline{A}$ and $\underline{B}$ are concrete representations. The abstract program is $f : A \to B$, and its concrete version is $\underline{f} : \underline{A} \to \underline{B}$. In the sequel, we use the terms 'abstract program' and 'specification' interchangeably.

2.1. An Example

Consider queue structures, represented abstractly as lists.

$$\begin{aligned}
&\textbf{type } \mathit{Queue}\ a = [a] \\
&\mathit{emptyQ} = [\,] \\
&\mathit{first} \quad = \mathit{head} \\
&\mathit{isEmpty} = \mathit{null} \\
&\mathit{enQ}\ a\ q = q +\!\!+ [a] \\
&\mathit{deQ} \quad = \mathit{tail}
\end{aligned}$$

(Note that $+\!\!+$ is a Haskell infix operator that concatenates two lists.) For a more efficient definition of enQ, a plausible concrete representation reads:

$$\begin{aligned}
&\textbf{type } \underline{\mathit{Queue}}\ a = ([a], [a]) \\
&\alpha :: \underline{\mathit{Queue}}\ a \to \mathit{Queue}\ a \\
&\alpha\ (\mathit{fq}, \mathit{bq}) = \mathit{fq} +\!\!+ \mathit{reverse}\ \mathit{bq}
\end{aligned}$$

The second list of the pair, representing the latter part of a queue, is reversed, so that enqueuing is simply list prefixing. The library operations can be implemented as follows:

$$\begin{aligned}
&\underline{\mathit{emptyQ}} && = ([\,], [\,]) \\
&\underline{\mathit{first}}\ ([\,], \mathit{bq}) && = \mathit{last}\ \mathit{bq} \\
&\underline{\mathit{first}}\ ((a : \mathit{fq}), \mathit{bq}) && = a \\
&\underline{\mathit{isEmpty}}\ ([\,], [\,]) && = \mathit{True} \\
&\underline{\mathit{isEmpty}}\ q && = \mathit{False} \\
&\underline{\mathit{enQ}}\ a\ (\mathit{fq}, \mathit{bq}) && = (\mathit{fq}, a : \mathit{bq}) \\
&\underline{\mathit{deQ}}\ ([\,], \mathit{bq})) && = \underline{\mathit{deQ}}\ (\mathit{reverse}\ \mathit{bq}, [\,]) \\
&\underline{\mathit{deQ}}\ (a : \mathit{fq}, \mathit{bq}) && = (\mathit{fq}, \mathit{bq})
\end{aligned}$$

Through standard equational reasoning, we can establish the correctness of the implementation by proving the promotion condition for each operation. For example,

$$
\begin{array}{ll}
 & \alpha\ (\underline{deQ}\ (a : fq, bq)) \\
\equiv & \quad \{\, \text{definition of } \underline{deQ} \,\} \\
 & \alpha\ (fq, bq) \\
\equiv & \quad \{\, \text{definition of } deQ \,\} \\
 & deQ\ (a : \alpha\ (fq, bq)) \\
\equiv & \quad \{\, \text{definition of } \alpha \,\} \\
 & deQ\ (a : (fq +\!\!+ reverse\ bq)) \\
\equiv & \quad \{\, \text{definition of } +\!\!+ \,\} \\
 & deQ\ ((a : fq) +\!\!+ reverse\ bq) \\
\equiv & \quad \{\, \text{definition of } \alpha \,\} \\
 & deQ\ (\alpha\ (a : fq, bq))
\end{array}
$$

Other than the handful of 'library' operations above, we have other abstract programs benefitting from the simple list representation. For example, the map function on queues:

$$
\begin{array}{l}
mapQ :: (a \to b) \to Queue\ a \to Queue\ b \\
mapQ\ f\ [\,] \quad\ = [\,] \\
mapQ\ f\ (a : q) = f\ a : mapQ\ f\ q
\end{array}
$$

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

$$
\begin{array}{l}
prioritise :: Ord\ a \Rightarrow Queue\ a \to Queue\ a \\
prioritise\ [\,] \quad\ = [\,] \\
prioritise\ (a : q) = insert\ a\ (prioritise\ q) \\
\quad \textbf{where}\ insert\ b\ [\,] \quad\ = [b] \\
\qquad\qquad insert\ b\ (a : q) = \textbf{if}\ b \leqslant a\ \textbf{then}\ b : (a : q) \\
\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\qquad\quad a : (insert\ b\ q)
\end{array}
$$

To maintain executability, all uses of the abstract representation have to be changed at once, even though some of the old definitions may not gain from the refactoring. One has to (re)implement all the functions either by pattern matching on the new representation (discouraging further refactoring) or by the use of library operations (at the cost of losing convenient equational

reasoning). For example, a definition of map using only the library operations is likely to be more clumsy:

$$
\begin{array}{l}
mapQ_{\text{prim}} :: (a \to b) \to Queue\ a \to Queue\ b \\
mapQ_{\text{prim}}\ f\ q = accum\ f\ q\ emptyQ \\
\quad \textbf{where}\ accum\ f\ q\ aq = \textbf{if}\ isEmpty\ q\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad aq \\
\qquad\qquad\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad accum\ f\ (deQ\ q)\ (enQ\ (f\ (first\ q))\ aq)
\end{array}
$$

Being restricted to an explicit interface reduced our options here: it is not easy to add to the front of a queue, for which we need an accumulating parameter.

In the next 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. Selective Refactoring

Our purpose is to allow incremental refactoring of a program, replacing a specification in stages by a more sophisticated implementation. We have seen in the previous section that once an implementation is given, the promotion condition is sufficient to guarantee its correctness. However, the promotion condition does not suggest a way of integrating the new definition into its original context, which may still operate on the old representation. We need a computation law of the form

$$f \ \equiv\ \alpha \circ \underline{f} \circ \alpha^{\circ}$$

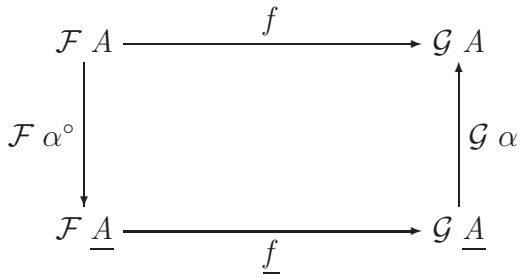
in order to replace specification f with its implementation $\alpha \circ \underline{f} \circ \alpha^{\circ}$.

Here, the function α° does the 'opposite' of α, mapping an abstract value down to a concrete value. The abstract program f is replaced by the composition $\alpha \circ \underline{f} \circ \alpha^{\circ}$, which converts an abstract value to the concrete representation, executes the concrete program, then converts the concrete result back to the abstract representation. For example, the abstract program fragment $deQ\ x$ is replaced with $(\alpha \circ \underline{deQ} \circ \alpha^{\circ})\ x$, where the two functions α° and α convert a value of type $Queue$ into one of type $\underline{Queue}$ and back again.

It is obvious that if α and α° are each other's inverses (a situation sometimes called *strong simulation* [**?**]), the computation law is equivalent to the promotion condition. However, requiring them to be inverses restricts the abstraction function to be bijective, which is rather a strong condition. In fact, one-sided invertibility is sufficient in this case. Specifically, the α° function should be a right inverse of α – that is, $\alpha \circ \alpha^\circ \equiv id$; it is not necessary for α° also to be a left inverse of α.

3.1. The Computation Law

Before going into the details of our results, we firstly generalise the notation used. The abstract operation f will not always have a type as simple as $Queue\ a \to Queue\ a$, like deQ does. Suppose we have datatypes A and $\underline{A}$, and conversion functions $\alpha :: \underline{A} \to A$ and $\alpha^\circ :: A \to \underline{A}$. In the general case, an abstract operation will take not just a single value in the abstract representation (of type A), but some combination of abstract values and other arguments. We capture this combination in terms of an operation $\mathcal{F}$ on datatypes A. Similarly, the operation will return a different combination $\mathcal{G}$ of abstract values and other results. The operations $\mathcal{F}$ and $\mathcal{G}$ are functors; they lift the conversion functions in the obvious way to $\mathcal{F}\ \alpha^\circ :: \mathcal{F}\ A \to \mathcal{F}\ \underline{A}$ and $\mathcal{G}\ \alpha :: \mathcal{G}\ \underline{A} \to \mathcal{G}\ A$, acting pointwise on combinations of values, and respecting identity and composition. Then an abstract operation f will have type $\mathcal{F}\ A \to \mathcal{G}\ A$, and the corresponding concrete operation $\underline{f} :: \mathcal{F}\ \underline{A} \to \mathcal{G}\ \underline{A}$ should satisfy the computation law $f \equiv \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ$, as shown in the following commuting diagram.



For example, for the operation $first :: Queue\ a \to a$, the input context $\mathcal{F}$ is the identity functor, because the source type $Queue\ a$ of *first* consists of a single occurrence of the datatype in question, and the output context $\mathcal{G}$ is a constant functor, because the target type a of *first* has no occurrences of the datatype. The operation must satisfy the following computation law:

$$\mathit{first} \equiv \underline{\mathit{first}} \circ \alpha^\circ$$

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

$$\begin{array}{lcl} \mathit{isEmpty} & \equiv & \underline{\mathit{isEmpty}} \circ \alpha^\circ \\ \mathit{emptyQ} & \equiv & \alpha \circ \underline{\mathit{emptyQ}} \\ \mathit{enQ}\ a & \equiv & \alpha \circ \underline{\mathit{enQ}}\ a \circ \alpha^\circ \\ \mathit{deQ} & \equiv & \alpha \circ \underline{\mathit{deQ}} \circ \alpha^\circ \end{array}$$

The computation law provides a way of replacing a specification with its implementation. But since it involves both α and α°, it seems more difficult to prove than the promotion condition. However, the following result allows us to derive the computation law from the promotion condition, using the right-invertibility property of α and α°.

Lemma 1 (Computation). *Given operation specifications $f :: \mathcal{F}\ A \to \mathcal{G}\ A$, and their implementations $\underline{f} :: \mathcal{F}\ \underline{A} \to \mathcal{G}\ \underline{A}$, for all conversion functions $\alpha :: \underline{A} \to A$ satisfying the promotion condition $\mathcal{G}\ \alpha \circ \underline{f} \equiv f \circ \mathcal{F}\ \alpha$, and their right inverses $\alpha^\circ :: A \to \underline{A}$, we can deduce the computation law $f \equiv \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ$.*

Proof.

$$\begin{array}{ll} & \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \\ \equiv & \quad \{\, \text{promotion} \,\} \\ & f \circ \mathcal{F}\ \alpha \circ \mathcal{F}\ \alpha^\circ \\ \equiv & \quad \{\, \mathcal{F} \text{ respects composition} \,\} \\ & f \circ \mathcal{F}\ (\alpha \circ \alpha^\circ) \\ \equiv & \quad \{\, \alpha \circ \alpha^\circ \equiv \mathit{id} \,\} \\ & f \circ \mathcal{F}\ \mathit{id} \\ \equiv & \quad \{\, \mathcal{F} \text{ respects identity} \,\} \\ & f \end{array}$$

□

Applying the computation law requires additional infrastructure (the right inverse α° of the abstraction function and its properties) in addition to that needed for a data abstraction framework based on the promotion condition. To minimise this extra obligation, we will explore a correctness-by-construction technique: in the next section, we will present a combinator-based language RINV implemented as a library, in which every definable function gets a right inverse for free. That is to say, the programmer carrying

out a refactoring writes only α, in RINV, and the corresponding α° is automatically generated. For the sake of completeness, in the case of the queue example presented above, a possible definition in RINV reads:

$$\alpha = app \circ (id \times reverse)$$

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

3.2. Refactoring by Translation

In our proposal, a programmer wishing to migrate to the concrete representation has the choice of keeping the original abstract definitions, or of refactoring them into the style using only the library operations, or of having a mixture of the two. For example, consider a queue that is read circularly for a certain amount of time, say repeatedly playing a piece of music.

$$\begin{aligned}
&play :: Time \to Queue\ (IO\ ()) \to IO\ () \\
&play\ 0 \qquad (a{:}\ _) \ = a \\
&play\ (n+1)\ (a : q) = \mathbf{do}\ a \\
&\qquad\qquad\qquad\qquad play\ n\ (q +\!\!+ [a])
\end{aligned}$$

We use Haskell's **do** notation above to sequence IO actions: the action a will be followed by the actions of $play\ n\ (q +\!\!+ [a])$. To refactor this function, we firstly have to rewrite it with library operations.

$$\begin{aligned}
&play_{\text{prim}} :: Time \to Queue\ (IO\ ()) \to IO\ () \\
&play_{\text{prim}}\ 0 \qquad q = first\ q \\
&play_{\text{prim}}\ (n+1)\ q = \mathbf{do}\ hd \\
&\qquad\qquad\qquad\qquad play_{\text{prim}}\ n\ (enQ\ hd\ tl) \\
&\quad \mathbf{where}\ hd = first\ q \\
&\qquad\quad\ \ tl\ \ = deQ\ q
\end{aligned}$$

There is no magic here. The programmer carrying out the refactoring has to write $play_{\text{prim}}$ and be responsible for its correctness: $play_{\text{prim}} \equiv play$.

The library operations are implemented using the concrete representation, allowing constant time performance of $\underline{enQ}$.

$$\begin{aligned}
&\underline{play_{\text{prim}}} :: Time \to \underline{Queue}\ (IO\ ()) \to IO\ () \\
&\underline{play_{\text{prim}}}\ 0\ q \qquad = \underline{first}\ q
\end{aligned}$$

$$
\begin{aligned}
&\underline{play_{\text{prim}}}\ (n+1)\ q = \mathbf{do}\ hd \\
&\qquad\qquad\qquad\qquad \underline{play_{\text{prim}}}\ n\ (\underline{enQ}\ hd\ tl) \\
&\quad \mathbf{where}\ hd = \underline{first}\ q \\
&\qquad\qquad\ tl\ \ = \underline{deQ}\ q
\end{aligned}
$$

Re-implementing every function to its "prim" version is certainly laborious, and is not necessarily beneficial. We perform selective refactoring, requiring $\underline{play_{\text{prim}}}$ to do the job of *play* in the original abstract context. For example, we may want to use $\underline{play_{\text{prim}}}$ together with *mapQ* (defined in Section **??**) in expressions like:

$$\underline{play_{\text{prim}}}\ n \circ mapQ\ f$$

The gap between different representations used by $\underline{play_{\text{prim}}}$ and *mapQ* can be bridged by a mechanical translation, following a rather straightforward scheme: each use of a library operation is replaced with its implementation precomposed with α and postcomposed with α° (subject to the appropriate functors).

The library operations that are defined on the concrete implementation require their inputs to be converted from the abstract representation before consumption, and the outputs converted back to the abstract representation. Effectively, all the refactored functions have the abstract representation as input and output types; the concrete representation remains only for intermediate structures. As an example, $\underline{play_{\text{prim}}}$ can be translated into the following.

$$
\begin{aligned}
&play' :: Time \to Queue\ (IO\ ()) \to IO\ () \\
&play'\ 0\ q \qquad\quad = (\alpha \circ \underline{first} \circ \alpha^\circ)\ q \\
&play'\ (n+1)\ q = \mathbf{do}\ hd \\
&\qquad\qquad\qquad\qquad play'\ n\ ((\alpha \circ \underline{enQ}\ hd \circ \alpha^\circ)\ tl) \\
&\quad \mathbf{where}\ hd = (\alpha \circ \underline{first} \circ \alpha^\circ)\ q \\
&\qquad\qquad\ tl\ \ = (\alpha \circ \underline{deQ} \circ \alpha^\circ)\ q
\end{aligned}
$$

Given the computation law, it is easy to conclude that $play'$ is equivalent to $play_{\text{prim}}$, in the sense that exactly the same output is produced for each input.

The original abstract program such as the definition of *play* is turned into a specification and can be used for equational reasoning. For example, one can continue to show

$$play\ n\ xs \equiv play\ n\ (xs +\!\!+ xs)$$

on the abstract level, without worrying about the refactoring of *play* into *play′*.

3.3. Optimization

The above translation of $\underline{play}_{\text{prim}}$ is semantically correct, but inefficient. There are conversions everywhere in the program, which are unnecessary because there is no reference to the abstract representation by the function at all. One way to remove the redundant conversions is fusion. Consider the expression

$$(\alpha \circ \underline{enQ}\ hd \circ \alpha^\circ)\ ((\alpha \circ \underline{deQ} \circ \alpha^\circ)\ q)$$

Our target is to fuse the intermediate conversions to produce

$$(\alpha \circ \underline{enQ}\ hd \circ \underline{deQ} \circ \alpha^\circ)\ q$$

This would clearly follow from $\alpha^\circ \circ \alpha \equiv id$, but this is not a property that we guarantee—for good reason, since requiring it in addition to the existing right inverse property $\alpha \circ \alpha^\circ \equiv id$ entails isomorphic abstract and concrete representations, 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 1 (Fusion Soundness). *Given operation specifications* $f :: \mathcal{F}\ A \to \mathcal{G}\ A$ *and* $g :: \mathcal{G}\ A \to \mathcal{H}\ A$*, and their implementations* $\underline{f} :: \mathcal{F}\ \underline{A} \to \mathcal{G}\ \underline{A}$ *and* $\underline{g} :: \mathcal{G}\ \underline{A} \to \mathcal{H}\ \underline{A}$*, for all conversion functions* $\alpha :: \underline{A} \to A$ *satisfying the promotion condition* $\mathcal{H}\ \alpha \circ \underline{g} \equiv g \circ \mathcal{G}\ \alpha$*, and their right inverses* $\alpha^\circ :: A \to \underline{A}$*, we have the following fusion law:*

$$\mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \alpha^\circ \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \equiv \mathcal{H}\ \alpha \circ \underline{g} \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ$$

Proof.

$$\begin{array}{ll} & \mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \alpha^\circ \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \\ \equiv & \quad \{\,\text{promotion}\,\} \\ & g \circ \mathcal{G}\ \alpha \circ \mathcal{G}\ \alpha^\circ \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \\ \equiv & \quad \{\,\mathcal{G} \text{ is a functor; } \alpha \circ \alpha^\circ \equiv id\,\} \end{array}$$

$$
\begin{array}{ll}
 & g \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \\
\equiv & \quad \{\,\text{promotion}\,\} \\
 & \mathcal{H}\ \alpha \circ \underline{g} \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ
\end{array}
$$

□

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

It is clear theoretically that repeatedly applying the above fusion law will eliminate all intermediate conversions until the point where the abstract representation is used. The absence of abstract representation is reflected in typing: for example the function $\underline{play_{\text{prim}}}$ is well-typed without translation, as we never have to unify the types $Queue$ and $\underline{Queue}$. In this case, $\underline{play_{\text{prim}}}$ is no different from library functions such as $\underline{deQ}$ translation-wise, where we lift the conversion out of the recursion. As a result, our implementation directly produces

$$play'\ n = \alpha \circ \underline{play_{\text{prim}}}\ n \circ \alpha^\circ$$

which is free from intermediate conversions.

It is also clear that not all conversions can be eliminated. In expressions like

$$\alpha \circ \underline{play_{\text{prim}}}\ n \circ \alpha^\circ \circ mapQ\ f$$

some performance overhead is inevitable. Such overheads may or may not be acceptable, depending on the situation. The programmer has the option of refactoring $mapQ$ as well to remove the conversion. We won't discuss this engineering trade-off further: although we advocate selective refactoring, we are not necessarily against complete refactoring; we simply provide the options for programmers to choose from.

3.4. More Examples

We now look at a few additional examples making use of the refactoring framework developed in this section.

3.4.1. Join Lists

Suppose we have the following definition of the reverse function:

$$\begin{aligned}&\mathit{reverse} :: [a] \to [a]\\ &\mathit{reverse}\ [\,] \quad\ \ = [\,]\\ &\mathit{reverse}\ (x : xs) = \mathit{append}\ (\mathit{reverse}\ xs)\ [x]\end{aligned}$$

It is well known that the above definition suffers from poor run-time performance due to the linear-time left-biased-list concatenation (*append*). One way to resolve such inefficiency is to transform individual concatenation-intensive definitions to reduce the calls to *append*, a technique that is systematised by Wadler [**?**]. Here we take a different approach by refactoring *append* itself based on a different representation of list, instead of trying to eliminate its usage.

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

$$\textbf{data}\ \underline{\mathit{List}}\ a = \mathit{Empty} \mid \mathit{Unit}\ a \mid \mathit{Join}\ (\underline{\mathit{List}}\ a)\ (\underline{\mathit{List}}\ a)$$

With this representation, a constant-time append function can be defined as

$$\underline{\mathit{append}}\ \mathit{l1}\ \mathit{l2} = \mathit{Join}\ \mathit{l1}\ \mathit{l2}$$

At the same time, we don't want to give up on the familiar notions of [ ] and (:). Instead, they can serve as a specification of the join representation.

$$\begin{aligned}&\textbf{type}\ \mathit{List}\ a = [a]\\ &\mathit{append}\ [\,] \quad\ \ y\ \ = \mathit{ys}\\ &\mathit{append}\ (x : xs)\ \mathit{ys} = x : \mathit{append}\ \mathit{xs}\ \mathit{ys}\end{aligned}$$

Now suppose a programmer would like to refactor certain list functions to make use of the constant-time append function; she will need to define an abstraction function $\alpha :: \underline{\mathit{List}}\ a \to \mathit{List}\ a$ (a corresponding α° will be automatically constructed by RINV, as we will see shortly in the next section), and verify its correctness by proving the promotion condition. The translation outlined in Section **??** will complete the refactoring, by converting calls of *append* into calls of $\underline{\mathit{append}}$. At the same time, many other list functions, including some that are yet to be defined, may still use pattern matching

on, and be manipulated using, the original *List* representation. For example, retrieving the head of a list can be defined as:

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

and calculated in the following manner, oblivious to the refactoring:

$$head\ (append\ (x : xs)\ ys)\ \equiv\ head\ (x : append\ xs\ ys)\ \equiv\ x$$

As we have seen, selective refactoring provides the option of keeping the widely used left-biased list, while taking advantage of alternative representations when needed. This flexibility frees programmers from the dilemma of committing to a single representation that is unlikely to be appropriate universally. In the sequel, we will see more examples where different representations showing distinct merits in different applications.

3.4.2. 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. In Haskell, it is an idiom to introduce a type synonym to document the different usages of the same type.

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

Effectively, in order to use the above definition with any other operations, we need to reverse the MSB representation. However, this implicitness of data representation is risky, as any incorrect usage won't be picked up by a compiler. At the same time, handling the two representations as different types can be cumbersome.

With our proposal, we program with only one representation (*Num*) and selectively refactor certain operations (such as *incr*), which effectively eliminates any possibility of misuse. We firstly create a new type for the LSB

representation (in Haskell the **newtype** declaration is a more efficient alternative to **data** when there is exactly one constructor with exactly one field inside it):

$$\textbf{newtype}\ \underline{Num} = L\ Num$$

and implement the increment function with the new representation

$$\begin{aligned}
&\underline{incr} :: \underline{Num} \to \underline{Num} \\
&\underline{incr}\ (L\ [\,]) &&= L\ [\,One\,] \\
&\underline{incr}\ (L\ (Zero : num)) &&= L\ (One : num) \\
&\underline{incr}\ (L\ (One : num)) &&= L\ (Zero : (\underline{incr}\ num))
\end{aligned}$$

As a result, only one representation is exposed to the programmer, and all the conversions between representations are handled implicitly by the refactoring translation.

3.4.3. Sized Trees

A size-annotated binary tree is suitable for fast indexing, as we can traverse it quickly by not entering any left subtree that has a smaller size than the index.

$$\begin{aligned}
\textbf{data}\ STree\ a = &\ Empty \\
\mid &\ SLeaf\ a \\
\mid &\ SFork\ (Int, (STree\ a, STree\ a))
\end{aligned}$$

(Note that the use of nested pairs in *SFork*'s parameter purely is a pragmatic decision, to avoid having to deal with triples in programming the conversion functions.)

$$\begin{aligned}
&\underline{index} :: Int \to STree\ a \to a \\
&\underline{index}\ 0\ (SLeaf\ a) &&= a \\
&\underline{index}\ n\ (SFork\ (s, (lt, rt))) \mid n \geqslant ls &&= \underline{index}\ (n - ls)\ rt \\
&\qquad\qquad\qquad\qquad\quad \mid otherwise &&= \underline{index}\ n\ lt \\
&\quad \textbf{where}\ ls = getSize\ lt \\
&\qquad\quad\ \ rs = getSize\ rt \\
&getSize\ Empty &&= 0 \\
&getSize\ (SLeaf\ _) &&= 1 \\
&getSize\ (SFork\ (s, _)) &&= s
\end{aligned}$$

This feature of fast indexing makes sized trees an attractive alternative to lists, when access to elements in the middle is required, with the following specification.

$$\begin{aligned} &\mathit{index}\ 0\ [x] &&= x \\ &\mathit{index}\ n\ (\_ : xs) &&= \mathit{index}\ (n-1)\ xs \end{aligned}$$

Again, once an abstraction function is defined and verified, we can enjoy smooth transitions between the two representations, and reap the benefit of having both.

4. The Right-Invertible Language RINV

Our approach to refactoring discussed above relies on the existence of a right inverse, α°, of the user-defined conversion function α. One can of course implement both functions α° and α separately and prove their consistency. For example, a definition of α° for the two representations of queues in Section **??** reads:

$$\begin{aligned} &\alpha^\circ :: [a] \to ([a], [a]) \\ &\alpha^\circ\ ls = (fq, bq) \\ &\quad \textbf{where}\ n\ = \mathit{length}\ ls\ `\mathit{div}`\ 2 \\ &\qquad\quad\ \ fq\ = \mathit{take}\ n\ ls \\ &\qquad\quad\ \ bq = \mathit{reverse}\ (\mathit{drop}\ n\ ls) \end{aligned}$$

An explicit proof verifying that α° is a right inverse of α will be several times longer than the definition above, and make non-trivial uses of properties of the functions involved. Such proofs have to be hand-crafted, and generally cannot be reused.

Instead, we take a linguistic approach to the problem of generating a definition and a consistency proof for α°, by writing α in a right-invertible language that automatically determines such a right inverse for each function constructed. As a result, the task of defining two functions and proving their consistency is reduced to writing just one of them. This approach to invertible languages is not new. As a matter of fact, there have been a number of proposals for similar purposes [**?** **?** **?** **?** **?**]. On the other hand, reversible languages are usually domain-specific; it is hard to directly reuse them for different applications other than the ones for which they were designed. Inspired by the literature, in this section we sketch a language,

namely RINV, for our purpose of refactoring pattern matching; with this language, the α in the above example can be programmed as

$$\alpha = app \circ (id \times reverse)$$

and its right inverse is automatically generated.

The language RINV is defined as a combinator library; its syntax is as below. (Non-terminals are indicated in small capitals.)

Language	RINV	$::=$ CSTR $\mid$ PRIM $\mid$ COMB
Constructors	CSTR	$::= nil \mid cons \mid snoc \mid wrap \mid ...$
Primitives	PRIM	$::= app \mid id \mid assocr \mid assocl \mid swap \mid$ $fst_g \mid snd_g \mid ...$
Combinators	COMB	$::=$ RINV $\circ$ RINV $\mid fold_X$ RINV $\mid$ RINV $\triangledown$ RINV $\mid$ RINV $\times$ RINV

The language is similar in flavour to the point-free 'algebra of programming' style [**?**], but with the additional feature that a right inverse can be automatically generated for each function that is defined. 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 do not distinguish between f and its forward reading $[\![f]\!]$.

The generated right inverses are intended to be total, so the forward functions have to be surjective; this property holds of the primitive functions, and is preserved by the combinators.

There is an extensible set of primitives (cf. Section **??**) defining the basic non-terminal building blocks of the language. In principle, any surjective function could be made a primitive in RINV. All primitives are uncurried; this suits an 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; and fst_g and snd_g are projection functions on pairs – discarding one element of a pair that can be regenerated by applying function g to the other one. As we will show, with just these few primitives we can define many interesting functions.

The set of constructor functions (cf. Section **??**) is also extensible, via new datatypes. We use lowercase names for the uncurried versions of constructors. In contrast to the primitives, constructor functions are inherently

non-surjective, and so require some special treatment that will be discussed shortly. We take advantage of this relaxation of the surjectivity rule to admit other non-surjective functions which are crucial for expressiveness. For example, in addition to the left-biased list constructor *cons*, coming 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.

$$wrap\ x = [x]$$

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 familiar backwards representation of lists, while *wrap*, *nil* and the primitive function *app* correspond to the join list representation [**?**]. As we will show in the sequel, this ability to admit non-surjective functions as constructors functions that do not directly arise from a datatype declaration gives us much freedom in altering structures without having to invent new types.

Since constructor functions are exceptions to the surjectivity rule, we additionally require that 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; such pairs of operands are called *jointly 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. There are built-in annotations in RINV that group constructors into constructor sets, and are used in static checking of surjectivity. For example, for the language presented above, we will have the following set of constructor groups:

$$\{\{nil, cons\}, \{nil, snoc\}, \{nil, wrap, app\}\}$$

The annotations are provided by the language designer, and extended when adding new constructors, but they do not appear in RINV programs. 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. Both the above requirements can be enforced by a rather straightforward syntactic check. The checking algorithm can be found in **??**.

Other than the two already mentioned combinators, $\times$ is the cartesian product of two functions (cf. Section **??**), and $fold_X\ f$ is the operator for regular structural recursion, which decomposes a structure of type X and replaces the constructors with its body f (cf. Section **??**); very often we omit the subscript X, when it can be understood from the context. In combination with *swap*, *assocl* and *assocr*, $\times$ is able to define all functions that rearrange the components of a tuple, while $\triangledown$ is useful in constructing the body of 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 2 (Right invertibility). *Given a function f in* RINV *consisting of forward and right-inverse functions $[\![f]\!]$ and $[\![f]\!]^\circ$, we have $[\![f]\!] \circ [\![f]\!]^\circ \equiv id$.*

The correctness of this theorem will become evident by the end of this section, as we discuss in detail the various constructs of RINV and their properties.

4.1. The Primitive Functions

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

$$
\begin{aligned}
&assocr :: ((a, b), c) \leftrightarrows (a, (b, c))\\
&[\![assocr]\!] = \lambda((a, b), c) \to (a, (b, c))\\
&[\![assocr]\!]^\circ = [\![assocl]\!]
\end{aligned}
$$

$$
\begin{aligned}
&assocl :: (a, (b, c)) \leftrightarrows ((a, b), c)\\
&[\![assocl]\!] = \lambda(a, (b, c)) \to ((a, b), c)\\
&[\![assocl]\!]^\circ = [\![assocr]\!]
\end{aligned}
$$

$$
\begin{aligned}
&swap :: (a, b) \leftrightarrows (b, a)\\
&[\![swap]\!] = \lambda(a, b) \to (b, a)\\
&[\![swap]\!]^\circ = [\![swap]\!]
\end{aligned}
$$

Function *app* is the uncurried append function, which is not injective. The admission of non-injective functions allows us to break away from the

isomorphism restriction. In this case, the right inverse for *app* is not unique; and the RINV programmer will need to choose the appropriate behaviour for the inverse among multiple variants of the function provided by a language implementation. This user-controlled flexibility is standard in invertible languages that admit non-injective functions [**? ?**]. For the purposes of demonstration, we pick the following definition for the right-inverse of *app*.

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

Pure projection functions such as *fst* and *snd* are difficult to handle in a total function setting; in a setting based on complete partial orders, one could replace the discarded element with an undefined value (for example, by defining $[\![fst]\!]^\circ = \lambda x \to (x, \bot)$), but when types are sets, no analogue of $\bot$ is available. Besides, although such a definition would satisfy right invertibility, it would not be practically very useful. Therefore, we restrict the use of projection functions to cases in which the discarded values are recoverable, giving rise to the following:

$$fst_g :: (a, b) \leftrightarrows a$$
$$[\![fst_g]\!]^\circ = \lambda x \to (x, g\ x)$$

and

$$snd_g :: (a, b) \leftrightarrows b$$
$$[\![snd_g]\!]^\circ = \lambda y \to (g\ y, y)$$

Thus, fst_g and snd_g remove duplications from a pair and reintroduce them in the other direction. Note that the function g itself is not necessarily in RINV; it just needs to be definable in the host language (in our case, Haskell).

4.2. The Constructors

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

$$[\![nil]\!] \quad = \lambda() \to [\,]$$
$$[\![cons]\!] = \lambda(x, xs) \to x : xs$$

Inverses of the primitive constructor functions are obtained simply by swapping the right- and left-hand sides of the definitions.

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

They are effectively partial 'guard' functions, succeeding when the input value matches the pattern.

Although *snoc* and *wrap* are not the primitive constructors for left-biased lists, they can be encoded:

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

The right inverses of *snoc* and *wrap* are

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

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.

4.3. The Combinators

The combinators in RINV are mostly standard.

4.3.1. 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:

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

Note that we have chosen not to define $\times$ in terms of the more primitive combinator $\triangle$ that executes both of its input functions on a single datum:

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

In the inverse direction, $[\![f]\!]^\circ\ x$ and $[\![g]\!]^\circ\ y$ would have to agree, 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 $\triangledown$ constructs a function that consumes an element of a sum type ('*Either*' in Haskell).

$$(\triangledown) :: (a \leftrightarrows c) \to (b \leftrightarrows c) \to (\mathit{Either}\ a\ b \leftrightarrows c)$$
$$[\![f \triangledown g]\!] = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Left}\ a \to [\![f]\!]\ a\ ;\ \mathit{Right}\ b \to [\![g]\!]\ b\}$$

In the inverse 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. In contrast to the case of $\triangle$, here the totality in the inverse direction can be recovered by choosing a non-failing branch. 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.

$$[\![f \triangledown g]\!]^\circ = \lambda x \to ([\![f]\!]^\circ\ x)\ \text{`}\mathit{choice}\text{`}\ ([\![g]\!]^\circ\ x)$$
$$\mathit{choice} :: \mathit{Maybe}\ a \to \mathit{Maybe}\ a \to \mathit{Maybe}\ a$$
$$\mathit{choice}\ (\mathit{Just}\ x)\ \_ = x$$
$$\mathit{choice}\ \_\ (\mathit{Just}\ y) = y$$

This shallow backtracking is sufficient because the guards of conditionals are only pattern matching outcomes, which are completely decided at each level. With the introduction of the *Maybe* monad, the types of the functions in RINV have to be lifted. As a result, the invertible function type $s \leftrightarrows t$ is actually the pairing of $s \to \mathit{Maybe}\ t$ and $t \to \mathit{Maybe}\ s$. For brevity, in the main text of this paper, we still use the non-monadic types for RINV functions to avoid the explicit handling of *Maybe*, with the understanding that all functions are lifted to the *Maybe* monad in the implementation. The details are spelt out in **??**.

4.3.2. Recursion

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

$$[\![\mathit{fold}_X\ f]\!]^\circ = \mathit{unfold}_X\ [\![f]\!]^\circ$$

The forward semantics of $\mathit{fold}_X\ f$ is the standard *fold* for a datatype X; its inverse semantics is defined by a corresponding unfold_X. Intuitively, *fold* disassembles a structure and replaces the constructors with applications of the body of the fold. Function *unfold*, on the other hand, takes a seed, splitting it with the body of the unfold 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 constructed mechanically. Our prototype automatically generates *fold* and *unfold* for each non-mutually recursive regular datatype. For example, consider the datatype of lists:

$$\begin{aligned}
&\mathit{fold}_{List} :: (\mathit{Either}\ ()\ (a, b) \to b) \to (\mathit{List}\ a \to b)\\
&\mathit{fold}_{List}\ f = \lambda xs \to \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\qquad [\,] \qquad\ \to f\ (\mathit{Left}\ ())\\
&\qquad\qquad (x : xs) \to f\ (\mathit{Right}\ (x, \mathit{fold}_{List}\ f\ xs))\\
&\mathit{unfold}_{List} :: (b \to \mathit{Either}\ ()\ (a, b)) \to (b \to \mathit{List}\ a)\\
&\mathit{unfold}_{List}\ f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of}\ \mathit{Left}\ () \qquad\ \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Right}\ (a, b) \to a : \mathit{unfold}_{List}\ f\ b
\end{aligned}$$

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

$$\begin{aligned}
&\mathbf{data}\ \mathit{LTree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{LTree}\ a, \mathit{LTree}\ a)\\
&\mathit{fold}_{LTree} :: (\mathit{Either}\ a\ (b, b) \to b) \to \mathit{LTree}\ a \to b\\
&\mathit{fold}_{LTree}\ f = \lambda t \to \mathbf{case}\ t\ \mathbf{of}\\
&\quad \mathit{Leaf}\ a \qquad\ \to f\ (\mathit{Left}\ a)\\
&\quad \mathit{Fork}\ (t_1, t_2) \to f\ (\mathit{Right}\ (\mathit{fold}_{LTree}\ f\ t_1, \mathit{fold}_{LTree}\ f\ t_2))\\
&\mathit{unfold}_{LTree} :: (b \to \mathit{Either}\ a\ (b, b)) \to b \to \mathit{LTree}\ a\\
&\mathit{unfold}_{LTree}\ f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of}\\
&\qquad\qquad\qquad \mathit{Left}\ a \qquad\ \to \mathit{Leaf}\ a\\
&\qquad\qquad\qquad \mathit{Right}\ (b_1, b_2) \to \mathit{Fork}\ (\mathit{unfold}_{LTree}\ f\ b_1, \mathit{unfold}_{LTree}\ f\ b_2)
\end{aligned}$$

We use *unfold* to construct *fold*'s right inverse. The following lemma is well known from the literature [**?**].

Lemma 2. $fold\ [\![f]\!] \circ unfold\ [\![f]\!]^\circ \sqsubseteq\ id.$

The use of $\sqsubseteq$ in the above lemma states that the left-hand side might be less defined than the right-hand side. Since both *fold* and *unfold* are case-exhaustive when their bodies are case-exhaustive, the only issue is the termination of *unfold*: when a body does not split a seed into 'smaller' seeds, unfolding that 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, the 'leads to' ordering on seeds induced by the body of the unfold should not admit an infinite descending chain), at least on polynomial datatypes. Such well-founded seed transformations are known as recursive coalgebras [**?**], and allow the above composition of *fold* and *unfold* to be well-defined in a setting of total functions. Static termination checkers exist in the literature [**? ?**] and are orthogonal to the discussion here.

4.4. Programming in RINV

We are now ready to look into the kinds of function we can define with RINV.

To start with, let's look first at a derived combinator *map* that can be defined in terms of *fold*. For example, *map* on lists, map_{List}, is defined as follows.

$$\begin{aligned} &map_{List} :: (a \leftrightarrows b) \to (List\ a \leftrightarrows List\ b) \\ &map_{List}\ f = fold_{List}\ (nil \triangledown (cons \circ (f \times id))) \end{aligned}$$

Function $map_{List}\ f$ 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_{LTree}, is defined as follows.

$$\begin{aligned} &map_{LTree} :: (a \leftrightarrows b) \to (Tree\ a \leftrightarrows Tree\ b) \\ &map_{LTree}\ f = fold_{LTree}\ ((leaf \circ f) \triangledown fork) \end{aligned}$$

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

$$\begin{aligned} reverse\ \ &= fold_{List}\ (nil \triangledown snoc) \\ [\![reverse]\!]^\circ &= unfold_{List}\ [\![nil \triangledown snoc]\!]^\circ \end{aligned}$$

Function *reverse* is one of many functions for which the use of derived constructors (such as *snoc*) is essential. If we restrict ourselves to the primitive constructors, we can only reverse a list into a different type. 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.

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

For example, we have:

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

The companion $[\![apprev]\!]^\circ$ function is

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

which can be reduced to

$$([\![id]\!]^\circ \times [\![reverse]\!]^\circ) \circ [\![app]\!]^\circ$$

In the inverse 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.

The other function we looked at in Section **??** yields the list view of a size-annotated binary tree. This abstraction function is defined as

$$\alpha = fold_{STree}\ (nil \triangledown wrap \triangledown (app \circ snd_{\lambda(x,y)\to size\ x + size\ y}))$$

Function α drops the size annotations from a tree and flattens it into a list. In the α° direction, the function $\lambda(x, y) \to size\ x + size\ y$ is used to recompute the lost size information.

As a remark, the primitive function *app* can be defined in Haskell with $foldr :: (a \to b \to b) \to b \to [a] \to b$ as:

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

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 constructs the fold body during execution, whereas in RINV, fold bodies are constructed syntactically so that they can be checked and inverted separately from the recurisve combinator. As a result, we include *app* as a primitive function in RINV.

4.5. Extending RINV

We use RINV for the purpose of constructing conversion functions between different representations of data. As a result, RINV has to be extended from time to time, when new abstract representations are introduced. (New concrete representations only require corresponding versions of *fold* and *unfold*, already captured by the parameterized $fold_X$ and $unfold_X$.) At the very least, a new abstract representation brings in a new set of primitive constructors. For example, a binary tree abstract representation might extend RINV with the following:

$$\begin{aligned} [\![tip]\!] &= \lambda x \to Tip\ x \\ [\![fork]\!] &= \lambda(lt, rt) \to Fork\ lt\ rt \end{aligned}$$

The inverses of primitive constructors are always straightforward; and the annotation that records the constructor sets will be extended with the new set $\{\, tip, fork \,\}$.

$$[\![tip]\!]^\circ = \lambda(Tip\ x) \quad \to x$$
$$[\![fork]\!]^\circ = \lambda(Fork\ lt\ rt) \to (lt, rt)$$

We can now start using binary trees, for example, as the abstract representation of sized trees.

$$\alpha = fold_{STree}\ (tip \triangledown (fork \circ snd_{\lambda(x,y)\to size\ x + size\ y}))$$

Only primitive constructors (together with the existing data projection function snd_g) are used in this case. In general, we may need to extend the set of primitive functions as well. For example, consider using natural numbers as an abstract representation:

$$\begin{aligned}&\textbf{data}\ Nat = Zero\\ &\qquad\qquad \mid\ Succ\ Nat\end{aligned}$$

which are then implemented as binary numbers with the least significant bit first:

$$\textbf{type}\ Binary = [Nat]$$

Since we are representing binary numbers, only *Zero* and *Succ Zero* can appear in the list. We do not use a specialized datatype for binary bits, as found in the example in Section **??**, because of the need to add binary bits to natural numbers in the process of converting between the two representations.

Again, primitive constructors arise from the datatype definition:

$$[\![zero]\!] = \lambda() \to Zero$$
$$[\![succ]\!] = \lambda n \to Succ\ n$$

and these can easily be inverted:

$$[\![zero]\!]^\circ = \lambda Zero \quad \to ()$$
$$[\![succ]\!]^\circ = \lambda Succ\ n \to n$$

To construct the conversion function from *Binary* to *Nat*, we need an additional primitive function

$$plusTwice :: (Nat, Nat) \leftrightarrows Nat$$
$$[\![plusTwice]\!] = \lambda(c, x) \to c + 2 * x$$
$$[\![plusTwice]\!]^\circ = \lambda y \to (mod\ y\ 2, div\ y\ 2)$$

Function *plusTwice* is a linear function with slope two, whereas its right inverse calculates the x-value and the y-intercept from a y-value. Similar to *app*, function *plusTwice* is surjective but not injective. Among the multiple choices, the user will have to choose one that suits the intended use. Our experience is that the appropriate choice is usually fairly obvious. For example, the above pattern of combining *mod* and *div* is quite common; it even appears as the function *divMod* in the Haskell standard prelude.

With the newly introduced primitive function, we can define the conversion from *Binary* to *Nat* as

$$\alpha = fold_{List}\ (zero \triangledown plusTwice)$$

In a sense, trying to refactor a datatype that is not already supported by RINV requires an extension to the language. As we have seen in this section, such extensions of primitive functions and constructors typically only generate small proof obligations. On the other hand, we also know that the nature of abstract representations means that they are relatively simple and few in number. It is a known fact that functional programmers "too often reach for lists when an ADT would be more appropriate" [**?**]. As a result, we might expect a library that is sufficient for most common cases can be built up reasonably quickly.

5. Discussion

5.1. RINV Expressiveness

The most general constraint on α functions is surjectivity, in order to ensure the existence of right inverses: valid abstract values are bounded by the actual range of the user-defined α function; invertibility is not guaranteed for abstract values outside this range. In the current proposal, RINV faithfully enforces surjectivity, which explains its restricted expressiveness compared to the standard point-free programming framework. An already-mentioned example that shows this difference 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 has no right inverse, despite the fact that we can easily guard against inconsistent input in the reverse direction:

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

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

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

$$\mathit{unzip} :: \mathit{List}\ (a,b) \leftrightarrows (\mathit{List}\ a, \mathit{List}\ b)$$
$$\mathit{unzip} = \mathit{fold}_{\mathit{List}}\ ((\mathit{nil} \vartriangle \mathit{nil}) \triangledown ((\mathit{cons} \times \mathit{cons}) \circ \mathit{trans}))$$
$$\mathit{trans} :: ((a,b1),(b2,c)) \leftrightarrows ((a,b2),(b1,c))$$
$$\mathit{trans} = \mathit{assocl} \circ (\mathit{id} \times \mathit{subr}) \circ \mathit{assocr}$$

This definition will be rejected in RINV, since $\mathit{cons} \times \mathit{cons}$ and $\mathit{nil} \vartriangle \mathit{nil}$ are not jointly surjective—indeed, *unzip* only produces pairs of lists of equal length.

If a value outside the range is constructed, the integrity of specification-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 specification too. For example, consider a program that requires balanced binary trees. An abstraction 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.

We do not include *unfold* as a combinator in RINV as the dual of *fold* because there is no combinator in RINV that creates values of a sum type: the *choice* operator introduced for the right-inverse of $\triangledown$ does not preserve surjectivity. An option is to define the bodies of *unfold*s as primitive functions; but this does require the programmers to deal with *Either* types explicitly, something we have tried to avoid.

5.2. The Dual Story

In this paper, we have picked the α function to be user-provided; the design of RINV and the subsequent discussion of refactoring is based on this decision. However, this choice is not absolute. One can well imagine a programmer coming up with α° functions first, and a left-invertible language generating the corresponding α functions; this would give the same invertibility property $\alpha \circ \alpha^\circ \equiv \mathit{id}$. The promotion condition can be adapted to involve only α°, as in $\underline{f} \circ \mathcal{F}\ \alpha^\circ \equiv \mathcal{G}\ \alpha^\circ \circ f$. Nevertheless, the crucial computation law and fusion law that form the foundation of the translation and optimization are still derivable; for computation, we have

$$
\begin{array}{ll}
 & \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \\
\equiv & \quad \{\, \underline{f} \circ \mathcal{F}\ \alpha^\circ \equiv \mathcal{G}\ \alpha^\circ \circ f \,\} \\
 & \mathcal{G}\ \alpha \circ \mathcal{G}\ \alpha^\circ \circ f \\
\equiv & \quad \{\, \mathcal{G} \text{ respects composition and identity; } \alpha \circ \alpha^\circ \equiv id \,\} \\
 & f
\end{array}
$$

and for fusion:

$$
\begin{array}{ll}
 & \mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \alpha^\circ \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ \\
\equiv & \quad \{\, \underline{f} \circ \mathcal{F}\ \alpha^\circ \equiv \mathcal{G}\ \alpha^\circ \circ f \,\} \\
 & \mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \alpha^\circ \circ \mathcal{G}\ \alpha \circ \mathcal{G}\ \alpha^\circ \circ f \\
\equiv & \quad \{\, \mathcal{G} \text{ respects composition and identity; } \alpha \circ \alpha^\circ \equiv id \,\} \\
 & \mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \alpha^\circ \circ f \\
\equiv & \quad \{\, \underline{f} \circ \mathcal{F}\ \alpha^\circ \equiv \mathcal{G}\ \alpha^\circ \circ f \,\} \\
 & \mathcal{H}\ \alpha \circ \underline{g} \circ \underline{f} \circ \mathcal{F}\ \alpha^\circ
\end{array}
$$

If we were to develop a left-invertible language in a similar style to RINV's, we expect that many problems will dualize. A notable difference is that users of the language will have to use *unfold* (as $\triangledown$ is not injective), with destruction functions combined together by the backtracking operator *choice*, resulting in a rather unusual programming style.

6. Further Applications

6.1. Pattern Matching for ADTs

The refactoring technique proposed in this paper rewrites selected definitions using pattern matching into ones using library operations; the concrete implementations of the library operations are not used in reasoning, and thus are not exposed. In this case, a more structured way of organizing the refactored program is with abstract datatypes (ADTs) [**?**].

As a matter of fact, the data abstraction framework developed in this paper can be used as a way of introducing pattern matching to certain ADTs. For example, we can construct an ADT for queues:

$$
\begin{array}{l}
\textbf{adt}\ \mathit{Queue}\ a = [\,a\,]\ \textbf{where} \\
\quad \mathit{emptyQ} :: \mathit{Queue}\ a \\
\quad \mathit{enQ} \quad :: a \to \mathit{Queue}\ a \to \mathit{Queue}\ a \\
\quad \mathit{deQ} \quad :: \mathit{Queue}\ a \to \mathit{Queue}\ a
\end{array}
$$

$$
\begin{aligned}
&\mathit{first} &&:: \mathit{Queue}\ a \to a\\
&\mathit{isEmpty} &&:: \mathit{Queue}\ a \to \mathit{Bool}
\end{aligned}
$$

with the following abstract programs as specifications:

$$
\begin{aligned}
&\mathit{emptyQ} &&= [\,]\\
&\mathit{first} &&= \mathit{head}\\
&\mathit{isEmpty} &&= \mathit{null}\\
&\mathit{enQ}\ a\ q &&= q +\!\!+ [\,a\,]\\
&\mathit{deQ} &&= \mathit{tail}
\end{aligned}
$$

This approach is known as *constructive specification* [**?**]: the semantics of operations are explicitly defined by expressing them in terms of a model. For example, the queue ADT is related to the list model. It is worth emphasising that the list 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.

Once an implementation of the ADT is shown correct (by proving the promotion condition), users of the ADT can pattern match on the model when defining non-library functions, and reason about these functions at the level of models. A translation based on the computation law, similar to the one in Section **??**, elaborates the semantics of programs using the ADT.

6.2. Stream Fusion

Streams are the dual of lists, and are a kind of *codata*. Instead of being built as data structures, streams encapsulate operations that can be unfolded to produce stream elements. Coutts et al. [**?**] introduce an ADT of streams in their work on fusion optimizations:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Stream}\ a = \exists s.\mathit{Strm}\ (s \to \mathit{Step}\ a\ s)\ s\\
&\textbf{data}\ \mathit{Step}\ a\ s = \mathit{Done}\\
&\qquad\qquad\quad\ \mid\ \mathit{Yield}\ a\ s\\
&\qquad\qquad\quad\ \mid\ \mathit{Skip}\ s
\end{aligned}
$$

To use the stream ADT as an implementation of lists, they have functions *stream* and *unstream* for converting lists to and from streams.

$$
\begin{aligned}
&\mathit{unstream} :: \mathit{Stream}\ a \to [\,a\,]\\
&\mathit{unstream}\ (\mathit{Strm}\ \mathit{next}\ s) = \mathit{unfold}\ s
\end{aligned}
$$

```
  where unfold s = case next s of
                     Done      →  []
                     Skip s'   →  unfold s'
                     Yield x s' → x : unfold s'

stream :: [a] → Stream a
stream xs = Strm next xs
  where next []       = Done
        next (x : xs) = Yield x xs
```

The *stream* function is non-recursive, corresponding to the non-recursive definition of *Stream*. The *unstream* function repeatedly calls the access operation of the stream, and produces a list by accumulating the elements this yields. The step *Skip* is unproductive, and therefore does not contribute to expressive power. However, it is crucial for the efficiency of their implementation, allowing intermediate stream/unstream conversions to be fused. For example, they want to reduce the expression

$$unstream \circ maps\ f \circ stream \circ unstream \circ maps\ g \circ stream$$

to

$$unstream \circ maps\ f \circ maps\ g \circ stream$$

This would follow from $stream \circ unstream \equiv id$; but this property is not satisfied by their conversion functions, and so their paper leaves open the question of soundness. Nevertheless, the result they want still holds; in fact, it is a consequence of our approach (Theorem **??**), with *unstream* the abstraction function and *stream* its right inverse.

It is worth mentioning that despite our fusion theorem developed in Section **??** can be used to prove the correctness of Coutts et al.'s optimization, the implementation of the *unstream* and *stream* function pair is not able to benefit from RINV. The main problem is that *stream* is non-recursive; and the information needed for constructing *Stream* data is stored in its function component (*next*). At the moment, RINV is not able to handle such higher-order cases.

7. Related Work

The theoretical foundation of this work goes all the way back to the dawn of the program correctness era [**?** **?** **?** **?**]. The promotion condition and

computation law are standard tools for establishing the correctness of program development. In particular, full invertibility of the abstraction function has been shown [**?**] to be a sufficient (although not necessary) precondition for the computation law. As expected, none of the early works deal specifically with pattern matching and its implications in program development, since this language feature first appeared only a decade later.

7.1. Pattern Matching with Data Abstraction

Efforts to combine data abstraction and pattern matching started over twenty years ago with Wadler's *views* proposal [**?**]; and it is still a hot research topic [**? ? ? ? ? ? ? ? ? ? ?**]. To avoid possible confusion over terminology, we always refer to Wadler's proposal [**?**] as "Wadler's views".

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 left- and right-biased 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, *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' function to 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' function from 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 this 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.

$$
\begin{aligned}
&\mathit{last}\ (\mathit{Snoc}\ \mathit{xs}\ x) = x \\
&\mathit{rotLeft}\ (\mathit{Cons}\ x\ \mathit{xs}) \ = \mathit{Snoc}\ \mathit{xs}\ x \\
&\mathit{rotRight}\ (\mathit{Snoc}\ \mathit{xs}\ x) = \mathit{Cons}\ x\ \mathit{xs} \\
&\mathit{rev}\ \mathit{Nil} \qquad\quad = \mathit{Lin} \\
&\mathit{rev}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Snoc}\ (\mathit{rev}\ \mathit{xs})\ x
\end{aligned}
$$

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 by the from function.

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:

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

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

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

To take advantage of the hybrid representation of views, Wadler's applications mostly involves cute examples with frequent changes of views in one function definition. When the hybrid structures fit with the recursive pattern of a function, the shallow conversion pays off by improving efficiency. On the other hand, in the applications we are looking at, it is more common to have only one view involved in a recursion, which certainly benefits from having homogeneous data.

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. 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 subsets 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 [**?**], and followed up by several others [**?** **?**], 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 implementation of selected primitives that are proven correct, and using only the view (our abstract representation) for pattern matching in user-defined functions. The language RINV for defining conversions guarantees right invertibility, a weaker condition that lifts the isomorphism restriction on abstract and concrete representations. However, in contrast to Wadler's views, our system only caters for linear refactoring—we cannot provide multiple views for the same implementation.

'Safe' variants of views [**?** **?**] have been proposed before. 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 explicit 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* [**?** **?**] and many of their variants [**?** **?** **?** **?** **?**] 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 focuses 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, because each active pattern is computed and matched independently.

Our proposal supports the full power of pattern matching while preserving safety. Yet, the additional expressiveness and guarantees come with a higher price tag: RINV is still rather restrictive, and can be somewhat awkward at times. Among alternative approaches, one can see higher-order recursion patterns such as hylomorphisms [**?**] as ways of imposing recursive views on possibly non-recursive abstract data. In the special case that the input is itself an inductively defined datatype, such recursive views can provide alternative decompositions to plain iteration—for example, *paramorphisms* or primitive recursion [**?**] and *histomorphisms* or course-of-value iteration [**?**]. Because RINV as presented here is first-order, it cannot capture these higher-order eliminators; we leave as future work the exploration of a higher-order version of RINV. McBride and McKinna [**?**] point out that view mechanisms providing case analysis are more informative in a dependently typed setting than a simply typed one, because the individual cases may have more refined types than the surrounding context; perhaps switching from polymorphic functional programming to dependent types will eliminate much of the need for the kind of language extension presented here.

7.2. Invertible Programming

The language RINV owes its origins to the rich literature on *invertible* programming [**?** **?**], 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 [**?** **?**]. The 2LT (Two-Level-Transformation) system [**?**] has a left-invertible combinator library at its core. Instead of addressing changes to individual values, the update is expressed as a format (type) evolution, which migrates a whole database to one in a new format. A typical usage of the system involves retrieving relational data into a hierarchical format, updating by enriching the hierarchical format, and putting back the updated data into a new relational format reflecting the enrichment. The whole process is specified in

terms of formats; the corresponding value-level transformations are induced. Invertible arrows [**?**] extend the arrow framework [**?**] (a generalization of monads) with a combinator that encodes pairs of functions being each other's inverses; the paper introducing invertible arrows [**?**] recognizes that, when full invertibility is not achievable (due to the non-isomorphic nature of the two sides), left- or right-biased semi-invertibility is nevertheless a useful approximation.

Right inverses have been studied as a component of the much more elaborate bidirectional programming framework of *lenses* [**?** **?** **?** **?**]; 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 [**?**] to give precise bounds to the ranges of forward functions (thus the domains of backwards functions). As a result, surjectivity now concerns the relationships between the domains and ranges of lenses connected by a combinator, instead of being a property between a function and its target datatype.

We have carefully chosen right invertibility as the central property our system is based on, to balance expressiveness and robustness. As we have seen, it has served our purpose well by being sufficient to establish the crucial computation and fusion laws.

On the other hand, RINV does not keep values on the concrete level stable: bringing a value on the concrete level up to the abstract level and putting it back without modification may induce changes. This has not been a problem with our design, but rules out the possibility of having multiple abstract representations for a single implementation. For example, as shown in Wadler's view proposal [**?**], it can be useful to have both *cons*-lists and *snoc*-lists as views at the same time.

Another omission of RINV is general data-projection functions. We can only discard information that is re-constructible, in other words redundant. At a glance, it may appear that both the above "flaws" would be fixed if we were to use a framework similar to that of *bidirectional programming* [**?** **?** **?** **?** **?** **?**], where a forward function $fwd :: A \to B$ is coupled with a "backwards" counterpart $bwd :: (B, A) \to A$. The idea is that *fwd* transforms a *source* into a *view*, which is then modified; the modification is to be saved in the form of an updated source, produced by *bwd*. Instead of being an inverse, *bwd* takes both the original source and a modified view as inputs, and tries to reinstate the source/view consistency by producing a new source. In this setting, stability of sources is achievable; and discarded values can be recovered

by extracting them from the copied source. However, in our case the modification operations, which are basically arbitrary abstract operations, are far more complicated than those in bidirectional programming, which assumes a single type for the view. As a result, an update from type $(List\ a, List\ a)$ to type $List\ a$ would not be allowed in bidirectional programming, but is perfectly sensible in programming with abstractions. If we use *bwd* on the resulting list, it is never clear what the source input is.

As we can see from the literature summarized above, building invertible languages using a combinatorial approach is by no means a new idea. Through coming from a common foundation, different proposals customize their designs to cater for specific applications. For example, lenses [**?** **?** **?** **?**] deal with data as records, whereas *Inv* [**?**], pointfree lenses [**?**], and RINV deal with data as algebraic datatypes. In a sense, any invertible language admitting algebraic datatypes can be used in place of RINV; our refactoring framework will still work, but not as well. For example, executing inverses in *Inv* may fail at run-time, which will cause failures in pattern matching, while point-free lenses only provide primitive constructors, which is limited in expressiveness. We have carefully designed RINV to balance safety and expressiveness for our application. The non-primitive constructor functions and annotated data projection functions are artifacts of such consideration; and both the new features are nicely integrated into the more standard combinator framework.

Program inversion, in the sense of generating an inverse of a program, is not the only way of obtaining an input from the output that it yields. Glück and Abramov [**?**] proposed a *universal resolving algorithm*, which, give an output and a program, enumerates the original inputs. The core technique they use, known as *positive driving* [**?**], is similar to the notion of *needed narrowing* [**?**] studied in the context of functional logic programming. Thus, their work can be seen as interpretation of a functional program by using the semantics of functional logic programs, a connection that is made more explicit in subsequent work [**?**]. Note that, if a function is surjective, the inverse computation by the universal resolving algorithm is always guaranteed to terminate.

Matsuda *et al.* [**?**] discussed right-inverse computation using grammar parsing. They construct a grammar from a program, so that the productions of the grammar abstract the evaluation of the program. Then, right-inverse computation is given by parsing with respect to the grammar. In their approach, an *affine* and *treeless* first-order functional program [**?**] is always

surjective onto the language of the derived grammar; and its inverse computation has linear complexity with respect to the original program's output.

8. 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 proper encapsulation: programmers are able to selectively reimplement original function definitions into primitive operations, 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 the framework of data abstraction. When an abstraction function is verified by the promotion condition, the computation law is able to replace abstract function calls with concrete ones. The soundness of such refactoring is based on the right-inverse property of the conversion pairs that bridge the abstract and concrete representations, for which we have designed RINV to guarantee right-invertibility by construction.

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 the anonymous reviewers of the earlier conference version of this paper [**?**] and of this revised and expanded version; we are especially 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 was supported by the UK Engineering and Physical Sciences Research Council through the *Generic and Indexed Programming* project (EP/E02128X) and a PhD Plus award on *Bidirectional Programming* via the Doctoral Training Grant (EP/P503876/1), and was partly conducted during Wang's internship

at National Institute of Informatics, Japan. Matsuda is supported by Grant-in-Aid for Start-up 22800003, and part of the work was done while he was at the University of Tokyo as JSPS Research Fellow supported by Grant-in-Aid for JSPS Fellows $20 \cdot 9584$.

AppendixA. Prototype Implementation

Our system is implemented as a preprocessor for the Glasgow Haskell Compiler, taking an input script and producing a Haskell program. The running example from the paper—of refactoring queues to use a non-list representation—can be fed to our system as a script structured as shown in Figure **??**. We deliberately use `typewriter` font rather than proper typesetting, because at the moment we see the input as a plain text script instead of a program in a rigorously defined extended language. Some familiarity with Haskell [**? ?**] and Template Haskell [**?**] is required to understand the details of this section.

AppendixA.1. Invoking Our System

The preprocessor can be applied from the command line as follows.

```
ghci -F -pgmF refactor QueueDemo.hs
```

Here, `refactor` is the name of the executable file of our prototype implementation, and `QueueDemo.hs` the input script, structured as shown in Figure **??**. This invokes the interactive version of the Glasgow Haskell Compiler `ghci`, and loads the translated code.

AppendixA.2. The Input File

As shown in Figure **??**, the input file consists mainly of three blocks, labelled `view`, `abstract` and `concrete`, specifying the intended refactoring; it also has some client code making use of the refactoring. In the following, we explain the various blocks in turn.

AppendixA.2.1. The `view` *Block*

The purpose of the `view` block, shown in Figure **??**, is to declare the abstract and concrete representations, and define the abstraction function. In the queue example, the abstract representation `Queue a` is implemented in terms of the concrete representation `Qi a`; the conversion between them is defined by RINV function `abstraction`. Note that the actual declarations of datatypes `Queue` and `Qi` are imported from separate modules `Queue` and `QueueImpl`, due to a restriction imposed by Template Haskell.

In the definition of `abstraction`, we use an ASCII encoding of standard RINV combinators: we write '`<**>`' for $\times$, '`.`' for $\circ$, and '`<||>`' for $\triangledown$. We use Template Haskell to generate appropriate fold functions and constructor

```
{-# OPTIONS -XTemplateHaskell -XMultiParamTypeClasses #-}
import UpdatableViewStub
import Queue
import QueueImpl

view Qi a as Queue a by
  abstraction = ...
  completeSets = ...
abstract
  ...
concrete
  ...

mapQueue = ...
play1 = ...
```

Figure A.1: An example input file: `QueueDemo.hs`

```
view Qi a as Queue a by
  abstraction = $(foldT ''Qi) (l2l . app . (id <**> reverse))
      where l2l = $(foldT ''[]) ( $(toC 'None) <||> $(toC 'More) )
  completeSets _ = [$(namesT ''Qi), $(namesT ''Queue)]
```

Figure A.2: An example input file: the `view` block

functions for given datatypes. For example, `foldT ''Qi` generates the fold for type `Qi`, and `toC 'None` generates the constructor function for constructor `None`. Note that ‘'’ refers to the *constructor* name and ‘''’ refers to the *type* name; for example, `'[]` means the name of the data constructor `[]` while `''[]` means the name of the type constructor `[]`.

The `view` block also contains the definition of `completeSets`, which specifies the constructor grouping introduced with the new datatypes.

Our preprocessor translates the `view` block into a RINV program, as described in **??**; the surjectivity check described in **??** guarantees the validity of the generated program, which is done statically through Template Haskell.

AppendixA.2.2. The `abstract` *and* `concrete` *Blocks*

The `abstract` and `concrete` blocks, shown in Figure **??**, contain the definitions of library functions on the specification and implementation level respectively. The specifications are simply copied to the output, while conversions are inserted into the implementations.

Instead of explicitly using functors, as in Section **??**, we overload the abstraction functions so that the Haskell type class mechanism will implicitly choose the appropriate instances. This is the reason for the type annotations of the library functions (no type inference or type checking is performed in the prototype). Another technical detail is that we group the α/α° pair into a single function `toView`, so that `toView f` is equivalent to $\alpha \circ f \circ \alpha^\circ$

AppendixA.2.3. Refactoring Programs

The `view`, `abstract` and `concrete` blocks prepare a refactoring system, which can be used to implement specifications. In the client code part of the example input file (shown in Figure **??**), `play1` represents the $play_{\text{prim}}$ function from Section **??**. As explained in Section **??**, $play_{\text{prim}}$ does not use the abstract representation at all; consequently, it is translated as if it were a library function. In Section **??** we explained how this absence of abstract representation can be inferred through type checking. As we perform no type checking in the prototype, such functions have to be annotated by their types and the libary functions they actually call.

Our system produces the code in Figure **??** for `play1`. A new function `play1_impl` ($\underline{play_{\text{prim}}}$ from Section **??**) is the implementation of `play1` using the concrete representation. Calls to `play1` are then redirected to `play1_impl` after proper conversions. The `mapQueue` definition is not refactored—it is copied straight to the output.

```
abstract
  emptyQueue :: Queue a
  emptyQueue = None

  enQ :: a -> Queue a -> Queue a
  enQ = More

  deQ :: Queue a -> Queue a
  deQ (More _ x) = x

  first :: Queue a -> a
  first (More a _) = a
concrete
  emptyQueue :: Qi a
  emptyQueue = Qi [] []

  enQ :: a -> Qi a -> Qi a
  enQ a (Qi x y) = Qi (a:x) y

  deQ :: Qi a -> Qi a
  deQ (Qi (a:x) y) = Qi x y
  deQ (Qi [] y)    = deQ (Qi (reverse y) [])

  first :: Qi a -> a
  first (Qi (a:x) y) = a
  first (Qi [] y)    = first (Qi (reverse y) [])
```

Figure A.3: An example input file: the `abstract` and `concrete` blocks

```
mapQueue f None       = None
mapQueue f (More a x) = More f (mapQueue f x)

{-@ IMPL Int -> Qi (IO ()) -> IO ()
      of Int -> Queue  (IO ()) -> IO ()
   USING enQ first deQ @-}
play1 0     q = first q
play1 (n+1) q = do hd
                   play1 n (enQ hd tl)
   where hd = first q
         tl = deQ   q
```

Figure A.4: An example input file: client code

```
play1_impl =
    let
        play1 0     q = first q
        play1 (n+1) q = do hd
                           play1 n (enQ hd tl)
           where hd = first q
                 tl = deQ   q
    in  play1
    where
        enQ = enQ_impl
        first = first_impl
        deQ = deQ_impl
play1 = (toView :: (Int -> Qi (IO ()) -> IO ()) ->
                   (Int -> Queue  (IO ()) -> IO ()))
        play1_impl
```

Figure A.5: An example of refactored output

AppendixB. RINV Translation

This section summarizes the translation of RINV to Haskell that was described in Section **??**. In the clauses below, the use of *Maybe* monad for failure handling is made explicit. As we can see, the only interesting use of the *Maybe* type is in the implementation of *choice*, where the right operand is chosen when the execution of the left operand fails (i.e., returning *Nothing*). The remaining clauses are simply lifted into the *Maybe* monad.

$$[\![\cdot]\!] :: (a \leftrightarrows b) \to a \to \mathit{Maybe}\ b$$
$$[\![\cdot]\!]^{\circ} :: (a \leftrightarrows b) \to b \to \mathit{Maybe}\ a$$

AppendixB.1. Constructors

$$\mathit{nil} :: () \leftrightarrows [a]$$
$$[\![\mathit{nil}]\!] = \lambda() \to \mathit{return}\ [\,]$$
$$[\![\mathit{nil}]\!]^{\circ} = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{[\,] \to \mathit{return}\ ()\ ;\ \_ \to \mathit{mfail}\ \texttt{"Unmatched"}\}$$

$$\mathit{cons} :: (a, [a]) \leftrightarrows [a]$$
$$[\![\mathit{cons}]\!] = \lambda(x, xs) \to \mathit{return}\ (x : xs)$$
$$[\![\mathit{cons}]\!]^{\circ} = \lambda l \to \mathbf{case}\ l\ \mathbf{of}\ \{x : xs \to \mathit{return}\ (x, xs)\ ;\ \_ \to \mathit{mfail}\ \texttt{"Unmatched"}\}$$

$$\mathit{snoc} :: (a, [a]) \leftrightarrows [a]$$
$$[\![\mathit{snoc}]\!] = \lambda(x, xs) \to \mathit{return}\ (xs +\!\!+ [x])$$
$$[\![\mathit{snoc}]\!]^{\circ} = \lambda l \to \mathbf{if}\ \mathit{length}\ l \geqslant 1\ \mathbf{then}\ (\mathit{last}\ l, \mathit{init}\ l)\ \mathbf{else}\ \mathit{mfail}\ \texttt{"Unmatched"}$$

$$\mathit{wrap} :: a \leftrightarrows [a]$$
$$[\![\mathit{wrap}]\!] = \lambda x \to \mathit{return}\ [x]$$
$$[\![\mathit{wrap}]\!]^{\circ} = \lambda l \to \mathbf{case}\ l\ \mathbf{of}\ \{[x] \to \mathit{return}\ x\ ;\ \_ \to \mathit{mfail}\ \texttt{"Unmatched"}\}$$

$$\mathit{app} :: ([a], [a]) \leftrightarrows [a]$$
$$[\![\mathit{app}]\!] = \lambda(x, y) \to \mathit{return}\ (x +\!\!+ y)$$
$$[\![\mathit{app}]\!]^{\circ} = \lambda xs \to \mathit{return}\ (\mathit{splitAt}\ ((\mathit{length}\ xs + 1)\ `\mathit{div}`\ 2)\ xs)$$

AppendixB.2. Primitives

$$\mathit{id} :: a \leftrightarrows a$$
$$[\![\mathit{id}]\!] = \mathit{return}$$
$$[\![\mathit{id}]\!]^{\circ} = \mathit{return}$$

$$\mathit{assocr} :: ((a, b), c)) \leftrightarrows (a, (b, c))$$

$$
\begin{aligned}
&[\![assocr]\!] = \lambda((a,b),c) \to return\ (a,(b,c))\\
&[\![assocr]\!]^\circ = [\![assocl]\!]
\end{aligned}
$$

$$
\begin{aligned}
&assocl :: (a,(b,c)) \leftrightarrows ((a,b),c)\\
&[\![assocl]\!] = \lambda(a,(b,c)) \to return\ ((a,b),c)\\
&[\![assocl]\!]^\circ = [\![assocr]\!]
\end{aligned}
$$

$$
\begin{aligned}
&swap :: (a,b) \leftrightarrows (b,a)\\
&[\![swap]\!] = \lambda(a,b) \to return\ (b,a)\\
&[\![swap]\!]^\circ = [\![swap]\!]
\end{aligned}
$$

$$
\begin{aligned}
&fst :: (a \to b) \to (a,b) \leftrightarrows a\\
&[\![fst]\!] = \lambda g \to return \circ fst\\
&[\![fst]\!]^\circ = \lambda g \to \lambda x \to return\ (x, g\ x)
\end{aligned}
$$

$$
\begin{aligned}
&snd :: (b \to a) \to (a,b) \leftrightarrows b\\
&[\![snd]\!] = \lambda g \to return \circ snd\\
&[\![snd]\!]^\circ = \lambda g \to \lambda x \to return\ (g\ x, x)
\end{aligned}
$$

AppendixB.3. Combinators

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

$$
\begin{aligned}
&(\times) :: (a \leftrightarrows b) \to (c \leftrightarrows d) \to ((a,c) \leftrightarrows (b,d))\\
&[\![f \times g]\!] = \lambda(w,x) \to liftM2\ (,)\ ([\![f]\!]\ w)\ ([\![g]\!]\ x)\\
&[\![f \times g]\!]^\circ = \lambda(y,z) \to liftM2\ (,)\ ([\![f]\!]^\circ\ y)\ ([\![g]\!]^\circ\ z)
\end{aligned}
$$

$$
\begin{aligned}
&(\triangledown) :: (a \leftrightarrows c) \to (b \leftrightarrows c) \to (Either\ a\ b \leftrightarrows c)\\
&[\![f \triangledown g]\!] = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{\, Left\ a \to [\![f]\!]\ a\ ;\ Right\ b \to [\![g]\!]\ a \,\}\\
&[\![f \triangledown g]\!]^\circ = \lambda x \to [\![f]\!]^\circ\ x\ \ `choice`\ \ [\![g]\!]^\circ\ x\\
&\quad \mathbf{where}\ choice\ (Just\ x)\ \_ \qquad\quad = Just\ x\\
&\qquad\qquad choice\ \_ \qquad (Just\ x) = Just\ x\\
&\qquad\qquad choice\ \_ \qquad \_ \qquad\quad = Nothing
\end{aligned}
$$

$$
\begin{aligned}
&fold_{List} :: (Either\ ()\ (a,b) \leftrightarrows b) \to [\,a\,] \leftrightarrows b\\
&[\![fold_{List}\ f]\!] = foldM_{List}\ [\![f]\!]\\
&[\![fold_{List}\ f]\!]^\circ = unfoldM_{List}\ [\![f]\!]^\circ
\end{aligned}
$$

$$
\begin{aligned}
&foldM_{List} :: (Either\ ()\ (a,b) \to Maybe\ b) \to [\,a\,] \to Maybe\ b\\
&foldM_{List}\ f\ [\,] = f\ (Left\ ())
\end{aligned}
$$

$$foldM_{List}\ f\ (a : x) = foldM_{List}\ f\ x \gg\!= \lambda r \to f\ (Right\ (a, r))$$

$$\begin{aligned}
&unfoldM_{List} :: (b \to Maybe\ (Either\ ()\ (a, b))) \to b \to Maybe\ [a] \\
&unfoldM_{List}\ f\ b = \\
&\quad f\ b \gg\!= \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
&\qquad Left\ () \qquad\ \to return\ [\,] \\
&\qquad Right\ (a, b) \to unfoldM_{List}\ f\ b \gg\!= \lambda x \to return\ (a : x)
\end{aligned}$$

AppendixC. Pseudocode for Surjectivity Check

This section describes the algorithm for the surjectivity check in RINV. Function *checkSurj* checks whether a constructor or a set of constructors (collected from operands of $\triangledown$) 'covers' a complete constructor set. In the case of $\triangledown$, when the two operands 'cover' a complete constructor set, we do not require them individually to be surjective. However, we still need to make sure that compositions with the operands are properly constructed (i.e., only surjective functions on the right of a composition). This is the reason for invoking *checkComp*, which in turn relies on *checkSurj* for checking the surjectivity of its component. One property of the two functions is that *checkSurj* x implies *checkComp* x.

$$
\begin{array}{ll}
\mathit{checkSurj}\ (\mathit{constructor}) & = \mathit{isComplete}\ \mathit{constructor}\ \mathit{completeSets} \\
\mathit{checkSurj}\ (\mathit{primitive}) & = \mathit{True} \\
\mathit{checkSurj}\ (f \triangledown g) & = (\mathit{isComplete}\ (f \triangledown g)\ \mathit{completeSets} \vee \\
 & \quad \mathit{checkSurj}\ f \vee \mathit{checkSurj}\ g) \wedge \\
 & \quad \mathit{checkComp}\ f \wedge \mathit{checkComp}\ g \\
\mathit{checkSurj}\ (f \circ g) & = \mathit{checkSurj}\ f \wedge \mathit{checkSurj}\ g \\
\mathit{checkSurj}\ (f \times g) & = \mathit{checkSurj}\ f \wedge \mathit{checkSurj}\ g \\
\mathit{checkSurj}\ (\mathit{fold}\ f) & = \mathit{checkSurj}\ f
\end{array}
$$

$$
\begin{array}{ll}
\mathit{checkComp}\ (\mathit{constructor}) & = \mathit{True} \\
\mathit{checkComp}\ (\mathit{primitive}) & = \mathit{True} \\
\mathit{checkComp}\ (f \triangledown g) & = \mathit{checkComp}\ f \wedge \mathit{checkComp}\ g \\
\mathit{checkComp}\ (f \circ g) & = \mathit{checkComp}\ f \wedge \mathit{checkSurj}\ g \\
\mathit{checkComp}\ (f \times g) & = \mathit{checkComp}\ f \wedge \mathit{checkComp}\ g \\
\mathit{checkComp}\ (\mathit{fold}\ f) & = \mathit{checkComp}\ f
\end{array}
$$

In the above, *isComplete* f cs checks whether the constructors provided by f cover one of the constructor sets in cs. We implement sets as lists, but still use the set notations for presentation. When constructors are paired into tuples, the behaviour of *isComplete* is slightly more complicated, as we need to compute the cartesian products of the constructor sets.

$$
\begin{array}{l}
\mathit{isComplete}\ f\ cs = \\
\quad \mathit{any}\ (\lambda s \rightarrow s \subseteq (\mathit{flatten}\ f))\ \mathit{completeTupleSets} \\
\quad\quad \textbf{where}
\end{array}
$$

$$
\begin{array}{l}
dim = length\ \$\ (flatten\ f)\ !!\ 0 \\
completeTupleSets \\
\quad = foldNat\ (\lambda r \rightarrow \{\{a : b \mid a \leftarrow c, b \leftarrow d\} \mid c \leftarrow cs, d \leftarrow r\}) \\
\qquad\qquad \{\{[\,]\}\} \\
\qquad\qquad dim \\
foldNat\ s\ n\ 0 = n \\
foldNat\ s\ n\ k = s\ (foldNat\ s\ n\ (k-1))
\end{array}
$$

In the above, *completeTupleSets* is the pairwise cartesian products of the elements in *cs* up to a given dimension. For example, given a set of sets of constructors

$$\{\{nil, cons\}, \{nil, snoc\}, \{nil, wrap, app\}\}$$

a *completeTupleSets* with dimension 2 is

$$
\begin{array}{l}
\{\{[nil, nil], [nil, cons], [cons, nil], [cons, cons]\}, \\
\ \{[nil, nil], [nil, snoc], [cons, nil], [cons, snoc]\}, \\
\ \{[nil, nil], [nil, wrap], [nil, app], [cons, nil], [cons, wrap], [cons\ app]\}, \\
\ ...\}
\end{array}
$$

The dimension represents the number of elements in a possibly nested tuple, for which we flatten nested tuples into lists.

$$
\begin{array}{ll}
flatten\ (constructor) & = \{[constructor]\} \\
flatten\ (f \triangledown g) & = flatten\ f \cup flatten\ g \\
flatten\ (f \times g) & = \{nf +\!\!+ ng \mid nf \leftarrow flatten\ f, ng \leftarrow flatten\ g\} \\
flatten\ (f \circ g) & = flatten\ f \\
flatten\ \_ & = \{[\,]\} \quad \text{-- primitive functions are ignored}
\end{array}
$$

The nesting of $\times$ increases the length of individual elements (dimension) in a complete set, whereas the nesting of $\triangledown$ increases the size of the complete sets. For example, we have the following executions

$$flatten\ (nil \times cons) = \{[nil, cons]\}$$

$$flatten\ (nil \triangledown cons) = \{[nil], [cons]\}$$

and

$$\mathit{flatten}\ ((\mathit{nil} \times \mathit{cons}) \bigtriangledown (\mathit{cons} \times \mathit{nil})) = \{[\mathit{nil}, \mathit{cons}], [\mathit{cons}, \mathit{nil}]\}$$

Among the above, $\{[\mathit{nil}], [\mathit{cons}]\}$ is a complete set, so $\mathit{nil} \bigtriangledown \mathit{cons}$ passes the *isComplete* test but the other ones fail. On the other hand, $(\mathit{nil} \times \mathit{nil}) \bigtriangledown (\mathit{nil} \times \mathit{cons}) \bigtriangledown (\mathit{cons} \times \mathit{nil}) \bigtriangledown (\mathit{cons} \times \mathit{cons})$ passes the *isComplete* test, because it flattens into $\{[\mathit{nil}, \mathit{nil}], [\mathit{nil}, \mathit{cons}], [\mathit{cons}, \mathit{nil}], [\mathit{cons}, \mathit{cons}]\}$, which covers a complete set of dimension 2.