Inverting Functions as Folds

Shin-Cheng Mu and Richard Bird

Programming Research Group, Oxford University
Wolfson Building, Parks Road, OX1 3QD, UK

Abstract. This paper is devoted to the proof and applications of a theorem giving conditions under which the inverse of a partial function can be expressed as a relational hylomorphism. The theorem is a generalisation of a previous result, due to Bird and de Moor, that gave conditions under which a total function can be expressed a relational fold. The theorem is illustrated with three problems, all dealing with constructing trees with various properties.

1 Introduction

Many problems in computation can be specified in terms of computing the inverse of an easily constructed function. The purpose of this paper is to describe one technique for inverting functions and to illustrate it with three examples. We will begin by describing the three problems. First, consider the following datatype *Tree A* of tip-valued binary trees:

data *Tree A* = *Tip A* | *Bin* (*Tree A*) (*Tree A*)

Suppose we are given two lists, one representing the depths of the tips of a tree in left-to-right order, and the other the tip values themselves. How can we reconstruct the tree from the two lists? This particular problem arises, for instance, in the final phase of the Hu-Tucker algorithm [18]. For simplicity, we will identify tip values with their depths, as in Figure 1. Of course, not every list of numbers corresponds to the depths of the tips of a tree.

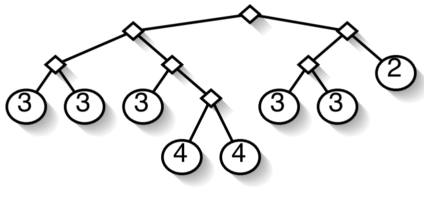

Fig. 1. A tree whose tips have depths $[3, 3, 3, 4, 4, 3, 3, 2]$

In the second problem, we are given a list of trees. The task is to combine them into a single tree, retaining the left-to-right order of the subtrees. How can we do this to make the height of the resulting tree as small as possible? Figure 2 illustrates one such tree, of height 11, for given subtrees of heights $[2, 9, 8, 3, 6, 9]$. As the actual content of the subtrees isn't important, we can think of them simply as numbers representing the heights. The problem is therefore also one of turning a list of numbers to a tree.

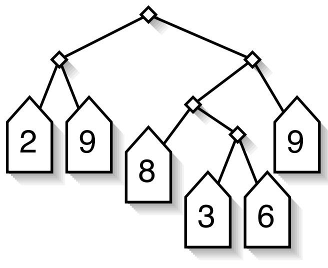

Fig. 2. A tree with height 11 built from trees with heights $[2, 9, 8, 3, 6, 9]$

The third problem is that of breadth-first labelling. Consider the following definition of internally and externally labelled binary trees:

$$\textbf{data}\ \textit{Tree}\ A = \textit{Tip}\ A \mid \textit{Bin}\ A\,(\textit{Tree}\ A)\,(\textit{Tree}\ A)$$

A breadth-first labelling of a tree with respect to a given list is the problem of augmenting the nodes of the tree with values in the list in breadth-first order. Figure 3 shows the result of breadth-first labelling a tree with 13 nodes with the infinite list $[1\cdot\cdot]$. While everybody knows how to do breadth-first traversal, the closely related problem of efficient breadth-first labelling is not so widely understood.

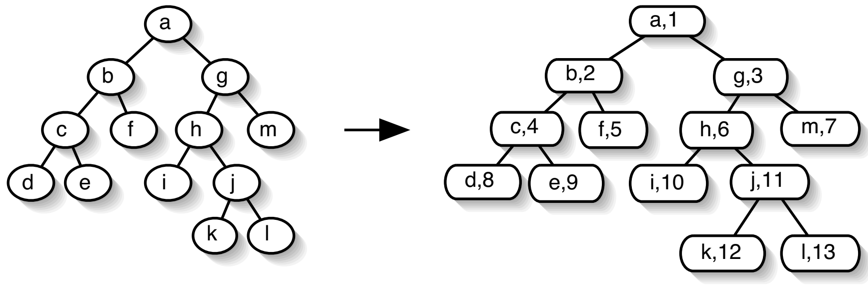

Fig. 3. Breadth-first labelling a tree on the left with $[1\cdot\cdot]$.

All three problems involve building (or rebuilding) a tree of some kind, and all can be specified in terms of the converse operation of flattening a tree into a list of its values. Functional programmers are aware that flattening a structure is usually performed by a fold operation. Consequently, building a structure is usually performed by the converse operation, unfold. However, there is no reason why the converse operation should necessarily involve an unfold. The converse-of-a-function theorem, to which this paper is devoted, gives us conditions under which the inverse of a function can be written as a fold.

In the following sections we will show how this theorem can be applied to derive solutions to the above problems. We claim that the converse-of-a-function theorem is useful because many problems can be specified in terms of an inverse of a known function. Functional programmers make use of a handful of laws and theorems to transform specifications to optimising code. The converse-of-a-function theorem is another useful tool worth adding to the functional programmer's arsenal. Its joint use with the fold fusion theorem turns out to be a recurring pattern in program derivation. Finally, we will present and prove a generalised theorem allowing one to write the inverse of a partial function as a hylomorphism.

2 Theory

The converse of a function is a relation, so our framework is of necessity a calculus of relational programs [3,5]. In this section we will present enough notation to describe the main ideas. Further concepts are introduced in Section 7.

2.1 Relations

Set-theoretically speaking, a relation $R :: A \leadsto B$ is a set of pairs (a, b) where a has type A and b type B. The *converse* of a relation is defined by flipping the pairs, that is,

$$(b, a) \in R^\circ \equiv (a, b) \in R$$

For $R :: B \leadsto A$ and $S :: C \leadsto B$, the composition $R \cdot S :: C \leadsto A$ is defined by

$$(c, a) \in R \cdot S \equiv (\exists b : b \in B : (c, b) \in S \land (b, a) \in R)$$

Converse is contravariant with respect to composition, so $(R \cdot S)^\circ = S^\circ \cdot R^\circ$.

For each type A, a relation id_A is defined by $id_A = \{(a, a) | a \in A\}$. We will omit the subscript when it is clear from the context. A relation $R :: A \leadsto B$ is called *simple* if $R \cdot R^\circ \subseteq id$. That is, every value in A is mapped to at most one value in B. In other words, R is a partial function. A relation R is called *entire* if $id \subseteq R^\circ \cdot R$, that is, every value in A is mapped to at least one value in B. A relation is a (total) function if it is both simple and entire.

In this paper we write the type of a function as $A \rightarrow B$, that of a partial function as $A \nrightarrow B$, and that of a relation as $A \leadsto B$.

A relation is called a *coreflexive* if it is a subset of id. We use coreflexives to model predicates. The ? operator converts a boolean-valued function to a coreflexive:

$$(a, a) \in p? \equiv p\ a$$

For convenience, we let $(a, a) \notin p?$ both when $p\ a$ yields *False* and when a is not in the domain of p. If we perform two consecutive tests, one of them being stronger than the other, the stronger one can absorb the weaker one:

$$(p\ a \Rightarrow q\ a) \Rightarrow p? \cdot q? = p? \tag{1}$$

Given a relation $R :: A \rightsquigarrow B$, the coreflexive $dom\ R :: A \mathrel{+\!\!\!\rightarrow} A$ determines the domain of R and is defined by

$$(a, a) \in dom\ R \equiv (\exists b : b \in B : (a, b) \in R)$$

Alternatively, $dom\ R = R^{\circ} \cdot R \cap id$, where $\cap$ denotes set intersection. It follows that

$$dom\ R \subseteq R^{\circ} \cdot R \tag{2}$$

The coreflexive $ran\ R$ determines the range of a relation and is defined by $ran\ R = dom\ R^{\circ}$.

When writing in a pointwise style, relations can be introduced by the choice operator $\Box$. The expression $x \Box y$ non-deterministically yields either x or y. For example, the following relation *prefix* maps a list to one of its prefixes:

```
$prefix :: List\ A \rightsquigarrow List\ A$
$prefix = foldr\ step\ [\,]$
    **where** $step\ \ \ \ \ :: A \rightarrow List\ A \rightsquigarrow List\ A$
          $step\ a\ x = (a : x)\ \Box\ [\,]$
```

In each step of the fold we can choose either to cons the current item to some prefix of the sublist, or just return the empty sequence [ ], which is a prefix of every list. For a more rigorous semantics of $\Box$, the reader is referred to [9].

2.2 Folds

Datatypes come with fold functions. For lists, the Haskell Prelude function $foldr :: (A \rightarrow B \rightarrow B) \rightarrow B \rightarrow List\ A \rightarrow B$ is well known. A slight variation for non-empty lists can be defined by

```
$foldrn\ \ \ \ \ \ \ \ \ \ \ \ :: (A \rightarrow B \rightarrow B) \rightarrow (A \rightarrow B) \rightarrow List^{+}\ A \rightarrow B$
$foldrn\ f\ g\ [a]\ \ \ \ \ = g\ a$
$foldrn\ f\ g\ (a : x) = f\ a\ (foldrn\ f\ g\ x)$
```

Here $List^{+}\ A$ denotes the type of non-empty lists. Recall the *Tree* datatype defined in the introduction; its fold function can be defined as:

```
$foldtree\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ :: (B \rightarrow B \rightarrow B) \rightarrow (A \rightarrow B) \rightarrow Tree\ A \rightarrow B$
$foldtree\ f\ g\ (Tip\ a)\ \ \ = g\ a$
$foldtree\ f\ g\ (Bin\ x\ y) = f\ (foldtree\ f\ g\ x)\ (foldtree\ f\ g\ y)$
```

All of these folds are instances of a more general definition. A regular datatype T can be defined as the fixed-point of a *base functor* F. That is to say, there is an isomorphism

$$\alpha_{\mathsf{F}} :: \mathsf{FT} \rightarrow \mathsf{T}$$

Datatypes are often parameterised. In that case α_{F} has type $\mathsf{F}_A(\mathsf{T}A) \rightarrow \mathsf{T}A$. For example, cons-lists over an arbitrary is the fixed-point of $\mathsf{F}_A\, X = 1 + (A \times X)$. When denoting types, we will write $\mathsf{F}(A, X)$ instead of $\mathsf{F}_A\, X$, thinking of F as a bifunctor. For more example, the base functor for non-empty lists is $\mathsf{F}(A, X) = A + (A \times X)$, and that for *Tree* is $\mathsf{F}(A, X) = A + (X \times X)$.

Given a base functor F for a datatype $\mathsf{T}A$ and a function f of type $\mathsf{F}(A, B) \rightarrow B$ for some B, the *catamorphism* $(\![f]\!)_{\mathsf{F}} :: \mathsf{T}A \rightarrow B$ is the unique function satisfying

$$(\![f]\!)_{\mathsf{F}} \cdot \alpha_{\mathsf{F}} = f \cdot \mathsf{F}(\![f]\!)_{\mathsf{F}}$$

The different folds are special cases of $(\![f]\!)_{\mathsf{F}}$ instantiated to different base functors, except that in Haskell, we usually divide f into several functions or constants, each of which corresponds to the operation on a particular operand of the coproduct in the base functor.

A functor on relations that takes functions to functions and is monotonic under relational inclusion is called a *relator*. By switching from functors to relators, the above theory extends to relations as well. A catamorphism $(\![R]\!)_{\mathsf{F}}$, where R is a relation of type $\mathsf{F}(A, B) \rightsquigarrow B$, now has type $\mathsf{T}A \rightsquigarrow B$. For a fuller account of relator theory and relational catamorphisms, the reader is referred to [2,3].

3 The Converse-of-a-Function Theorem

The converse-of-a-function theorem, introduced in [5,9], tells us how we can write the inverse of a function as a fold. It reads:

Theorem 1 (Converse of a Function). Let $f :: B \rightarrow \mathsf{T}A$ be a function and F the base functor for T. If $R :: \mathsf{F}(A, B) \rightsquigarrow B$ is surjective and $f \cdot R \subseteq \alpha_{\mathsf{F}} \cdot \mathsf{F}f$, then $f^{\circ} = (\![R]\!)_{\mathsf{F}}$.

The specialisation of this theorem to functions over lists reads as follows: let $f :: B \rightarrow \mathit{List}\, A$ be given. If $\mathit{base} :: B$ and $\mathit{step} :: A \rightarrow B \rightsquigarrow B$ are jointly surjective (meaning that $\{\mathit{base}\} \cup \{b' \mid \exists\, a, b : (b', b) \in \mathit{step}\, a\} = B$) and satisfy

$$\begin{aligned} &f\, \mathit{base} &&= [\,] \\ &f(\mathit{step}\, a\, x) &&= a : f\, x \end{aligned}$$

then $f^{\circ} = \mathit{foldr}\; \mathit{step}\; \mathit{base}$.

Similarly, to invert a total function f on non-empty lists, Theorem 1 states that if $\mathit{base} :: A \rightsquigarrow B$ and $\mathit{step} :: A \rightarrow B \rightsquigarrow B$ are jointly surjective (that is, $\mathit{ran}\; \mathit{base} \cup \mathit{ran}\; \mathit{step} = \mathit{id}_B$) and satisfy

$$\begin{aligned} &f\,(\mathit{base}\, a) &&= [a] \\ &f(\mathit{step}\, a\, x) &&= a : f\, x \end{aligned}$$

then $f^\circ = \mathit{foldrn\ step\ base}$.

We will postpone the proof of Theorem 1 to Sect. 7, where in fact a more general result is proved. For now, let us see some of its applications.

4 Building a Tree from Its Depths

We will start with a formal specification of the problem of a building a tree given the depths of its tips. First of all, the familiar function *flatten*, which takes a tree and returns its tips in left-to-right order, can be written as a fold:

$$\begin{aligned}&\mathit{flatten} :: \mathit{Tree}\ A \to \mathit{List}^+\ A\\ &\mathit{flatten} = \mathit{foldtree}\ (+\!\!+)\ \mathit{wrap}\end{aligned}$$

Here $\mathit{wrap}\ x = [x]$ wraps an item into a singleton list.

A tree of integers is *well-formed* if one can assign to it a *level*, where the level of a tip is the number at the tip, and the level of a non-tip is defined only if its two subtrees have the same level, in which case it is one less than the levels. The partial function *level* can be defined by:

$$\begin{aligned}&\mathit{level} :: \mathit{Tree\ Int} \nrightarrow \mathit{Int}\\ &\mathit{level} = \mathit{foldtree\ up\ id}\\ &\qquad\quad \mathbf{where}\ \mathit{up}\ a\ b = \mathbf{if}\ \ a == b\ \mathbf{then}\ a - 1\end{aligned}$$

Note that the **if** clause in the definition of *up* has only one branch. Therefore, *level* is a partial function which only returns a value for a tree when its left and right subtrees have been assigned the same level.

We call a tree *well-formed* if it is in the domain of *level*. Our problem can thus be specified by

$$\mathit{build} = \mathit{dom\ level} \cdot \mathit{flatten}^\circ$$

We have generalised the problem a little, allowing the level number of the resulting tree to be other than zero.

The relation $\mathit{flatten}^\circ$ maps a list to an arbitrary tree that flattens to the list. For a given list, there will be many such trees. The coreflexive *dom level* acts as a filter picking those that are well-formed. Our specification is therefore an instance of the "generator – filter" paradigm that recurs frequently in functional programming.

Now we have got the problem specification, we are left with two problems: how to compute $\mathit{flatten}^\circ$, and how to fuse *dom level* into the computation.

4.1 Building a Tree with a Fold

Our aim is to apply the converse-of-a-function theorem to invert *flatten*. We need a pair of relations $\mathit{one} :: A \leadsto \mathit{Tree}\ A$ and $\mathit{add} :: A \to \mathit{Tree}\ A \leadsto \mathit{Tree}\ A$ that are jointly surjective and satisfy

$$\begin{aligned}&\mathit{flatten}\ (\mathit{one}\ a) &&= [a]\\ &\mathit{flatten}\ (\mathit{add}\ a\ x) &&= a : \mathit{flatten}\ x\end{aligned}$$

Look at the second equation. It says that if we have a tree x which flattens to some list as, the relation *add* must be able to create a new tree y out of a and x such that y flattens to $a : as$. One way to do that is illustrated in Fig. 4. We divide the left spine of x in two parts, move down the lower part for one level, and attach a to the end.

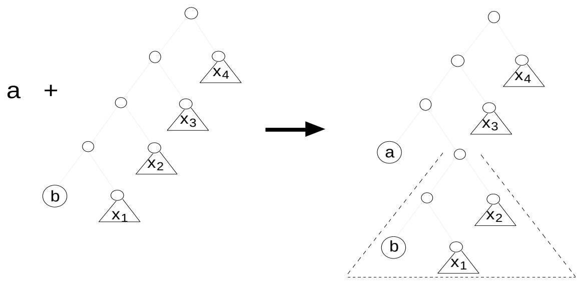

Fig. 4. Adding a new node to a tree

To facilitate this operation, we introduce an alternative *spine representation*. A tree is represented by the list of subtrees along the left spine, plus the left-most tip. The function *roll* converts a spine back into a single tree, and is in fact an isomorphism between *Spine A* and *Tree A*.

$$\textbf{type}\ \mathit{Spine}\ A = A \times \mathit{List}(\mathit{Tree}\ A)$$

$$\begin{aligned} &\mathit{roll} && :: \mathit{Spine}\ A \to \mathit{Tree}\ A \\ &\mathit{roll}(a, x) && = \mathit{foldl}\ \mathit{Bin}\ (\mathit{Tip}\ a)\ x \end{aligned}$$

The advantage of this representation is that we can trace the spine upward from the left-most leaf, rather than downwards from the root. As we will see in the end of the next section, this is necessary for an efficient algorithm.

The function *flatten* · *roll* flattens a spine tree. Our task now is to invert it as a fold. We need a pair of relations $\mathit{one} :: A \leadsto \mathit{Spine}\ A$ and $\mathit{add} :: A \to \mathit{Spine}\ A \leadsto \mathit{Spine}\ A$ satisfying

$$\mathit{flatten}\,(\mathit{roll}\,(\mathit{one}\ a)) = [a] \tag{3}$$

$$\mathit{flatten}\,(\mathit{roll}\,(\mathit{add}\ a\,(b, \mathit{xs}))) = a : \mathit{flatten}\,(\mathit{roll}\,(b, \mathit{xs})) \tag{4}$$

We claim that the following definition for *one* and *add* does the job:

$$\begin{aligned} &\mathit{one}\ a && = (a, [\,]) \\ &\mathit{add}\ a\,(b, \mathit{xs}) && = (a, \mathit{roll}\,(b, \mathit{ys}) : \mathit{zs}) \\ & && \quad \textbf{where}\ \mathit{ys} \mathbin{+\!\!+} \mathit{zs} = \mathit{xs} \end{aligned}$$

The non-deterministic pattern in the definition of *add*, dividing the list *xs* into two parts, indicates that *add* is a relation. The relations *one* and *add* are jointly surjective because *roll*, being an isomorphism, is surjective; thus, given any spine tree (a, ws), either ws is empty, in which case it is covered by *one* a, or there always exists a pair of (b, ys) such that they roll into the head of ws, in which case (a, ws) would be one of the results of $add\ a\ (b, ys \mathbin{+\!\!+} tail\ ws)$.

It is clear that the function *one* satisfies (3). To show that *add* satisfies (4), we will need the following fact, whose proof is left to the diligent reader:

$$flatten(roll\ (a, xs)) = a : concat(map\ flatten\ xs) \tag{5}$$

Now we will show that *add* satisfies (4):

$$\begin{array}{ll}
& a : flatten(roll\ (b, ys \mathbin{+\!\!+} zs)) \\
= & \{(5)\} \\
& a : b : concat(map\ flatten\ (ys \mathbin{+\!\!+} zs)) \\
= & \{concat \text{ and } map \text{ distributes over } \mathbin{+\!\!+}\} \\
& a : b : concat(map\ flatten\ ys) \mathbin{+\!\!+} concat(map\ flatten\ zs) \\
= & \{(5)\} \\
& a : flatten(roll\ (b, ys)) \mathbin{+\!\!+} concat(map\ flatten\ zs) \\
= & \{\text{definition of } concat \text{ and } map\} \\
& a : concat(map\ flatten\ (roll(b, ys) : zs)) \\
= & \{(5)\} \\
& flatten\ (roll\ (a, roll(b, ys) : zs)) \\
= & \{\text{definition of } add\} \\
& flatten\ (roll\ (add\ a\ (b, ys \mathbin{+\!\!+} zs)))
\end{array}$$

Thus $(flatten \cdot roll)^\circ = foldrn\ add\ one$ by Theorem 1.

4.2 The Derivation

Having inverted $flatten \cdot roll$, we can start the derivation:

$$\begin{array}{ll}
& build \\
= & \{\text{definition}\} \\
& dom\ level \cdot flatten^\circ \\
= & \{roll \text{ is an isomorphism}\} \\
& dom\ level \cdot (flatten \cdot roll \cdot roll^\circ)^\circ \\
= & \{\text{converse is contravariant}\} \\
& dom\ level \cdot roll \cdot (flatten \cdot roll)^\circ \\
= & \{\text{inverting } flatten \cdot roll \text{ as in the last section}\} \\
& dom\ level \cdot roll \cdot foldrn\ add\ one \\
= & \{\text{since } dom\ R \cdot f = f \cdot dom\ (R \cdot f), \text{ let } wellform = dom\ (level \cdot roll)\} \\
& roll \cdot wellform \cdot foldrn\ add\ one
\end{array}$$

Except for the introduction of *roll*, the derivation so far is mostly mechanical. Whereas *dom level* checks whether a tree is well-formed, *wellform* is its counterpart defined on spine trees. Intuitively, a spine tree (b, xs) is well-formed if and only if all the trees in xs are well-formed, and the first tree in xs has a level number b, the second tree has a level number $b - 1$, and so on.

As $roll \cdot wellform$ is a partial function, it can be easily implemented in Haskell. However, *add* is still a relation. If we can fuse *wellform* into the fold and thereby refine *add* to a partial function, the whole expression will be implementable.

However, *wellform* is a rather strong condition to enforce. It is not possible to maintain this invariant within the fold before and after each application of *add*. It is time to take the second inventive step: to invent a weaker condition. The predicate *decform* holds for a spine tree (b, xs) if the level number of the first tree in xs is at most b and the trees in xs have strictly decreasing level numbers:

$$\begin{aligned} &decform\,(b, xs) &&= leading\; b\; xs \wedge decreasing\,(map\; level\; xs)\\ &leading\; b\; xs &&= null\; xs \vee level(head\; xs) \leq b \end{aligned}$$

Note that the application of *level* to all the trees in xs implicitly states the requirement that all the trees are well-formed.

The predicate *decform* is weaker than *wellform*. We can thus derive:

$$\begin{aligned} & roll \cdot wellform \cdot foldrn\; add\; one\\ = \quad & \{(1)\}\\ & roll \cdot wellform \cdot decform? \cdot foldrn\; add\; one\\ = \quad & \{\text{fold fusion, see below}\}\\ & roll \cdot wellform \cdot foldrn\; add'\; one \end{aligned}$$

The equality established by fold fusion in the last step ensures that no result is lost from the refinement. Fortunately, it can be shown that the following fusion condition is valid:

$$decform? \cdot add\; a = add'\; a \cdot decform?$$

where add' is defined by rolling the given spine tree up to the point when the two left-most trees do not have the same level number:

$$\begin{aligned} &add'\; a\,(b, xs) &&= leading?\,(a, decRoll\,(Tip\; b)\; xs)\\ &decRoll\; x\; [\,] &&= [x]\\ &decRoll\; x\,(y : zs) &&\mid (level\; x == level\; y) = decRoll\,(Bin\; x\; y)\; zs\\ & &&\mid otherwise \qquad\qquad\quad = x : y : zs \end{aligned}$$

The code is shown in Fig. 5. We refine the data structure to avoid recomputing *level* by defining type *SpineI* and maintain the invariant that $level\, x = n$ for all pairs (x, n) along the spine. Constructors *Tip* and *Bin* are lifted accordingly. The function *rollwf* implements $roll \cdot wellform$. The check is performed implicitly by *bin* each time two trees are joined. This algorithm is linear in the number of nodes in the tree, as each call to *join* either stops or builds a new node.

```
data Tree a = Tip a | Bin (Tree a) (Tree a) deriving Show
type SpineI = (Int, [(Tree Int, Int)])

build :: [Int] -> Tree Int
build = rollwf . foldrn add' one

one a = (a,[])

add' a (b,xs) | leading (a,zs) = (a,zs)
                where zs = decRoll (tip b) xs
decRoll x [] = [x]
decRoll x (y:zs) | level x == level y = decRoll (bin x y) zs
                 | otherwise          = x:y:zs
leading (a,xs) = level (head xs) <= a

tip a = (Tip a, a)
bin (x,m) (y,n) | m == n = (Bin x y, m-1)
level = snd

rollwf :: SpineI -> Tree Int
rollwf (b,xs) = fst (foldl bin (tip b) xs)

foldrn f g [x] = g x
foldrn f g (a:x) = f a (foldrn f g x)
```

Fig. 5. Code for rebuilding a tree from the depths of its tips

5 Building Trees with Minimum Height

Next we consider the second problem of building a tree with minimum height. A linear-time algorithm to this problem has been proposed in [4], but here we will demonstrate how a similar algorithm can be derived.

Given a tip-valued binary tree whose tip values represent the heights of trees, the function computing the height of the combined tree can be defined as a fold in the obvious way:

$$\begin{aligned}&\mathit{height} :: \mathit{Tree\ Int} \to \mathit{Int}\\&\mathit{height} = \mathit{foldtree\ ht\ id}\\&\qquad \textbf{where}\ \ \mathit{ht}\ a\ b = (a \sqcup b) + 1\end{aligned}$$

where $\sqcup$ returns the larger of its two arguments. The problem is thus to find, among all the trees which flatten to the given list, one for which *height* yields the minimal value. The specification needs to consider all possible results. For that we need the power transpose operator Λ, also called the *breadth* function.

The power transpose operator Λ converts a relation $R :: A \leadsto B$ to a function $\Lambda R :: A \to \mathit{Set}\,B$. For $a \in A$, the set $(\Lambda R) a$ contains all values in B to which a is mapped:

$$(\Lambda R) a = \{b \mid (a,b) \in R\}$$

To extract a value from a set we need the relation $\mathit{min}\,(\preceq) :: \mathit{Set}\,A \leadsto A$, defined by

$$(xs, x) \in \mathit{min}\,(\preceq) \equiv x \in xs \wedge (\forall y : y \in xs : x \preceq y)$$

For this definition to be of any use, $(\preceq)$ has to be a *connected preorder*, meaning an ordering which is reflexive, transitive, and compares everything of the correct type. The relation $\mathit{min}\,(\preceq)$ will not in general be a function because a preorder is not necessarily anti-symmetric.

For our problem, define $(\preceq)$ to be a comparison between the heights of two trees:

$$x \preceq y \equiv \mathit{height}\,x \leq \mathit{height}\,y$$

Our problem can then be specified as:

$$\mathit{bmh} = \mathit{min}\,(\preceq) \cdot \Lambda(\mathit{flatten}^{\circ})$$

The reasoning in Sect. 4.1 can be reused: we introduce the spine representation and invert *flatten* to $\mathit{roll} \cdot \mathit{foldrn}\ \mathit{add}\ \mathit{one}$. Furthermore, *roll* can be factored out of Λ, and we get:

$$\mathit{bmh} = \mathit{roll} \cdot \mathit{min}\,(\preceq') \cdot \Lambda(\mathit{foldrn}\ \mathit{add}\ \mathit{one})$$

where $xs \preceq' ys \equiv \mathit{roll}\,xs \preceq \mathit{roll}\,ys$, i.e., $(\preceq')$ is the counterpart of $(\preceq)$ defined on spine trees.

Since the relation *add* has $n+1$ choices when given a spine tree of length n, the above specification generates an exponential number of trees. To eliminate the non-determinism in *add* and thereby improve the efficiency, we make use of the following *greedy theorem*. Presented below is a special case of the more general version proved in [5].

Theorem 2 (The Greedy Theorem (For Non-empty Lists)). Let $\mathit{base} :: A \leadsto A$ and $\mathit{step} :: A \to B \leadsto B$ be two relations. If *step* is monotonic on a connected preorder $(\trianglelefteq)$, that is,

$$(x \trianglelefteq y\ \wedge\ (y, y') \in \mathit{step}\ a) \Rightarrow (\exists x' : (x, x') \in \mathit{step}\ a : x' \trianglelefteq y') \tag{6}$$

then we have

$$\mathit{foldrn}\,(\mathit{min}\,(\trianglelefteq) \cdot \Lambda \mathit{step})\,(\mathit{min}\,(\trianglelefteq) \cdot \Lambda \mathit{base}) \subseteq \mathit{min}\,(\trianglelefteq) \cdot \Lambda(\mathit{foldrn}\ \mathit{step}\ \mathit{base})$$

Informally, the monotonicity condition means that a worse partial solution in some stage of the fold always gives a worse result. If this condition holds, then at each stage of the fold we need only retain one of the best results computed so far. Thus $\mathit{min}\,(\trianglelefteq)$ gets promoted into *foldrn*.

Had *add* satisfied the monotonicity condition (6) with respect to $(\preceq')$, we could apply the greedy theorem. However, that is not true: a tree with the smallest height does not always remain the smallest after being extended by *add*.

This is where human ingenuity gets involved. Fortunately, *add* is monotonic on a stronger ordering. We define:

$$heights\ (a, xs) = (reverse \cdot map\ height \cdot scanl\ Bin\ (Tip\ a))\ xs$$

In words, *heights* returns a list of heights along the left spine, starting from the root. The relation *add* is then monotonic on $\ll$, defined by:

$$x \ll y \equiv heights\ x \trianglelefteq heights\ y$$

where $(\trianglelefteq)$ is the lexicographic ordering on sequences. This choice does make sense: to ensure monotonicity, we need to optimise not only the whole tree, but also all the subtrees on the left spine. The proof that *add* is monotonic on $(\ll)$, however, is quite involved and will not be presented here. The reader is referred to [6] for more detailed discussion.

Applying the greedy theorem, we get:

$$bmh = roll \cdot foldrn\ (min\ (\trianglelefteq) \cdot \Lambda add)\ (min\ (\trianglelefteq) \cdot \Lambda one)$$

Since *one* is a function, $min\ (\ll) \cdot \Lambda one = one$. With some analysis, we can further optimise $min\ (\ll) \cdot \Lambda add$. Let $(b, [x_1, x_2, \cdot\cdot, x_n])$ be the spine tree to which we are about to insert a value a. It can be shown that in order to construct the best tree under the ordering $(\ll)$, we do not need to actually check through all the $n + 1$ possibilities. We can always break the list between x_i and x_{i+1} such that i is the smallest index such that $a < height\ x_{i+1}$ and $height\ (roll\ (b, [x_1, x_2, \cdot\cdot, x_i])) < height\ x_{i+1}$. We will also omit the details and refer the interested readers to [6].

The code is shown in Fig. 6. As in the first problem, we annotate each tree with its height to avoid re-computation. By the same argument as that in the end of Sect. 4.2, this algorithm is also linear in the number of nodes in the tree.

6 Breadth-First Labelling

To breadth-first label a tree with respect to a given list is to label the nodes in the tree in breadth-first order, using the values in the list. Jones and Gibbons [13] proposed a neat solution to this problem, based on a clever use of cyclic data structures. The problem was recently revisited by Okasaki [22]. We are going to show how Okasaki's algorithm can be derived using the converse-of-a-function theorem.

Recall the data structure for binary trees:

$$\mathbf{data}\ Tree\ A = Tip\ A \mid Bin\ A\ (Tree\ A)\ (Tree\ A)$$

The queue-based algorithm for breadth-first traversal is well-known:

$$\begin{aligned} &bft\ \ :: Tree\ A \to List\ A \\ &bft\ x = bftF\ [x] \end{aligned}$$

$$\mathbf{type}\ Forest\ A = List (Tree\ A)$$

```
bmh :: [Int] -> (Tree Int, Int)
bmh = roll . foldrn minadd one

one a = (a,[])

minadd :: Int -> SpineI -> SpineI
minadd a (b,xs) = (a, minsplit (tip b) xs)
  where minsplit x [] = [x]
        minsplit x (y:xs) | a < height y
                            && height x < height y = x:y:xs
                          | otherwise = minsplit (bin x y) xs

tip a = (Tip a, a)
bin (x,a) (y,b) = (Bin x y, ht a b)
height = snd

ht a b = (a `max` b) + 1

roll :: SpineI -> (Tree Int, Int)
roll (a,x) = foldl bin (tip a) x
```

Fig. 6. Code for building trees with minimum height

$$\begin{array}{ll}
\mathit{bftF} & :: \mathit{Forest}\ A \rightarrow \mathit{List}\ A \\
\mathit{bftF}\,[\,] & = [\,] \\
\mathit{bftF}\,(\mathit{Tip}\ a : \mathit{xs}) & = a : \mathit{bftF}\ \mathit{xs} \\
\mathit{bftF}\,(\mathit{Bin}\ a\ x\ y : \mathit{xs}) & = a : \mathit{bftF}\,(\mathit{xs} \mathbin{+\!\!+} [x, y])
\end{array}$$

To perform the labelling, we use the following partial function *zipTree*:

$$\begin{array}{ll}
\mathit{zipTree} & :: \mathit{Tree}\ A \rightarrow \mathit{Tree}\ B \mathrel{+\!\!\!\rightarrow} \mathit{Tree}\,(A \times B) \\
\mathit{zipTree}\,(\mathit{Tip}\ a)\,(\mathit{Tip}\ b) & = \mathit{Tip}\,(a, b) \\
\mathit{zipTree}\,(\mathit{Bin}\ a\ x\ y)\,(\mathit{Bin}\ b\ u\ v) & = \mathit{Bin}\,(a, b)\,(\mathit{zipTree}\ x\ u)\,(\mathit{zipTree}\ y\ v)
\end{array}$$

Breadth-first labelling of a tree x can then be seen as zipping x with another tree y, in which the breadth-first traversal of y is a prefix of the given list *as*:

$$\begin{array}{ll}
\mathit{bfl} & :: \mathit{List}\ A \rightarrow \mathit{Tree}\ B \mathrel{+\!\!\!\rightarrow} \mathit{Tree}\,(A \times B) \\
\mathit{bfl}\ \mathit{as}\ x & = \mathit{zipTree}\ y\ x \\
& \quad \textbf{where}\,(\mathit{bft}\ y) \mathbin{+\!\!+} \mathit{bs} = \mathit{as}
\end{array}$$

Equivalently,

$$\begin{array}{ll}
\mathit{bfl}\ \mathit{as}\ x & = \mathit{zipTree}\,((\mathit{bft}^{\circ} \cdot \mathit{prefix})\ \mathit{as})\ x \\
& = (\mathit{zipTree} \cdot \mathit{bft}^{\circ} \cdot \mathit{prefix})\ \mathit{as}\ x
\end{array}$$

This completes the specification. The relation *prefix* non-deterministically maps a list to one of its finite prefixes. The prefix is then passed to bft°, yet again being non-deterministically mapped to a tree whose breadth-first traversal equals the chosen prefix. It is important that *zipTree* is a partial function which yields a value only when the given two trees are of exactly the same shape. Therefore, the tree composed by $bft^{\circ} \cdot \mathit{prefix}$ can be zipped with the input tree only if it is of the correct size and shape. The partial function *zipTree* plays the role of the filter.

Since breadth-first traversal is an algorithm more naturally defined in terms of queues of trees (or forests) rather than of a single tree, it is reasonable to try to invert *bftF* rather than *bft*. The problem can be rephrased in terms of *bftF*:

$$\mathit{bfl}\ \mathit{as}\ x = \mathit{wrap}^{\circ}\ ((\mathit{zipForest} \cdot \mathit{bftF}^{\circ} \cdot \mathit{prefix})\ \mathit{as}\ [x])$$

Here $\mathit{zipForest} :: \mathit{Forest}\ A \to \mathit{Forest}\ B \mathrel{+\!\!\!\to} \mathit{Forest}\ (A, B)$ is a simple extension of *zipTree* to forests, which, like *zipTree*, is a partial function:

$$\begin{aligned}
&\mathit{zipForest}\ [\,]\ [\,] &&= [\,]\\
&\mathit{zipForest}\ (x : \mathit{xs})\ (y : \mathit{ys}) &&= \mathit{zipTree}\ x\ y : \mathit{zipForest}\ \mathit{xs}\ \mathit{ys}
\end{aligned}$$

Once the decision to focus on *bftF* is made, the rest is mechanical. To invert *bftF*, we are to find *base* and *step* such that

$$\begin{aligned}
&\mathit{bftF}\ \mathit{base} &&= [\,]\\
&\mathit{bftF}\ (\mathit{step}\ a\ \mathit{xs}) &&= a : \mathit{bftF}\ \mathit{xs}
\end{aligned}$$

The value of *base* can only be $[\,]$. The derivation for *step* is not too difficult either. We start with the general case which does not assume any structure in *xs*:

$$\begin{aligned}
&\quad a : \mathit{bftF}\ \mathit{xs}\\
=&\quad \{\text{definition of } \mathit{bftF}\}\\
&\quad \mathit{bftF}\ (\mathit{Tip}\ a : \mathit{xs})
\end{aligned}$$

Therefore $\mathit{step}\ a\ \mathit{xs}$ might contain $(\mathit{Tip}\ a : \mathit{xs})$ as one of the possible values. But this choice alone does not make *step* jointly surjective with $[\,]$, since it cannot generate a forest with a non-tip tree as its head. We therefore consider the case when *xs* contains contains more than two trees:

$$\begin{aligned}
&\quad a : \mathit{bftF}(\mathit{xs} \mathbin{+\!\!+} [x, y])\\
=&\quad \{\text{definition of } \mathit{bftF}\}\\
&\quad \mathit{bftF}\ (\mathit{Bin}\ a\ x\ y : \mathit{xs})
\end{aligned}$$

Therefore we define *step* to be:

$$\begin{aligned}
&\mathit{step} &&:: A \to \mathit{Forest}\ A \leadsto \mathit{Forest}\ A\\
&\mathit{step}\ a\ \mathit{xs} &&= (\mathit{Tip}\ a : \mathit{xs})\ \Box\ (\mathit{Bin}\ a\ x\ y : \mathit{xs}')\\
& &&\quad \mathbf{where}\ (\mathit{xs}' \mathbin{+\!\!+} [x, y]) = \mathit{xs}
\end{aligned}$$

Since a forest either begins with a tip tree, begins with a non-tip tree, or is empty, *step* is jointly surjective with $[\,]$. The converse of *bftF* is thus constructed as $\mathit{bftF}^{\circ} = \mathit{foldr}\ \mathit{step}\ [\,]$.

Knowing that $\mathit{bftF}^{\circ} :: \mathit{List}\ A \leadsto \mathit{Forest}\ A$ is a fold, we can fuse *zipForest* and bftF° as a fold :

$$\begin{array}{l}
\mathit{zipForest} \cdot \mathit{bftF}^{\circ} = \mathit{foldr}\ \mathit{revZip}\ \mathit{stop} \\
\qquad \textbf{where}\ \mathit{stop}\,[\,] = [\,] \\
\qquad\qquad \mathit{revZip}\ a\,f\ (\mathit{Tip}\ b : \mathit{ts}) = \mathit{Tip}\,(a,b) : f\ \mathit{ts} \\
\qquad\qquad \mathit{revZip}\ a\,f\ (\mathit{Bin}\ b\ u\ v : \mathit{ts}) = \mathit{Bin}\,(a,b)\ x\ y : \mathit{ys} \\
\qquad\qquad\quad \textbf{where}\ \mathit{ys} \mathbin{+\!\!+} [x,y] = f\ (\mathit{ts} \mathbin{+\!\!+} [u,v])
\end{array}$$

The expression $\mathit{zipForest} \cdot \mathit{bftF}^{\circ}$ has type $\mathit{List}\ A \rightarrow \mathit{Forest}\ B \leadsto \mathit{Forest}\,(A \times B)$. Consider $(\mathit{zipForest} \cdot \mathit{bftF}^{\circ})\,x$ where x is a list of labels. Constructors building x are replaced by *revZip* and *stop*, yielding a relation mapping an unlabelled forest to a labelled forest. A pattern matching error will be invoked by *stop* if x is too short, and by *revZip* if x is too long. Applying fold fusion again to fuse $\mathit{zipForest} \cdot \mathit{bftF}^{\circ}$ with *prefix* in effect adds another case for *revZip*, that is, $\mathit{revZip}\ a\,f\ [\,] = [\,]$, which cuts the list of labels when the forest is consumed earlier than the list. Still, the list of labels cannot be too short.

The resulting code is shown in Fig. 7. It can be made linear if we use an implementation of deques supporting constant-time addition and deletion [8,21] for both the input and output of *revzip*. For clarity, we will just leave it as it is. It is nothing more than an adaption of Okasaki's algorithm in [22] to lists. In his paper, Okasaki raised the question why most people did not come up with this algorithm but instead appealed to more complicated approaches. Our answer is because they did not know the converse-of-a-function theorem.

```
data Tree a = Tip a | Bin a (Tree a) (Tree a) deriving Show

bfl :: [a] -> Tree b -> Tree (a,b)
bfl xs = unwrap . foldr revzip stop xs . wrap
 where stop [] = []
       revzip a f [] = []
       revzip a f (Tip b:ts) = Tip (a,b) : f ts
       revzip a f (Bin b u v :ts) = Bin (a,b) x y : ys'
          where ys = f (ts ++ [u,v])
                (ys',x,y) = (init (init ys), last (init ys), last ys)

wrap a = [a]
unwrap [a] = a
```

Fig. 7. Code for breadth-first labelling

7 The Hylomorphism Theorem

By definition, a *hylomorphism* is the composition of a fold with the converse of a fold. The hylomorphism $(\![R]\!)_{\mathsf{F}} \cdot (\![S]\!)_{\mathsf{F}}{}^{\circ}$ can be characterised as the least solution for X of the inequation $R \cdot \mathsf{F}X \cdot S^\circ \subseteq X$. In other words, we have:

$$(\!|R|\!)_{\mathsf{F}} \cdot (\!|S|\!)_{\mathsf{F}}{}^\circ \subseteq X \Leftarrow R \cdot \mathsf{F}X \cdot S^\circ \subseteq X \tag{7}$$

The aim of this section is to prove the following generalisation of Theorem 1:

Theorem 3 (Hylomorphism Theorem). Let $S :: A \leadsto B$ be a simple relation. If relation $R :: \mathsf{F}(C, A) \leadsto A$ and function $f :: \mathsf{F}(C, B) \rightarrow B$ are such that (i) $dom\, S = ran\, R$; (ii) $S \cdot R \subseteq f \cdot \mathsf{F}S$; and (iii) $\delta_{\mathsf{F}} \cdot R^\circ$ is inductive, then

$$S = (\!|f|\!)_{\mathsf{F}} \cdot (\!|R|\!)_{\mathsf{F}}{}^\circ$$

In words, Theorem 3 gives conditions under which a simple relation can be expressed as a hylomorphism. The new ingredients in Theorem 3 are the *membership* relation δ_{F} of a relator F, and the notion of an *inductive* relation. Both are described below in Sect. 7.1. The main proof is given in Sect. 7.2.

Theorem 1 follows as a special instance of Theorem 3 by taking $f = \alpha$ and S to be an entire relation as well as a simple one, that is, a function. An entire relation S is one for which $dom\, S = id$, so condition (i) translates to the requirement that R be a surjective relation. In Sect. 7.2, we will prove that condition (iii) holds if both (i) and (ii) do and if $\delta_{\mathsf{F}} \cdot f^\circ$ is inductive. Fact 1 below gives us that $\delta_{\mathsf{F}} \cdot \alpha_{\mathsf{F}}{}^\circ$ is inductive. Since $(\!|\alpha_{\mathsf{F}}|\!)_{\mathsf{F}} = id$, we then obtain the result $S = (\!|R|\!)_{\mathsf{F}}{}^\circ$. Taking converses, this is the conclusion of Theorem 1.

7.1 Inductivity and Membership

We say that a relation admits induction, or is inductive, if we can use it to perform induction[11]. Formally, inductivity is defined by:

Definition 1 (Inductivity). A relation $R :: A \leadsto A$ is inductive if for all $X :: B \leadsto A$,

$$R \backslash X \subseteq X \Rightarrow \Pi \subseteq X$$

Here Π denotes the largest relation of its type, and the left division operator ($\backslash$) is defined by the Galois connection:

$$S \subseteq R \backslash T \equiv R \cdot S \subseteq T$$

The definition can be translated to the point level to aid understanding. It says that R is inductive if the property

$$(\forall c :: (c, a) \in R \Rightarrow (c, b) \in X) \Rightarrow (a, b) \in X$$

where a and b are arbitrary, implies X contains all the pairs of its type. As an example, take R to be $<$, the ordering on natural numbers, and $P\, a = (a, b) \in X$ to be some property we want to prove for all a and some fixed b. The definition specialises to the claim that if

$$(\forall c :: c < a \Rightarrow P\, c) \Rightarrow P\, a$$

then $P\, a$ holds for all natural numbers a. Thus we can see that inductivity captures the principle of induction.

Three facts we will need are the following:

Fact 1 The relation $\delta_{\mathsf{F}} \cdot \alpha_{\mathsf{F}}{}^{\circ}$ is inductive.

Fact 2 If R is inductive and $S \subseteq R$, then S is inductive.

Fact 3 If R is inductive, so is $S^{\circ} \cdot R \cdot S$ for any simple relation S.

The other concept we need, due to Hoogendijk and de Moor [17], is the membership relation of a datatype. For example, a membership relation δ_{List} for lists can be specified informally by:

$$(a, [a_0, a_1, \ldots a_n]) \in \delta_{List} \equiv (\exists i :: a = a_i)$$

The formal definition of membership is not at all intuitive, and we refer the reader to [17] for more discussion. A fact about membership we will use is that it is a lax natural transformation, which is to say,

$$\delta_{\mathsf{F}} \cdot \mathsf{F}R \subseteq R \cdot \delta_{\mathsf{F}} \qquad (8)$$

for all R.

7.2 The Proof

We begin by reciting some basic facts about a simple relation S. First, for any X and Y,

$$S \cdot X \subseteq Y \Leftarrow X \subseteq S^{\circ} \cdot Y \qquad (9)$$

The proof is immediate from the fact that $S \cdot S^{\circ} \subseteq id$. More generally,

$$S \cdot X \subseteq Y \equiv dom\, S \cdot X \subseteq S^{\circ} \cdot Y \qquad (10)$$

When S is also entire, i.e., $dom\, S = id$, this reduces to the usual shunting rule for functions. The following shunting lemma will be used a number of times:

Lemma 1. Let S be simple and suppose R satisfies (i) $ran\, R \subseteq dom\, S$, and (ii) $S \cdot R \subseteq f \cdot \mathsf{F}S$. Then $R \subseteq S^{\circ} \cdot f \cdot \mathsf{F}S$.

Proof.

$$\begin{array}{ll}
& R \subseteq S^{\circ} \cdot f \cdot \mathsf{F}S \\
\equiv & \quad \{\text{using } R = ran\, R \cdot R\} \\
& ran\, R \cdot R \subseteq S^{\circ} \cdot f \cdot \mathsf{F}S \\
\Leftarrow & \quad \{\text{assumption (i)}\} \\
& dom\, S \cdot R \subseteq S^{\circ} \cdot f \cdot \mathsf{F}S \\
\equiv & \quad \{\text{shunting (10)}\} \\
& S \cdot R \subseteq f \cdot \mathsf{F}S
\end{array}$$

□

Now comes the main proof of Theorem 3. In one direction, the proof is relatively easy:

$$\begin{array}{cl}
& (\!|f|\!) \cdot (\!|R|\!)^\circ \subseteq S \\
\Leftarrow & \{(7)\} \\
& f \cdot \mathsf{F}S \cdot R^\circ \subseteq S \\
\Leftarrow & \{\text{shunting (9), since } f \cdot \mathsf{F}S \text{ simple if } S \text{ is}\} \\
& R^\circ \subseteq (f \cdot \mathsf{F}S)^\circ \cdot S \\
\equiv & \{\text{converses; Lemma 1}\} \\
& \mathit{true}
\end{array}$$

For the other direction, we reason:

$$\begin{array}{cl}
& S \subseteq (\!|f|\!) \cdot (\!|R|\!)^\circ \\
\equiv & \{\text{shunting (10)}\} \\
& \mathit{dom}\, S \subseteq S^\circ \cdot (\!|f|\!) \cdot (\!|R|\!)^\circ \\
\equiv & \{\text{assumption (i): } \mathit{dom}\, S = \mathit{ran}\, R\} \\
& \mathit{ran}\, R \subseteq S^\circ \cdot (\!|f|\!) \cdot (\!|R|\!)^\circ \\
\Leftarrow & \{\text{claim : } \mathit{ran}\, R \subseteq \mathit{ran}\, (\!|R|\!)\} \\
& \mathit{ran}\, (\!|R|\!) \subseteq S^\circ \cdot (\!|f|\!) \cdot (\!|R|\!)^\circ \\
\Leftarrow & \{(2)\text{: } \mathit{ran}\, X \subseteq X \cdot X^\circ\} \\
& (\!|R|\!) \cdot (\!|R|\!)^\circ \subseteq S^\circ \cdot (\!|f|\!) \cdot (\!|R|\!)^\circ \\
\Leftarrow & \{\text{monotonicity}\} \\
& (\!|R|\!) \subseteq S^\circ \cdot (\!|f|\!) \\
\equiv & \{\text{converses and shunting, since } (\!|f|\!) \text{ is a function if } f \text{ is}\} \\
& (\!|R|\!) \cdot (\!|f|\!)^\circ \subseteq S \\
\equiv & \{\text{proved above}\} \\
& \mathit{true}
\end{array}$$

We still need to prove the claim that $\mathit{ran}\, R \subseteq \mathit{ran}\, (\!|R|\!)$ under the given conditions. We will appeal to the following lemma[1], whose proof is postponed to the appendix.

Lemma 2. If $\delta_{\mathsf{F}} \cdot R^\circ$ is inductive and $\mathit{dom}\, R \subseteq \mathsf{F}(\mathit{ran}\, R)$, then

$$\mathit{ran}\, (R \cdot \mathsf{F}C) \subseteq C \Rightarrow \mathit{ran}\, R \subseteq C \tag{11}$$

for coreflexives C.

To check that $\mathit{dom}\, R \subseteq \mathsf{F}(\mathit{ran}\, R)$, we reason:

$$\begin{array}{cl}
& \mathit{dom}\, R \\
\subseteq & \{\text{Lemma 1 : } R \subseteq S^\circ \cdot f \cdot \mathsf{F}S\}
\end{array}$$

[1] Property (11) is called F-inductivity in [11].

$$\begin{array}{ll} & dom\,(S^\circ \cdot f \cdot \mathsf{F}S) \\ \subseteq & \{\text{since } dom\,(X \cdot Y) \subseteq dom\,Y\} \\ & dom\,(\mathsf{F}S) \\ \subseteq & \{\text{relators preserve domains: } dom\,(\mathsf{F}S) = \mathsf{F}(dom\,S)\} \\ & \mathsf{F}(dom\,S) \\ = & \{\text{by assumption (i): } dom\,S = ran\,R\} \\ & \mathsf{F}(ran\,R) \end{array}$$

That relators preserve domains is given in [5] as an exercise on tabulation.

Finally, the left-hand side of property (11), namely $ran\,(R \cdot \mathsf{F}C) \subseteq C$, actually holds for all R when C is $ran\,(\!|R|\!)$.

$$\begin{array}{ll} & ran\,(\!|R|\!) \\ = & \{\text{definition of } (\!|R|\!)\} \\ & ran\,(R \cdot \mathsf{F}(\!|R|\!) \cdot \alpha^\circ) \\ = & \{\text{since } ran\,(X \cdot Y) = ran\,(X \cdot ran\,Y)\} \\ & ran\,(R \cdot ran\,(\mathsf{F}(\!|R|\!) \cdot ran\,(\alpha^\circ))) \\ = & \{\text{since } ran\,(\alpha^\circ) = id\} \\ & ran\,(R \cdot ran\,(\mathsf{F}(\!|R|\!))) \\ = & \{\text{relators preserve domains}\} \\ & ran\,(R \cdot \mathsf{F}(ran\,(\!|R|\!))) \end{array}$$

We therefore conclude that $ran\,R \subseteq ran\,(\!|R|\!)$ under the given assumptions.

We will now prove a lemma which shows that condition (iii) of Theorem 3 holds if conditions (i) and (ii) do and if $\delta_\mathsf{F} \cdot f^\circ$ is inductive. It is this lemma that establishes the connection between Theorem 1 and Theorem 3.

Lemma 3. The relation $\delta_\mathsf{F} \cdot R^\circ$ is inductive if (i) $ran\,R \subseteq dom\,S$; (ii) $S \cdot R \subseteq f \cdot \mathsf{F}S$; and (iii) $\delta_\mathsf{F} \cdot f^\circ$ is inductive.

Proof. We reason:

$$\begin{array}{ll} & \delta_\mathsf{F} \cdot R^\circ \\ \subseteq & \{\text{Lemma 1, converse}\} \\ & \delta_\mathsf{F} \cdot \mathsf{F}S^\circ \cdot f^\circ \cdot S \\ \subseteq & \{(8)\} \\ & S^\circ \cdot \delta_\mathsf{F} \cdot f^\circ \cdot S \end{array}$$

Since $\delta_\mathsf{F} \cdot f^\circ$ is inductive, so is $S^\circ \cdot \delta_\mathsf{F} \cdot f^\circ \cdot S$ by Fact 3. We then obtain that $\delta_\mathsf{F} \cdot R^\circ$ is inductive by Fact 2.

□

8 Conclusions and Related Work

The idea of program inversion can be traced back to Dijkstra [10]. However, given the importance of inversion as a specification technique, relatively few papers have been devoted to the topic, and of those that have, most deal with program inversion in an imperative setting. A program is inverted by running it "backwards" and the challenging part is when we encounter a branch or a loop [24]. The classic example was to construct a binary tree given its inorder and preorder traversal [14,15,7,26,25]. Inversion of functional programs has received even less attention. Most published results (e.g. [20,16]) are based on a "compositional" approach, which is essentially the same as its imperative counterpart: if h is defined by $f \cdot g$, then $h^\circ = g^\circ \cdot f^\circ$. The inverse of f and g are then recursively constructed until we reach primitives whose inverses are pre-defined. Efforts have also been made to automate the process, such as in [1]. This paper also contains a detailed bibliography.

The converse-of-a-function theorem, however, takes a non-compositional approach to invert a function. To invert a function, what matters is not how it is defined but what properties it satisfies. This technique is not new. Similar techniques have been adopted in, for example, [19] and [23]. However, to the best our knowledge, it was de Moor [5,9] who first presented the technique as a theorem, suggesting a wider range of application. The problem dealt with in [9] was precedence parsing, leading to a derivation of Floyd's algorithm.

We have applied the converse-of-a-function theorem to three examples. The inverted function is usually a non-deterministic fold. To make it useful, it is often composed before some other function which acts as a filter. The fold fusion theorem is then applied to fuse the filter into the fold to remove the non-determinism, refining the specification to an implementable function. This pattern of derivation turned out to be useful in solving many problems.

One natural question is how widely the theorem can be applied. In other words, how to determine whether the converse-of-a-function theorem can be applied a particular function. Part of the answer is given in [12]: if the converse of a function can be written as a fold, the function itself must be an unfold. The necessary and sufficient conditions for a function to be an unfold given in [12] can thus be used as a test before applying the converse-of-a function theorem.

We have not fully exploited the generality of Theorem 3. It can potentially be very useful since it allows the functor F, which determines the pattern of recursion, to be independent from the input and output types. A much wider class of algorithms can thus be covered. However, the theorem itself offers no clue how F and f could be chosen. It is therefore less useful for program derivation and probably more helpful in proving the correctness of known algorithms. We have applied the theorem to some simple cases, such as letting $\mathsf{F}(A, X) = A + X$ to verify some loop-based algorithms. The authors are enthusiastic to see more examples for which the more general theorem is necessary.

Acknowledgements

Thanks are due to members of the Algebra of Programming group in Oxford University Computing Laboratory, to Oege de Moor, for his interest, encouragement and comments throughout the development of this paper, and to Roland Backhouse, who filled in a key step in an earlier proof of the theorem based on F-reductivity and F-inductivity. The authors would also like to thank the anonymous referees for detailed and useful advices.

References

1. S. M. Abramov and R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. C. Backhouse and J. N. F. d. Oliveira, editors, *Mathematics of Program Construction 2000*, number 1837 in Lecture Notes in Computer Science, pages 187–212. Springer-Verlag, 2000.
2. R. C. Backhouse, P. de Bruin, G. Malcolm, T. S. Voermans, and J. van der Woude. Relational catamorphisms. In B. Moller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers B.V., 1991.
3. R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Moller, H. Partsch, and S. A. Schuman, editors, *Formal Program Development. Proc. IFIP TC2/WG 2.1 State of the Art Seminar.*, number 755 in Lecture Notes in Computer Science, pages 7–42. Springer-Verlag, January 1992.
4. R. S. Bird. On building trees with minimum height. *Journal of Functional Programming*, 7(4):441–445, 1997.
5. R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.
6. R. S. Bird, J. Gibbons, and S.-C. Mu. Algebraic methods for optimization problems. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, number 2297 in Lecture Notes in Computer Science, pages 281–307. Springer-Verlag, January 2002.
7. W. Chen and J. T. Udding. Program inversion: more than fun! *Science of Computer Programming*, 15:1–13, 1990.
8. T.-R. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.
9. O. de Moor and J. Gibbons. Pointwise relational programming. In *Proceedings of Algebraic Methodology and Software Technology 2000*, number 1816 in Lecture Notes in Computer Science, pages 371–390. Springer-Verlag, May 2000.
10. E. W. Dijkstra. Program inversion. Technical Report EWD671, Eindhoven University of Technology, 1978.
11. H. Doornbos and R. C. Backhouse. Induction and recursion on datatypes. In B. Moller, editor, *Mathematics of Program Construction, 3rd International Conference*, number 947 in Lecture Notes in Computer Science, pages 242–256. Springer-Verlag, July 1995.
12. J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? In A. Corradini, M. Lenisa, and U. Montanari, editors, *Coalgebraic Methods*

in Computer Science, number 44.1 in Electronic Notes in Theoretical Computer Science, April 2001.
13. J. Gibbons and G. Jones. Linear-time breadth-first tree algorithms: an exercise in the arithmetic of folds and zips. Technical report, University of Auckland, 1993. University of Auckland Computer Science Report No. 71, and IFIP Working Group 2.1 working paper 705 WIN-2.
14. D. Gries. *The Science of Programming*. Springer Verlag, 1981.
15. D. Gries and J. L. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 37–42. Addison Wesley, 1990.
16. P. G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.
17. P. F. Hoogendijk and O. de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, March 2000.
18. T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.
19. E. Knapen. Relational Programming, Program Inversion, and the Derivation of Parsing Algorithms. Master's thesis, Eindhoven University of Technology, 23 November 1993.
20. R. E. Korf. Inversion of applicative programs. In *Proceedings of the Seventh Intern. Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 1007–1009. William Kaufmann, Inc., 1981.
21. C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.
22. C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 131–136. ACM Press, September 2000.
23. C. Pareja-Flores and J. Á. Velázquez-Iturbide. Synthesis of functions by transformations and constraints. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, page 317, Amsterdam, The Netherlands, June 1997. ACM Press.
24. B. J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing Journal*, 9:331–348, 1997.
25. B. Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In *Mathematics of Program Construction 1992*, number 669 in Lecture Notes in Computer Science, pages 291–301. Springer-Verlag, 1993.
26. J. L. van de Snepscheut. Inversion of a recursive tree traversal. Technical Report JAN 171a, California Institute of Technology, May 1991. Available online at `ftp://ftp.cs.caltech.edu/tr/cs-tr-91-07.ps.Z`.

A Proof of Lemma 2

For completeness we will record the proof of Lemma 2, namely that if $\delta_{\mathsf{F}} \cdot R^{\circ}$ is inductive and $dom\, R \subseteq \mathsf{F}(ran\, R)$, then

$$ran\,(R \cdot \mathsf{F}C) \subseteq C \Rightarrow ran\, R \subseteq C$$

for coreflexives C.

To carry out the proof, we need to appeal to some properties left out earlier in the paper. Firstly, there is yet another definition of *ran* via a Galois connection:

$$\mathit{ran}\, R \subseteq S \equiv R \subseteq S \cdot \Pi$$

Once a Galois connection (f, g) is established, many properties follows. Since any coreflexive can be a result of a *ran*, we have

$$C \subseteq D \equiv C \cdot \Pi \subseteq D \cdot \Pi \tag{12}$$

for coreflexives C and D. Secondly, from the definition of left division it follows that

$$(S \cdot R)\backslash T = R\backslash(S\backslash T) \tag{13}$$

It also follows that division is anti-monotonic, that is

$$S \subseteq R \Rightarrow R\backslash T \subseteq S\backslash T$$

Finally, the equality below is proved in [17].

$$\delta_{\mathsf{F}}\backslash(R \cdot S) = \mathsf{F}R \cdot \delta_{\mathsf{F}}\backslash S \tag{14}$$

The proof of Lemma 2 proceeds by proving $\mathit{ran}\, R \subseteq C$, given $\mathit{ran}\,(R \cdot \mathsf{F}C)$, $\mathit{dom}\, R \subseteq \mathsf{F}(\mathit{ran}\, R)$ and $\delta_{\mathsf{F}} \cdot R^{\circ}$ inductive.

Proof.

$$\begin{aligned}
& \mathit{ran}\, R \subseteq C \\
\equiv \quad & \{(12)\} \\
& \mathit{ran}\, R \cdot \Pi \subseteq C \cdot \Pi \\
\equiv \quad & \{\text{division}\} \\
& \Pi \subseteq \mathit{ran}\, R\backslash(C \cdot \Pi) \\
\Leftarrow \quad & \{\text{since } \delta_{\mathsf{F}} \cdot R^{\circ} \text{ inductive}\} \\
& (\delta_{\mathsf{F}} \cdot R^{\circ})\backslash(\mathit{ran}\, R\backslash(C \cdot \Pi)) \subseteq \mathit{ran}\, R\backslash(C \cdot \Pi) \\
\Leftarrow \quad & \{\text{claim 1: } (\delta_{\mathsf{F}} \cdot R^{\circ})\backslash(\mathit{ran}\, R\backslash(C \cdot \Pi)) \subseteq R^{\circ}\backslash\mathsf{F}C \cdot \Pi\} \\
& R^{\circ}\backslash\mathsf{F}C \cdot \Pi \subseteq \mathit{ran}\, R\backslash(C \cdot \Pi) \\
\equiv \quad & \{\text{division}\} \\
& \mathit{ran}\, R \cdot R^{\circ}\backslash(\mathsf{F}C \cdot \Pi) \subseteq C \cdot \Pi \\
\Leftarrow \quad & \{\text{claim 2: } \mathit{ran}\, R \cdot R^{\circ}\backslash(\mathsf{F}C \cdot \Pi) \subseteq R \cdot \mathsf{F}C \cdot \Pi\} \\
& R \cdot \mathsf{F}C \cdot \Pi \subseteq C \cdot \Pi \\
\equiv \quad & \{\text{range}\} \\
& \mathit{ran}\,(R \cdot \mathsf{F}C \cdot \Pi) \subseteq C \\
\equiv \quad & \{\text{since } \mathit{ran}\,(X \cdot Y) = \mathit{ran}\,(X \cdot \mathit{ran}\, Y) \text{ and } \mathit{ran}\, \Pi = \mathit{id}\} \\
& \mathit{ran}\,(R \cdot \mathsf{F}C) \subseteq C
\end{aligned}$$

To prove claim 1, we reason:

$$
\begin{array}{cl}
 & (\delta_{\mathsf{F}} \cdot R^{\circ})\backslash(\mathit{ran}\, R\backslash(C \cdot \mathit{\Pi})) \\
= & \{(13)\} \\
 & (\mathit{ran}\, R \cdot \delta_{\mathsf{F}} \cdot R^{\circ})\backslash(C \cdot \mathit{\Pi}) \\
\subseteq & \{(8), \text{division is anti-monotonic}\} \\
 & (\delta_{\mathsf{F}} \cdot \mathsf{F}(\mathit{ran}\, R) \cdot R^{\circ})\backslash(C \cdot \mathit{\Pi}) \\
\subseteq & \{\text{by assumption: } \mathit{dom}\, R \subseteq \mathsf{F}(\mathit{ran}\, R), \text{division is anti-monotonic}\} \\
 & (\delta_{\mathsf{F}} \cdot \mathit{dom}\, R \cdot R^{\circ})\backslash(C \cdot \mathit{\Pi}) \\
= & \{\mathit{dom}\, R \cdot R^{\circ} = R^{\circ}\} \\
 & (\delta_{\mathsf{F}} \cdot R^{\circ})\backslash(C \cdot \mathit{\Pi}) \\
= & \{(13)\} \\
 & R^{\circ}\backslash(\delta_{\mathsf{F}}\backslash(C \cdot \mathit{\Pi})) \\
= & \{(14)\} \\
 & R^{\circ}\backslash(\mathsf{F}C \cdot \delta_{\mathsf{F}}\backslash\mathit{\Pi}) \\
= & \{\text{since } \delta_{\mathsf{F}}\backslash\mathit{\Pi} = \mathit{\Pi} \text{ by division}\} \\
 & R^{\circ}\backslash(\mathsf{F}C \cdot \mathit{\Pi})
\end{array}
$$

The proof for claim 2 goes:

$$
\begin{array}{cl}
 & \mathit{ran}\, R \cdot R^{\circ}\backslash(\mathsf{F}C \cdot \mathit{\Pi}) \\
\subseteq & \{\text{since } \mathit{ran}\, R = (R \cdot R^{\circ}) \cap \mathit{id}\} \\
 & R \cdot R^{\circ} \cdot R^{\circ}\backslash(\mathsf{F}C \cdot \mathit{\Pi}) \\
\subseteq & \{\text{division}\} \\
 & R \cdot \mathsf{F}C \cdot \mathit{\Pi}
\end{array}
$$

□