Theory and Applications of 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, applications, and generalisation of a theorem, due to Bird and de Moor, that gave conditions under which a total function can be expressed as a relational fold. The theorem is illustrated with three problems, all dealing with constructing trees with various properties. It is then generalised to give conditions under which the inverse of a partial function can be expressed as a relational hylomorphism. The proof makes use of Doornbos and Backhouse's theory on well-foundedness and reductivity. Possible applications of the generalised theorem is then discussed.

1 Introduction

The purpose of this paper is to describe one technique for inverting functions. Why bother with inverse functions? The reader might ask. The reason is that many problems in computation can be specified in terms of computing the inverse of an easily constructed function. Indeed, compression is best specified as the inverse of decompression, parsing the inverse of printing, and so on. But these are not the only applications; inverse sometimes arise in unexpected situations. To illustrate this, we will discuss three problems and solve them as instances of a sinlge technique.

The first problem is that of breadth-first labelling. To breadth-first label a tree with respect to a given list is to augment the nodes of the tree with values in the list in breadth-first order. Figure 1 shows the result of breadth-first labelling a tree with 13 nodes with the infinite list [1..]. While everybody knows how to do breadth-first traversal, the closely related problem of efficient breadth-first labelling is not so widely understood.

How would one specify this problem, and what does it have to do with inverse functions? Let us call the type of binary trees *Tree A* and assume that we

Preprint submitted to Elsevier Science *16 May 2003*

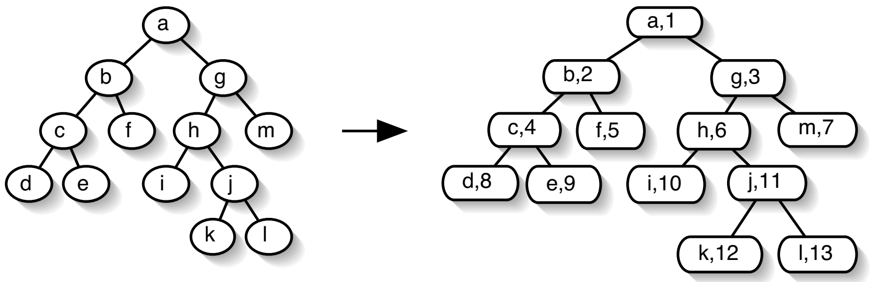

Fig. 1. Breadth-first labelling a tree on the left with [1..].

have at hand the function $bft :: Tree\ A \rightarrow List\ A$, for breadth-first traversal, and $zipTree :: Tree\ A \rightarrow Tree\ B \rightarrow Tree\,(A \times B)$, a *partial* function zipping together two trees of the same shape. To perform breadth-first labelling given a tree t and a list x, we want to zip t with another tree u. What, then, does this tree u has to satisfy? Firstly, it must be of the right shape, a condition that can be enforced by *zipTree*. Secondly, its breadth-first traversal must be a prefix of the given list x. We thus come up with the following specification:

$$\begin{aligned} &bfl\ t\ x = zipTree\ t\ u \\ &\qquad \mathbf{where}\ \ bft\ u = y \\ &\qquad\qquad\quad y \mathbin{+\!\!+} z = x \end{aligned}$$

Now look at the flow of information in the above specification. The functions *bft* and $+\!\!+$ appear on the left-hand side, meaning that we wish the data to go *backwards* through them. Let us denote the inverse of a function f by f°, pronounced “the converse of f” or more briefly “f wok”. The formal definition of f° will be delayed to Section 2.1. For now, let us say that $f^\circ\, y$ non-deterministically yields some x such that $f\ x = y$. We rewrite the specification as a pipeline from the right to the left, resulting in the following equivalent point-free specification:

$$bfl\ t = zipTree\ t \cdot bft^\circ \cdot fst \cdot cat^\circ$$

where $cat = uncurry\,(+\!\!+)$. Here cat° non-deterministically splits the input list in two, therefore $fst \cdot cat^\circ$ takes an arbitrary prefix of the input list. The inverse of *bft* gives us a tree whose breadth-first traversal matches the prefix. The tree is then zipped with the input t.

This is an example where inverses arise unexpectedly in specification. Concise as it is, how does one derive an algorithm from it? The answer, among two other examples, is to be presented in this paper.

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

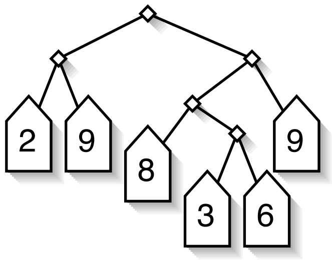

Fig. 2. A tree with height 11 built from trees with heights $[2, 9, 8, 3, 6, 9]$

$[2, 9, 8, 3, 6, 9]$. As the actual content of the subtrees isn't important, we can think of them simply as numbers representing the heights.

The third problem is a classical one. It is well-known that given the preorder and inorder traversals of an internally labelled binary tree, the tree can be reconstructed uniquely if it contains no duplicated elements. The challenge is to derive a linear time algorithm to do this.

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. 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 [4,6]. 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

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

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 $R :: A \leadsto B$, the coreflexive $dom\ R :: A \nrightarrow A$ 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:

$$\begin{array}{l} prefix :: List\ A \leadsto List\ A \\ prefix = foldr\ step\ [\,] \\ \quad \textbf{where}\ \begin{array}{lcl} step & :: & A \to List\ A \leadsto List\ A \\ step\ a\ x & = & (a : x)\ \Box\ [\,] \end{array} \end{array}$$

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 [29].

2.2 Folds

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

$$\begin{array}{lcl} foldrn & :: & (A \to B \to B) \to (A \to B) \to List^+\ A \to B \\ foldrn\ f\ g\ [a] & = & g\ a \\ foldrn\ f\ g\ (a : x) & = & f\ a\ (foldrn\ f\ g\ x) \end{array}$$

Here $List^+\ A$ denotes the type of non-empty lists.

Define tip-valued binary tree by the following datatype:

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

Its fold function can be defined as:

$$\begin{array}{lcl} foldTree & :: & (B \to B \to B) \to (A \to B) \to Tree\ A \to B \\ foldTree\ f\ g\ (Tip\ a) & = & g\ a \\ foldTree\ f\ g\ (Bin\ x\ y) & = & f\ (foldTree\ f\ g\ x)\ (foldTree\ f\ g\ y) \end{array}$$

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

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

Datatypes are often parameterised. In that case α_{F} has type $\mathsf{F}_A(\mathsf{T}A) \to \mathsf{T}A$. 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 $\mathsf{F}(A, B) \leadsto B$, now has type $\mathsf{T}A \leadsto B$. For a fuller account of relator theory and relational catamorphisms, the reader is referred to [3,4].

3 The Converse-of-a-Function Theorem

The converse-of-a-function theorem, introduced in [6,29], 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 $R :: \mathsf{F}(A, B) \leadsto B$ 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 $\mathit{step} :: A \rightarrow B \leadsto B$ are jointly surjective (meaning that $\{(\mathit{base}, \mathit{base})\} \cup ((\bigcup_{a \in A} \mathit{ran}\,(\mathit{step}\ a)) = \mathit{id}_B)$ 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 $\mathit{base} :: A \leadsto B$ and $\mathit{step} :: A \rightarrow B \leadsto B$ are jointly surjective (that is,

$ran\ base \cup (\bigcup_{a \in A} ran\ (step\ a)) = id_B)$ and satisfy

$$\begin{array}{ll} f\ (base\ a) & = [a] \\ f(step\ a\ x) & = a : f\ x \end{array}$$

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

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

4 Rebuilding a Tree from its Traversals

It is well known that, given the inorder and preorder traversal of a binary tree whose labels are all distinct, one can reconstruct the tree uniquely. The problem has been recorded in [27, Section 2.3.1, Exercise 7] as an exercise, where Knuth briefly described why it can be done and commented that it "would be an interesting exercise" to write a program for the task. Indeed, it has become a classical problem to tackle for those who study program inversion to derive a linear time algorithm, such as in [8,36]. As van de Snepscheut noted in [36], one class of solution attempts to invert an iterative algorithm while the other class delivers a recursive algorithm. In this section we will see how our theorem helps to derive a functional program to solve the problem. Interestingly, although we start with a recursive specification, Theorem 1 delivers an algorithm falling into the first category.

We define the following datatype for internally labelled binary trees and its fold function *foldTree*:

$$\mathbf{data}\ Tree\ A = Null \mid Node\ A\ (Tree\ A)\ (Tree\ A)$$

$$\begin{array}{ll} foldTree\ f\ e\ Null & = e \\ foldTree\ f\ e\ (Node\ a\ t\ u) & = f\ a\ (foldTree\ f\ e\ t)\ (foldTree\ f\ e\ u) \end{array}$$

Inorder and preorder traversal on the trees can then be defined in terms of *foldTree*:

$$\begin{array}{ll} inorder & = foldTree\ inf\ [\,] \\ & \quad \mathbf{where}\ \ inf\ a\ x\ y = x \mathbin{+\!\!+} [a] \mathbin{+\!\!+} y \\ preorder & = foldTree\ pre\ [\,] \\ & \quad \mathbf{where}\ \ pre\ a\ x\ y = [a] \mathbin{+\!\!+} x \mathbin{+\!\!+} y \end{array}$$

Assume a predicate *distinct* which yields true for a tree whose values in the nodes are all distinct. The aim is to construct $distinct? \cdot (fork\ (preorder, inorder))^{\circ}$, where $fork\ (f, g)\ x = (f\ x, g\ x)$.

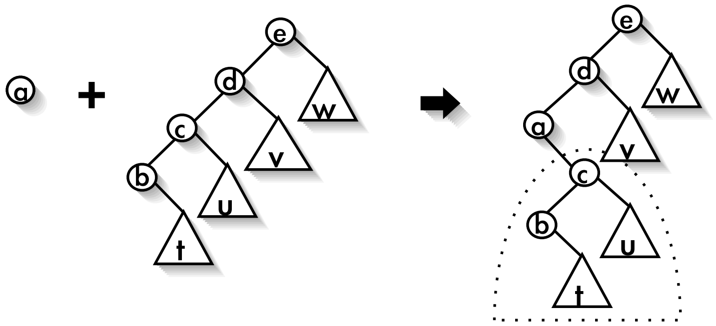

Fig. 3. Adding a new node to a tree

We will try to apply Theorem 1 to construct the converse of a function as a relational fold. However, due to its type, $(fork\,(preorder, inorder))^{\circ}$ apparently cannot be a fold on a recursive datatype. Instead, we define *rebuild* to be

$$rebuild\,x = ((x\ \texttt{==}) \cdot preorder)? \cdot inorder^{\circ}$$

The relation $inorder^{\circ}$ constructs all trees whose inorder traversals meet a given list. The coreflexive $((x\ \texttt{==}) \cdot preorder)?$ then picks the one whose preorder traversal is the list x. Apparently, $(fork\,(preorder, inorder))^{\circ} = uncurry\ rebuild$. Furthermore, the predicate *distinct* can be enforced by the constrain that x must not contain duplicated elements. The aim now is thus to derive $rebuild\,x$.

The derivation proceeds in two parts: to invert *inorder* as a fold on lists, and to fuse $((x\ \texttt{==}) \cdot preorder)?$ into the resulting fold.

4.1 Building a Tree by a Fold

According to Theorem 1, in order to invert *inorder*, we need a tree *zero* and a relation $add :: A \to Tree\,A \leadsto Tree\,A$ that are jointly surjective and satisfy

$$\begin{aligned} inorder\ zero &= [\,] \\ inorder\,(add\ a\ x) &= a : inorder\ x \end{aligned}$$

Look at the second equation. It says that if we have a tree x whose inorder traversal is as, the relation *add* must be able to create a new tree y out of a and x such that the order traversal of y is $a : as$. One way to do that is illustrated in Figure 3. We divide the left spine of x in two parts, move down the lower part for one level, and attach a to the end.

To facilitate this operation, we introduce an alternative *spine representation*. A tree is represented by the list of values and subtrees along the left spine.

$$\mathbf{type}\ Spine\ A = List\,(A \times Tree\ A)$$

For example, The tree to the left in Figure 3 is represented by the list

$$[(b,t),(c,u),(d,v),(e,w)]$$

The function *roll* converts a spine back into a single tree, and is in fact an isomorphism between *Spine A* and *Tree A*.

$$\begin{array}{l} \mathit{roll} :: \mathit{Spine}\,A \rightarrow \mathit{Tree}\,A \\ \mathit{roll} = \mathit{foldl}\,\mathit{join}\,\mathit{Null} \\ \qquad \textbf{where}\ \ \mathit{join}\,u\,(a,v) = \mathit{Node}\,a\,u\,v \end{array}$$

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 $\mathit{inorder} \cdot \mathit{roll}$ flattens a spine tree. Our task now is to invert it as a fold. We need a spine tree $\mathit{zero} :: \mathit{Spine}\,A$ and a relation $\mathit{add} :: A \rightarrow \mathit{Spine}\,A \leadsto \mathit{Spine}\,A$ satisfying

$$\begin{array}{rcl} \mathit{inorder}\,(\mathit{roll}\,\mathit{zero}) & = & [\,] \\ \mathit{inorder}\,(\mathit{roll}\,(\mathit{add}\,a\,\mathit{us})) & = & a : \mathit{inorder}\,(\mathit{roll}\,\mathit{us}) \end{array} \tag{3}$$

An easy choice for *zero* would be $[\,]$. As for *add*, we claim that the following definition satisfies (3):

$$\begin{array}{l} \mathit{add} \qquad :: A \rightarrow \mathit{Spine}\,A \leadsto \mathit{Spine}\,A \\ \mathit{add}\,a\,\mathit{us} = (a, \mathit{roll}\,\mathit{vs}) : \mathit{ws} \\ \qquad \textbf{where}\ \mathit{vs} +\!\!+ \mathit{ws} = \mathit{us} \end{array}$$

The non-deterministic pattern in the definition of *add*, dividing the list *us* into two parts, indicates that $\mathit{add}\,a$ is a relation. For example, the tree to the right in Figure 3 results from cutting the spine in the middle, yielding $[(a, \mathit{roll}\,[(b,t),(c,u)]),(d,v),(e,w)]$.

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

$$\mathit{inorder} \cdot \mathit{roll} = \mathit{concat} \cdot \mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder})) \tag{4}$$

where $\mathit{cons} = \mathit{uncurry}\,(:)$. The proof of (3) goes:

$$\begin{array}{cl} & a : \mathit{inorder}\,(\mathit{roll}\,(\mathit{vs} +\!\!+ \mathit{ws})) \\ = & \quad \{(4)\} \\ & a : \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,(\mathit{vs} +\!\!+ \mathit{ws})) \\ = & \quad \{\text{since } \mathit{concat} \text{ and } \mathit{map} \text{ distributes over } +\!\!+\} \\ & a : \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,\mathit{vs}) +\!\!+ \\ & \quad \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,\mathit{ws}) \end{array}$$

$$
\begin{array}{ll}
= & \{(4)\} \\
& a : \mathit{inorder}\,(\mathit{roll}\,\mathit{vs}) \mathbin{+\!\!+} \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,\mathit{ws}) \\
= & \{\text{definition of } \mathit{concat} \text{ and } \mathit{map}\} \\
& \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,((a, \mathit{roll}\,\mathit{vs}) : \mathit{ws})) \\
= & \{(4)\} \\
& \mathit{inorder}\,(\mathit{roll}\,((a, \mathit{roll}\,\mathit{vs}) : \mathit{ws}))
\end{array}
$$

It is also not difficult to see that $[\,]$ and add are jointly surjective: any non-null tree can be a result of add. We therefore conclude that $(\mathit{inorder} \cdot \mathit{roll})^{\circ} = \mathit{foldr}\,\mathit{add}\,[\,]$.

4.2 Enforcing a Preorder

Having inverted $\mathit{inorder} \cdot \mathit{roll}$, we can start the derivation:

$$
\begin{array}{ll}
& \mathit{rebuild}\,x \\
= & \{\text{definition}\} \\
& ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot \mathit{inorder}^{\circ} \\
= & \{\mathit{roll} \text{ is an isomorphism}\} \\
& ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot (\mathit{inorder} \cdot \mathit{roll} \cdot \mathit{roll}^{\circ})^{\circ} \\
= & \{\text{converse is contravariant}\} \\
& ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot \mathit{roll} \cdot (\mathit{inorder} \cdot \mathit{roll})^{\circ} \\
= & \{\text{inverting } \mathit{inorder} \cdot \mathit{roll} \text{ as in the last section}\} \\
& ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot \mathit{roll} \cdot \mathit{foldr}\,\mathit{add}\,[\,] \\
= & \{\text{since } p? \cdot f = f \cdot (p \cdot f)?, \\
& \quad \text{let } \mathit{hasPreorder}\,x = (x \mathrel{==}) \cdot \mathit{preorder} \cdot \mathit{roll}\} \\
& \mathit{roll} \cdot (\mathit{hasPreorder}\,x)? \cdot \mathit{foldr}\,\mathit{add}\,[\,]
\end{array}
$$

Except for the introduction of roll, the derivation so far is mostly mechanical. As $\mathit{roll} \cdot (\mathit{hasPreorder}\,x)?$ is a partial function, it can be easily implemented in Haskell. However, add is still a relation. If we can fuse $(\mathit{hasPreorder}\,x)?$ into the fold and thereby refine add to a partial function, the whole expression will be implementable. Unfortunately, $(\mathit{hasPreorder}\,x)?$ 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 – obviously, after adding a new element to a tree, the new tree will certainly have a different preorder traversal. Can we invent something weaker that can be fused into the fold?

Define $\mathit{preorderF}$ to be the preorder traversal of forests:

$$\mathit{preorderF} = \mathit{concat} \cdot \mathit{map}\,\mathit{preorder}$$

Look at Figure 3 again. The preorder traversal of the tree on the left-hand side is $[e, d, c, b] \mathbin{+\!\!+} \mathit{preorderF}\,[t, u, v, w]$ — that is, to go down along the left spine, then traverse through the subtrees upwards. In general, given a spine tree us, its preorder traversal is $\mathit{reverse}\,(\mathit{map}\,\mathit{fst}\,us) \mathbin{+\!\!+} \mathit{preorderF}\,(\mathit{map}\,\mathit{snd}\,us)$. We will call the part before $\mathbin{+\!\!+}$ the *prefix* and that after $\mathbin{+\!\!+}$ the *suffix* of the traversal. Now look at the tree on the right-hand side. Its preorder traversal is $[e, d, a, c, b] \mathbin{+\!\!+} \mathit{preorderF}\,[t, u, v, w]$. It is not difficult to see that when we add a node a to a spine tree us, the suffix of its preorder traversal does not change. The new node a is always inserted to the prefix.

With this insight, we split $\mathit{hasPreorder}$ into two parts:

$$
\begin{array}{lcl}
\mathit{hasPreorder} & :: & \mathit{List}\ A \rightarrow \mathit{Spine}\ A \rightarrow \mathit{Bool} \\
\mathit{hasPreorder}\ x\ us & = & \mathit{prefixOk}\ x\ us \wedge \mathit{suffixOk}\ x\ us \\
\mathit{suffixOk}\ x\ us & = & \mathit{preorderF}\,(\mathit{map}\,\mathit{snd}\,us)\ \textbf{isSuffixOf}\ x \\
\mathit{prefixOk}\ x\ us & = & \mathit{reverse}\,(\mathit{map}\,\mathit{fst}\,us) == (x \ominus \mathit{preorderF}\,(\mathit{map}\,\mathit{snd}\,us))
\end{array}
$$

where $x \ominus y$ removes y from the tail of x and is defined by:

$$x \ominus y = z\ \textbf{where}\ z \mathbin{+\!\!+} y = x$$

The expression x **isSuffixOf** y yields true if x is a suffix of y. The use of boldface font here indicates that it is an infix operator (and binds looser than function application). The plan is to fuse only $\mathit{suffixOk}\ x$ into the fold while leaving $\mathit{prefixOk}\ x$ outside.

There is a slight problem, however. The invariant $\mathit{suffixOk}\ x$ does not prevent the fold from generating, say, a leftist tree with all *Null* along the spine, since the empty list is indeed a suffix of any list. Such a tree may be bound to be rejected later. Look again at the right-hand side of Figure 3. Assume we know that the preorder traversal of the tree we want is $x = [..\, d, c, b] \mathbin{+\!\!+} \mathit{preorderF}\,[t, u, v, w]$. The tree in Figure 3, although satisfying $\mathit{suffixOk}\ x$, is bound to be wrong because d is the next immediate symbol but a now stands in the way between d and c, and there is no way to change the order afterwards. Thus when we find a proper location to insert a new node, we shall be more aggressive and consume as much suffix of x as possible. The following predicate $\mathit{lookahead}\ x$ ensures that in the constructed tree, the next immediate symbol in x will be consumed:

$$
\begin{array}{lcl}
\mathit{lookahead} & :: & \mathit{List}\ A \rightarrow \mathit{Spine}\ A \rightarrow \mathit{Bool} \\
\mathit{lookahead}\ x\ us & = & \mathit{length}\ us \leq 1\ \vee\ (\mathit{map}\,\mathit{fst}\,us)\ !!\ 1 \neq \mathit{last}\ x' \\
\quad \textbf{where}\ x' = x \ominus \mathit{preorderF}\,(\mathit{map}\,\mathit{snd}\,us) & &
\end{array}
$$

Apparently $\mathit{lookahead}\ x$ is compatible with $\mathit{hasPreorder}\ x$. We will use both *suffixOk* x and *lookahead* x as our invariant. Define

$$ok\ x\ us\ =\ suffixOk\ x\ us \wedge lookhead\ x\ us$$

The derivation continues:

$$\begin{array}{cl} & rebuild\ x \\ = & \quad \{\text{as in the beginning of this section}\} \\ & roll \cdot (hasPreorder\ x)? \cdot foldr\ add\ [\,] \\ = & \quad \{\text{since } hasPreorder\ x\ us = prefixOk\ x\ us \wedge ok\ x\ us\} \\ & roll \cdot (prefixOk\ x)? \cdot (ok\ x)? \cdot foldr\ add\ [\,] \\ = & \quad \{\text{fold fusion, assume } nodup\ x\} \\ & roll \cdot (prefixOk\ x)? \cdot foldr\ (add'\ x)\ [\,] \end{array}$$

The fold fusion theorem used in the last step is well-known (see, for example, [6, Chapter 6]):

$$R \cdot (\![S]\!)_{\mathsf{F}} = (\![T]\!)_{\mathsf{F}} \Leftarrow R \cdot S = T \cdot \mathsf{F}R$$

To justify the fusion step, it can be shown that if x contains no duplicated elements, the following fusion condition holds:

$$(ok\ x)?\ (add\ a\ us)\ =\ add'\ x\ a\ ((ok\ x)?\ us)$$

where add' is defined by:

$$\begin{array}{lcl} add' & :: & List\ A \rightarrow A \rightarrow Spine\ A \rightarrow Spine\ A \\ add'\ x\ a\ us & = & up\ a\ Null\ (us, x \ominus preorderF\ (map\ snd\ us)) \end{array}$$

$$\begin{array}{l} up\ ::\ A \rightarrow Tree\ A \rightarrow (Spine\ A \times List\ A) \rightarrow Spine\ A \\ up\ a\ v\ ([\,], x) \qquad\qquad\qquad = [(a, v)] \\ up\ a\ v\ ((b, u) : us, x +\!\!+ [b']) \ \mid\ b == b' \qquad = \ up\ a\ (Node\ b\ v\ u)\ (us, x) \\ \qquad\qquad\qquad\qquad\qquad\quad\ \mid\ otherwise = \ (a, v) : (b, u) : us \end{array}$$

In words, the function *up* traces the left spine upwards and consumes the values on the spine if they match the tail of x. It tries to roll as much as possible before adding a to the end of the spine.

We are now ready for the final optimisation. To avoid computing $x \ominus preorderF\ (map\ snd\ us)$ from scratch each time, we can apply a tupling transformation (see, for example [22] or [6, Chapter 3]), having the fold returning a pair. The Haskell implementation is shown in Figure 4. The fold in *rebuild* returns a pair, the first component being a tree and the second component being a list representing $x \ominus preorderF\ (map\ snd\ us)$. Since the list is consumed from the end, we represent it in reverse. The function *rollpf* implements $roll \cdot (prefixOk\ x)?$.

```
data Tree a = Null | Node a (Tree a) (Tree a) deriving (Show,Eq)

rebuild :: Eq a => [a] -> [a] -> Tree a
rebuild x = rollpf . foldr add' ([],reverse x)
  where add' a (us,x) = up a Null (us,x)
        up a v ([],x) = ([(a,v)],x)
        up a v ((b,u):us, b':x)
           | b == b' = up a (Node b v u) (us, x)
           | otherwise = ((a,v):(b,u):us, b':x)

rollpf :: Eq a => ([(a,Tree a)],[a]) -> Tree a
rollpf (us,x) = rp Null (us,x)
 where rp v ([],[]) = v
       rp v ((b,u):us, b':x)
          | b == b' = rp (Node b v u) (us,x)
```

Fig. 4. Rebuilding a tree from its traversals via a fold.

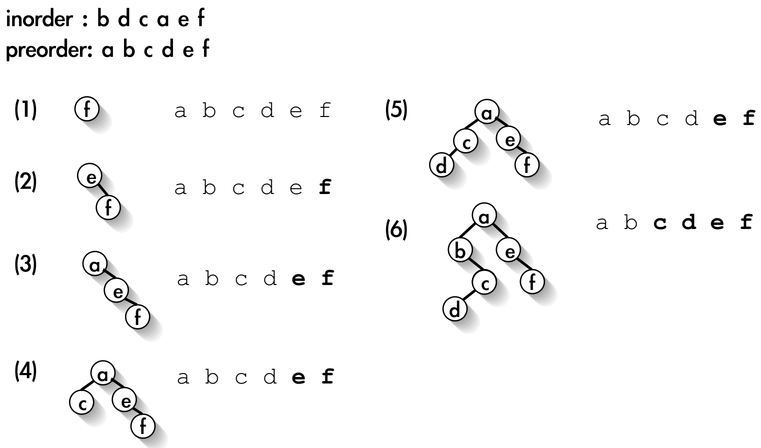

Fig. 5. Building a tree from its preorder. The preorder traversals of the trees under the spine is printed in boldface font.

Figure 5 shows an example of this algorithm in action. The part in boldface font indicates *preorderF* (*map snd us*). Notice how the preorder traversals of the trees under the spine always form a suffix of the given list $[a, b, c, d, e, f]$.

We have actually reinvented the algorithm proposed in [8], but in a functional style. The first step in [8] was to transform the recursive definition of *fork* (*preorder*, *inorder*) into an iteration by introducing a stack. The same effect we achieved by introducing the spine representation.

Hold on! The reader might complain: the derivation works because, by luck, we choose the correct order. Had we started with:

$$((x \mathtt{==}) \cdot \mathit{inorder})? \cdot \mathit{preorder}^{\circ}$$

Now, we would have to invert *preorder*, and then enforce, on the resulting fold, the constraint that the tree built must have a given inorder traversal. Does it still work? In fact, it does, and the result is a new but complicated algorithm. Therefore, we are only going to sketch an outline of its development.

We first seek to invert *preorder*. For this problem it turns out that it makes more sense to work on forests rather than trees. Abbreviate $\mathit{List}\,(\mathit{Tree}\,A)$ to $\mathit{Forest}\,A$. Recall $\mathit{preorderF} :: \mathit{Forest}\,A \to \mathit{List}\,A$ defined by $\mathit{preorderF} = \mathit{concat} \cdot \mathit{map}\,\mathit{preorder}$. The reader can easily verify that *preorderF* can be inverted as below:

$$\begin{array}{l}
\mathit{preorderF}^{\circ} = \mathit{foldr}\ \mathit{step}\ [\,] \\
\quad\quad \textbf{where}\ \ \mathit{step}\,(a, \mathit{us}) = \mathit{tip}\ a : \mathit{us} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \square\ \mathit{lbr}\,(a, \mathit{head}\ \mathit{us}) : \mathit{tail}\ \mathit{us} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \square\ \mathit{rbr}\,(a, \mathit{head}\ \mathit{us}) : \mathit{tail}\ \mathit{us} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \square\ \mathit{Node}\ a\,(\mathit{us}!!0)\,(\mathit{us}!!1) : \mathit{tail}\,(\mathit{tail}\ \mathit{us})
\end{array}$$

where the helper functions *tip*, *lbr* and *rbr* respectively creates a tip tree, a tree with only the left branch, and a tree with only the right branch. They are defined by:

$$\begin{array}{lcl}
\mathit{tip}\ a & = & \mathit{Node}\ a\ \mathit{Null}\ \mathit{Null} \\
\mathit{lbr}\,(a, t) & = & \mathit{Node}\ a\ t\ \mathit{Null} \\
\mathit{rbr}\,(a, t) & = & \mathit{Node}\ a\ \mathit{Null}\ t
\end{array}$$

In words, *step* extends a forest in one of the four possible ways, when applicable: adding a new tip tree, extending the left-most tree in the forest by making it a left-subtree or a right-subtree, or combining the two left-most trees.

The next step is to find out a rule deciding which of the four operations to perform when adding a new value to a forest. We need to invent an invariant to enforce in the body of the fold. To begin with, we reason:

$$\begin{array}{cl}
 & ((x \mathtt{==}) \cdot \mathit{inorder})? \cdot \mathit{preorder}^{\circ} \\
= & \quad \{\text{since } \mathit{preorder} = \mathit{preorderF} \cdot \mathit{wrap} \text{ where } \mathit{wrap}\ a = [a]\} \\
 & ((x \mathtt{==}) \cdot \mathit{inorder})? \cdot \mathit{wrap}^{\circ} \cdot \mathit{preorderF}^{\circ} \\
= & \quad \{\text{some trivial manipulation}\} \\
 & \mathit{wrap}^{\circ} \cdot ((x \mathtt{==}) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder})? \cdot \mathit{preorderF}^{\circ}
\end{array}$$

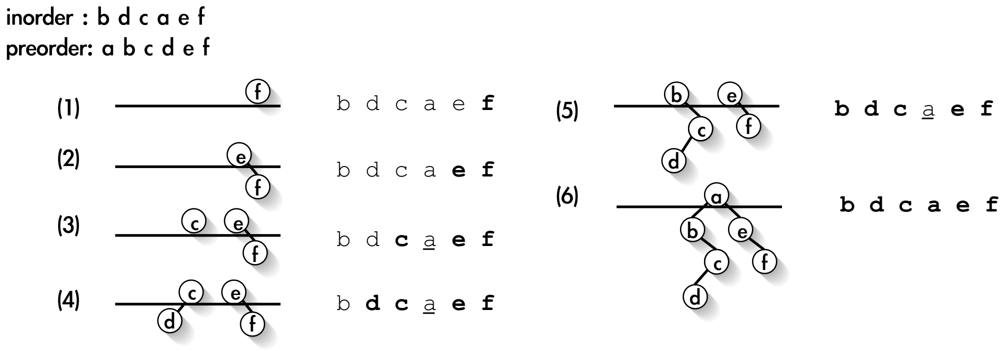

Fig. 6. Building a tree from its preorder. The inorder traversals of the constructed subtrees are printed in boldface font. The "skipped" subsequences between subtrees are underlined. In the optimised code in Figure 7, they are paired with the subtrees. Thus the type *AForest*.

Again, the condition $(x \mathrel{==}) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder}$ is too strong to maintain. Luckily, it turns out that the weaker constraint

$$(\mathbf{isSubSeqOf}\ x) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder}$$

will do, where $(\mathbf{isSubSeqOf}\ x)\ y \ = \ y\ \mathbf{isSubSeqOf}\ x$ yields true if y is a subsequence of x. That is, we require that during the construction of the forest, the inorder traversal of each tree shall always form segments of x, in correct order. Figure 6 demonstrates the process of constructing the same tree as that in Figure 5. This time notice how the inorder traversal of the constructed forest always forms a subsequence of the given list $[b, d, c, a, e, f]$.

After some pencil-and-paper work, it is not difficult to work out the rules to extend the forest while maintaining the invariant. However, the rules consists of totally eight cases and is relatively complicated comparing to the simpler algorithms in Section 4.2. It is owing to the fact that we have four possible operations to choose from, while in Section 4.2 there were only two – either to go upwards one node along the spine or to stop and attach a new node. For that reason we will just present the result.

A program implementing the algorithm is presented in Figure 7, where each tree in the forest is annotated with some extra information to avoid recomputing them (represented by the type *AForest*). After this optimisation, the program runs in linear time, but with a bigger constant overhead than that in Section 4.2.

```
tip a   = Node a Null Null
rbr a x = Node a Null x
lbr a x = Node a x Null

type AForest a = [(Tree a, [a])]

rebuild :: Eq a => [a] -> [a] -> Tree a
rebuild x = fst . unwrap . snd . foldr add (reverse x, [])
 where add :: Eq a => a -> ([a],AForest a) -> ([a],AForest a)
       add a xu@(x, [])  = newtree a xu
       add a xu@(x, (t,[]):us)
         | isNext x a = (tail x, (rbr a t, []):us)
         | otherwise = newtree a xu
       add a xu@(x,(t,b:bs):us)
         | a == b = (x, join a (t,bs) us)
         | isNext x a = (tail x, (rbr a t, b:bs):us)
         | otherwise = newtree a xu

       join a (t,[]) [] = [(lbr a t,[])]
       join a (t,[]) ((u,y):us) = (Node a t u, y) : us
       join a (t,bs) us = (lbr a t, bs):us

       newtree a (x,us) = (x', (tip a, y):us)
          where (x',y) = skip x a

       isNext [] a = False
       isNext (b:bs) a = a == b

skip x a = locate a [] x
 where locate a y [] = ([],y)
       locate a y (b:x) | a == b = (x,y)
                        | otherwise = locate a (b:y) x
```

Fig. 7. Another way to rebuild a tree from its traversals via a fold.

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 [5], but here we will demonstrate how a similar algorithm can be derived.

We start with giving a formal specification of the problem. Define tip-valued binary tree by the following datatype:

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

The 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 and *foldTree* is the fold function for *Tree*, defined in Section 2.2.

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

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

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

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

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:

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

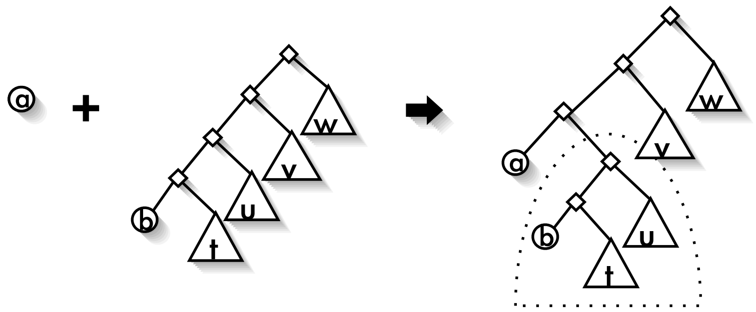

Fig. 8. Adding a new node to a tip-valued binary tree

Similar to the last problem, the derivation also proceeds in two steps: to invert *flatten* as a relational fold, and fusing something into the fold to eliminate its non-determinism.

The function *flatten* can be inverted in a way similar to that in Section 4.1. It is also helpful to switch to a spine representation. We define the following:

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

A tree is represented by the list of subtrees along the spine, together with the leftmost leaf. The conversion from a spine tree to the ordinary representation can be performed by:

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

Since the range of *flatten* is the set of non-empty lists, we seek to invert it to *foldrn*, the fold on non-empty lists. Theorem 1 says that $(\mathit{flatten} \cdot \mathit{roll})^{\circ} = \mathit{foldrn}\ \mathit{add}\ \mathit{one}$ if the relations *add* and *one* satisfy:

$$\begin{aligned} \mathit{flatten}\,(\mathit{roll}\,(\mathit{one}\ a)) &= [a] \\ \mathit{flatten}\,(\mathit{roll}\,(\mathit{add}\ a\,(b, \mathit{xs}))) &= a : \mathit{flatten}\,(\mathit{roll}\,(b, \mathit{xs})) \end{aligned}$$

Figure 8 illustrates the idea. We claim that the following definition satisfies the requirement.

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

The proof is similar to that in Section 4.1 and is left to the reader as an exercise.

Having inverted *flatten*, we get:

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

Furthermore, *roll* can be factored out of Λ:

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

where $xs \preceq' ys \equiv roll\,xs \preceq roll\,ys$, 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 [6].

Theorem 2 (The Greedy Theorem (for non-empty lists)) Let $base :: A \rightsquigarrow B$ and $step :: A \rightarrow B \rightsquigarrow B$ be two relations. If *step* is monotonic on a connected preorder $(\trianglelefteq)$, that is,

$$(x \trianglelefteq y \ \wedge \ (y, y') \in step\ a) \Rightarrow (\exists x' : (x, x') \in step\ a : x' \trianglelefteq y') \tag{5}$$

then we have

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

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 $min\,(\trianglelefteq)$ gets promoted into *foldrn*.

Had *add* satisfied the monotonicity condition (5) 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*.

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 [7] for more detailed discussion.

```
type SpineI a = (a, [(Int, 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. 9. Code for building trees with minimum height

Applying the greedy theorem, we get:

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

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 [7].

The code is shown in Figure 9. As in the first problem, we annotate each tree with its height to avoid re-computation. 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 [17] proposed a neat solution to this problem, based on a clever use of cyclic data structures. The problem was recently revisited by Okasaki [31]. We are going to show how Okasaki's algorithm can be derived using the converse-of-a-function theorem.

Let us go through again the specification in finer detail. Recall the data structure for internally and externally labelled binary trees:

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

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

$$\begin{array}{lcl} \mathit{bft} & :: & \mathit{Tree}\ A \rightarrow \mathit{List}\ A \\ \mathit{bft}\ x & = & \mathit{bftF}\ [x] \end{array}$$

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

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

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

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

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*:

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

Equivalently,

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

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 $bft^{\circ} \cdot prefix$ 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 a 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 $bftF$:

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

Here $zipForest :: Forest\ A \rightarrow Forest\ B \leftrightarrow Forest\ (A \times B)$ is a simple extension of $zipTree$ to forests, which, like $zipTree$, is a partial function:

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

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

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

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:

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

Therefore $step\ a\ xs$ might contain $(Tip\ a : xs)$ 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 more than two trees:

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

Therefore we define *step* to be:

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

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

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

The expression $\mathit{zipForest} \cdot \mathit{bftF}^\circ$ has type $\mathit{List}\ A \to \mathit{Forest}\ B \rightsquigarrow \mathit{Forest}\ (A \times B)$. 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 Figure 10. It can be made linear if we use an implementation of deques supporting constant-time addition and deletion [9,30] 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 [31] 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.

7 The Generalised Converse-of-a-Function Theorem

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

Theorem 3 (Generalised converse-of-a-function theorem) Let $S :: B \rightsquigarrow A$ be a simple relation. If there exists a relation $R :: \mathsf{F}(C,B) \rightsquigarrow B$ and a simple relation $T :: \mathsf{F}(C,A) \leftrightarrow A$ are such that (i) $\mathit{dom}\,S = \mathit{ran}\,R$; (ii)

```
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. 10. Code for breadth-first labelling

$S \cdot R \subseteq T \cdot \mathsf{F}S$; and (iii) R° is F-well-founded, then

$$S^\circ = (\![R]\!)_\mathsf{F} \cdot (\![T]\!)_\mathsf{F}^\circ$$

In words, Theorem 3 gives conditions under which the converse of a simple relation can be expressed as a hylomorphism. Its relationship with Theorem 1, as well as other notions we need to establish such a connection, will be given in Section 7.1.

The new ingredient in Theorem 3 is F-*well-foundedness*. The notions of well-foundedness and admitting induction are of great importance in computing science. In [13], Doornbos gave a careful analysis of the relationship of these two notions and proposed several different generalisations of them. The notion of F-well-foundedness is defined in [13, page 102] as

Definition 4 (F-well-foundedness) A relation R is F-well-founded if and only if, for all relations T, the equation $X = T \cdot \mathsf{F}X \cdot R$ has a unique solution for X.

Now we will give a proof of Theorem 3. Taking converses of both sides, the aim is to prove that $S = (\![T]\!)_\mathsf{F} \cdot (\![R]\!)_\mathsf{F}^\circ$ under the given conditions. We know that the hylomorphism $(\![T]\!)_\mathsf{F} \cdot (\![R]\!)_\mathsf{F}^\circ$ can be characterised as the least solution for X of the equation $X = T \cdot \mathsf{F}X \cdot R^\circ$. Since, by assumption (iii) that R° is F-well-founded, we know that $(\![T]\!)_\mathsf{F} \cdot (\![R]\!)_\mathsf{F}^\circ$ is in fact the unique solution. Now we will show that S is also a solution. The proof goes:

$$\begin{array}{ll} & S \\ = & \{\text{since } S = S \cdot \mathit{dom}\, S = S \cdot \mathit{ran}\, R \text{ by assumption (i)}\} \\ & S \cdot \mathit{ran}\, R \\ \subseteq & \{\text{since } \mathit{ran}\, R \subseteq R \cdot R^\circ\} \end{array}$$

$$
\begin{aligned}
& S \cdot R \cdot R^{\circ} \\
\subseteq \quad & \{\text{by assumption (ii): } S \cdot R \subseteq T \cdot \mathsf{F}S\} \\
& T \cdot \mathsf{F}S \cdot R^{\circ} \\
= \quad & \{\text{since } R = \mathit{ran}\, R \cdot R = \mathit{dom}\, S \cdot R \text{ by assumption (i)}\} \\
& T \cdot \mathsf{F}S \cdot R^{\circ} \cdot \mathit{dom}\, S \\
\subseteq \quad & \{\text{since } \mathit{dom}\, S \subseteq S^{\circ} \cdot S\} \\
& T \cdot \mathsf{F}S \cdot R^{\circ} \cdot S^{\circ} \cdot S \\
\subseteq \quad & \{\text{by assumption (ii): } S \cdot R \subseteq T \cdot \mathsf{F}S\} \\
& T \cdot \mathsf{F}S \cdot (T \cdot \mathsf{F}S)^{\circ} \cdot S \\
\subseteq \quad & \{\text{since } T \cdot \mathsf{F}S \text{ is simple}\} \\
& S
\end{aligned}
$$

Theorem 3 is thus proved.

7.1 Well-foundedness and Reductivity

The proof of Theorem 3 makes use of F-well-foundedness to guarantee the uniqueness of solution. For practical proposes, however, a stronger property is needed. The reason is that F-well-foundedness alone does not guarantee termination if we view a hylo-equation as a left-to-right rewrite rule and evaluation of non-determinism as demonic. As a counterexample[1], take $\mathsf{F}X = X \times X$ and consider the relation $\langle f, id \rangle\ a = (f\ a, a)$. The recursive program $X = S \cdot \mathsf{F}X \cdot \langle f, id \rangle$ does not terminate if S is strict on the second component of the input pair. Yet $\langle f, id \rangle$ is F-well-founded.

It was shown in [13, Section 7.5], however, that termination *is* guaranteed for F-*reductive* relations. The notion of F-reductivity was introduced in [14,15] and [13, Section 6.3] as one of the ways to generalise the notion of admitting induction to arbitrary datatypes.

Definition 5 (F-reductivity) A relation $R :: A \leadsto \mathsf{F}A$ is said to be F-reductive if and only if for all coreflexives $C \subseteq id_A$:

$$R \backslash\!\!\!\!\!\!\;\;\;\!\!\mathsf{F}C \subseteq C \Rightarrow id_A \subseteq C \qquad (6)$$

Here the *monotype factor* operator $\backslash$ is defined by the Galois connection[2]:

$$D \subseteq R \backslash C \equiv \mathit{dom}\,(R \cdot D) \subseteq C$$

[1] Due to [13, page 104].

[2] For more intuition behind the monotype factor, the reader is referred to [14].

for all relations R and coreflexives C and D. Property (6) can be translated to point-level to aid understanding[3]:

$$(\forall a : (\forall a' : (a, a') \in R : (a', a') \in \mathsf{F}C) : (a, a) \in C) \Rightarrow (\forall a :: (a, a) \in C)$$

In words, R is F-reductive if it can be used for inductive proofs in the following way: we may conclude that a property C is universally true if we can show that all a satisfies C, given that any "R-predecessor" a' of a is an F-structure containing only elements satisfying C.

Several properties concerning F-reductivity are handy for our purpose. The following facts are respectively Theorem 6.25, Theorem 6.19, and Theorem 6.22 of [13]:

Fact 6 F-reductivity implies F-well-foundedness.

Fact 7 If T is F-reductive, so is $\mathsf{F}S^\circ \cdot T \cdot S$ for simple S.

Fact 8 α_F° is F-reductive.

Since F-reductivity guarantees termination and, according to Fact 6, is stronger than F-well-foundedness, the F-well-foundedness requirement in Theorem 3 is always strengthened to F-reductivity in practice. The question of how to construct F-reductive relations has been discussed in depth in [14,13].

We have yet talked about the relationship between Theorem 1 and Theorem 3. To begin with, note the following lemma:

Lemma 9 $R \subseteq S^\circ \cdot T \cdot \mathsf{F}S$ if $\mathit{ran}\, R \subseteq \mathit{dom}\, S$ and $S \cdot R \subseteq T \cdot \mathsf{F}S$.

Theorem 1 follows as a special instance of Theorem 3 by taking $T = \alpha_\mathsf{F}$ 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 $\mathit{dom}\, S = \mathit{id}$, so condition (i) translates to the requirement that R be a surjective relation. As for condition (iii), R° is F-reductive if T° is, according to Lemma 9 and Fact 7. However, Fact 8 says that $T^\circ = \alpha_\mathsf{F}^\circ$ is indeed F-reductive. Since $(\![\alpha_\mathsf{F}]\!)_\mathsf{F} = \mathit{id}$, we then obtain the result $S^\circ = (\![R]\!)_\mathsf{F}$, the conclusion of Theorem 1.

The proof for Lemma 9 can simply be extracted from the proof for Theorem 3. For completeness, it is given below:

$$\begin{array}{ll} & R \\ \subseteq & \{\text{since } R = \mathit{ran}\, R \cdot R \subseteq \mathit{dom}\, S \cdot R\} \\ & \mathit{dom}\, S \cdot R \end{array}$$

[3] On the other hand, (6) can be written more concisely as $\mu(R \backslash \cdot (\mathsf{F}\cdot)) = \mathit{id}$, where μ stands for the least fixed-point operator.

$$
\begin{aligned}
&\subseteq \quad \{\text{since } \mathit{dom}\, S \subseteq S^\circ \cdot S\}\\
&\quad S^\circ \cdot S \cdot R\\
&\subseteq \quad \{\text{since } S \cdot R \subseteq T \cdot \mathsf{F}S\}\\
&\quad S^\circ \cdot T \cdot \mathsf{F}S
\end{aligned}
$$

8 Applications of the Generalised Theorem

Theorem 3 can potentially be very powerful 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. One application we have found for Theorem 3 is to prove that a loop implements the inverse of some function. A loop can be specified relationally by

$$T \cdot R^* \cdot S$$

The relation S initialises the loop, while R serves as the loop body. The domain of T represents the terminating condition and therefore ought to be disjoint from the domain of R. Given a relation R, the reflexive transitive closure R^* is the smallest reflexive transitive relation containing R. More generally, the relation $R^* \cdot S :: A \rightsquigarrow B$, where $S :: A \rightsquigarrow B$ and $R :: B \rightsquigarrow B$, can be defined as a least fixed-point:

$$R^* \cdot S = \mu(X : S \cup R \cdot X)$$

A key observation here is that a closure can also be written as a hylomorphism, with the base functor $\mathsf{F}_A X = A + X$:

$$
\begin{aligned}
&\quad R^* \cdot S\\
&= \quad \{\text{definition of closure}\}\\
&\quad \mu(X : S \cup R \cdot X)\\
&= \quad \{\text{coproduct}\}\\
&\quad \mu(X : [S, R] \cdot (id + X) \cdot [id, id]^\circ)\\
&= \quad \{\text{hylomorphism, let } \mathsf{F}_A X = A + X\}\\
&\quad (\![S, R]\!)_{\mathsf{F}} \cdot (\![id, id]\!)_{\mathsf{F}}^\circ
\end{aligned}
$$

Here the unfolding phase wraps the input value with an *inl*, before wrapping it with an indefinite number of *inr*s. The folding phase then replaces the *inl* with S and each *inr* with an R. The exact number of iterations performed is determined the termination test T.

Given a function f, let us instantiate Theorem 3 to discover the conditions under which $f^\circ = (\![S, R]\!)_{\mathsf{F}} \cdot (\![id, id]\!)_{\mathsf{F}}^\circ$:

- Since $dom\,f = id$, condition (i) instantiates to $ran\,[S, R] = id$. That is, S and R shall be jointly surjective.
- Condition (ii) can be divided into two parts:

  $$f \cdot S \subseteq id \wedge f \cdot R \subseteq f$$

  Shunting the functions to the other side, we get:

  $$S \subseteq f^\circ \wedge R \cdot f^\circ \subseteq f^\circ$$

  which looks familiar enough! Think of f° as an invariant. The first half says that the initial values satisfies the invariant, while the second half says that given inputs satisfying the invariant, the loop body R maintains the invariant.
- Condition (iii) requires that $[S, R]^\circ$ be F-well-founded. Intuitively speaking, we want R to "decrease" the loop variables in some sense.

Assume we wish to prove that $T \cdot R^* \cdot S$ correctly implements a specification X. As will be shown in the next two sections, in some occasions X can be quite naturally factored into $T \cdot f^\circ$ for some f. We then just need to check the three conditions above.

8.1 The String Edit Problem

The string edit problem [10, Chapter 15] is a typical example for dynamic programming. Recently it has drawn much attention due to its application in DNA sequence matching. In its simplest form, we are given two strings, one as the source and one as the target, and some available commands. Imagine a cursor positioned to the left of the source string. We assume the following commands:

- *Ins* c: to insert a character c at the current position.
- *Del* c: to delete the character c in the current position.
- *Cpy* c: to skip the current character c and move the cursor one position to the right.

The task is to find the shortest sequence of commands to transform the source string to the target string. In more complicated variations we might be given more commands and their weights may vary.

We represent the three commands with a datatype Op:

$$\mathbf{data}\ Op = Ins\ Char \mid Del\ Char \mid Cpy\ Char$$

To specify the problem, one might attempt to construct a relation taking the pair of strings and return an arbitrary sequence of commands relating the strings. In fact, it is easier to construct its inverse. The function *exec* below executes a sequence of commands, starting from a pair of empty strings, and yields two strings:

$$
\begin{array}{l}
\mathit{exec} :: \mathit{List}\ \mathit{Op} \to (\mathit{String} \times \mathit{String}) \\
\mathit{exec} = \mathit{foldl}\ \mathit{step}\ ([\,],[\,]) \\
\qquad \textbf{where}\ \begin{array}{ll}
\mathit{step}\,(x,y)\,(\mathit{Ins}\ c) & = (x, y \mathbin{+\!\!+} [c]) \\
\mathit{step}\,(x,y)\,(\mathit{Del}\ c) & = (x \mathbin{+\!\!+} [c], y) \\
\mathit{step}\,(x,y)\,(\mathit{Cpy}\ c) & = (x \mathbin{+\!\!+} [c], y \mathbin{+\!\!+} [c])
\end{array}
\end{array}
$$

The *exec* function starts with two empty strings and tries to reconstruct the original source and target strings. After an *Ins* operation, an extra character is added to the target string. The *Del* operation is treated as a statement that the source string has an extra character. A *Cpy* command can be viewed as saying that the two strings has a common character at the current position. The converse of *exec*, on the other hand, takes two strings and yields a sequence of commands reducing them to a pair of empty strings (thus showing that the commands transform one string to another). The string edit problem is thus defined by:

$$
\mathit{stredit} = \mathit{min}\ R \cdot \Lambda \mathit{exec}^{\circ}
$$

In [6, Section 9.2], Bird and de Moor derived from this specification a dynamic programming algorithm using their dynamic programming theorem for converse of folds.

Yet some others prefer to describe exec° as an iterative process. That is, they claim that $\mathit{exec}^{\circ} = \mathit{end} \cdot \mathit{move}^{*} \cdot \mathit{start}$, where

$$
\begin{array}{lll}
\mathit{start}\,(x,y) & = & (x, y, [\,]) \\
\mathit{move}\,(x,y,\mathit{ops}) & = & (x, \mathit{init}\ y, \mathit{Ins}\,(\mathit{last}\ y) : \mathit{ops}) \\
 & & \Box\, (\mathit{init}\ x, y, \mathit{Del}\,(\mathit{last}\ x) : \mathit{ops}) \\
 & & \Box\, ((\mathit{init}\ x, \mathit{init}\ y, \mathit{Cpy}\,(\mathit{last}\ x) : \mathit{ops}), \mathit{if}\ \mathit{last}\ x == \mathit{last}\ y) \\
\mathit{end}\,([\,],[\,],\mathit{ops}) & = & \mathit{ops}
\end{array}
$$

The loop starts with the two strings and an empty list of commands. The non-deterministic loop body *move* then try to recover what the last command might be by trying all possible commands. The iteration repeats until both strings become empty. Notice that *move* is defined as a partial relation which yields value only when not both of x and y are empty. This was the view taken by Curtis in [11]. Once a specification is written in terms of a $\mathit{min}\,R$ after a loop, theories in [11] are ready to transform it to a dynamic programming algorithm, if certain conditions are satisfied.

We will not go into how the problem can be solved using the developed theories. Instead we will bridge the gap between the two views on *exec*. In other words, how do we know the claim that $exec^\circ = end \cdot move^* \cdot start$ is true?

With the discussions in the opening of Section 8 in mind, we generalise *exec* to *execWith* such that

$$exec = execWith \cdot end^\circ$$

The function *execWith* has type $(String \times String \times List\ Op) \to (String \times String)$ and is defined by:

$$execWith\,(x, y, ops) = foldl\ step\,(x, y)\ ops$$

It is just replacing the constant $([\,], [\,])$ in the definition of *exec* with a given argument (x, y). The task is then to show that $execWith^\circ = move^* \cdot start$. One may also think of it as that we have just invented and proposed $execWith^\circ$ to be the loop invariant, and are about to check whether this invariant works. The invariant says that, denoting the input pair of strings by (x, y), and the intermediate values at any point of computing $move^* \cdot start$ by (x', y', ops), executing the commands *ops* on (x', y') shall always yield (x, y).

Now we will check the conditions one by one:

- Condition (i) holds: *start* and *move* are jointly surjective.
- Condition (ii) requires:

  $$\begin{aligned} execWith \cdot start &\subseteq id \\ execWith \cdot move &\subseteq execWith \end{aligned}$$

  The first one trivially holds. The second inclusion holds because *move* undoes the last step of execution. Thus the domain of the left-hand side is restricted to triples where one of the two strings is not empty. The execution still yields the same result.
- For condition (iii): *move* is well-founded because it always reduces the length of the first two components of the triple.

Therefore, we conclude that $execWith^\circ = (\![start, move]\!)_{\mathsf{F}} \cdot (\![id, id]\!)_{\mathsf{F}}^\circ = move^* \cdot start$.

8.2 Building Trees by Combining Pairs

Recall again the following datatype for leaf-valued binary trees:

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

And yes, we are about to introduce yet another approach to building trees out of a list.

The majority of this paper has been focusing on inverting *flatten* to a fold. There is yet another alternative way to build a tree from a list: starting from a list of tips, keep combining adjacent trees until only one is left. The process can be characterised by

$$\mathit{wrap}^{\circ} \cdot \mathit{join}^{*} \cdot \mathit{map}\ \mathit{Tip}$$

where $\mathit{join}\,(x \mathbin{+\!\!+} [a, b] \mathbin{+\!\!+} y) = x \mathbin{+\!\!+} [\mathit{Bin}\ a\ b] \mathbin{+\!\!+} y$.

Our aim is, of course, to show that $\mathit{flatten}^{\circ} = \mathit{wrap}^{\circ} \cdot \mathit{join}^{*} \cdot \mathit{map}\ \mathit{Tip}$. Observe that

$$\mathit{flatten} = \mathit{flattenF} \cdot \mathit{wrap}$$

where $\mathit{flattenF} = \mathit{concat} \cdot \mathit{map}\,\mathit{flatten}$. We have just proposed this invariant for the loop: that during the iterations, the forest always flattens to the given list. Now we check that $\mathit{flattenF}^{\circ} = \mathit{join}^{*} \cdot \mathit{map}\ \mathit{Tip}$:

- Indeed, *map Tip* and *join* are jointly surjective. The former covers any lists of tip trees while the latter covers the rest.
- We need to verify that:

$$\begin{aligned} \mathit{concat} \cdot \mathit{map}\,\mathit{flatten} \cdot \mathit{map}\ \mathit{Tip} &\subseteq \mathit{id} \\ \mathit{concat} \cdot \mathit{map}\,\mathit{flatten} \cdot \mathit{join} &\subseteq \mathit{concat} \cdot \mathit{map}\,\mathit{flatten} \end{aligned}$$

  The first inclusion obviously holds. The second holds because *join* restricts the domain of the left-hand side to lists with at least two trees, but not affecting the result returned.
- Finally, *join* is well-founded because it reduces the length of the forest.

It then follows that $\mathit{flattenF}^{\circ} = \mathit{join}^{*} \cdot \mathit{map}\ \mathit{Tip}$ and, consequently, $\mathit{flatten}^{\circ} = \mathit{wrap}^{\circ} \cdot \mathit{join}^{*} \cdot \mathit{map}\ \mathit{Tip}$.

One might relate this small exercise to merge sort. There are two ways to implement merge sort: one is to implement it as a hylomorphism, where the unfolding phase expands a tree and the folding phase performs merging at each node. The other is to implement it as a loop: to start with a *map wrap*, converting the input to a list of singleton lists, and then to iteratively merge adjacent lists until only one list is left. The first can be said to be top-down and the second bottom-up. A similar reasoning converts the former to the latter. However, an additional distributivity property of list merging will be needed in the proof. A similar problem was treated in [21], where a top-down algorithm was also transformed to a bottom-up one.

9 Conclusions and Related work

The idea of program inversion can be traced back to Dijkstra [12]. However, given the importance of inversion as a specification technique, relatively few papers have been devoted to the topic. Of those that have, most deal with program inversion in the context of imperative programs and refinement calculus. A program is inverted by running it "backwards" and the challenging part is when we encounter a branch or a loop [37,2,34]. The classic example was to construct a binary tree given its inorder and preorder traversal [18,19,8,36,35].

Inversion of functional programs has received even less attention. Most published results (e.g. [28,20]) 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. This rather control-oriented view is complemented by a more data-oriented view in [23,24]. The paper generalised functions to arrows. They then considered polytypic operations on datatypes and ensured that an operation and its inverse carrying things out in reverse order (such as "map from the left" and "map from the right") are always constructed in pairs. 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. 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.

This technique is not new. Similar techniques have been adopted in, for example, [25] and [32]. However, to the best our knowledge, it was de Moor [6,29] who first presented the technique as a theorem, suggesting a wider range of application. The problem dealt with in [29] was precedence parsing, leading to a derivation of Floyd's algorithm. It is therefore not a coincidence that the algorithms we developed in Section 4 resemble parsing. The authors believe that it is possible, although a tiresome task, to derive a shift-reduce parsing algorithm by generalising the reasoning in Section 4.

It was also pointed out that the problem of building trees of minimum height can be seen as a special case of Knuth's generalised shortest path problem [26]. The problem addressed was, given a context-free grammar and a cost function on parse trees, to construct a word and its parse tree whose cost is minimum. Given a list of numbers, we can construct an ambiguous grammar whose only word is the list, while the possible parse trees include all binary trees. The cost of a parse tree would simply be its height. Knuth's algorithm can thus be applied to find the best parse yielding the minimum height. It would be interesting to investigate whether the linear time algorithm in Section 5 is an optimised special case and how they relate to each other.

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 to a particular function. Part of the answer is given in [16]: 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 [16] can thus be used as a test before applying the converse-of-a function theorem.

One possible reason why inverting imperative programs were more often talked about could be that theories about non-determinism in the context of refinement calculus are more established. In [37,2], for instance, Dijkstra's guarded command language was extended to include angelic choices as well as demonic choice. It was then shown that the inverse of a demonic program is angelic. Corresponding theories for relations are still being developed [33]. It is interesting to see how that would benefit the research about inverses for relations.

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. We have applied the theorem to the simple cases that $\mathsf{F}(A, X) = A + X$ 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 kindly pointed out the relation with Henk Doornbos's work and the advantage of basing the theorem on F-reductivity. The authors would also like to thank the anonymous referees for detailed and useful advices.

References

[1] S. M. Abramov. 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 187–202. Springer-Verlag, 2000.

[2] R. J. R. Back and J. von Wright. Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.

[3] R. C. Backhouse, P. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers, 1991.

[4] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. A. 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.

[5] R. S. Bird. On building trees with minimum height. *Journal of Functional Programming*, 7(4):441–445, 1997.

[6] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[7] 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.

[8] W. Chen and J. T. Udding. Program inversion: more than fun! *Science of Computer Programming*, 15:1–13, 1990.

[9] 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.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, July 2001.

[11] S. Curtis. *A Relational Approach to Optimization Problems*. PhD thesis, Oxford University Computing Laboratory, 1995.

[12] E. W. Dijkstra. Program inversion. Technical Report EWD671, Eindhoven University of Technology, 1978.

[13] H. Doornbos. *Reductivity Arguments and Program Construction*. PhD thesis, Eindhoven University of Technology, 1996.

[14] H. Doornbos and R. C. Backhouse. Induction and recursion on datatypes. In B. Möller, editor, *Mathematics of Program Construction, 3rd International Conference*, number 947 in Lecture Notes in Computer Science, pages 242–256. Springer-Verlag, July 1995.

[15] H. Doornbos and R. C. Backhouse. Reductivity. *Science of Computer Programming*, 26:217–236, 1996.

[16] 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.

[17] 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.

[18] D. Gries. *The Science of Programming*. Springer Verlag, 1981.

[19] 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.

[20] P. G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.

[21] R. Hinze. Constructing tournament representations: An exercise in pointwise relational programming. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, Dagstuhl, Germany, July 2002. Springer-Verlag.

[22] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science*, number 1113 in Lecture Notes in Computer Science, pages 407–418, Cracow, September 1996. Springer-Verlag.

[23] P. Jansson and J. T. Jeuring. Polytypic compact printing and parsing. In S. D. Swierstra, editor, *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, number 1576 in Lecture Notes in Computer Science, pages 273–287. Springer-Verlag, 1999.

[24] P. Jansson and J. T. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002. Technical report Utrecht University UU-CS-2001-34, 2001.

[25] E. Knapen. Relational Programming, Program Inversion, and the Derivation of Parsing Algorithms. Master's thesis, Eindhoven University of Technology, 23 November 1993.

[26] D. E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[27] D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms, 3rd Edition*. Addison Wesley, 1997.

[28] 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.

[29] 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.

[30] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[31] 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.

[32] 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.

[33] I. W. Rewitzky. Binary multirelations. Work in progress.

[34] B. J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing Journal*, 9:331–348, 1997.

[35] 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.

[36] 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` .

[37] J. von Wright. Program inversion in the refinement calculus. *Information Processing Letters*, 37:95–100, 1991.