Rebuilding a Tree from Its Traversals:
A Case Study of Program Inversion

Shin-Cheng Mu[1] and Richard Bird[2]

[1] Information Processing Lab, Dep. of Math. Informatics, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
[2] Programming Research Group, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract. Given the inorder and preorder traversal of a binary tree whose labels are all distinct, one can reconstruct the tree. This article examines two existing algorithms for rebuilding the tree in a functional framework, using existing theory on function inversion. We also present a new, although complicated, algorithm by trying another possibility not explored before.

1 Introduction

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 [10, Sect. 2.3.1, Exercise 7] as an exercise; 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 classic problem to tackle for those who study program inversion. For example, see [5,15].

All of the above work on the problem is based on program inversion in an imperative style. As van de Snepscheut noted in [15], one class of solutions attempts to invert an iterative algorithm, while the other class delivers a recursive algorithm. In this article we will look at the problem in a functional style, and attempt to derive these algorithms using existing theory on function inversion.

2 Problem Specification

To formalise the problem, consider internally labelled binary trees defined by the following datatype:

$$\textbf{data}\ \mathit{Tree}\,\alpha = \mathit{Null} \mid \mathit{Node}\,(\alpha, \mathit{Tree}\,\alpha, \mathit{Tree}\,\alpha)$$

Inorder and preorder traversal on the trees can then be defined as:

$$\begin{aligned}
&\mathit{preorder}, \mathit{inorder} && :: \mathit{Tree}\,\alpha \to [\alpha] \\
&\mathit{preorder}\ \mathit{Null} && = [\,] \\
&\mathit{preorder}\,(\mathit{Node}\,(a, u, v)) && = [a] +\!\!+ \mathit{preorder}\ u +\!\!+ \mathit{preorder}\ v \\
&\mathit{inorder}\ \mathit{Null} && = [\,] \\
&\mathit{inorder}\,(\mathit{Node}\,(a, u, v)) && = \mathit{inorder}\ u +\!\!+ [a] +\!\!+ \mathit{inorder}\ v
\end{aligned}$$

A. Ohori (Ed.): APLAS 2003, LNCS 2895, pp. 265–282, 2003.

Define *pinorder* to be the function returning both the preorder and inorder of a tree:

$$\begin{aligned} &pinorder :: Tree\,\alpha \to ([\alpha], [\alpha]) \\ &pinorder = fork\,(preorder, inorder) \end{aligned}$$

where $fork\,(f, g)\,a = (f\,a, g\,a)$. The task involves constructing the inverse of *pinorder*. But what exactly does *the inverse* mean?

A function $f :: \alpha \to \beta$ has inverse $f^{-1} :: \beta \to \alpha$ if for all $a :: \alpha$ and $b :: \beta$ we have[1]:

$$f\,a = b \equiv f^{-1}\,b = a$$

This definition implies that f^{-1} is also a (possibly partial) function, a restriction which we will relax in Sect. 4.1. This function f^{-1} exists if and only if f is an injection – that is, for all b there exists at most one a such that $f\,a = b$. For instance, *id*, the identity function, is inverse to itself. In the case of *pinorder*, its inverse exists only when we restrict the domain of input trees to those containing no duplicated labels. To be more formal, we should have included this constraint as a predicate in the definition of *pinorder*. To reduce the amount of details, however, we will allow a bit of hands waving and assume that *pinorder* is a partial function taking only trees with labels all distinct.

3 The Compositional Approach

Most published work on program inversion is based on what we call the *compositional approach*, be it procedural [6,7,5,16,15,14] or functional [8]. The basic strategy, from a functional perspective, is to exploit various distributivity laws. In particular, provided that f^{-1} and g^{-1} both exist, we have that

$$(f \cdot g)^{-1} = g^{-1} \cdot f^{-1} \tag{1}$$

and that

$$(f \cup g)^{-1} = f^{-1} \cup g^{-1} \tag{2}$$

if f and g have disjoint ranges. Here the $\cup$ operator denotes set union, if we see a function as a set of pairs of input and output. A function defined in many clauses can be seen as the union of the separate clauses. Defining the product operator on pairs of functions to be:

$$(f \times g)\,(a, b) = (f\,a, g\,b)$$

(which generalises to triples in the obvious way), we also have that inverses distribute into products:

$$(f \times g)^{-1} = (f^{-1} \times g^{-1}) \tag{3}$$

[1] The symbol $\equiv$ is read "if and only if".

Finally, we have $(f^{-1})^{-1} = f$ and $id^{-1} = id$. In order to compute f^{-1}, the compositional approach to function inversion starts with expanding the definition of f and try to push the inverse operator to the leaves of the expression. The process continues until we reach some primitive whose inverse is either predefined or trivial. With this approach, a sequentially composed program is "run backwards". The challenging part is when we encounter conditionals, in such cases we have somehow to decide which branch the result was from. This is the approach we will try in this section.

3.1 Unfolding a Tree

Standard transformations yield the following recursive definition of *pinorder*:

$$\begin{aligned}
&\mathit{pinorder}\ \mathit{Null} &&= ([\,],[\,])\\
&\mathit{pinorder}\ (\mathit{Node}(a,x,y)) &&= \mathit{pi}\ (a, \mathit{pinorder}\ x, \mathit{pinorder}\ y)\\
&\quad \textbf{where}\ \mathit{pi} &&:: (\alpha,([\alpha],[\alpha]),([\alpha],[\alpha])) \to ([\alpha],[\alpha])\\
&\qquad\qquad \mathit{pi}\ (a,(x_1,y_1),(x_2,y_2)) &&= ([a] +\!\!+ x_1 +\!\!+ x_2, y_1 +\!\!+ [a] +\!\!+ y_2)
\end{aligned}$$

Notice first that the two clauses of *pinorder* have disjoint ranges – the second clause always generates pairs of non-empty lists. According to (2), to construct the inverse of *pinorder* we can invert the two clauses separately and join them together. The first clause can simply be inverted to a partial function taking $([\,],[\,])$ to *Null*. The second clause of *pinorder* can also be written in point-free style as

$$\mathit{pi} \cdot (a \times \mathit{pinorder} \times \mathit{pinorder}) \cdot \mathit{Node}^{-1}$$

where Node^{-1} corresponds to pattern matching. Its inverse, according to (1) and (3), is

$$\mathit{Node} \cdot (a \times \mathit{pinorder}^{-1} \times \mathit{pinorder}^{-1}) \cdot \mathit{pi}^{-1}$$

We can invert *pinorder* if we can invert *pi*. In summary, define:

$$\begin{aligned}
&\mathit{rebuild}\ ([\,],[\,]) &&= \mathit{Null}\\
&\mathit{rebuild}\ (x,y) &&= (\mathit{Node} \cdot (\mathit{id} \times \mathit{rebuild} \times \mathit{rebuild}) \cdot \mathit{pi}^{-1})\ (x,y)
\end{aligned}$$

The function *rebuild* is the inverse of *pinorder* if pi^{-1} exists.

Note that the wish that pi^{-1} exists is a rather strong one. Every injective function f has a functional inverse f^{-1}. As we shall see in Sect. 4.1, however, it is possible that a non-injective function f does not have a functional inverse, but its *converse*, the notion of inverse generalised to relations, reduces to a function after being composed with something else. In the particular case here, however, *pi* does have a functional inverse if its domain is restricted to triples $(a,(x_1,y_1),(x_2,y_2))$ where a does not present in either y_1 or y_2, and that the length of x_i equals y_i for $i \in \{1,2\}$ (it can be proved inductively that it is indeed the case in the usage of *pi* in *pinorder* if the input tree has distinct labels). Such a partial function *pi* has as its inverse:

$$\begin{aligned}
&\mathit{pi}^{-1}\ (a:x,y) = (a,(x_1,y_1),(x_2,y_2))\\
&\quad \textbf{where}\ y_1 +\!\!+ [a] +\!\!+ y_2 = y\\
&\qquad\quad x_1 +\!\!+_{(\mathit{len}\ y_1)} x_2 = x
\end{aligned}$$

where the pattern $x_1 +\!\!+_n x_2 = x$ splits into two such that x_1 has length n. Haskell programmers would have written $(x_1, x_2) = splitAt\ n\ x$.

This is how in [10] Knuth explained why indeed the tree can be uniquely constructed. However, a naive implementation would result in a cubic time algorithm, because searching for an a in y and splitting x are both linear-time operations. In the next sections we will remove this linear-time overhead by a common functional program derviation technique.

3.2 Eliminating Repeated Traversals

Directly implementing *rebuild* as defined in the last section results in a slow algorithm because the input lists are repeatedly traversed. For example, the list x is split into x_1 and x_2. This requires traversing x from the front. In the next level of recursion, however, x_1 will also be split into two, and the prefix of x is thus traversed again.

It is a common technique in functional program derivation to reorder the splitting of lists such that the repeated traversal can be avoided by introducing extra outputs and exploiting the associativity of $+\!\!+$. Define *reb* as a partial function on non-empty x and y as:

$$
\begin{array}{l}
reb\ a\ (x, y) = (rebuild\ (x_1, y_1), x_2, y_2) \\
\quad \textbf{where}\ y_1 +\!\!+ [a] +\!\!+ y_2 = y \\
\qquad\quad\ \ x_1 +\!\!+_{(len\ y_1)} x_2 = x
\end{array}
$$

The aim now is to derive a recursive definition for *reb*. The case when y starts with a is relatively simple:

$$
\begin{array}{ll}
 & reb\ a\ (x, a : y) \\
= & \quad \{\text{by definition, since } [\,] +\!\!+ [a] +\!\!+ y = a : y\} \\
 & (rebuild\ ([\,], [\,]), x, y) \\
= & \quad \{\text{by definition of } rebuild\} \\
 & (Null, x, y)
\end{array}
$$

For the general case, we derive:

$$
\begin{array}{ll}
 & reb\ a\ (b : x, y) \\
= & \quad \{\text{by definition, let } y_1 +\!\!+ [a] +\!\!+ y_2 = y \\
 & \qquad \text{and } x_1 +\!\!+_{(len\ y_1)} x_2 = x\} \\
 & (rebuild\ (b : x_1, y_1), x_2, y_2) \\
= & \quad \{\text{by definition of } rebuild, \text{ let } y_3 +\!\!+ [b] +\!\!+ y_4 = y_1 \\
 & \qquad \text{and } x_3 +\!\!+_{(len\ y_3)} x_4 = x_1\} \\
 & (Node\ (b, rebuild\ (x_3, y_3), rebuild\ (x_4, y_4)), x_2, y_2) \\
= & \quad \{\text{since } y_3 +\!\!+ [b] +\!\!+ (y_4 +\!\!+ [a] +\!\!+ y_2) = y \\
 & \qquad \text{and } x_3 +\!\!+_{(len\ y_3)} (x_4 +\!\!+ x_2) = x; \text{ we have} \\
 & \qquad (t_3, x_4 +\!\!+ x_2, y_4 +\!\!+ [a] +\!\!+ y_2) = reb\ b\ (x, y) \text{ for some } t_3\}
\end{array}
$$

$$
\begin{aligned}
& (Node\,(b, t_3, rebuild\,(x_4, y_4)), x_2, y_2) \\
= \quad & \{\text{let } (t_4, x_2, y_2) = reb\; a\,(x_4 +\!\!+ x_2, y_4 +\!\!+ [a] +\!\!+ y_2)\} \\
& (Node\,(b, t_3, t_4), x_2, y_2)
\end{aligned}
$$

Denoting $x_4 +\!\!+ x_2$ by x_5 and $y_4 +\!\!+ [a] +\!\!+ y_2$ by y_5, we have just derived the following recursive definition of *reb*:

$$
\begin{aligned}
reb\; a\,(x, a : y) &= (Null, x, y) \\
reb\; a\,(b : x, y) &= \mathbf{let}\ (t_3, x_5, y_5) = reb\; b\,(x, y) \\
&\qquad\quad (t_4, x_2, y_2) = reb\; a\,(x_5, y_5) \\
&\quad \mathbf{in}\ \ (Node\,(b, t_3, t_4), x_2, y_2)
\end{aligned}
$$

The use of duplicated as in the first pattern is non-standard. In Haskell we would need an explicit equality test.

To be complete, we still need to work out a definition of *rebuild* in term of *reb*. Since *reb* is defined on non-empty input only, we deal with the empty case separately:

$$rebuild\,([\,], [\,]) = Null$$

For the non-empty case, the following equivalence is trivial to verify:

$$
\begin{aligned}
rebuild\,(a : x, y) = \mathbf{let}\ & (t_1, x', y') = reb\; a\,(x, y) \\
& t_2 = rebuild\,(x', y') \\
\mathbf{in}\ \ & Node\,(a, t_1, t_2)
\end{aligned}
$$

A Haskell implementation of the algorithm is given in Fig. 1. We have actually reinvented the recursive algorithm in [15] in a functional style. In the next section, we will continue to see how the other, iterative, class of algorithms can be derived functionally.

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

rebuild :: Eq a => ([a],[a]) -> Tree a
rebuild ([],[]) = Null
rebuild (a:x, y) = let (t, x', y')   = reb a (x,y)
                   in Node a t (rebuild (x',y'))

reb :: Eq a => a -> ([a],[a]) -> (Tree a,[a],[a])
reb a (x@(~(b:x1)),y)
| head y == a  =  (Null, x, tail y)
| otherwise    =  let (t1, x', y')   = reb b (x1, y)
                      (t2, x'', y'') = reb a (x', y')
                  in (Node b t1 t2, x'', y'')
```

Fig. 1. Haskell code implementing $pinorder^{-1}$.

4 Two Folding Algorithms

In Sect. 3, we started from the original program, and constructed a program in one go that builds a tree having both the specified inorder and preorder traversal. The expressiveness of relations, on the other hand, allows us to deal with the problem in a more modular way. In this section we will try a different approach, by first constructing a non-deterministic program that builds a tree with the specified inorder traversal, and then refine the program such that it generates only the tree with the required preorder traversal.

To talk about a program that *non-deterministically* generates a possible result, we will introduce relations in Sect. 4.1. The development of the algorithm will be presented in Sect. 4.2 and Sect. 4.3. The derived algorithm, expressed as a fold on in the input list, turns out to be the functional counterpart of Chen and Udding's algorithm in [5]. Finally in Sect. 4.4, we will briefly discuss an alternative algorithm dual to the on in Sect. 4.3.

4.1 From Functions to Relations

A relation of type $\alpha \to \beta$ is a set of pairs (a, b), where $a :: \alpha$ is drawn from the domain and $b :: \beta$ from the range. A function is a relation that is *simple* (a value in the domain is mapped to no more than one value in the range) and *entire* (every value in the domain is mapped to something). Composition of relations is defined by:

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

If we relax the entireness condition, we get partial functions. A useful class of partial functions is given by the *coreflexsives*. A coreflexive is a sub-relation of id. It serves as a filter, letting through only those values satisfying certain constraints. We denote the conversion from predicates (boolean-valued functions) to coreflexives by the operator ?. For a predicate p, we have

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

Given a relation $R :: \alpha \to \beta$, its *converse*, denoted by $R^\circ :: \beta \to \alpha$, is obtained by swapping the pairs in R:

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

It is the generalisation of inverse to relations.

Real programs are deterministic. Therefore the result of a derivation has to be a function in order to be mapped to a program. Being able to have relations as intermediate results, however, allows the derivation to proceed in a more flexible way, as we will see in the incoming sections.

4.2 Unflattening an Internally Labelled Binary Tree

Back to the problem of building trees from its traversals. Or aim was to construct $pinorder^\circ = (fork\,(preorder, inorder))^\circ$, which takes a pair of traversals and

yields a tree. The familiar Haskell function *curry* converts a function on pairs to a higher-order function, defined below[2]:

$$\begin{aligned} &\mathit{curry} \quad :: ((\alpha, \beta) \to \gamma) \to \alpha \to \beta \to \gamma \\ &\mathit{curry}\, f\ a\ b = f\,(a, b) \end{aligned}$$

When the function is the converse of a fork, we have the following property:

$$\mathit{curry}\,(\mathit{fork}\,(R, f))^\circ\ a = ((a\ ==) \cdot R)? \cdot f^\circ \tag{4}$$

To get an intuition of (4), let R, f and a have types $\gamma \to \alpha$, $\gamma \to \beta$, and α, respectively. The term $(\mathit{fork}\,(R, f))^\circ$ thus has type $(\alpha, \beta) \to \gamma$. The left-hand side of (4) takes a value $b :: \beta$, passes the pair (a, b) to $(\mathit{fork}\,(R, f))^\circ$, and returns $c :: \gamma$ such that c is mapped to a by R and $f\ c = b$. The right-hand side does the same by mapping b to an arbitrary c using f°, and taking only those cs mapped to a by R. The proof of (4) relies on expressing forks as *intersections* of relations and applying more primitive algebraic relational calculus. Interested readers are referred to [12].

Define $\mathit{rebuild} = \mathit{curry}\,\mathit{pinorder}$. By (4), we have

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

The converse of *inorder* takes a list as the input, and maps it to an arbitrary tree whose inorder traversal is the given list. The coreflexive $((x\ ==) \cdot \mathit{preorder})?$ then acts as a filter, picking the tree whose preorder traversal is the list x. Our aim is to construct $\mathit{rebuild}\ x$, assuming that x does not contain duplicated elements. Writing *rebuild* this way is possible only with the expressiveness of relations.

The aim of this section is to invert *inorder*. In [13], the same approach was applied to solve similar problems, building trees under some optimal or validity constraints. They all use the same core algorithm developed in this section but differ in the refinement step to be described in the next section.

How does one build a tree with a given inorder traversal? In the compositional approach, since *inorder* is defined as a fold, its inverse would be constructed as an unfold, like what we did in Sect. 3.1.

Alternatively, one can build such a tree as a fold over the given list. So, given a tree whose inorder traversal is x, how does one add another node to the tree, such that the inorder traversal of the new tree is $a : x$? One way to do it is illustrated in Fig. 2: we divide the left spine of the tree in two parts, move down the lower part for one level, and attach a to the end. If we call this operation *add*, the relational fold *foldr add Null* constructs a tree whose inorder traversal is the given list, i.e., *foldr add Null* is a sub-relation of $\mathit{inorder}^\circ$. But can we construct *all* the legal trees this way? In other words, does $\mathit{inorder}^\circ$ equal *foldr add Null*?

The *converse-of-a-function theorem* [4,11,13] gives conditions under which the converse of a function equals a fold. What matters is whether the function satisfies certain properties, rather than its particular definition. Its specialised version for lists reads:

[2] Cautious readers would notice that f here is a function while *curry* in (4) takes relational arguments. Factoring out the variable b and substitute f for R, we get $\mathit{curry}\ R\ a = R \cdot \mathit{fork}\,(\mathit{const}\ a, \mathit{id})$, which can be taken as its generalisation to relations.

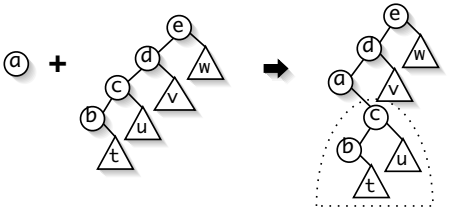

Fig. 2. Spine representation for internally labelled trees.

Theorem 1. Given a function $f :: \beta \to [\alpha]$, if we can find a relation $step :: \alpha \to \beta \to \beta$ and a value $base :: \beta$ that are jointly surjective and :

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

then we have:

$$f^\circ = foldr\ step\ base$$

By *jointly surjectiveness* we mean that *base* and the range of *step* covers all the values in β.

Returning to *add*, we could have start with deriving $inorder^\circ$ directly using Theorem 1. To allow an efficient implementation of *add*, however, it turns out that it is preferable to introduce another representation for trees. The following type $Spine\ a$ represents the spine of an internally labelled binary tree:

$$\mathbf{type}\ Spine\ \alpha = [(\alpha, Tree\ \alpha)]$$

For example, the tree on the left-hand side of Fig. 2 is represented by the list

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

The conversion from a spine tree to the ordinary representation can be performed by the function *roll* defined below:

$$\begin{aligned} &roll :: Spine\ \alpha \to Tree\ \alpha \\ &roll = foldl\ join\ Null \\ &\quad \mathbf{where}\ join\ u\ (a,v) = Node\ (a,u,v) \end{aligned}$$

The reader is reminded that the introduction of *Spine* is merely for the sake of efficiency. As we will see later, with *Spine* it is easier to attach leaves as shown in Fig. 2. Without *Spine*, we still get the same algorithm, but on a different datatype.

Our aim now is to find a relation *add* and a value *zero* such that

$$(inorder \cdot roll)^\circ = foldr\ add\ zero$$

According to the converse-of-a-function theorem, the above equality holds if

$$inorder\,(roll\,zero) = [\,] \tag{5}$$
$$inorder\,(roll\,(add\,a\,us)) = a : inorder\,(roll\,us) \tag{6}$$

To satisfy (5), we can choose $zero = Null$. To derive an add satisfying (6), we will need a property distributing $inorder$ into the subtrees on the spine:

$$inorder \cdot roll = concat \cdot map\,(cons \cdot (id \times inorder)) \tag{7}$$

where the functor $\times$ is define by $(f \times g)\,(x, y) = (f\,x, g\,y)$. Starting from the left-hand side of (6) and assume $us = vs +\!\!+ ws$, we derive:

$$\begin{aligned}
& a : inorder\,(roll\,(vs +\!\!+ ws)) \\
= \quad & \{(7)\} \\
& a : concat\,(map\,(cons \cdot (id \times inorder))\,(vs +\!\!+ ws)) \\
= \quad & \{\text{since } concat \text{ and } map \text{ distributes over } +\!\!+\} \\
& a : concat\,(map\,(cons \cdot (id \times inorder))\,vs) +\!\!+ \\
& \quad\; concat\,(map\,(cons \cdot (id \times inorder))\,ws) \\
= \quad & \{(7)\} \\
& a : inorder\,(roll\,vs) +\!\!+ \\
& \quad\; concat\,(map\,(cons \cdot (id \times inorder))\,ws) \\
= \quad & \{\text{definition of } concat \text{ and } map\} \\
& concat\,(map\,(cons \cdot (id \times inorder))\,((a, roll\,vs) : ws)) \\
= \quad & \{(7)\} \\
& inorder\,(roll\,((a, roll\,vs) : ws))
\end{aligned}$$

We therefore have $add\,a\,(vs +\!\!+ ws) = (a, roll\,vs) : ws$. The cases for $add\,a\,[\,]$ and $add\,a\,[u]$ turn out to be absorbed by the above case, and the derivations are left as exercises for the reader. Thus we choose:

$$\begin{aligned}
& add && :: \alpha \to Spine\,\alpha \to Spine\,\alpha \\
& add\,a\,(us +\!\!+ vs) && = (a, roll\,us) : vs
\end{aligned}$$

It is not a function because of the non-deterministic pattern $us +\!\!+ vs$.

It is also not difficult to see that $Null$ and add are jointly surjective, since if a tree is not $Null$, it must be a result of adding its leftmost element on the spine to some tree. We therefore conclude that

$$(inorder \cdot roll)^{\circ} = foldr\;add\;Null$$

Since $roll$ is an isomorphism, we have $inorder^{\circ} = roll \cdot foldr\;add\;Null$.

For some intuition what add does, consider Fig. 2 again. As mentioned before, the tree on the left-hand side is represented in the spine representation as $[(b, t), (c, u), (d, v), (e, w)]$. The one on the right-hand side, on the other hand, is represented by $[(a, roll\,[(b, t), (c, u)]), (d, v), (e, w)]$ and results from splitting the former list, one of the five ways to split it into two.

4.3 Enforcing a Preorder

Now recall our the specification of *rebuild*

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

In the last section we have inverted *inorder* as a relational fold and switched to a spine representation, yielding:

$$\begin{aligned} & rebuild\ x \\ = & ((x\ ==) \cdot preorder)? \cdot roll \cdot foldr\ add\ Null \\ = & roll \cdot (hasPreorder\ x)? \cdot foldr\ add\ Null \end{aligned}$$

where $hasPreorder\ x = (x\ ==) \cdot preorder \cdot roll$. We now want to fuse $(hasPreorder\ x)?$ into the fold to efficiently generate the one tree which has the correct preorder traversal. The *fold-fusion* theorem for lists says:

Theorem 2.

$$R \cdot foldr\ S\ e = foldr\ T\ d \Leftarrow R \cdot S\ a = T\ a \cdot R \wedge ((e, d') \in R \equiv d' = d)$$

However, $hasPreorder\ x$ is too strong an invariant to enforce within the fold: it is impossible to make the constructed tree to have the same preorder traversal in each iteration.

Instead, we will try to find a weaker constraint to fuse into the fold. Define *preorderF* to be the preorder traversal of forests:

$$preorderF = concat \cdot map\ preorder$$

Look at Fig. 2 again. The preorder traversal of the tree on the left-hand side is

$$[e, d, c, b] +\!\!+ preorderF\ [t, u, v, w]$$

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

$$reverse\ (map\ fst\ us) +\!\!+ preorderF\ (map\ snd\ us)$$

We will call the part before $+\!\!+$ the *prefix* and that after $+\!\!+$ the *suffix* of the traversal. Now look at the tree on the right-hand side. Its preorder traversal is

$$[e, d, a, c, b] +\!\!+ preorderF\ [t, u, v, w]$$

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 *hasPreorder* into two parts:

$$\begin{aligned} & hasPreorder && :: Eq\ \alpha \Rightarrow [\alpha] \rightarrow Spine\ \alpha \rightarrow Bool \\ & hasPreorder\ x\ us && = prefixOk\ x\ us \wedge suffixOk\ x\ us \\ & suffixOk\ x\ us && = preorderF\ (map\ snd\ us)\ \mathbf{isSuffixOf}\ x \\ & prefixOk\ x\ us && = reverse\ (map\ fst\ us) == (x \ominus preorderF\ (map\ snd\ us)) \end{aligned}$$

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

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

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 applications). 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 Fig. 2. Assume we know that the preorder traversal of the tree we want is $x = [..\, d, c, b] +\!\!+ \mathit{preorderF}\, [t, u, v, w]$. The tree in the right-hand side of Fig. 2, 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:

$$\begin{aligned}
&\mathit{lookahead} && :: \mathit{Eq}\, \alpha \Rightarrow [\alpha] \to \mathit{Spine}\, \alpha \to \mathit{Bool} \\
&\mathit{lookahead}\, x\, us &&= \mathit{length}\, us \leq 1 \vee \\
&&&\quad (\mathit{map}\, \mathit{fst}\, us)\, !!\, 1 \neq \mathit{last}\, x' \\
&\quad \textbf{where } x' = x \ominus \mathit{preorderF}\, (\mathit{map}\, \mathit{snd}\, us)
\end{aligned}$$

Apparently $\mathit{lookahead}\, x$ is weaker than $\mathit{hasPreorder}\, x$. Define

$$\mathit{ok}\, x\, us = \mathit{suffixOk}\, x\, us \wedge \mathit{lookhead}\, x\, us$$

which will be our invariant in the fold, we have

$$\mathit{hasPreorder}\, x\, us = \mathit{prefixOk}\, x\, us \wedge \mathit{ok}\, x\, us \tag{8}$$

The derivation goes:

$$\begin{aligned}
& \mathit{rebuild} \\
= \quad & \{\text{definition}\} \\
& ((x ==) \cdot \mathit{preorder})? \cdot \mathit{inorder}^{\circ} \\
= \quad & \{\text{inverting } \mathit{inorder}\text{, moving } \mathit{roll} \text{ to the left}\} \\
& \mathit{roll} \cdot (\mathit{hasPreorder}\, x)? \cdot \mathit{foldr}\, \mathit{add}\, \mathit{null} \\
= \quad & \{\text{by (8)}\} \\
& \mathit{roll} \cdot (\mathit{prefixOk}\, x)? \cdot (\mathit{ok}\, x)? \cdot \mathit{foldr}\, \mathit{add}\, \mathit{null} \\
= \quad & \{\text{fold fusion, assume } \mathit{nodup}\, x\} \\
& \mathit{roll} \cdot (\mathit{prefixOk}\, x)? \cdot \mathit{foldr}\, (\mathit{add}'\, x)\, \mathit{null}
\end{aligned}$$

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

$$(\mathit{ok}\, x)? \cdot \mathit{add}\, a = \mathit{add}'\, x\, a \cdot (\mathit{ok}\, x)?$$

where add' is defined by:

$$
\begin{aligned}
&add' :: Eq\,\alpha \Rightarrow [\alpha] \to (\alpha, Spine\,\alpha) \to Spine\,\alpha \\
&add'\,x\,(a, us) = up\;a\;null\;(us, x \ominus preorderF\;(map\;snd\;us))
\end{aligned}
$$

$$
\begin{aligned}
&up\;\; :: \;\; Eq\,\alpha \Rightarrow \alpha \to Tree\,\alpha \to (Spine\,\alpha, [\alpha]) \to Spine\,\alpha \\
&up\;a\;v\;([\,], x) \;\; = \;\; [(a, v)] \\
&up\;a\;v\;((b, u) : us, x +\!\!+ [b']) \\
&\quad \mid b == b' \quad\;\; = \;\; up\;a\;(node\;(b, (v, u))\;(us, x) \\
&\quad \mid otherwise = \;\; (a, v) : (b, u) : us
\end{aligned}
$$

In words, the function *up* traces the left spine upwards and consume 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.

As a final optimisation, we can avoid re-computing $x \ominus preorderF\;(map\;snd\;us)$ from scratch by applying a tupling transformation, having the fold returning a pair. The Haskell implementation is shown in Fig. 3. The fold in *rebuild* returns a pair of a tree and 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. 3. Rebuilding a tree from its traversals via a fold.

Figure 4 shows an example of this algorithm in action. The part in boldface font indicates $preorderF\;(map\;snd\;us)$. Notice how the preorder traversals on 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 [5], but in a functional style.

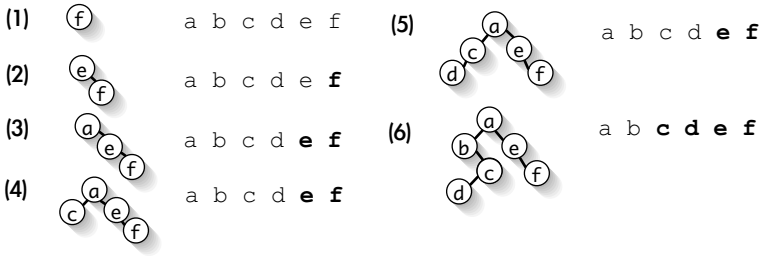

Fig. 4. Building a tree from preorder `abcdef` and inorder `bdcaef`. The preorder traversals of the trees under the spine is written in boldface font.

4.4 Building a Tree with a Given Preorder

The reader might justifiably complain that the derivation works because, by luck, we choose to invert *inorder* first. Had we started with defining $pinorder = fork\,(inorder, preorder)$, we would have come up with

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

We would then have to invert *preorder*, and then enforce, on the resulting fold, the constraint that the tree built must have a given inorder traversal. Does the alternative still work? In fact, it does, and the result is a new, though more complicated, algorithm. We sketch an outline of its development in this section.

We first seek to invert *preorder*. For this problem it turns out that it makes more sense to work on forests rather than trees. Recall $preorderF :: [Tree\,\alpha] \to [\alpha]$ defined by $preorderF = concat \cdot map\,preorder$. The reader can easily verify, with the converse-of-a-function theorem, that *preorderF* can be inverted as below:

$$\begin{aligned} preorderF^{\circ} &= foldr\ step\ [\,] \\ step\ a\ us &= tip\ a : us \\ &\quad \Box\ lbr\,(a, head\ us) : tail\ us\ \Box\ rbr\,(a, head\ us) : tail\ us \\ &\quad \Box\ node(a, (us!!0, us!!1)) : tail\,(tail\ us) \end{aligned}$$

where the $\Box$ operator denotes non-deterministic choice. 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:

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

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

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

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

Again, the condition $(x\ ==) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder}$ 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 5 demonstrates the process of constructing the same tree as that in Fig. 4. This time notice how the inorder traversal of the constructed forest always forms a subsequence of the given list $[b, d, c, a, e, f]$.

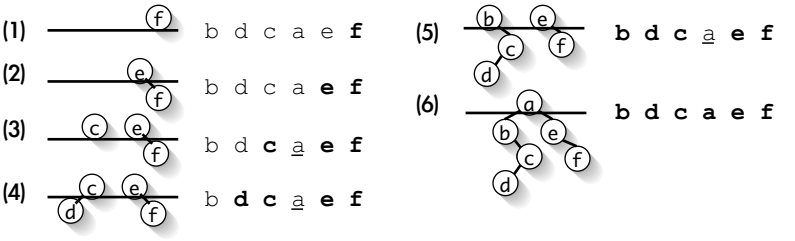

Fig. 5. Yet another way to build a tree from preorder `abcdef` and inorder `bdcaef`.

After some pencil-and-paper work, one will realise that to decide how to extend the forest while maintaining the invariant above, it is necessary to take into consideration two more entities in the given inorder traversal: the skipped segments between the trees in the current forest (the underlined parts in Fig. 5), and the element immediately next to the first tree (e.g., `a` in (2) and `d` in (3) of Fig. 5).

The algorithm is implemented in Fig. 6, where the functions *add* and *join* reflect the rules. In each step of the fold in *rebuild*, the variable x represents the reversed prefix of the given inorder traversal to the left of the first tree, i.e., the "unprocessed" prefix. The function $\mathit{isNext}\ x\ a$ tests whether a is the next element. The skipped segments, on the other hand, is paired with each tree in the forest. The fold thus yields the type $([\alpha], [(\mathit{Tree}\ \alpha, [\alpha])])$.

Let a be the new node to add to the forest. When the forest is empty, the function *newtree* is called to create a new singleton tree. Otherwise denote by t the leftmost tree in the forest. If a is the next element in x, the function *rbr* is called to make t a right branch of a. If a equals the first element in the skipped segment next to t, on the other hand, the function *join* is called to decide whether to attach t as a left subtree or to attach two subtrees to a. When none of the cases hold, we make a a separate singleton tree by *newtree*, which also computes the new x and the new skipped segment.

```
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. 6. An implementation of the second folding algorithm.

All functions called by *add*, apart from *skip*, are constant-time operations, and each element in x is skipped only once. Therefore, the program runs in linear time, but with a bigger constant overhead than that in Sect. 4.3. To the best of our knowledge, this algorithm is new. However, the rules consists of totally eight cases and are relatively complicated comparing to other algorithms in this article. It is due to that we have four possible operations to choose from, while

in Sect. 4.3 there are only two – either to walk upward along the spine or to stop and attach a new node. For that reason we will not go into more detail but just present the result.

5 Conclusion and Related Work

We have looked at two classes of algorithms for the problem of building a binary tree from its traversals. The recursive algorithm turns out to be a result of the compositional approach to function inversion and a technique to eliminate repeated traversals. In [15], similar transform was presented in a procedural style.

The iterative algorithm, on the other hand, is related to the converse-of-a-function theorem. It allows us to consider inorder and preorder traversals separately. The first step in [5] was to transform the recursive definition of *pinorder* into an iteration by introducing a stack. The same effect we achieved by introducing the spine representation. By exploiting symmetricity, we also discovered a new, but complicated, linear-time algorithm. In both cases, the actions we perform on the spine or the forest resemble a shift-reduce parser. In fact, the one of the early motivating example of the converse-of-a-function theorem was precedence parsing [11]. It is certainly possible to derive a full shift-reduce parser using the theorem, although it will be a laborious exercise.

The reader might complain that introducing a spine representation is too inventive a step, if not itself the answer to the problem. Our defence is that the spine representation is there to enable traversing the tree upwards from the left-most tip, which is more a concern of efficiency than an algorithmic one. In fact, for some applications in [12], where the converse-of-a-function theorem is also applied, we actually prefer not to use *Spine*. Some earlier algorithms solve problems similar to that addressed in this paper without the use of the spine representation, at least not explicitly. The derivation of [5] introduced a stack represented by a list, which served the same purpose of the spine we use. In [9] another similar problem was considered. The resulting algorithm does not use the spine explicitly. Instead, the author introduced pairs using a tupling transform. One might see it as implicitly storing the spine in the machine stack.

One of the purposes of this article is to advocate relational program derivation. So far, its most successful application area is probably in dealing with optimisation problems [3,4]. Program inversion may be another application where a relational approach is useful [12]. An alternative way to talk about inverses is by set-valued functions[8]. At least within the two application fields, the relations provide a natural and concise framework, avoiding having to take care of the bookkeeping details of maintaining a set of results. The algebra of relations, however, is notorious for having too many rules, and it remains to see whether the complexity can be kept within a manageable scale. See [2,4] for a fuller introduction to the relational theory of program derivation.

This article concerns only manual program derivation, where the algorithm is the delivered result. Complementarily, [1] introduced their *universal resolving algorithm* to construct an inverse interpreter that actually compute the values

delivered by the inversed function. Furthermore, the inverse interpreter can be partially evaluated to produce a *program* that performs the inverted task.

Acknowledgement The converse-of-a-function theorem was first proposed by Oege de Moor [4,11]. He also significantly simplified the derivation in Sect. 3 after reading an earlier draft of the first author's thesis. The authors would also like to thank the members of the Algebra of Programming group in Oxford University Computing Laboratory for valuable discussions, Robert Glück for encouraging research on this topic, and Zhenjiang Hu for his edits and comments.

References

1. S. M. Abramov and R. Glück. The universal resolving algorithm: inverse computation in a functional language. *Science of Computer Programming*, 43:193–299, 2002.
2. 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.
3. R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, editor, *Procs. State-of-the-Art Seminar on Formal Program Development*, number 755 in Lecture Notes in Computer Science. Springer-Verlag, 1993.
4. R. S. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.
5. W. Chen and J. T. Udding. Program inversion: more than fun! *Science of Computer Programming*, 15:1–13, 1990.
6. E. W. Dijkstra. Program inversion. Technical Report EWD671, Eindhoven University of Technology, 1978.
7. 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.
8. P. G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.
9. 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.
10. D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms, 3rd Edition.* Addison Wesley, 1997.
11. 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.
12. S.-C. Mu. *A Calculational Approach to Program Inversion.* PhD thesis, Oxford University Computing Laboratory, 2003.
13. S.-C. Mu and R. S. Bird. Inverting functions as folds. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, pages 209–232. Springer-Verlag, July 2002.

14. 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.
15. 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` .
16. J. von Wright. Program inversion in the refinement calculus. *Information Processing Letters*, 37:95–100, 1991.