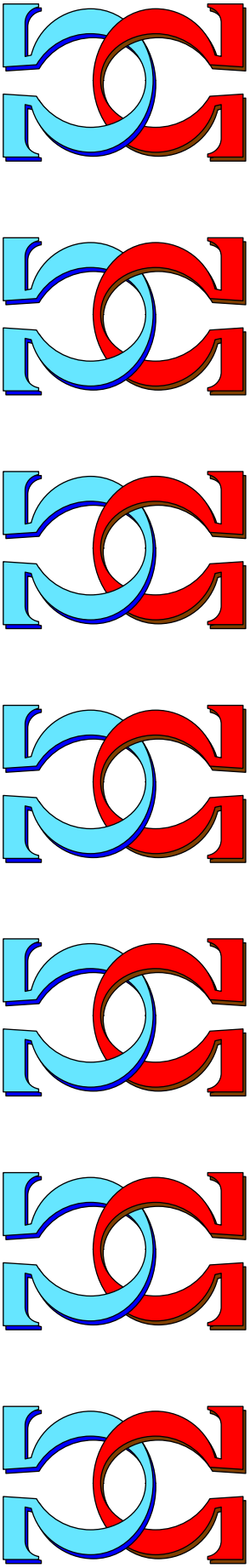


CDMTCS
Research
Report
Series

The Suitability of Different Binary Tree Obfuscations

Stephen Drape
Department of Computer Science
The University of Auckland
Auckland, New Zealand

CDMTCS-310
June 2007

Centre for Discrete Mathematics and
Theoretical Computer Science

The Suitability of Different Binary Tree Obfuscations

Stephen Drape
Department of Computer Science
The University of Auckland, New Zealand
stephen@cs.auckland.ac.nz

Abstract

An obfuscation aims to transform a program, without affecting the functionality, so that some secret information within the program can be hidden for as long as possible from an adversary. Proving that an obfuscating transform is correct (*i.e.* it preserves functionality) is considered to be a challenging task. We use data refinement to specify data obfuscations, model our operations using the functional language Haskell and consider obfuscating abstract data-types. This approach allows us to prove properties, including correctness, of our operations easily.

In this paper our focus is on how to obfuscate a data-type of binary trees for which we specify a set of operations and a list of properties that these operations satisfy. We consider different tree transformations and discuss their suitability as obfuscations. In particular we show what our tree operations would be like under these different transformations. We also discuss various ways of defining obfuscated operations including the use of folds and unfolds and how we can exploit properties of Haskell to add extra confusion in our obfuscated definitions.

1 Introduction

Skype's internet telephony client [2], SDC Java DRM (according to [15]), and most software license-control systems rely heavily on obfuscation for their security. After the landmark proof of Barak *et al.* [1], there seems little hope of designing a perfectly-secure software black-box, for any broad class of programs. To date, no one has devised an alternative to Barak's model, in which we would be able to derive proofs of security for systems of practical interest. These theoretical difficulties do not lessen practical interest in obfuscation, nor should it prevent us from placing appropriate levels of reliance on obfuscated systems in cases where the alternative of a hardware black-box is infeasible or uneconomic.

In this paper we define obfuscation as a heuristic method whose objective is to transform a program, without affecting relevant aspects of its functionality, in such a way that some secret information in the program can be preserved as long as possible from

some set of adversaries. The second clause in our objective implies that theoretical study of the effectiveness of an obfuscation will be impossible until we have a validated, and theoretically-tractable, model of adversarial attack. The first clause is, by contrast, an appropriate domain for theoretical study. We expect our compilers to accurately preserve program semantics when they transform our source codes into object codes. We have a similar expectation of obfuscating compilers and object-code obfuscators. Theoretical study of the correctness of obfuscating systems is as yet in its infancy. In this paper, we expand on previous work [8] to present a discussion of possible obfuscations for a data-type of binary trees — this data-type consists of a type declarations and a set of operations that can be performed on the binary trees. We consider obfuscation to be refinement [6] and we consider obfuscating abstract data-types. This means we obfuscate a set of operations (contained in the data-type) rather than just individual pieces of code. We model our operations using the functional language Haskell [14] in which we can elegantly specify our tree data-types and operations. Our approach allows us to establish a framework for the proving the properties, including correctness, of our tree obfuscations. Our proof system is highly constructive, so that it may someday be used as a method for generating obfuscated programs.

We explore some possible tree transformations and discuss their suitability as potential obfuscations. We show how our binary tree operations are changed under each of the tree transformations. We will see that a suitable obfuscation should not produce definitions for the tree operations that are similar to the original definitions. Also we would like that our transformations have a degree of flexibility so that, for example, we can have many different definitions of the same operation or we can create operations that have bogus parts.

In [7] the following definition for obfuscation was given:

Definition 1 (Assertion Obfuscation)**.** Let f be an operation and $\mathcal{A}$ be an assertion that f satisfies. We transform f to obtain f^O by using an obfuscation O and let $\mathcal{A}^O$ be the assertion corresponding to $\mathcal{A}$ which f^O satisfies. The obfuscation O is said to be an *assertion obfuscation* if the proof that f^O satisfies $\mathcal{A}^O$ is more complicated than the proof that f satisfies $\mathcal{A}$.

We do not give further details in this paper (in particular how to measure the complexity of proofs). Thus as a proposal for an attack model we could suppose that an adversary will be armed with proof tools (such as theorem provers) and so our goal will be to try to make proofs harder to construct for our obfuscated operations.

2 Preliminaries

Before we describe our data-type for binary trees, we discuss some standard list operations from Haskell which will be need later — these operations are taken from [3]. We can define a list data-type and this is presented in Figure 1.

There are some shorthands which are commonly used with Haskell lists. Instead of $Empty$ we use [ ] and we can write $Cons\ 1\ (Cons\ 2\ Empty)$ as $1 : (2 : [\,])$ or just $[1, 2]$

$$List\ \alpha ::= Empty \mid Cons\ \alpha\ (List\ \alpha)$$

$$\begin{aligned}
(:) &:: \alpha \to List\ \alpha \to List\ \alpha \\
\mathsf{head} &:: List\ \alpha \to \alpha \\
\mathsf{tail} &:: List\ \alpha \to List\ \alpha \\
\mathsf{length} &:: List\ \alpha \to \mathbb{N} \\
(+\!\!+) &:: List\ \alpha \to List\ \alpha \to List\ \alpha \\
\mathsf{map} &:: (\alpha \to \beta) \to List\ \alpha \to List\ \beta \\
\mathsf{concat} &:: List\ (List\ \alpha) \to List\ \alpha
\end{aligned}$$

Figure 1: Data-Type for Lists

and so $x : xs \equiv Cons\ x\ xs$. Note that for inductive proofs we will have two cases: one for *Empty* and one for *Cons*.

Let us briefly describe each of the operations in our list data-type. We have already seen that (:) is just an infix version of *Cons* and is often used in the definitions of operations. The operation head returns the first element of a list and tail returns everything but the head. They are easily defined as follows:

$$\mathsf{head}\ (x : xs) = x \qquad\qquad \mathsf{tail}\ (x : xs) = xs$$

Note that these operations are only defined for non-empty lists.

The length of a list can be defined as follows:

$$\begin{aligned}
\mathsf{length}\ [\,] &= 0 \\
\mathsf{length}\ (x : xs) &= 1 + \mathsf{length}\ xs
\end{aligned}$$

We can join two lists together using the concatenate operator (++):

$$\begin{aligned}
[\,] +\!\!+\ ys &= ys \\
(x : xs) +\!\!+\ ys &= x : (xs +\!\!+\ ys)
\end{aligned}$$

The operations length and ++ satisfy:

$$\mathsf{length}\ (xs +\!\!+\ ys) = \mathsf{length}\ xs + \mathsf{length}\ ys$$

This can be proved by induction on xs.

We can apply ++ to a list of lists by using concat:

$$\begin{aligned}
\mathsf{concat}\ [\,] &= [\,] \\
\mathsf{concat}\ (xs : xss) &= xs +\!\!+\ \mathsf{concat}\ xss
\end{aligned}$$

A property of this operation is that

$$\mathsf{concat}\ (xss +\!\!+\ yss) = \mathsf{concat}\ xss +\!\!+\ \mathsf{concat}\ yss$$

The operation map takes a function and a list as input and returns a list which is obtained by applying the function to every element of the input list.

$$\begin{aligned}\mathsf{map}\ f\ [\,] &= [\,]\\ \mathsf{map}\ f\ (x : xs) &= f\ x : \mathsf{map}\ f\ xs\end{aligned}$$

Some properties of map are:

$$\begin{aligned}\mathsf{map}\ f\ (xs +\!\!+ ys) &= \mathsf{map}\ f\ xs +\!\!+ \mathsf{map}\ f\ ys\\ \mathsf{map}\ f \cdot \mathsf{map}\ g &= \mathsf{map}\ (f \cdot g)\\ \mathsf{map}\ f \cdot \mathsf{concat} &= \mathsf{concat} \cdot \mathsf{map}\ (\mathsf{map}\ f)\end{aligned}$$

A fuller discussion of list operations can be found in [3]. In Section 6 we discuss the fold functions for lists (and other data-types).

2.1 Obfuscation as Data Refinement

Suppose that we have a data-type D and we want to obfuscate it to obtain the data-type O. To provide a framework for obfuscating data-types (and establishing the correctness of the obfuscations) we use *data refinement* [6] and, in particular, we consider obfuscation as *functional refinement*. So for obfuscation we require an abstraction function $af :: O \to D$ and a data-type invariant dti such that for elements $x :: D$ and $y :: O$

$$x \rightsquigarrow y \iff (x = af(y)) \wedge dti(y) \tag{1}$$

The arrow $\rightsquigarrow$ is read as "...is data refined by ..." (or in our case, "...is obfuscated by...") which expresses how the data-types are related. In our situation, it turns out that af is a surjective function so we can find a conversion function $of :: D \to O$ that satisfies $of(x) = y \Rightarrow x \rightsquigarrow y$ and thus

$$af \cdot cf = id \tag{2}$$

Suppose that we have an operation $f :: D \to D$ defined in our data-type. Then to obfuscate f we want an operation $f^O :: O \to O$ which preserves the correctness of f. In terms of data refinement, we say that f^O is *correct* (with respect to f) if it satisfies:

$$(\forall x :: D; y :: O) \bullet x \rightsquigarrow y \Rightarrow f(x) \rightsquigarrow f^O(y) \tag{3}$$

If f^O is a correct refinement (obfuscation) of f then we write $f \rightsquigarrow f^O$. and for Equation (3), we can draw the commuting diagram in Figure 2. From (1) and (3), we have the following equation:

$$f \cdot af = af \cdot f^O \tag{4}$$

where $\cdot$ means functional composition. Thus we can prove that a definition of f^O is correct by using this equation.

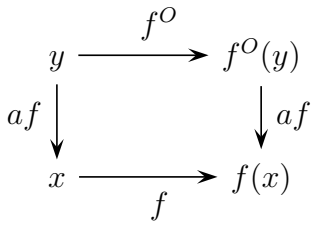


Figure 2: A commuting diagram for data obfuscation

2.1.1 Non-homogeneous operations

Suppose that we have a operation

$$f :: D \to E$$

where D and E are the state spaces of two data-types. Let af_D and af_E be abstraction functions for some obfuscations of D and E. How do we define a correct obfuscation f^O of f? Suppose $x :: D$ and $x \rightsquigarrow y$ and consider:

$$\begin{aligned} & f(x) \rightsquigarrow f^O(y) \\ \equiv \quad & \{\text{Equation (1) using } af_E\} \\ & f(x) = af_E(f^O(y)) \\ \equiv \quad & \{\text{Equation (1) using } af_D\} \\ & f(af_D(y)) = af_E(f^O(y)) \end{aligned}$$

Thus

$$f \cdot af_D = af_E \cdot f^O \tag{5}$$

We will use this equation in Section 4.1 when defining obfuscated tree operations.

3 Binary Tree Data-Type

In previous work [8] we consider binary trees which contained values on the internal nodes. As an alternative, we will use binary trees which have the following type:

$$Tree\ \alpha ::= Leaf\ \alpha \mid Fork\ (Tree\ \alpha)\ (Tree\ \alpha)$$

Thus the values for these binary trees are on the tips. Our data-type for binary trees is contained in Figure 3. Note that when using inductive proofs with binary trees the base case will involve $Leaf$ and the inductive case will involve $Fork$.

$$Tree\ \alpha ::= Leaf\ \alpha \mid Fork\ (Tree\ \alpha)\ (Tree\ \alpha)$$

$$\begin{aligned}
\mathsf{flatten} &:: Tree\ \alpha \to List\ \alpha \\
\mathsf{mkTree} &:: List\ \alpha \to Tree\ \alpha \\
\mathsf{member} &:: \alpha \to Tree\ \alpha \to \mathbb{B} \\
\mathsf{size} &:: Tree\ a \to \mathbb{N} \\
\mathsf{mapTree} &:: (\alpha \to \beta) \to Tree\ \alpha \to Tree\ \beta
\end{aligned}$$

Figure 3: Data-Type for Binary Trees

3.1 Binary Tree Operations

The function mkTree takes a finite, non-empty list and builds a binary tree using elements of the list as leaf nodes. We can define it as follows:

$$\begin{aligned}
&\mathsf{mkTree}\ xs \\
&\quad \mid m == 0 \quad = Leaf\ (\mathsf{head}\ zs) \\
&\quad \mid \text{otherwise} = Fork\ (\mathsf{mkTree}\ ys)\ (\mathsf{mkTree}\ zs) \\
&\qquad\qquad \text{where}\ (ys, zs) = \mathsf{splitAt}\ m\ xs \\
&\qquad\qquad\qquad\quad m \qquad = \mathsf{div}\ (\mathsf{length}\ xs)\ 2
\end{aligned}$$

The function splitAt has type

$$\mathsf{splitAt} :: \mathbb{N} \to List\ \alpha \to (List\ \alpha, List\ \alpha)$$

and can be defined as follows:

$$\begin{aligned}
\mathsf{splitAt}\ 0\ xs &= ([\,], xs) \\
\mathsf{splitAt}\ n\ [\,] &= ([\,], [\,]) \\
\mathsf{splitAt}\ n\ (x : xs) &= (x : ys, zs) \\
&\qquad \text{where}\ (ys, zs) = \mathsf{splitAt}\ (n-1)\ xs
\end{aligned}$$

Note that using the definition of $+\!\!+$ we can show that

$$(ys, zs) = \mathsf{splitAt}\ n\ xs \Rightarrow ys +\!\!+ zs = xs$$

We use splitAt in the definition of mkTree so that we do not produce unbalanced trees. In fact we find from the definition of mkTree that

$$\begin{aligned}
&\mathsf{length}\ xs \text{ is even} \Rightarrow \mathsf{length}\ zs = \mathsf{length}\ ys \\
&\mathsf{length}\ xs \text{ is odd} \Rightarrow \mathsf{length}\ zs = \mathsf{length}\ ys + 1
\end{aligned} \tag{6}$$

where $(ys, zs) = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs$.

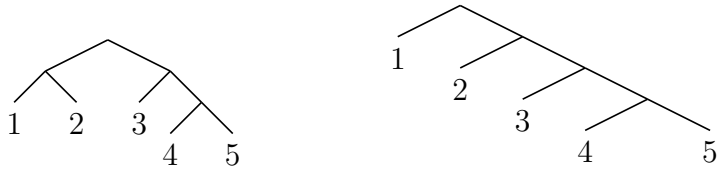
Our definition of mkTree (taken from [3]) creates a minimal height binary tree but for our data-type, we do not insist that our trees have minimal height (although many tree

operations would be more efficient). In fact we have many ways to build a binary tree from a list. For example, we could define

$$\begin{aligned}&\mathsf{mkTree'}\ [a] &&= Leaf\ a\\ &\mathsf{mkTree'}\ (a : xs) &&= Fork\ (Leaf\ a)\ (\mathsf{mkTree'}\ xs)\end{aligned}$$

Thus,

$\mathsf{mkTree}\ [1,2,3,4,5]$ $\mathsf{mkTree'}\ [1,2,3,4,5]$



The function flatten produces a list from the leaf values of a binary tree in left to right order. We can define the function as follows:

$$\begin{aligned}&\mathsf{flatten}\ (Leaf\ a) &&= [a]\\ &\mathsf{flatten}\ (Fork\ xt\ yt) &&= \mathsf{flatten}\ xt +\!\!+ \mathsf{flatten}\ yt\end{aligned}$$

So, using the example above,

$$\mathsf{flatten}\ (\mathsf{mkTree}\ [1..5]) = [1..5]$$

In fact, we can prove that

$$\mathsf{flatten} \cdot \mathsf{mkTree} = id$$

But it is not the case that

$$\mathsf{mkTree} \cdot \mathsf{flatten} = id$$

For instance if $t = \mathsf{mkTree'}\ [1..5]$ then

$$\mathsf{mkTree}\ (\mathsf{flatten}\ t) \neq t$$

Also, flatten is not injective as two different trees may flatten to the same list. For example,

$$\mathsf{flatten}\ \left(\begin{array}{ccc} & & \\ 1 & & \\ & 2 & 3 \end{array}\right) = [1,2,3] = \mathsf{flatten}\ \left(\begin{array}{ccc} & & \\ & & 3 \\ 1 & 2 & \end{array}\right)$$

The function member checks whether a particular value matches one of the leaf values of the tree:

$$\begin{aligned}&\mathsf{member}\ v\ (Leaf\ a) &&= v == a\\ &\mathsf{member}\ v\ (Fork\ xt\ yt) &&= \mathsf{member}\ v\ xt \vee \mathsf{member}\ v\ yt\end{aligned}$$

We can measure the number of leaf nodes in a binary tree by using the function size:

$$\begin{aligned}\mathsf{size}\ (Leaf\ a) &= 1\\ \mathsf{size}\ (Fork\ xt\ yt) &= \mathsf{size}\ xt + \mathsf{size}\ yt\end{aligned}$$

This function satisfies:

$$\mathsf{size} = \mathsf{length} \cdot \mathsf{flatten}$$

Finally, we define a function mapTree which satisfies:

$$\mathsf{flatten}\ (\mathsf{mapTree}\ f\ zt) = \mathsf{map}\ f\ (\mathsf{flatten}\ zt) \tag{7}$$

for all finite trees zt and functions f. We define it as follows:

$$\begin{aligned}\mathsf{mapTree}\ f\ (Leaf\ a) &= Leaf\ (f\ a)\\ \mathsf{mapTree}\ f\ (Fork\ xt\ yt) &= Fork\ (\mathsf{mapTree}\ f\ xt)\ (\mathsf{mapTree}\ f\ yt)\end{aligned}$$

Now let us show that this definition does indeed satisfy Equation (7).

Proof. We will prove Equation (7) by induction on zt.

Base Case Suppose that $zt = Leaf\ a$. Then

$$\begin{aligned}&\mathsf{map}\ f\ (\mathsf{flatten}\ (Leaf\ a))\\ =\quad &\{\text{definition of flatten}\}\\ &\mathsf{map}\ f\ [a]\\ =\quad &\{\text{definition of map}\}\\ &[f\ a]\\ =\quad &\{\text{definition of flatten}\}\\ &\mathsf{flatten}\ (Leaf\ (f\ a))\\ =\quad &\{\text{definition of mapTree}\}\\ &\mathsf{flatten}\ (\mathsf{mapTree}\ f\ (Leaf\ a))\end{aligned}$$

Step Case Suppose that $zt = Fork\ xt\ yt$ and, for the induction hypothesis, xt and yt satisfy Equation (7). Then

$$\begin{aligned}&\mathsf{map}\ f\ (\mathsf{flatten}\ zt)\\ =\quad &\{\text{definition}\}\\ &\mathsf{map}\ f\ (\mathsf{flatten}\ (Fork\ xt\ yt))\\ =\quad &\{\text{definition of flatten}\}\\ &\mathsf{map}\ f\ (\mathsf{flatten}\ xt +\!\!+ \mathsf{flatten}\ yt)\\ =\quad &\{\mathsf{map}\ f\ (xs +\!\!+ ys) = (\mathsf{map}\ f\ xs) +\!\!+ (\mathsf{map}\ f\ ys)\}\\ &(\mathsf{map}\ f\ (\mathsf{flatten}\ xt)) +\!\!+ (\mathsf{map}\ f\ (\mathsf{flatten}\ yt))\end{aligned}$$

$$
\begin{aligned}
&= \quad \{\text{induction hypothesis}\} \\
&\quad \mathsf{flatten}\ (\mathsf{mapTree}\ f\ xt) +\!\!+ (\mathsf{flatten}\ (\mathsf{mapTree}\ f\ yt) \\
&= \quad \{\text{definition of } \mathsf{flatten}\} \\
&\quad \mathsf{flatten}\ (Fork\ (\mathsf{mapTree}\ f\ xt)\ (\mathsf{mapTree}\ f\ yt)) \\
&= \quad \{\text{definition of } \mathsf{mapTree}\} \\
&\quad \mathsf{flatten}\ (\mathsf{mapTree}\ f\ (Fork\ xt\ yt)) \\
&= \quad \{\text{definition}\} \\
&\quad \mathsf{flatten}\ (\mathsf{mapTree}\ f\ zt)
\end{aligned}
$$

□

By the definition from [7], an operation is said to be obfuscated if an assertion proof for that operation is more complicated. Using these operations, some assertions for binary trees could be:

$$
\begin{aligned}
\mathsf{flatten} \cdot \mathsf{mkTree} &= id \\
\mathsf{size} &= \mathsf{length} \cdot \mathsf{flatten} \\
\mathsf{flatten} \cdot (\mathsf{mapTree}\ f) &= (\mathsf{map}\ f) \cdot \mathsf{flatten}
\end{aligned}
$$

So the proof of Equation (7) will be a useful comparison for seeing whether our transformations produce good obfuscations.

3.2 Using Tree Transformations

Now that we have defined our binary tree data-type, we need to decide what tree transformations we are going to use as obfuscations. For our first attempt, let us consider a tree reflection. If we imagine placing a line of symmetry through the uppermost $Fork$ of a tree then to reflect a binary tree we just recursively swap the left and right subtrees at every $Fork$.

$$
\begin{aligned}
&\mathsf{reflect}\ (Leaf\ a) &&= Leaf\ a \\
&\mathsf{reflect}\ (Fork\ xt\ yt) &&= Fork\ (\mathsf{reflect}\ yt)\ (\mathsf{reflect}\ xt)
\end{aligned}
$$

For example,

We can prove (by structural induction) that reflect is self-inverse and so it is suitable for an abstraction (and conversion) function.

How are the definitions of the operations from Figure 3 changed by this transformation? The flatten operation is similar to the earlier version except that the nodes are concatenated in a different order:

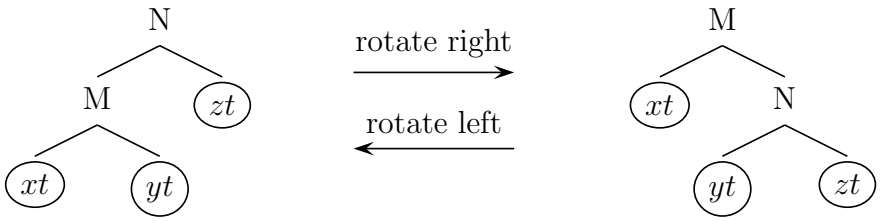
$$\begin{aligned}&\mathsf{flattenRef}\ (Leaf\ a) &&= [a]\\ &\mathsf{flattenRef}\ (Fork\ xt\ yt) &&= \mathsf{flattenRef}\ yt +\!\!+ \mathsf{flattenRef}\ xt\end{aligned}$$

For a function to make reflected trees we can use the definition of mkTree from Section 3 except that in the "where" clause we swap over ys and zs:

$$\text{where } (zs, ys) = \mathsf{splitAt}\ m\ xs$$

Since $+$ and $\vee$ are commutative then we can use the same definitions of size and member as before. We can also use the previous definition for mapTree as well. So, the operations for reflected binary trees are very similar to the original operations defined in Section 3 and so this means that reflection is not a good binary tree obfuscation.

Let us now consider tree rotations. We can have two possible rotations (left and right) which should perform the following transformations:



On the left we have tree of the form $Fork\ (Fork\ xt\ yt)\ zt$ and on the right we have $Fork\ xt\ (Fork\ yt\ zt)$. Thus we can define rotation functions as follows:

$$\begin{aligned}&\mathsf{rotRight}\ (Fork\ (Fork\ xt\ yt)\ zt) &&= Fork\ xt\ (Fork\ yt\ zt)\\ &\mathsf{rotRight}\ xt &&= xt\end{aligned}$$

and

$$\begin{aligned}&\mathsf{rotLeft}\ (Fork\ xt\ (Fork\ yt\ zt)) &&= Fork\ (Fork\ xt\ yt)\ zt\\ &\mathsf{rotLeft}\ xt &&= xt\end{aligned}$$

For a refinement, let us suppose that we take rotRight to be the abstraction function and rotLeft to be the conversion function. Now consider

$$\begin{aligned}&\mathsf{flatten}\ (\mathsf{rotRight}\ (Fork\ (Fork\ xt\ yt)\ zt))\\ =\quad &\{\text{definition of rotRight}\}\\ &\mathsf{flatten}\ (Fork\ xt\ (Fork\ yt\ zt))\\ =\quad &\{\text{definition of flatten}\}\\ &\mathsf{flatten}\ xt +\!\!+ \mathsf{flatten}\ (Fork\ yt\ zt)\\ =\quad &\{\text{definition of flatten}\}\\ &\mathsf{flatten}\ xt +\!\!+ (\mathsf{flatten}\ yt +\!\!+ \mathsf{flatten}\ zt)\end{aligned}$$

$$
\begin{array}{ll}
= & \quad \{+\!\!+ \text{ is associative}\} \\
& (\mathsf{flatten}\ xt +\!\!+ \mathsf{flatten}\ yt) +\!\!+ \mathsf{flatten}\ zt \\
= & \quad \{\text{definition of } \mathsf{flatten}\} \\
& (\mathsf{flatten}\ (Fork\ xt\ yt)) +\!\!+ \mathsf{flatten}\ zt \\
= & \quad \{\text{definition of } \mathsf{flatten}\} \\
& \mathsf{flatten}\ (Fork\ (Fork\ xt\ yt)\ zt)
\end{array}
$$

Thus we have shown that $\mathsf{flatten} \cdot \mathsf{rotRight} = \mathsf{flatten}$ and so for the rotated tree we have the same definition of flatten as before. This means that we can use the same definition of mkTree and, in fact, all the other binary tree operations. A more serious problem with the rotation transformation is that we do not actually have that $\mathsf{rotRight} \cdot \mathsf{rotLeft} = id$ or $\mathsf{rotLeft} \cdot \mathsf{rotRight} = id$. For example,

$$
\begin{array}{ll}
& \mathsf{rotLeft}\ (\mathsf{rotRight}\ (Fork\ (Leaf\ a)\ (Fork\ xt\ yt))) \\
= & \mathsf{rotLeft}\ (Fork\ (Leaf\ a)\ (Fork\ xt\ yt)) \\
= & Fork\ (Fork\ (Leaf\ a)\ xt)\ yt \\
\neq & Fork\ (Leaf\ a)\ (Fork\ xt\ yt)
\end{array}
$$

This means that we cannot use rotLeft or rotRight as abstraction function and so rotation is unsuitable as a refinement for obfuscation.

3.3 Splitting

An obfuscation suitable for arrays is called an *array split* [4]. This obfuscation has been generalised so that it is applicable to more data-types [9] and in particular, we can apply it to trees. The aim of a split is break an object t into smaller objects which are called the *split components*. If t is split into two components l and r then we write

$$t \rightsquigarrow \langle l, r \rangle$$

The idea of a split is to spread the "information" contained in t across the split components.

A natural tree split would be:

$$
\begin{array}{rcl}
Leaf\ a & \rightsquigarrow & \langle Leaf\ a \rangle \\
Fork\ xt\ yt & \rightsquigarrow & \langle xt, yt \rangle
\end{array}
$$

For this split, we can define a conversion function (split) and an abstraction function (unsplit) immediately:

$$
\begin{array}{ll}
\mathsf{split}\ (Leaf\ x) = \langle Leaf\ a \rangle & \mathsf{unsplit}\ \langle Leaf\ a \rangle = (Leaf\ x) \\
\mathsf{split}\ (Fork\ xt\ yt) = \langle xt, yt \rangle & \mathsf{unsplit}\ \langle xt, yt \rangle = (Fork\ xt\ yt)
\end{array}
$$

We can see that $\mathsf{unsplit} \cdot \mathsf{split} = id$.

How easy is it to define operations for split trees? For an operation mkSpTree which makes split trees, we can use the following equation

$$\mathsf{mkSpTree} = \mathsf{split} \cdot \mathsf{mktree}$$

to derive a definition. Using this equation, we obtain the following definition:

$$\begin{array}{l}\mathsf{mkSpTree}\ xs \\ \quad \left| \begin{array}{l} m == 0 \quad = \langle Leaf\ (\mathsf{head}\ xs)\rangle \\ \mathsf{otherwise} = \langle(\mathsf{mkTree}\ ys),\ (\mathsf{mkTree}\ zs)\rangle \end{array} \right. \\ \qquad\quad \mathsf{where}\ (ys, zs) = \mathsf{splitAt}\ m\ xs \\ \qquad\qquad\quad\ m \qquad = \mathsf{div}\ (\mathsf{length}\ xs)\ 2\end{array}$$

We can derive a flattening operation flattenSp using:

$$\mathsf{flattenSp} = \mathsf{flatten} \cdot \mathsf{unsplit}$$

This gives us:

$$\begin{array}{l}\mathsf{flattenSp}\ \langle Leaf\ a\rangle = [a] \\ \mathsf{flattenSp}\ \langle xt, yt\rangle \quad = \mathsf{flatten}\ xt +\!\!+ \mathsf{flatten}\ yt\end{array}$$

This definition closely matches the definition of flatten given in Section 3. This is true for the other tree operations — for example, here is a definition for mapSpTree:

$$\begin{array}{l}\mathsf{mapSpTree}\ f\ \langle Leaf\ a\rangle = \langle Leaf\ (f\ a)\rangle \\ \mathsf{mapSpTree}\ f\ \langle xt, yt\rangle \quad = \langle \mathsf{mapTree}\ f\ xt,\ \mathsf{mapTree}\ f\ yt\rangle\end{array}$$

So, this tree split is not a particularly good obfuscation as the definitions for the split tree obfuscations are similar to the original operations. To produce a good tree obfuscation we really need a recursive transformation so that the definitions under this transformation will also be recursive (rather than just relying on the original tree operations). It is hard to define a recursive tree split as, by the definition of a split, the split components also need to have the same type and we have to ensure that our transformation is invertible.

In general, splits are used mainly to change the order of data values by partitioning the values. For trees, since many trees can be flattened to the same list, the structure of the data values as well as the order of the values is important. Thus splits are not really suitable as tree obfuscations as we need to use a transformation that changes the structure as well as the order of the data values.

4 Ternary Trees

The tree transformations discussed in the last section did not really affect the "shape" of binary trees and did not produce suitable obfuscations. Now we will define a transformation that changes the structure of a binary tree in such a way that allows us to recover the original binary tree if we wish.

We will define an obfuscation which transforms a binary tree into a ternary tree. The use of ternary trees provides us with an opportunity to add extra information so that we can have "real" information and "junk" information. We will define ternary trees with the following type:

$$Tree_3\ \alpha ::= Leaf_3\ \alpha \mid Fork_3\ (Tree_3\ \alpha)\ (Tree_3\ \alpha)\ (Tree_3\ \alpha)$$

In the conversion and abstraction functions we will need to define how a binary tree should be transformed at each node in the tree and what "junk" to add. In previous work on obfuscating binary trees [8], the tree data-type contained values at each node and the transformation relied on properties of the node values (such as whether the node value was even). Since the trees from our data-type only contain values in the leaves, we will have to define conversion functions based on the structure of the trees.

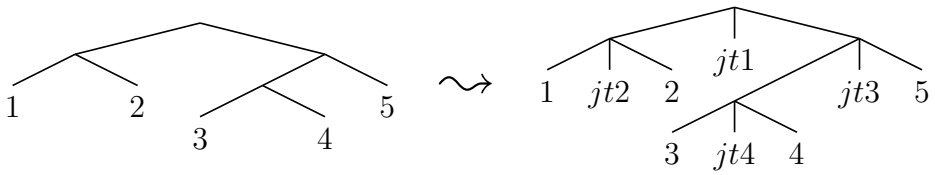
A simple conversion function is:

$$\begin{aligned}\mathsf{cf}\ (Leaf\ a) &= Leaf_3\ a\\ \mathsf{cf}\ (Fork\ xt\ yt) &= Fork_3\ (\mathsf{cf}\ xt)\ jt\ (\mathsf{cf}\ yt)\end{aligned}$$

where jt is a random ternary trees. For this conversion function, we have the following abstraction function:

$$\begin{aligned}\mathsf{af}\ (Leaf_3\ a) &= Leaf\ a\\ \mathsf{af}\ (Fork_3\ xt\ ct\ yt) &= Fork\ (\mathsf{af}\ xt)\ (\mathsf{af}\ yt)\end{aligned}$$

So, for example,



where $jt1, \ldots, jt4$ are arbitrary ternary trees.

For the rest of this section we will use a more complicated transformation which has the following conversion function:

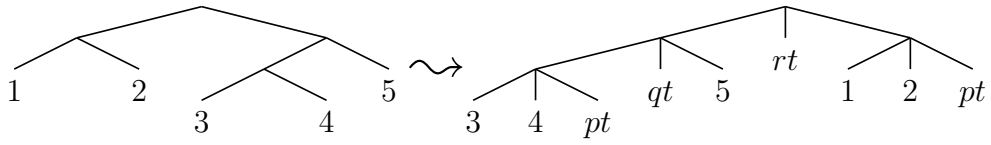
$$\begin{aligned}\mathsf{to3}\ (Leaf\ a) &= Leaf_3\ a\\ \mathsf{to3}\ (Fork\ (Leaf\ a)\ xt) &= Fork_3\ (Leaf_3\ a)\ (\mathsf{to3}\ xt)\ pt\\ \mathsf{to3}\ (Fork\ xt\ (Leaf\ a)) &= Fork_3\ (\mathsf{to3}\ xt)\ qt\ (Leaf_3\ a)\\ \mathsf{to3}\ (Fork\ xt\ yt) &= Fork_3\ (\mathsf{to3}\ yt)\ rt\ (\mathsf{to3}\ xt)\end{aligned}$$

This function pattern matches on the structure of the binary tree enabling us to do different transformations on different nodes. The ternary trees pt, qt and rt can be defined arbitrarily, in fact we can build trees which are functions of the subtrees xt and yt and the leaf node a. However, if efficiency is a concern then we will need to restrict the size of these new trees. Also, we should ensure that our abstraction function for this transformation "ignores" these extra ternary trees.

We can define the following abstraction function to2 for this transformation:

$$\begin{aligned}
&\mathsf{to2}\ (Leaf_3\ a) &&= Leaf\ a\\
&\mathsf{to2}\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) &&= Fork\ (Leaf\ a)\ (\mathsf{to2}\ yt)\\
&\mathsf{to2}\ (Fork_3\ xt\ yt\ (Leaf_3\ a)) &&= Fork\ (\mathsf{to2}\ xt)\ (Leaf\ a)\\
&\mathsf{to2}\ (Fork_3\ xt\ yt\ zt) &&= Fork\ (\mathsf{to2}\ zt)\ (\mathsf{to2}\ xt)
\end{aligned}$$

We can verify that $\mathsf{to2} \cdot \mathsf{to3} = id$ but in general $\mathsf{to3} \cdot \mathsf{to2} \neq id$. As an example, under this transformation,



Note that for a particular abstraction function we can define many conversion functions that are right inverses. For example, in our definition of to3 we can specify different instances for pt, qt and rt.

When designing a conversion function for a refinement it is important to ensure that an inverse function can be constructed. For instance, suppose for the definition of to3 we had the following middle cases:

$$\begin{aligned}
&\mathsf{to3}\ (Fork\ (Leaf\ a)\ xt) = Fork_3\ (Leaf_3\ a)\ (\mathsf{to3}\ xt)\ pt\\
&\mathsf{to3}\ (Fork\ xt\ (Leaf\ a)) = Fork_3\ (Leaf_3\ a)\ qt\ (\mathsf{to3}\ xt)
\end{aligned}$$

Then for the definition of to2 we would need a case:

$$\mathsf{to2}\ (Fork_3\ (Leaf_3\ a)\ jt\ kt)$$

which is ambiguous as it has to decide whether jt or kt is "real" or not.

Another source for ambiguity is to have overlapping cases in the function definitions. In the earlier definition for to3, the last case matches binary trees of the form: $Fork\ xt\ yt$. As Haskell pattern matches from top to bottom, we are guaranteed that the previous cases (in which one of the subtrees is a leaf node) do not hold and there will be no difficultly in writing a suitable abstraction function. If we are not guaranteed that this top down matching holds then we could write a definition with explicit guards, such as:

$$\begin{aligned}
&\mathsf{to3}\ (Leaf\ a) = Leaf_3\ a\\
&\mathsf{to3}\ (Fork\ jt\ kt)\\
&\quad \mid jt == Leaf\ a = Fork_3\ (Leaf_3\ a)\ (\mathsf{to3}\ kt)\ pt\\
&\quad \mid kt == Leaf\ a = Fork_3\ (\mathsf{to3}\ jt)\ qt\ (Leaf_3\ a)\\
&\quad \mid \text{otherwise} \quad = Fork_3\ (\mathsf{to3}\ kt)\ rt\ (\mathsf{to3}\ jt)
\end{aligned}$$

However we choose to exploit the pattern matching of Haskell so that we can produce a seemingly ambiguous definition.

4.1 Ternary tree operations

The first ternary operation we will define is a flattening operation:

$$\mathsf{flatten}_3 :: Tree_3\ \alpha \to List\ \alpha$$

Using Equation (5) from Section 2.1.1 (with $af_D = \mathsf{to2}$ and $af_E = id$) we have that

$$\mathsf{flatten}_3\ t_3 = \mathsf{flatten}\ (\mathsf{to2}\ t_3) \tag{8}$$

We can use this equation to derive a definition for $\mathsf{flatten}_3$ by structural induction.

Base Case Suppose that $t_3 = Leaf_3\ a$. Then

$$\begin{array}{ll} & \mathsf{flatten}_3\ (Leaf_3\ a) \\ = & \quad \{\text{definition}\} \\ & \mathsf{flatten}\ (\mathsf{to2}\ (Leaf_3\ a)) \\ = & \quad \{\text{definition of } \mathsf{to2}\} \\ & \mathsf{flatten}\ (Leaf\ a) \\ = & \quad \{\text{definition of } \mathsf{flatten}\} \\ & [a] \end{array}$$

Step Case We suppose that $t_3 = Fork_3\ xt\ yt\ zt$ and we have three subcases.

Subcase 1 Suppose that xt is a leaf node, *i.e.* for some a, $xt = Leaf_3\ a$. For the induction hypothesis we suppose that yt satisfies Equation (8).

$$\begin{array}{ll} & \mathsf{flatten}_3\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) \\ = & \quad \{\text{definition}\} \\ & \mathsf{flatten}\ (\mathsf{to2}\ (Fork_3\ (Leaf_3\ a)\ yt\ zt)) \\ = & \quad \{\text{definition of } \mathsf{to2}\} \\ & \mathsf{flatten}\ (Fork\ (Leaf\ a)\ (\mathsf{to2}\ yt)) \\ = & \quad \{\text{definition of } \mathsf{flatten}\} \\ & (\mathsf{flatten}\ (Leaf\ a)) +\!\!+ (\mathsf{flatten}\ (\mathsf{to2}\ yt)) \\ = & \quad \{\text{definition of } \mathsf{flatten}\} \\ & [a] +\!\!+ (\mathsf{flatten}\ (\mathsf{to2}\ yt)) \\ = & \quad \{\text{induction hypothesis}\} \\ & [a] +\!\!+ (\mathsf{flatten}_3\ yt) \\ = & \quad \{\text{property: } [x] +\!\!+ xs = x : xs\} \\ & a : (\mathsf{flatten}_3\ yt) \end{array}$$

Subcase 2 Suppose that $zt = Leaf_3\ b$ for some b and xt is not a leaf node. For the induction hypothesis we suppose that xt satisfies Equation (8). We find that

$$\mathsf{flatten}_3\ (Fork_3\ xt\ yt\ (Leaf_3\ b)) = \mathsf{flatten}_3\ xt +\!\!+ [b]$$

Subcase 3 Suppose that xt and zt are not leaf nodes and that these two trees satisfy Equation (8). We find that

$$\mathsf{flatten}_3\ (Fork_3\ xt\ yt\ zt) = \mathsf{flatten}_3\ zt +\!\!+ \mathsf{flatten}_3\ xt$$

Putting the cases together we have the following definition for $\mathsf{flatten}_3$:

$$\begin{array}{ll}
\mathsf{flatten}_3\ (Leaf_3\ a) & = [a] \\
\mathsf{flatten}_3\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) & = a : (\mathsf{flatten}_3\ yt) \\
\mathsf{flatten}_3\ (Fork_3\ xt\ yt\ (Leaf_3\ b)) & = \mathsf{flatten}_3\ xt +\!\!+ [b] \\
\mathsf{flatten}_3\ (Fork_3\ xt\ yt\ zt) & = \mathsf{flatten}_3\ zt +\!\!+ \mathsf{flatten}_3\ xt
\end{array}$$

As with the definition of $\mathsf{to3}$, the definition of $\mathsf{flatten}_3$ has overlapping cases and so the order of the cases is important. (As before we could rewrite the definition using guards.)

4.2 Making ternary trees from lists

Now we need to define a function

$$\mathsf{mkTree}_3 :: List\ \alpha \to Tree_3\ \alpha$$

which converts a list into a ternary tree. Using Equation (5), this function has to satisfy

$$\mathsf{mkTree} = \mathsf{to2} \cdot \mathsf{mkTree}_3 \tag{9}$$

The function also needs to satisfy

$$\mathsf{flatten}_3 \cdot \mathsf{mkTree}_3 = id$$

There are many functions which satisfy these equations and we use the definition of $\mathsf{to3}$ to derive one such definition using the equation:

$$\mathsf{mkTree}_3\ xs = \mathsf{to3}\ (\mathsf{mkTree}\ xs) \tag{10}$$

Since $\mathsf{to2} \cdot \mathsf{to3} = id$ we know that this definition will satisfy Equation (9).

Using the definition of mkTree, let $m = \mathsf{div}\ (\mathsf{length}\ xs)\ 2$ and $(ys, zs) = \mathsf{splitAt}\ m\ xs$. We derive this definition by induction on $\mathsf{length}\ xs$.

Case 1 Suppose that $m = 0$ and so $xs = [a] = zs$ for some a (and $ys = [\,]$). Thus, $\mathsf{length}\ xs = 1$ and

$$\begin{array}{ll}
 & \mathsf{to3}\ (\mathsf{mkTree}\ xs) \\
= & \quad \{\text{definition of } \mathsf{mkTree} \text{ with } m = 0\} \\
 & \mathsf{to3}\ (Leaf\ (\mathsf{head}\ zs)) \\
= & \quad \{\text{definition of } \mathsf{to3}\} \\
 & Leaf_3\ (\mathsf{head}\ zs)
\end{array}$$

Case 2 From the definition of to3 we need to match binary trees which have the patterns $Fork\ (Leaf\ \_)\ \_$ and $Fork\ \_\ (Leaf\ \_)$. To obtain the first pattern from mktree we have to have the length of ys equal to 1 and the length of zs non-zero. Thus we have that $m = 1$ and $1 < \mathsf{length}\ xs \leq 3$. For an induction hypothesis, we suppose that zs satisfies Equation (10).

$$
\begin{aligned}
& \mathsf{to3}\ (\mathsf{mkTree}\ xs) \\
= \quad & \{\text{definition of mkTree with } m \neq 0\} \\
& \mathsf{to3}\ (Fork\ (\mathsf{mkTree}\ ys)\ (\mathsf{mkTree}\ zs)) \\
= \quad & \{\text{definition of mkTree with length } ys = 1\} \\
& \mathsf{to3}\ (Fork\ (Leaf\ (\mathsf{head}\ ys))\ (\mathsf{mkTree}\ zs)) \\
= \quad & \{\text{definition of to3}\} \\
& Fork_3\ (Leaf_3\ (\mathsf{head}\ ys))\ (\mathsf{to3}\ (\mathsf{mkTree}\ zs))\ pt \\
= \quad & \{\text{induction hypothesis}\} \\
& Fork_3\ (Leaf_3\ (\mathsf{head}\ ys))\ (\mathsf{mkTree}_3\ zs)\ pt
\end{aligned}
$$

Now let us consider the pattern $Fork\ \_\ (Leaf\ \_)$ and so from the definition of mkTree we need that length $zs = 1$ and length $ys > 1$. However from Equation (6), we must have that length $ys \leq$ length zs and so this pattern is never produced from mkTree.

Case 3 For the final case, we suppose that length $xs > 3$ and, for the induction hypothesis, we suppose that ys and zs satisfy Equation (10).

$$
\begin{aligned}
& \mathsf{to3}\ (\mathsf{mkTree}\ xs) \\
= \quad & \{\text{definition of mkTree with } m \neq 0\} \\
& \mathsf{to3}\ (Fork\ (\mathsf{mkTree}\ ys)\ (\mathsf{mkTree}\ zs)) \\
= \quad & \{\text{definition of to3 where } ys \text{ and } zs \text{ are not leaf nodes}\} \\
& Fork_3\ (\mathsf{to3}\ (\mathsf{mkTree}\ zs))\ rt\ (\mathsf{to3}\ (\mathsf{mkTree}\ ys)) \\
= \quad & \{\text{induction hypothesis}\} \\
& Fork_3\ (\mathsf{mkTree}_3\ zs)\ rt\ (\mathsf{mkTree}_3\ ys)
\end{aligned}
$$

Putting these cases we obtain:

$$
\begin{aligned}
& \mathsf{mkTree}_3\ xs \\
& \quad \mid m == 0 \quad = Leaf_3\ (\mathsf{head}\ zs) \\
& \quad \mid m == 1 \quad = Fork_3\ (Leaf_3\ (\mathsf{head}\ ys))\ (\mathsf{mkTree}_3\ zs)\ pt \\
& \quad \mid \text{otherwise} = Fork_3\ (\mathsf{mkTree}_3\ zs)\ rt\ (\mathsf{mkTree}_3\ ys) \\
& \qquad\qquad \text{where } m = \mathsf{div}\ (\mathsf{length}\ xs)\ 2 \\
& \qquad\qquad\qquad (ys, zs) = \mathsf{splitAt}\ m\ xs
\end{aligned}
$$

Since we know that when $m > 1$ then length $zs \neq 1$ and so we can add a bogus case which checks the length of zs and this extra case adds to the obfuscation of the operation. To

match, we can replace the tests for m with length checks. So, for example:

$$
\begin{array}{l}
\mathsf{mkTree}_3\ xs \\
\quad \mid \mathsf{length}\ xs == 1 = Leaf_3\ (\mathsf{head}\ zs) \\
\quad \mid \mathsf{length}\ ys == 1 = Fork_3\ (Leaf_3\ (\mathsf{head}\ ys))\ (\mathsf{mkTree}_3\ zs)\ pt \\
\quad \mid \mathsf{length}\ zs == 1 = Fork_3\ (Leaf_3\ (\mathsf{head}\ zs))\ qt\ (\mathsf{mkTree}_3\ ys) \\
\quad \mid \mathsf{otherwise} \qquad = Fork_3\ (\mathsf{mkTree}_3\ zs)\ rt\ (\mathsf{mkTree}_3\ ys) \\
\qquad\quad \text{where } m = \mathsf{div}\ (\mathsf{length}\ xs)\ 2 \\
\qquad\qquad\quad (ys, zs) = \mathsf{splitAt}\ m\ xs
\end{array}
$$

As before pt, qt and rt are arbitrary ternary trees. For the "bogus" case in this definition a ternary tree is built which is not "correct" for our particular refinement. Of course since this case is never taken then we can build whatever ternary tree we like. The test $\mathsf{length}\ zs == 1$ is a kind of *opaque* predicate [4] as the value of the predicate is known to the creator of the function but it is not obvious to others inspecting the definition that the value of the test is always false. Again this bogus test relies on the top-down matching of Haskell.

4.3 The map function for ternary trees

For $\mathsf{mapTree}_3$ we could use the following simple definition:

$$
\begin{array}{l}
\mathsf{mapTree}_3\ f\ (Leaf_3\ x) \qquad\quad = Leaf_3\ (f\ x) \\
\mathsf{mapTree}_3\ f\ (Fork_3\ xt\ yt\ zt) = Fork_3\ (\mathsf{mapTree}_3\ f\ xt)\ (\mathsf{mapTree}_3\ f\ yt)\ (\mathsf{mapTree}_3\ f\ zt)
\end{array}
$$

In Section 3.1 we gave an assertion relating $\mathsf{mapTree}$ and $\mathsf{flatten}$. We have a similar assertion for ternary trees:

$$\mathsf{flatten}_3\ (\mathsf{mapTree}_3\ f\ wt) = \mathsf{map}\ f\ (\mathsf{flatten}_3\ wt) \tag{11}$$

From the definition of $\mathsf{flatten}_3$ we would have four cases in the proof of Equation (11). For brevity let us just consider one case.

So, suppose that $wt = Fork_3\ (Leaf_3\ a)\ yt\ zt$ and, for the induction hypothesis, yt and zt satisfy Equation (11). For the left-hand side:

$$
\begin{array}{ll}
 & \mathsf{flatten}_3\ (\mathsf{mapTree}_3\ f\ wt) \\
= & \quad \{\text{definition}\} \\
 & \mathsf{flatten}_3\ (\mathsf{mapTree}_3\ f\ (Fork_3\ (Leaf_3\ a)\ yt\ zt)) \\
= & \quad \{\text{definition of } \mathsf{mapTree}_3\} \\
 & \mathsf{flatten}_3\ (Fork_3\ (\mathsf{mapTree}_3\ f\ (Leaf_3\ a))\ (\mathsf{mapTree}_3\ f\ yt)\ (\mathsf{mapTree}_3\ f\ zt)) \\
= & \quad \{\text{definition of } \mathsf{mapTree}_3\} \\
 & \mathsf{flatten}_3\ (Fork_3\ (Leaf_3\ (f\ a))\ (\mathsf{mapTree}_3\ f\ yt)\ (\mathsf{mapTree}_3\ f\ zt)) \\
= & \quad \{\text{definition of } \mathsf{flatten}_3\} \\
 & (f\ a) : (\mathsf{flatten}_3\ (\mathsf{mapTree}_3\ f\ yt)) \\
= & \quad \{\text{induction hypothesis}\}
\end{array}
$$

$$(f\ a) : (\mathsf{map}\ f\ (\mathsf{flatten}_3\ yt))$$

For the right hand side:

$$\begin{array}{ll} & \mathsf{map}\ f\ (\mathsf{flatten}_3\ wt) \\ = & \quad \{\text{definition}\} \\ & \mathsf{map}\ f\ (\mathsf{flatten}_3\ (Fork_3\ (Leaf_3\ a)\ yt\ zt)) \\ = & \quad \{\text{definition of } \mathsf{flatten}_3\} \\ & \mathsf{map}\ f\ (a : (\mathsf{flatten}_3\ yt)) \\ = & \quad \{\text{definition of } \mathsf{map}\} \\ & (f\ a) : (\mathsf{map}\ f\ (\mathsf{flatten}_3\ yt)) \end{array}$$

The proofs for the other three cases are similar. Comparing this proof to the one for binary trees given in Section 3.1 (which had only two cases to consider) we can see that the ternary tree proof is more complicated.

As an alternative, we can define $\mathsf{mapTree}_3$ using the equation:

$$\mathsf{mapTree}_3 = \mathsf{to3} \cdot \mathsf{mapTree} \cdot \mathsf{to2}$$

If we use the generalised version of to3 from Section 4 then we can produce a more general version of $\mathsf{mapTree}_3$:

$$\begin{array}{ll} \mathsf{mapTree}_3\ f\ (Leaf_3\ a) & = Leaf_3(f\ a) \\ \mathsf{mapTree}_3\ f\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) & = Fork_3\ (Leaf_3\ (f\ a))\ (\mathsf{mapTree}_3\ f\ yt)\ pt \\ \mathsf{mapTree}_3\ f\ (Fork_3\ xt\ yt\ (Leaf_3\ a)) & = Fork_3\ (\mathsf{mapTree}_3\ f\ xt)\ qt\ (Leaf_3\ (f\ a)) \\ \mathsf{mapTree}_3\ f\ (Fork_3\ xt\ yt\ zt) & = Fork_3\ (\mathsf{mapTree}_3\ f\ xt)\ rt\ (\mathsf{mapTree}_3\ f\ zt) \end{array}$$

The trees pt, qt and rt represent arbitrary ternary trees which could depend on the subtrees (such as xt or $Leaf_3\ a$) or on the function f. Since these ternary trees are "bogus" we can define them in anyway we choose — such as seeming to produce expression which are "incorrect". Note that the earlier definition of $\mathsf{mapTree}_3$ above is a particular instance of the generalised form.

4.4 Other tree operations

Let us briefly consider how to refine the other binary tree operations.

In Section 3 we stated a relationship between size and flatten and so by using the corresponding relationship for ternary operations:

$$\mathsf{size}_3 = \mathsf{length} \cdot \mathsf{flatten}_3$$

we can derive a definition for size_3. So, for instance (from the third clause in the definition of $\mathsf{flatten}_3$):

$$\mathsf{length}\ (\mathsf{flatten}_3\ (Fork_3\ xt\ yt\ (Leaf_3\ b)))$$

$$
\begin{array}{ll}
= & \{\text{definition of } \mathsf{flatten}_3\} \\
& \mathsf{length}\ (\mathsf{flatten}_3\ (xt +\!\!+ [b])) \\
= & \{\text{property: } \mathsf{length}\ (xs +\!\!+ ys) = \mathsf{length}\ xs + \mathsf{length}\ ys\} \\
& \mathsf{length}\ (\mathsf{flatten}_3\ xt) + \mathsf{length}\ [b] \\
= & \{\text{definition of } \mathsf{length}\} \\
& \mathsf{length}\ (\mathsf{flatten}_3\ xt) + 1 \\
= & \{\text{definition for } \mathsf{size}_3\} \\
& 1 + \mathsf{size}_3\ xt
\end{array}
$$

The other cases are similar and so we obtain the following definition:

$$
\begin{array}{ll}
\mathsf{size}_3\ (Leaf_3\ a) & = 1 \\
\mathsf{size}_3\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) & = 1 + \mathsf{size}_3\ yt \\
\mathsf{size}_3\ (Fork_3\ xt\ yt\ (Leaf_3\ b)) & = 1 + \mathsf{size}_3\ xt \\
\mathsf{size}_3\ (Fork_3\ xt\ yt\ zt) & = \mathsf{size}_3\ xt + \mathsf{size}_3\ zt
\end{array}
$$

Suppose that we have the following instance of $\mathsf{to3}$:

$$
\begin{array}{ll}
\mathsf{to3}\ (Leaf\ a) & = Leaf_3\ a \\
\mathsf{to3}\ (Fork\ (Leaf\ a)\ xt) & = Fork_3\ (Leaf_3\ a)\ (\mathsf{to3}\ xt)\ (Leaf_3\ p) \\
\mathsf{to3}\ (Fork\ xt\ (Leaf\ a)) & = Fork_3\ (\mathsf{to3}\ xt)\ (Leaf_3\ q)\ (Leaf_3\ a) \\
\mathsf{to3}\ (Fork\ xt\ yt) & = Fork_3\ (\mathsf{to3}\ yt)\ (Leaf_3\ r)\ (\mathsf{to3}\ xt)
\end{array}
$$

where p, q and r are arbitrary values with type α With this conversion function, we can use the following definition of size_3:

$$
\begin{array}{ll}
\mathsf{size}_3\ (Leaf_3\ x) & = 1 \\
\mathsf{size}_3\ (Fork_3\ xt\ yt\ zt) & = \mathsf{size}_3\ xt + \mathsf{size}_3\ yt + \mathsf{size}_3\ zt - 1
\end{array}
$$

So, by using different instances of the conversion function we can produce different specialisations of our ternary operations.

The ternary membership operation satisfies:

$$\mathsf{member}_3 = \mathsf{member} \cdot \mathsf{to2}$$

Using this equation we can derive the following operation:

$$
\begin{array}{ll}
\mathsf{member}_3\ p\ (Leaf_3\ a) & = p == a \\
\mathsf{member}_3\ p\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) & = p == a \vee \mathsf{member}_3\ p\ yt \\
\mathsf{member}_3\ p\ (Fork_3\ xt\ yt\ (Leaf_3\ a)) & = \mathsf{member}_3\ p\ xt \vee p == a \\
\mathsf{member}_3\ p\ (Fork_3\ xt\ yt\ zt) & = \mathsf{member}_3\ p\ zt \vee \mathsf{member}_3\ p\ xt
\end{array}
$$

We can collapse the last two cases into one and so we have

$$
\begin{array}{ll}
\mathsf{member}_3\ p\ (Leaf_3\ a) & = p == a \\
\mathsf{member}_3\ p\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) & = p == a \vee \mathsf{member}_3\ p\ yt \\
\mathsf{member}_3\ p\ (Fork_3\ xt\ yt\ zt) & = \mathsf{member}_3\ p\ xt \vee \mathsf{member}_3\ p\ zt
\end{array}
$$

5 Rose Trees

In [3], Haskell operations for a special kind of multi-way branching trees called *rose trees* is discussed in detail. (Note that rose tree are often known as *general trees*.) Each element of a rose tree contains a node and a (possibly empty) list of subtrees. We can define the type for rose trees as follows:

$$Rose\ \alpha = Node\ \alpha\ (List\ (Rose\ \alpha))$$

Note that many of the definitions for rose tree operation will use folds and so we will rely on various folds properties which are given in Section 6.

Despite seeming to be a more complicated structure, there is a correspondence between rose trees and binary trees. This means that we should be able to use rose trees to obfuscate binary trees. One way to convert from binary trees to rose trees is by using the following function:

$$\mathsf{toB}\ (Node\ a\ xts) = \mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ xts)$$

For converting back again we can use:

$$\begin{array}{lcl} \mathsf{toR}\ (Leaf\ a) & = & Node\ a\ [\,] \\ \mathsf{toR}\ (Fork\ xb\ yb) & = & Node\ a\ (xts +\!\!+ [\mathsf{toR}\ yb]) \\ & & \text{where } Node\ a\ xts = \mathsf{toR}\ xb \end{array}$$

Note that in [3] a more efficient version of toR is discussed but we will use the version given above to simplify our proofs and derivations.

If we want to use rose trees to obfuscate binary trees then we must have that the conversion function (in this case toR) is a right inverse of the abstraction function (toB).

Property 1 (Right inverse of toB)**.** The function toR is a right inverse for toB, *i.e.*

$$\mathsf{toB} \cdot \mathsf{toR} = id$$

Proof. Consider toB (toR zb) where zb is a binary tree.

Base Case Suppose that $zb = Leaf\ a$.

$$\begin{array}{ll} & \mathsf{toB}\ (\mathsf{toR}\ (Leaf\ a)) \\ = & \quad \{\text{definition of toR}\} \\ & \mathsf{toB}\ (Node\ a\ [\,]) \\ = & \quad \{\text{definition of toB}\} \\ & \mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ [\,]) \\ = & \quad \{\text{definition of map}\} \\ & \mathsf{foldl}\ Fork\ (Leaf\ a)\ [\,] \\ = & \quad \{\text{definition of foldl}\} \\ & Leaf\ a \end{array}$$

Step Case Suppose that $zb = Fork\ xb\ yb$ where xb and yb are binary trees. For the induction hypothesis, we suppose that xb and yb satisfy Property 1.

$$
\begin{aligned}
& \mathsf{toB}\ (\mathsf{toR}\ (Fork\ xb\ yb)) \\
= \quad & \{\text{definition of } \mathsf{toR} \text{ with } Node\ a\ xts = \mathsf{toR}\ xb\} \\
& \mathsf{toB}\ (Node\ a\ (xts +\!\!+ [\mathsf{toR}\ yb])) \\
= \quad & \{\text{definition of } \mathsf{toB}\} \\
& \mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ (xts +\!\!+ [\mathsf{toR}\ yb])) \\
= \quad & \{\mathsf{map}\ f\ (xs +\!\!+ [y]) = (\mathsf{map}\ f\ xs) +\!\!+ [f\ y]\} \\
& \mathsf{foldl}\ Fork\ (Leaf\ a)\ ((\mathsf{map}\ \mathsf{toB}\ xts) +\!\!+ [\mathsf{toB}\ (\mathsf{toR}\ yb)]) \\
= \quad & \{\text{Property 7: } \mathsf{foldl}\ f\ e\ (xs +\!\!+ ys) = \mathsf{foldl}\ f\ (\mathsf{foldl}\ f\ e\ xs)\ ys\} \\
& \mathsf{foldl}\ Fork\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ xts))\ [\mathsf{toB}\ (\mathsf{toR}\ yb)] \\
= \quad & \{\text{definition of } \mathsf{toB}\} \\
& \mathsf{foldl}\ Fork\ (\mathsf{toB}\ (Node\ a\ xts))\ [\mathsf{toB}\ (\mathsf{toR}\ yb)] \\
= \quad & \{Node\ a\ xts = \mathsf{toR}\ xb\} \\
& \mathsf{foldl}\ Fork\ (\mathsf{toB}\ (\mathsf{toR}\ xb))\ [\mathsf{toB}\ (\mathsf{toR}\ yb)] \\
= \quad & \{\text{induction hypothesis}\} \\
& \mathsf{foldl}\ Fork\ xb\ [yb] \\
= \quad & \{\text{definition of } \mathsf{foldl}\} \\
& Fork\ xb\ yb
\end{aligned}
$$

□

For this conversion, we also have the following property.

Property 2 (Left inverse of toB)**.** The function toR is a left inverse for toB, *i.e.*

$$\mathsf{toR} \cdot \mathsf{toB} = id$$

Proof. Consider $\mathsf{toR}\ (\mathsf{toB}\ (Node\ a\ xts))$ where xts is a list of rose trees.

Base Case Suppose that $xts = [\,]$.

$$
\begin{aligned}
& \mathsf{toR}\ (\mathsf{toB}\ (Node\ a\ [\,])) \\
= \quad & \{\text{definition of } \mathsf{toB}\} \\
& \mathsf{toR}\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ [\,])) \\
= \quad & \{\text{definition of } \mathsf{map}\} \\
& \mathsf{toR}\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ [\,]) \\
= \quad & \{\text{definition of } \mathsf{foldl}\} \\
& \mathsf{toR}\ (Leaf\ a)
\end{aligned}
$$

$=$ {definition of toR}
$Node\ a\ [\,]$

Step Case 1 Suppose that $xts = [yr]$ and, for the induction hypothesis, that the rose tree yr satisfies Property 2.

$\mathsf{toR}\ (\mathsf{toB}\ (Node\ a\ [yr]))$
$=$ {definition of toB}
$\mathsf{toR}\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ [yr]))$
$=$ {definition of map}
$\mathsf{toR}\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ [\mathsf{toB}\ yr])$
$=$ {definition of foldl}
$\mathsf{toR}\ (Fork\ (Leaf\ a)\ (\mathsf{toB}\ yr))$
$=$ {definition of toR with $Node\ a\ [\,] = \mathsf{toR}\ (Leaf\ a)$}
$Node\ a\ ([\,] +\!\!+ [\mathsf{toR}\ (\mathsf{toB}\ yr)])$
$=$ {definition of $+\!\!+$ and induction hypothesis}
$Node\ a\ [yr]$

Step Case 2 Suppose that $xr = Node\ a\ (yts +\!\!+ [yr])$ and, for the induction hypothesis, that $Node\ a\ yts$ and yr satisfy Property 2.

$\mathsf{toR}\ (\mathsf{toB}\ (Node\ a\ (yts +\!\!+ [yr])))$
$=$ {definition of toB}
$\mathsf{toR}\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ (yts +\!\!+ [yr])))$
$=$ {$\mathsf{map}\ f\ (xs +\!\!+ [y]) = (\mathsf{map}\ f\ xs) +\!\!+ [f\ y]$}
$\mathsf{toR}\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ ((\mathsf{map}\ \mathsf{toB}\ yts) +\!\!+ [\mathsf{toB}\ yr]))$
$=$ {Property 7: $\mathsf{foldl}\ f\ e\ (xs +\!\!+ ys) = \mathsf{foldl}\ f\ (\mathsf{foldl}\ f\ e\ xs)\ ys$}
$\mathsf{toR}\ (\mathsf{foldl}\ Fork\ (\mathsf{foldl}\ Fork\ (Leaf\ a)\ (\mathsf{map}\ \mathsf{toB}\ yts))\ [\mathsf{toB}\ yr])$
$=$ {definition of toB}
$\mathsf{toR}\ (\mathsf{foldl}\ Fork\ (\mathsf{toB}\ (Node\ a\ yts))\ [\mathsf{toB}\ yr])$
$=$ {definition of foldl}
$\mathsf{toR}\ (Fork\ (\mathsf{toB}\ (Node\ a\ yts))\ (\mathsf{toB}\ yr))$
$=$ {definition of toR with the induction hypothesis}
$Node\ a\ (yts +\!\!+ [yr])$

□

5.1 Rose Tree Operations

Now that we have our abstraction and conversion functions we can define operations for rose trees. Many of the operation definitions are taken from [3] but in the following sections we will prove various assertions for these operations.

5.1.1 Making Rose Trees

Let us consider how to define a function mkRose which builds a rose tree from a list. For refinement, we will insist that this function builds a rose tree that is equivalent to the corresponding binary tree built by mkTree. Since mkTree is a non-homogeneous operation, we use Equation (5) with the abstraction function toB for trees (and id for lists) to obtain:

$$\mathsf{mkTree} = \mathsf{toB} \cdot \mathsf{mkRose}$$

Since $\mathsf{toR} \cdot \mathsf{toB} = id$ (Property 2), we have that:

$$\mathsf{mkRose} = \mathsf{toR} \cdot \mathsf{mkTree}$$

which allows us to derive a definition for mkRose.

So, for all non-empty finite lists xs, we want

$$\mathsf{mkRose}\ xs = \mathsf{toR}\ (\mathsf{mkTree}\ xs) \tag{12}$$

Let $m = \mathsf{div}\ (\mathsf{length}\ xs)\ 2$ and $(ys, zs) = \mathsf{splitAt}\ m\ xs$. We will derive Equation (12) by induction on the length of xs.

Base Case Suppose that length xs is 1. Then $m = 0$ and so

$$\begin{array}{ll} & \mathsf{toR}\ (\mathsf{mkTree}\ xs) \\ = & \quad \{\text{definition of mkTree with } m = 0\} \\ & \mathsf{toR}\ (Leaf\ (\mathsf{head}\ zs)) \\ = & \quad \{\text{definition of toR}\} \\ & Node\ (\mathsf{head}\ zs)\ [\,] \end{array}$$

Step Case Suppose that the length of xs is greater than 1. By the definition of splitAt (and m) the lengths of ys and zs are less than the length of xs. So, for the induction hypothesis, we suppose that ys and zs satisfy Equation (12).

$$\begin{array}{ll} & \mathsf{toR}\ (\mathsf{mkTree}\ xs) \\ = & \quad \{\text{definition of mkTree with } m \neq 0\} \\ & \mathsf{toR}\ (Fork\ (\mathsf{mkTree}\ ys)\ (\mathsf{mkTree}\ zs)) \\ = & \quad \{\text{definition of toR with } Node\ y\ yts = \mathsf{toR}\ (\mathsf{mkTree}\ ys)\} \end{array}$$

$$
\begin{aligned}
& Node\ y\ (yts +\!\!+ [\mathsf{toR}\ (\mathsf{mkTree}\ zs)]) \\
= \quad & \{\text{induction hypothesis}\} \\
& Node\ y\ (yts +\!\!+ [\mathsf{mkRose}\ zs])
\end{aligned}
$$

By the induction hypothesis, $\mathsf{toR}\ (\mathsf{mkTree}\ ys) = \mathsf{mkRose}\ ys$ and so $Node\ y\ yts = \mathsf{mkRose}\ ys$. Putting both cases together we have:

$$
\begin{aligned}
& \mathsf{mkRose}\ xs \\
& \quad \mid m == 0 \quad = Node\ (\mathsf{head}\ zs)\ [\,] \\
& \quad \mid \text{otherwise} = Node\ y\ (yts +\!\!+ [\mathsf{mkRose}\ zs]) \\
& \qquad\qquad \text{where}\ Node\ y\ yts = \mathsf{mkRose}\ ys \\
& \qquad\qquad\qquad\quad (ys, zs) \quad\ = \mathsf{splitAt}\ m\ xs \\
& \qquad\qquad\qquad\quad m \qquad\quad\ = \mathsf{div}\ (\mathsf{length}\ xs)\ 2
\end{aligned}
$$

5.1.2 Membership

We can derive a membership operation memRose using Equation (5) :

$$\mathsf{memRose}\ v = (\mathsf{member}\ v) \cdot \mathsf{toB}$$

Proof. We derive a definition for memRose that satisfies

$$\mathsf{memRose}\ v\ (Node\ x\ xts) = \mathsf{member}\ v\ (\mathsf{toB}\ (Node\ x\ xts)) \tag{13}$$

by using induction on xts.

Base Case We will suppose that xts is empty.

$$
\begin{aligned}
& \mathsf{member}\ v\ (\mathsf{toB}\ (Node\ x\ [\,])) \\
= \quad & \{\text{definition of toB}\} \\
& \mathsf{member}\ v\ (\mathsf{foldl}\ Fork\ (Leaf\ x)\ (\mathsf{map}\ \mathsf{toB}\ [\,])) \\
= \quad & \{\text{definition of map}\} \\
& \mathsf{member}\ v\ (\mathsf{foldl}\ Fork\ (Leaf\ x)\ [\,]) \\
= \quad & \{\text{definition of foldl}\} \\
& \mathsf{member}\ v\ (Leaf\ x) \\
= \quad & \{\text{definition of member}\} \\
& v == x
\end{aligned}
$$

Thus

$$\mathsf{memRose}\ v\ (Node\ x\ [\,]) = v == x$$

Step Case Suppose that xts is non-empty and, for the induction hypothesis, that all of the trees in xts satisfy Equation (13). Consider

$$M = \mathsf{member}\ v\ (\mathsf{foldl}\ Fork\ (Leaf\ x)\ (\mathsf{map}\ \mathsf{toB}\ xts))$$

We will use the Fusion Theorem for Fold Left (Property 3) so that we can fuse member into the fold. For this theorem, we take $f = \mathsf{member}\ v$ (which is strict), $g = Fork$ and $a = Leaf\ x$. Now

$$b = \mathsf{member}\ v\ (Leaf\ x) = v == x$$

We now need a function h such that $f\ (g\ xt\ yt) = h\ (f\ xt)\ yt$.

$$f\ (g\ xt\ yt) = \mathsf{member}\ v\ (Fork\ xt\ yt) = \mathsf{member}\ v\ xt \vee \mathsf{member}\ v\ yt$$

We take h to be $(\lambda\ xt\ yt.\ xt \vee \mathsf{member}\ v\ yt)$ and so:

$$\begin{array}{lll}
M & = & \mathsf{foldl}\ (\lambda\ xt\ yt.\ xt \vee \mathsf{member}\ v\ yt)\ (v == x)\ (\mathsf{map}\ \mathsf{toB}\ xts) \\
& = & \quad \{\text{FoldLeft-Map Theorem (Property 5)}\} \\
& & \mathsf{foldl}\ (\lambda\ xt\ yt.\ xt\ \vee \mathsf{member}\ v\ (\mathsf{toB}\ yt))\ (v == x)\ xts \\
& = & \quad \{\text{induction hypothesis}\} \\
& & \mathsf{foldl}\ (\lambda\ xt\ yt.\ xt \vee \mathsf{memRose}\ v\ yt)\ (v == x)\ xts \\
& = & \quad \{\text{FoldLeft-Map Theorem (Property 5)}\} \\
& & \mathsf{foldl}\ (\lambda\ xt\ yt.\ xt \vee yt)\ (v == x)\ (\mathsf{map}\ (\mathsf{memRose}\ v)\ xts) \\
& = & \quad \{\text{simplification}\} \\
& & \mathsf{foldl}\ (\vee)\ (v == x)\ (\mathsf{map}\ (\mathsf{memRose}\ v)\ xts)
\end{array}$$

Thus from this case, we have:

$$xts \neq [\,] \Rightarrow \mathsf{memRose}\ v\ (Node\ x\ xts) = \mathsf{foldl}\ (\vee)\ (v == x)\ (\mathsf{map}\ (\mathsf{memRose}\ v)\ xts))$$

Let us consider

$$\begin{array}{lll}
 & & \mathsf{foldl}\ (\vee)\ (v == x)\ (\mathsf{map}\ (\mathsf{memRose}\ v)\ [\,]) \\
= & & \quad \{\text{definition of map}\} \\
 & & \mathsf{foldl}\ (\vee)\ (v == x)\ [\,] \\
= & & \quad \{\text{definition of foldl}\} \\
 & & v == x
\end{array}$$

This matches the expression for the base case. Thus our definition is:

$$\mathsf{memRose}\ v\ (Node\ x\ xts) = \mathsf{foldl}\ (\vee)\ (v == x)\ (\mathsf{map}\ (\mathsf{memRose}\ v)\ xts)$$

□

5.1.3 Flattening

To flatten rose trees, we can define an operation which satisfies

$$\mathsf{flattenRose} = \mathsf{flatten} \cdot \mathsf{toB}$$

By the definition of flatten in Section 3, $\mathsf{flatten}\ (Fork\ xt\ yt) = \mathsf{flatten}\ xt +\!\!+ \mathsf{flatten}\ yt$ and by following a similar derivation to memRose, we find that

$$\mathsf{flattenRose}\ (Node\ x\ xts) = \mathsf{foldl}\ (+\!\!+)\ [x]\ (\mathsf{map}\ \mathsf{flattenRose}\ xts)$$

Now $\mathsf{concat} = \mathsf{foldl}\ (+\!\!+)\ [\,]$ and so we can define

$$\mathsf{flattenRose}\ (Node\ x\ xts) = x : \mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ xts)$$

which matches the definition given in [3].

As with the operations for binary trees, we have the following property for all finite non-empty lists xs:

$$\mathsf{flattenRose}\ (\mathsf{mkRose}\ xs) = xs \tag{14}$$

Proof. Let $m = \mathsf{div}\ (\mathsf{length}\ xs)\ 2$ and $(ys, zs) = \mathsf{splitAt}\ m\ xs$. We will prove Equation (14) by induction on the length of xs.

Base Case Suppose that $\mathsf{length}\ xs$ is 1. Then $m = 0$ and so

$$\begin{aligned}
& \mathsf{flattenRose}\ (\mathsf{mkRose}\ xs) \\
= \quad & \{\text{definition of mkRose with } m = 0\} \\
& \mathsf{flattenRose}\ (Node\ (\mathsf{head}\ zs)\ [\,]) \\
= \quad & \{\text{definition of flattenRose}\} \\
& (\mathsf{head}\ zs) : \mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ [\,]) \\
= \quad & \{\text{definitions of concat and map}\} \\
& [\mathsf{head}\ zs] \\
= \quad & \{\text{definitions}\} \\
& xs
\end{aligned}$$

Step Case Suppose that the length of xs is greater than 1. By the definition of splitAt (and m) the lengths of ys and zs are less than the length of xs. So, for the induction hypothesis, we suppose that ys and zs satisfy Equation (14).

$$\begin{aligned}
& \mathsf{flattenRose}\ (\mathsf{mkRose}\ xs) \\
= \quad & \{\text{definition of mkRose with } Node\ y\ yts = \mathsf{mkRose}\ ys\} \\
& \mathsf{flattenRose}\ (Node\ y\ (yts +\!\!+ [\mathsf{mkRose}\ zs])) \\
= \quad & \{\text{definition of flattenRose}\}
\end{aligned}$$

$$
\begin{aligned}
& y : \mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ (yts +\!\!+ [\mathsf{mkRose}\ zs])) \\
= \quad & \{\text{property: } \mathsf{map}\ f\ (xs +\!\!+ [x]) = (\mathsf{map}\ f\ xs) +\!\!+ [f\ x]\} \\
& y : \mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ yts +\!\!+ [\mathsf{flattenRose}\ (\mathsf{mkRose}\ zs)]) \\
= \quad & \{\text{induction hypothesis}\} \\
& y : \mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ yts +\!\!+ [zs]) \\
= \quad & \{\text{property: } \mathsf{concat}\ (ts +\!\!+ xs) = \mathsf{concat}\ ts +\!\!+ \mathsf{concat}\ xs\} \\
& y : (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ yts) +\!\!+ \mathsf{concat}\ [zs]) \\
= \quad & \{\text{definition of } \mathsf{concat}\} \\
& y : (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ yts) +\!\!+ zs) \\
= \quad & \{\text{definition of } +\!\!+\} \\
& (y : \mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ yts)) +\!\!+ zs \\
= \quad & \{\text{definition of } \mathsf{flattenRose}\} \\
& \mathsf{flattenRose}\ (Node\ y\ yts) +\!\!+ zs \\
= \quad & \{\text{definition of } Node\ y\ yts\} \\
& \mathsf{flattenRose}\ (\mathsf{mkRose}\ ys) +\!\!+ zs \\
= \quad & \{\text{induction hypothesis}\} \\
& ys +\!\!+ zs \\
= \quad & \{\text{definitions and property of } \mathsf{splitAt}\} \\
& xs
\end{aligned}
$$

□

5.1.4 Size

From Section 3, we know that $\mathsf{size} = \mathsf{length} \cdot \mathsf{flatten}$. Thus for rose trees, we want

$$\mathsf{sizeRose} = \mathsf{length} \cdot \mathsf{flattenRose}$$

Using the above foldl definition for flattenRose, we have

$$\mathsf{sizeRose}\ (Node\ x\ xts) = \mathsf{length}\ (\mathsf{foldl}\ (+\!\!+)\ [x]\ (\mathsf{map}\ \mathsf{flattenRose}\ xts))$$

As length is strict, we can use the fusion theorem for fold left (Property 3). Now $\mathsf{length}\ [x] = 1$ and

$$\mathsf{length}\ (xs +\!\!+ ys) = \mathsf{length}\ xs + \mathsf{length}\ ys$$

So we take $h = (\lambda\ xs\ ys.\ xs + \mathsf{length}\ ys)$ and the right hand side becomes

$$\mathsf{foldl}\ (\lambda\ xs\ ys.\ xs + \mathsf{length}\ ys)\ 1\ (\mathsf{map}\ \mathsf{flattenRose}\ xts)$$

Using the FoldLeft-Map Theorem (Property 5) and $\mathsf{map}\ f \cdot \mathsf{map}\ g = \mathsf{map}\ (f \cdot g)$:

$$\mathsf{foldl}\ (+)\ 1\ (\mathsf{map}\ (\mathsf{length} \cdot \mathsf{flattenRose})\ xts)$$

Now $\mathsf{sum}\ xs$ can be defined as $\mathsf{foldl}\ (+)\ 0\ xs$ and so $\mathsf{foldl}\ (+)\ 1\ xs$ is equivalent to $1 + \mathsf{sum}\ xs$. Since we assume that $\mathsf{sizeRose} = \mathsf{length} \cdot \mathsf{flattenRose}$, then:

$$\mathsf{sizeRose}\ (Node\ x\ xts) = 1 + \mathsf{sum}\ (\mathsf{map}\ \mathsf{sizeRose}\ xts)$$

So to find the size of $Node\ x\ xts$ we sum together the size of each of the rose trees in xts and add 1 for the value x. Note that this matches the definition given in [3].

5.1.5 Map for Rose Trees

We define an operation $\mathsf{mapRose}\ f$ which applies a function f to all of the elements in a rose tree. Consider $\mathsf{mapRose}\ f\ (Node\ x\ xts)$. The function f needs to be applied to the value x and then $\mathsf{mapRose}\ f$ needs to be applied to each rose tree in xts. This leads us to the following definition:

$$\mathsf{mapRose}\ f\ (Node\ x\ xts) = Node\ (f\ x)\ (\mathsf{map}\ (\mathsf{mapRose}\ f)\ xts)$$

We can show this definition is correct by using Equation (4) to prove that

$$\mathsf{mapTree}\ f \cdot \mathsf{toB} = \mathsf{toB} \cdot \mathsf{mapRose}\ f$$

(the details are omitted).

As with the previous tree mapping and flattening functions, mapRose and flattenRose satisfy the following property:

$$\mathsf{flattenRose}\ (\mathsf{mapRose}\ f\ rts) = \mathsf{map}\ f\ (\mathsf{flattenRose}\ rts) \tag{15}$$

for all finite rose trees rts.

Proof. We will show that Equation (15) holds by induction on rts.

Base Case We suppose that $rts = Node\ x\ [\,]$ and then for the left hand side

$$\begin{aligned}
& \mathsf{flattenRose}\ (\mathsf{mapRose}\ f\ (Node\ x\ [\,])) \\
= \quad & \{\text{definition of mapRose}\} \\
& \mathsf{flattenRose}\ (Node\ (f\ x)\ (\mathsf{map}\ (\mathsf{mapRose}\ f)\ [\,])) \\
= \quad & \{\text{definition of map}\} \\
& \mathsf{flattenRose}\ (Node\ (f\ x)\ [\,]) \\
= \quad & \{\text{definition of flattenRose}\} \\
& (f\ x) : (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ [\,])) \\
= \quad & \{\text{definitions of map and concat}\} \\
& (f\ x) : [\,] \\
= \quad & \{\text{notation}\} \\
& [f\ x]
\end{aligned}$$

For the right hand side

$$
\begin{aligned}
& \mathsf{map}\ f\ (\mathsf{flattenRose}\ (Node\ x\ [\,])) \\
= \quad & \{\text{definition of } \mathsf{flattenRose}\} \\
& \mathsf{map}\ f\ (x : (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ [\,]))) \\
= \quad & \{\text{definition of } \mathsf{map} \text{ and } \mathsf{concat}\} \\
& \mathsf{map}\ f\ (x : [\,]) \\
= \quad & \{\text{definitions of } \mathsf{map} \text{ and notation}\} \\
& [f\ x]
\end{aligned}
$$

Step Case Let $rts = Node\ x\ xts$ and for the induction hypothesis we suppose that every tree in xts satisfies Equation (15). Starting with the left hand side

$$
\begin{aligned}
& \mathsf{flattenRose}\ (\mathsf{mapRose}\ f\ (Node\ x\ xts)) \\
= \quad & \{\text{definition of } \mathsf{mapRose}\} \\
& \mathsf{flattenRose}\ (Node\ (f\ x)\ (\mathsf{map}\ (\mathsf{mapRose}\ f)\ xts)) \\
= \quad & \{\text{definition of } \mathsf{flattenRose}\} \\
& (f\ x) : (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ (\mathsf{map}\ (\mathsf{mapRose}\ f)\ xts))) \\
= \quad & \{\text{property of } \mathsf{map}\text{: } \mathsf{map}\ f \cdot \mathsf{map}\ g = \mathsf{map}\ (f \cdot g)\} \\
& (f\ x) : (\mathsf{concat}\ (\mathsf{map}\ (\mathsf{flattenRose} \cdot (\mathsf{mapRose}\ f))\ xts)) \\
= \quad & \{\text{induction hypothesis}\} \\
& (f\ x) : (\mathsf{concat}\ (\mathsf{map}\ ((\mathsf{map}\ f) \cdot \mathsf{flattenRose})\ xts)) \\
= \quad & \{\text{property of } \mathsf{map}\text{: } \mathsf{map}\ f \cdot \mathsf{map}\ g = \mathsf{map}\ (f \cdot g)\} \\
& (f\ x) : (\mathsf{concat}\ (\mathsf{map}\ (\mathsf{map}\ f)\ (\mathsf{map}\ \mathsf{flattenRose}\ xts))) \\
= \quad & \{\text{property of } \mathsf{map}\text{: } \mathsf{map}\ f \cdot \mathsf{concat} = \mathsf{concat} \cdot \mathsf{map}\ (\mathsf{map}\ f)\} \\
& (f\ x) : (\mathsf{map}\ f\ (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ xts))) \\
= \quad & \{\text{definition of } \mathsf{map}\} \\
& \mathsf{map}\ f\ (x : (\mathsf{concat}\ (\mathsf{map}\ \mathsf{flattenRose}\ xts))) \\
= \quad & \{\text{definition of } \mathsf{flattenRose}\} \\
& \mathsf{map}\ f\ (\mathsf{flattenRose}\ (Node\ x\ xts))
\end{aligned}
$$

□

Comparing this proof to the one for Equation (7) given in Section 3.1 we can see that the proof for rose trees is much more complicated and it uses some properties for map (stated in Section 2) which also need to be proved.

5.2 Suitability for obfuscation

We have seen that we can use rose trees to obfuscate binary trees. Since rose trees have a more complicated data structure than binary trees then the definitions of the operations were slightly harder to understand. The proofs for properties for rose trees are generally hard to construct than those for binary trees (in Section 5.1.5 we gave an example proof) and so by the definition given in [7] these operations are obfuscated.

Some of the definitions for rose tree operations relied on the fold functions. This has two immediate consequences: we have to rely on many fold properties (given in Section 6) which constructing proofs for rose tree operations and we have less flexibility when defining rose tree operations. Further discussion of the use of folds with obfuscation can be found in Section 6.5.

6 Folds and Unfolds

In this section we discuss the fold and unfold functions. In particular we give various properties that these functions satisfy — many of these properties have been used throughout Section 5 in the discussion of rose trees.

6.1 Definitions and properties of fold

The fold left function foldl is defined as follows:

$$\begin{aligned}
&\mathsf{foldl} && :: (\beta \to \alpha \to \beta) \to \beta \to \mathit{List}\ \alpha \to \beta \\
&\mathsf{foldl}\ f\ a\ [\,] && = a \\
&\mathsf{foldl}\ f\ a\ (x : xs) && = \mathsf{foldl}\ f\ (f\ a\ x)\ xs
\end{aligned}$$

Similarly the fold right function foldr is defined as:

$$\begin{aligned}
&\mathsf{foldr} && :: (\alpha \to \beta \to \beta) \to \beta \to \mathit{List}\ \alpha \to \beta \\
&\mathsf{foldr}\ f\ a\ [\,] && = a \\
&\mathsf{foldr}\ f\ a\ (x : xs) && = f\ x\ (\mathsf{foldr}\ f\ a\ xs)
\end{aligned}$$

The fold functions replace the list constructors : and [ ] with f and a respectively. So, for example,

$$\begin{aligned}
\mathsf{foldl}\ \oplus\ b\ [x_1, x_2, \ldots, x_n] &= (\ldots((b \oplus x_1) \oplus x_2)\ldots) \oplus x_n \\
\mathsf{foldr}\ \otimes\ b\ [x_1, x_2, \ldots, x_n] &= x_1 \otimes (x_2 \otimes (\ldots(x_n \otimes b)\ldots))
\end{aligned}$$

We can define some of the standard list operations from Section 2 as folds. For example,

$$\begin{aligned}
\mathsf{map}\ f &= \mathsf{foldr}\ ((:).f)\ [\,] \\
\mathsf{length} &= \mathsf{foldl}\ (\lambda\ x\ y.\ x + 1)\ 0 \\
xs +\!\!+ ys &= \mathsf{foldr}\ (:)\ ys\ xs \\
\mathsf{concat} &= \mathsf{foldl}\ (+\!\!+)\ [\,]
\end{aligned}$$

We can even write head as foldr $(\lambda\ x\ y.x)$ $(\bot)$ and the identity for lists can be written as foldr (:) [ ]. The fold functions satisfy various properties (some of which were used in Section 5 when we discussed rose trees). Many of the properties, which are called *fusion theorems*, allow us to combine folds with other functions.

Property 3 (Fusion Theorem for Fold Left)**.** For a function f

$$f \cdot \mathsf{foldl}\ g\ a = \mathsf{foldl}\ h\ b$$

if f is strict, $f\ a = b$ and h satisfies the relationship:

$$f\ (g\ x\ y) = h\ (f\ x)\ y$$

for all x and y.

Property 4 (Fusion Theorem for Fold Right)**.** For a function f

$$f \cdot \mathsf{foldr}\ g\ a = \mathsf{foldr}\ h\ b$$

if f is strict, $f\ a = b$ and h satisfies the relationship:

$$f\ (g\ x\ y) = h\ x\ (f\ y)$$

for all x and y.

We can construct two other fusion rules for the folds — one using map and the other using $+\!\!+$. Consider the expression $\mathsf{foldl}\ f\ a\ (\mathsf{map}\ g\ [x_1, x_2, \ldots, x_n])$. By the definition of map, we have $\mathsf{foldl}\ f\ a\ (g\ x_1, g\ x_2, \ldots, g\ x_n)$. Expanding out the fold gives

$$f\ (\ldots (f\ (f\ a\ (g\ x_1))\ (g\ x_2))\ldots)\ (g\ x_n)$$

This is equivalent to $\mathsf{foldl}\ (\lambda\ x\ y.\ f\ x\ (g\ y))\ a\ [x_1, x_2, \ldots, x_n]$. We can construct a similar argument for foldr and so we have the following two theorems for map.

Property 5 (FoldLeft-Map Theorem)**.** For functions f and g

$$(\mathsf{foldl}\ f\ a) \cdot (\mathsf{map}\ g) = \mathsf{foldl}\ (\lambda\ x\ y.\ f\ x\ (g\ y))\ a$$

Property 6 (FoldRight-Map Theorem)**.** For functions f and g

$$(\mathsf{foldr}\ f\ a) \cdot (\mathsf{map}\ g) = \mathsf{foldr}\ (f \cdot g)\ a$$

Let us now consider how to combine $+\!\!+$ and foldr so that we can rewrite $\mathsf{foldr}\ p\ e\ (xs +\!\!+ ys)$ without using $+\!\!+$. We can write $xs +\!\!+ ys$ as $\mathsf{foldr}\ (:)\ ys\ xs$ and so we can use the Fusion Theorem for Fold Right (Property 4). So we would like

$$(\mathsf{foldr}\ p\ e) \cdot (\mathsf{foldr}\ (:)\ ys) = \mathsf{foldr}\ h\ b$$

Since $\mathsf{foldr}\ p\ e$ is strict we can use the fusion theorem with $f = \mathsf{foldr}\ p\ e$, $g = (:)$ and $a = ys$. Thus $b = \mathsf{foldr}\ p\ e\ ys$.

$$\begin{aligned} f\ (g\ x\ xs) &= \mathsf{foldr}\ p\ e\ ((:)\ x\ xs) \\ &= \mathsf{foldr}\ p\ e\ (x : xs) \\ &= p\ x\ (\mathsf{foldr}\ p\ e\ xs) \\ &= h\ x\ (f\ xs) \end{aligned}$$

and so $h = p$. By the fusion theorem:

$$\mathsf{foldr}\ p\ e\ (\mathsf{foldr}\ (:)\ ys\ xs) = \mathsf{foldr}\ p\ (\mathsf{foldr}\ p\ e\ ys)\ xs$$

Therefore

$$\mathsf{foldr}\ p\ e\ (xs +\!\!+\ ys) = \mathsf{foldr}\ p\ (\mathsf{foldr}\ p\ e\ ys)\ xs$$

We can construct a similar argument for foldl and so we have the following two theorems.

Property 7 (FoldLeft-Cat Theorem)**.** For finite lists xs and ys:

$$\mathsf{foldl}\ p\ e\ (xs +\!\!+\ ys) = \mathsf{foldl}\ p\ (\mathsf{foldl}\ p\ e\ xs)\ ys$$

Property 8 (FoldRight-Cat Theorem)**.** For finite lists xs and ys:

$$\mathsf{foldr}\ p\ e\ (xs +\!\!+\ ys) = \mathsf{foldr}\ p\ (\mathsf{foldr}\ p\ e\ ys)\ xs$$

6.2 List Unfolds

A list unfold produces a list from another data structure — it can be seen as an "inverse" for fold. We follow the definition for unfolds given in [11]:

$$\begin{array}{l}\mathsf{unfold} :: (\alpha \to \mathbb{B}) \to (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to \mathit{List}\ \beta \\ \mathsf{unfold}\ p\ f\ g\ x \\ \quad \left| \begin{array}{ll} p\ x & = [\,] \\ \text{otherwise} & = (f\ x) : (\mathsf{unfold}\ p\ f\ g\ (g\ x)) \end{array} \right.\end{array}$$

As an example

$$\mathsf{map}\ f = \mathsf{unfold}\ (== [\,])\ (f.\mathsf{head})\ (\mathsf{tail})$$

Property 9 (Unfold Fusion)**.** For a function f,

$$(\mathsf{unfold}\ p\ g\ h) \cdot f = \mathsf{unfold}\ p'\ g'\ h'$$

if $p \cdot f = p'$, $g \cdot f = g'$ and $h \cdot f = f \cdot h'$

Property 10 (Unfold Map Fusion)**.** For map:

$$(\mathsf{map}\ f) \cdot (\mathsf{unfold}\ p\ g\ h) \quad = \quad \mathsf{unfold}\ p\ (f \cdot g)\ h$$

A fold followed by an unfold is called a *hylomorphism* [13]. In [10], a hylomorphism is defined to be:

$$\mathsf{hylo}\ f\ e\ p\ g\ h \ = \ \mathsf{fold}\ f\ e \cdot \mathsf{unfold}\ p\ g\ h$$

which gives

$$\mathsf{hylo}\ f\ e\ p\ g\ h\ x = \text{if } p\ x \text{ then } e \text{ else } f\ (g\ x)\ (\mathsf{hylo}\ f\ e\ p\ g\ h\ (h\ x)) \tag{16}$$

6.3 Folds for Binary Trees

To define a fold for binary trees we need to define two functions f and g which describe how the constructors $Null$ and $Fork$ are transformed. We define foldTree as follows:

$$\begin{aligned}&\mathsf{foldTree}\ f\ g\ (Leaf\ a) &&= f\ a\\ &\mathsf{foldTree}\ f\ g\ (Fork\ xt\ yt) &&= g\ (\mathsf{foldTree}\ f\ g\ xt)\ (\mathsf{foldTree}\ f\ g\ yt)\end{aligned}$$

and it has type

$$\mathsf{foldTree} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to Tree\ \alpha \to \beta$$

We can easily write flatten as an instance of foldTree as follows:

$$\mathsf{flatten} = \mathsf{foldTree}\ (\lambda\ x.\ [x])\ (+\!\!+)$$

For mapTree we have that

$$\mathsf{mapTree}\ f = \mathsf{foldTree}\ (Leaf \cdot f)\ Fork$$

and for member:

$$\mathsf{member}\ v = \mathsf{foldTree}\ (== v)\ (\vee)$$

As with fold for lists, we have a fusion theorem for foldTree.

Property 11 (Fusion Theorem for binary trees)**.** For a strict function h

$$h \cdot (\mathsf{foldTree}\ f_1\ g_1) = \mathsf{foldTree}\ f_2\ g_2$$

if $h \cdot f_1 = f_2$ and

$$h\ (g_1\ r\ s) = g_2\ (h\ r)\ (h\ s)$$

for all r and s.

Proof. We prove $h\ (\mathsf{foldTree}\ f_1\ g_1\ zt) = \mathsf{foldTree}\ f_2\ g_2\ zt$ by induction on zt.

Case ($\perp$) We suppose that $zt = \perp$.

$$\begin{aligned}&h\ (\mathsf{foldTree}\ f_1\ g_1\ \perp)\\ =\quad &\{\text{case exhaustion}\}\\ &h\ \perp\\ =\quad &\{\text{assumption: } h \text{ is strict}\}\\ &\perp\\ =\quad &\{\text{case exhaustion}\}\\ &\mathsf{foldTree}\ f_2\ g_2\ \perp\end{aligned}$$

Case $(Leaf)$ Suppose that $zt = Leaf\ a$.

$$
\begin{array}{ll}
 & h\ (\mathsf{foldTree}\ f_1\ g_1\ (Leaf\ a)) \\
= & \quad \{\text{definition of } \mathsf{foldTree}\} \\
 & h\ (f_1\ a) \\
= & \quad \{\text{assumption: } h \cdot f_1 = f_2\} \\
 & f_2\ a \\
= & \quad \{\text{definition of } \mathsf{foldTree}\} \\
 & \mathsf{foldTree}\ f_2\ g_2\ (Leaf\ a)
\end{array}
$$

Case $(Fork)$ Suppose that $zt = Fork\ xt\ yt$.

$$
\begin{array}{ll}
 & h\ (\mathsf{foldTree}\ f_1\ g_1\ (Fork\ xt\ yt)) \\
= & \quad \{\text{definition of } \mathsf{foldTree}\} \\
 & h\ (g_1\ (\mathsf{foldTree}\ f_1\ g_1\ xt)\ (\mathsf{foldTree}\ f_2\ g_2\ yt)) \\
= & \quad \{\text{assumption: } h\ (g_1\ r\ s) = g_2\ (h\ r)\ (h\ s)\} \\
 & g_2\ (h\ (\mathsf{foldTree}\ f_1\ g_1\ xt))\ (h\ (\mathsf{foldTree}\ f_1\ g_1\ yt)) \\
= & \quad \{\text{induction hypothesis}\} \\
 & g_2\ (\mathsf{foldTree}\ f_2\ g_2\ xt)\ (\mathsf{foldTree}\ f_2\ g_2\ yt) \\
= & \quad \{\text{definition of } \mathsf{foldTree}\} \\
 & \mathsf{foldTree}\ f_2\ g_2\ (Fork\ xt\ yt)
\end{array}
$$

□

For an example of the fusion theorem in action, let us consider size. From Section 3, we know that

$$\mathsf{size} = \mathsf{length} \cdot \mathsf{flatten}$$

So, using the fold version of flatten, we would like function f_2 and g_2 such that

$$\mathsf{foldTree}\ f_2\ g_2 = \mathsf{length} \cdot (\mathsf{foldTree}\ (\lambda\ x.\ [x])\ (+\!\!+))$$

and $\mathsf{length}\ \bot = \bot$. For the fusion theorem, we need

$$f_2 = \mathsf{length} \cdot (\lambda\ x.\ [x])$$

and since $\mathsf{length}\ [x] = 1$ then

$$f_2 = (\lambda\ x.\ 1)$$

Also, we need

$$g_2\ (\mathsf{length}\ r)\ (\mathsf{length}\ s) = \mathsf{length}\ (r +\!\!+ s)$$

but $\mathsf{length}\ (r +\!\!+ s) = \mathsf{length}\ r + \mathsf{length}\ s$ we have that $g_2 = (+)$. Thus by the fusion theorem size can be written as

$$\mathsf{size} = \mathsf{foldTree}\ (\lambda\ x.\ 1)\ (+)$$

6.3.1 Unfolds and mktree

So far, we have not mentioned mkTree — can it written as a fold? Since mkTree changes a list into a tree we might expect to use a list fold. However we have two problems: firstly we do not have a mapping for [ ]; and secondly our definition of mkTree uses two parts of a list rather than just using the head and tail of a list. The first problem can be fixed by allowing a $Null$ tree (so that $\mathsf{mkTree}\ Null = [\,]$).

We can use fold fusion to try to "force" a fold definition for mkTree. From Section 3 we know that $\mathsf{flatten} \cdot \mathsf{mkTree} = id$. So let us suppose that $\mathsf{mkTree} = \mathsf{foldr}\ g\ a$ for some g and a. We can the identity as $\mathsf{foldr}\ (:)\ [\,]$. So we would like

$$\mathsf{flatten} \cdot (\mathsf{foldr}\ g\ a) = \mathsf{foldr}\ (:)\ [\,]$$

The function flatten is strict and we take $a = Null$ so that $\mathsf{flatten}\ Null = [\,]$. For fusion, we have that

$$\mathsf{flatten}\ (g\ x\ yt) = x : (\mathsf{flatten}\ yt)$$

By the definition of flatten, we have that $\mathsf{flatten}\ (Fork\ (Leaf\ x)\ yt) = [x] +\!\!+ \mathsf{flatten}\ yt$. If we take $g\ x\ yt = Fork\ (Leaf\ x)\ yt$ then

$$\mathsf{foldr}\ (\lambda\ x\ yt.\ Fork\ (Leaf\ x)\ yt)\ Null$$

should produce a function for making trees. In fact this function matches up with mkTree' from Section 3. We can actually write mkTree as a tree unfold. An unfold for binary trees has the type:

$$\mathsf{unfoldTree} :: (\alpha \to \mathbb{B}) \to (\alpha \to \beta) \to (\alpha \to (\alpha, \alpha)) \to \alpha \to Tree\ \beta$$

and we can define it as:

$$\begin{array}{l}\mathsf{unfoldTree}\ p\ f\ g\ x \\ \quad \mid p\ x \qquad\ = Leaf\ (f\ x) \\ \quad \mid \mathsf{otherwise} = Fork\ (\mathsf{unfoldTree}\ p\ f\ g\ (\mathsf{fst}\ (g\ x)))\ (\mathsf{unfoldTree}\ p\ f\ g\ (\mathsf{snd}\ (g\ x)))\end{array}$$

Using the definition from Section 3 we can write mkTree as an instance of unfoldTree:

$$\begin{array}{l}\mathsf{mkTree} = \mathsf{unfoldTree}\ ((==1) \cdot \mathsf{length})\ (\mathsf{head})\ g \\ \qquad\qquad \text{where } g\ xs = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs\end{array}$$

We saw earlier that the function mapTree written as a tree fold but it is also an instance of a tree unfold:

$$\begin{array}{l}\mathsf{mapTree}\ h = \mathsf{unfoldTree}\ p\ f\ g \\ \qquad \text{where } p\ (Leaf\ a) = True \\ \qquad\qquad\ \ p\ (Fork\ xt\ yt) = False \\ \qquad\qquad\ \ f\ (Leaf\ a) = h\ a \\ \qquad\qquad\ \ g\ (Fork\ xt\ yt) = (xt, yt)\end{array}$$

As with lists we have a fusion theorem for unfoldTree.

Property 12 (Binary Tree Unfold Fusion)**.** For a function h,

$$(\mathsf{unfoldTree}\ p\ f\ g) \cdot h = \mathsf{unfoldTree}\ p'\ f'g'$$

if $p \cdot h = p'$, $f \cdot h = f'$, $h \cdot \mathsf{fst} \cdot g' = g \cdot \mathsf{fst} \cdot h$ and $h \cdot \mathsf{snd} \cdot g' = g \cdot \mathsf{snd} \cdot h$

Proof. We will prove

$$\mathsf{unfoldTree}\ p\ f\ g\ (h\ x) = \mathsf{unfoldTree}\ p'\ f'\ g'\ x \tag{17}$$

by induction on x.

Case 1 Suppose that $p'\ x = True$ and so by one of our assumptions $p\ (h\ x) = True$.

$$\begin{aligned}
& \mathsf{unfoldTree}\ p'\ f'\ g'\ x \\
= \quad & \{\text{definition of } \mathsf{foldTree} \text{ with } p'\ x = True\} \\
& Leaf\ (f'\ x) \\
= \quad & \{\text{assumption: } f'\ x = f\ (h\ x)\} \\
& Leaf\ (f\ (h\ x)) \\
= \quad & \{\text{definition of } \mathsf{unfoldTree} \text{ with } p\ (h\ x) = True\} \\
& \mathsf{unfoldTree}\ p\ f\ g\ (h\ x)
\end{aligned}$$

Case 2 Suppose that $p'\ x = False$ and let $(y, z) = g'\ x$. For the induction hypothesis, we suppose that y and z satisfy Equation (17).

$$\begin{aligned}
& \mathsf{unfoldTree}\ p'\ f'\ g'\ x \\
= \quad & \{\text{definition of } \mathsf{foldTree} \text{ with } p'\ x = False\} \\
& Fork\ (\mathsf{unfoldTree}\ p'\ f'\ g'\ (\mathsf{fst}\ (g'\ x)))\ (\mathsf{unfoldTree}\ p'\ f'\ g'\ (\mathsf{snd}\ (g'\ x))) \\
= \quad & \{\text{induction hypothesis}\} \\
& Fork\ (\mathsf{unfoldTree}\ p\ f\ g\ h\,(\mathsf{fst}\ (g'\ x)))\ (\mathsf{unfoldTree}\ p\ f\ g\ h\,(\mathsf{snd}\ (g'\ x))) \\
= \quad & \{\text{assumptions: } h \cdot \mathsf{fst} \cdot g' = g \cdot \mathsf{fst} \cdot h \text{ and } h \cdot \mathsf{snd} \cdot g' = g \cdot \mathsf{snd} \cdot h\} \\
& Fork\ (\mathsf{unfoldTree}\ p\ f\ g\ g\,(\mathsf{fst}\ (h\ x)))\ (\mathsf{unfoldTree}\ p\ f\ g\ g\,(\mathsf{snd}\ (h\ x))) \\
= \quad & \{\text{definition of } \mathsf{foldTree} \text{ with } p\ (h\ x) = False\} \\
& \mathsf{unfoldTree}\ p\ f\ g\ (h\ x)
\end{aligned}$$

□

6.4 Proving correctness using folds

The abstraction and conversion functions for the ternary tree conversion (given in Section 4) produce and consume binary trees recursively and so they can be written using binary

tree folds and unfolds. The conversion function $\mathsf{to3}$ can be written as a binary tree fold:

$$\begin{aligned}\mathsf{to3} &= \mathsf{foldTree}\ (Leaf_3)\ g \\ &\quad \text{where } g\ (Leaf_3\ a)\ xt = Fork_3\ (Leaf_3\ a)\ xt\ pt \\ &\qquad\quad g\ xt\ (Leaf_3\ a) = Fork_3\ xt\ qt\ (Leaf_3\ a) \\ &\qquad\quad g\ xt\ yt\ \ \ \ \ \ \ \ = Fork_3\ yt\ rt\ xt\end{aligned}$$

and the abstraction function $\mathsf{to2}$ can be written as a binary tree unfold:

$$\begin{aligned}\mathsf{to2} &= \mathsf{unfoldTree}\ p\ f\ g \\ &\quad \text{where } p\ (Leaf_3\ a) &&= True \\ &\qquad\quad p\ xt &&= False \\ &\qquad\quad f\ (Leaf_3\ a) &&= a \\ &\qquad\quad g\ (Fork_3\ (Leaf_3\ a)\ yt\ zt) &&= (Leaf_3\ a, yt) \\ &\qquad\quad g\ (Fork_3\ xt\ yt\ (Leaf_3\ a)) &&= (xt, Leaf_3\ a) \\ &\qquad\quad g\ (Fork_3\ xt\ yt\ zt) &&= (zt, xt)\end{aligned}$$

Writing the conversion and abstraction functions using folds and unfolds means that we can use fusion theorems for proofs. For example, to prove the ternary tree operation op_3 is correct with respect to the binary tree operation op_2 then using Equation (4):

$$op_2 \cdot \mathsf{to2} = \mathsf{to2} \cdot op_3$$

and since $\mathsf{to2} \cdot \mathsf{to3} = id$, we post compose by $\mathsf{to2}$ to obtain:

$$op_2 = \mathsf{to2} \cdot op_3 \cdot \mathsf{to3}$$

So this means we will have an equation of the form:

$$op_2 = \mathsf{unfoldTree} \cdot op_3 \cdot \mathsf{foldTree}$$

Thus we can use unfold fusion followed by fold fusion (or vice versa) to prove the equation.

Derivations are usually harder to do with folds and unfolds because functions are in the "wrong place" for fusion. For example, consider the derivation of $\mathsf{flatten}_3$ from Section 4.1:

$$\mathsf{flatten}_3\ t_3 = \mathsf{flatten}\ (\mathsf{to2}\ t_3)$$

The operation $\mathsf{flatten}$ can be written as a tree fold and $\mathsf{to2}$ is a tree unfold. Thus we have an expression of the form

$$\mathsf{foldTree} \cdot \mathsf{unfoldTree}$$

We cannot use either of the fusion rules to simplify this expression. Instead, as with lists, we can give an expression for a tree hylomorphism.

Property 13 (Tree Hylomorphism)**.** For a binary trees:

$$\mathsf{hyloTree} = (\mathsf{foldTree}\ r\ s) \cdot (\mathsf{unfoldTree}\ p\ f\ g)$$

Proof. Consider $\mathsf{hyloTree}\ x$ for some x. For the first case, suppose that $p\ x$ is $True$.

$$
\begin{array}{ll}
 & \mathsf{hyloTree}\ x \\
= & \quad \{\text{definition of } \mathsf{hyloTree}\} \\
 & \mathsf{foldTree}\ r\ s\ (\mathsf{unfoldTree}\ p\ f\ g\ x) \\
= & \quad \{\text{definition of } \mathsf{unfoldTree} \text{ with } p\ x = True\} \\
 & \mathsf{foldTree}\ r\ s\ (Leaf\ (f\ x)) \\
= & \quad \{\text{definition of } \mathsf{foldTree}\} \\
 & r\ (f\ x)
\end{array}
$$

Now suppose that $p\ x$ does not hold and let $(lt, rt) = g\ x$

$$
\begin{array}{ll}
 & \mathsf{hyloTree}\ x \\
= & \quad \{\text{definition of } \mathsf{hyloTree}\} \\
 & \mathsf{foldTree}\ r\ s\ (\mathsf{unfoldTree}\ p\ f\ g\ x) \\
= & \quad \{\text{definition of } \mathsf{unfoldTree} \text{ with } p\ x = False\} \\
 & \mathsf{foldTree}\ r\ s\ (Fork\ (\mathsf{unfoldTree}\ p\ f\ g\ lt)\ (\mathsf{unfoldTree}\ p\ f\ g\ rt)) \\
= & \quad \{\text{definition of } \mathsf{foldTree}\} \\
 & s\ (\mathsf{foldTree}\ r\ s\ (\mathsf{unfoldTree}\ p\ f\ g\ lt))\ (\mathsf{foldTree}\ r\ s\ (\mathsf{unfoldTree}\ p\ f\ g\ rt)) \\
= & \quad \{\text{definition of } \mathsf{hyloTree}\} \\
 & s\ (\mathsf{hyloTree}\ lt)\ (\mathsf{hyloTree}\ rt)
\end{array}
$$

Thus,

$$
\begin{array}{l}
\mathsf{hyloTree}\ x \\
\quad \left| \begin{array}{ll}
p\ x & = r\ (f\ x) \\
\text{otherwise} & = s\ (\mathsf{hyloTree}\ (\mathsf{fst}\ (g\ x)))\ (\mathsf{hyloTree}\ (\mathsf{snd}\ (g\ x)))
\end{array} \right.
\end{array}
$$

□

As an example of a hylomorphism we can show that

$$\mathsf{flatten} \cdot \mathsf{mkTree} = id$$

From the fold definition for flatten and the unfold definition for mkTree, we have that

$$
\begin{array}{ll}
r & = (\lambda\ x.\ [x]) \\
s & = (+\!\!+) \\
p & = (== 1) \cdot \mathsf{length} \\
f & = \mathsf{head} \\
g\ xs & = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs
\end{array}
$$

So let us consider $\mathsf{hyloTree}\ xs$ for some non-empty list xs. We will use $\mathsf{hyloTree}$ by induction on the length of xs. Suppose that $p\ xs = True$ and so this means that $xs = [a]$ for some a. Then $r\ (f\ xs) = (\lambda\ x.[x])\ (\mathsf{head}\ [a]) = [a]$ and so $r \cdot f = id$.

Now suppose that $p\ xs = False$. We will use the following property of $\mathsf{splitAt}$:

$$(ys, zs) = \mathsf{splitAt}\ n\ xs \Rightarrow xs = ys +\!\!+ zs$$

Let us suppose that

$$(ys, zs) = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs$$

So

$$\begin{array}{ll} & \mathsf{hyloTree}\ xs \\ = & \quad \{\text{definition of } \mathsf{hyloTree}\} \\ & (+\!\!+)\ (\mathsf{hyloTree}\ (\mathsf{fst}\ (g\ x)))\ (\mathsf{hyloTree}\ (\mathsf{snd}\ (g\ x))) \\ = & \quad \{\text{definition of } g\} \\ & (\mathsf{hyloTree}\ ys) +\!\!+ (\mathsf{hyloTree}\ zs) \\ = & \quad \{\text{induction hypothesis}\} \\ & ys +\!\!+ zs \\ = & \quad \{\text{property of } \mathsf{splitAt}\} \\ & xs \end{array}$$

6.5 Obfuscation and Folds

In the previous section we have seen various laws of folds and unfolds and how these laws may be used in proofs using folds or unfolds. We used folds for the definition of our rose tree operations in Section 5.1.1 but we did not use them to define operations for ternary trees. Why was this? There are two main reasons.

Firstly, folds and unfolds satisfy a "universal property" which means that the functions $\mathsf{fold}\ f\ e$ and $\mathsf{unfold}\ p\ f\ g$ have unique (strict) solutions. For example (taken from [11]), for strict h,

$$\begin{array}{ll} & h = \mathsf{fold}\ f\ e \\ \equiv & \\ & h\ [\,] = e\ \wedge\ h\ (a : xs) = f\ a\ (h\ xs) \end{array}$$

Thus when defining an operation using folds or unfolds we essentially have one way to construct the operation — in the case of fold we have some flexibility in how we define the function argument f. This is not ideal for obfuscation as we would like to have some flexibility when creating our obfuscated operations. For instance, for our definition for mkTree_3 in Section 4.2 we deliberately added in bogus cases which would be harder to do with operations using folds.

Secondly, as we saw in Section 6.4, proving a property using folds and unfolds requires the use of many different fold and unfold laws (such as fusion). The proofs that we constructed for the ternary operations in Section 4.1 mainly relied on the definitions of our operations and not with a set of properties that the definitions satisfies and so, generally it was found that the correctness proofs in Section 4.1 were easier to construct. However using folds and unfolds mean that the proofs could be automated by using a series of fusion laws (see for example [5]).

7 Conclusions

In this contribution we have expanded on previous work [8] to present a more extensive study of how to create obfuscations for a binary tree data-type. We have presented a number of transformations and discussed their suitability as obfuscations. In Section 3.2 we saw that general tree transformations such as rotations and reflections are not suitable for obfuscations as the operation definitions using these transformations are similar to the original definitions and so are not very well obfuscated. We also saw that using ternary trees and rose trees enabled us to produce obfuscations for binary trees. In Sections 4.1 and 4.2 we exploited the properties of ternary trees and the pattern matching of Haskell to add bogus cases in our definitions. Our ternary tree conversion allowed us to add in junk elements meaning that we have some flexibility when defining our obfuscated operations. However, for our rose trees conversion, we have less flexibility when defining operations. In general when writing obfuscations for trees (and other data-types) we should use a variety of styles and formats, including folds, to enable us to create different obfuscations.

In Section 1 we mentioned that one possible definition for obfuscation involves constructing proofs for assertions. This definition says that proofs of assertions for an obfuscated operation should be harder to construct (for some measure of construction) than for an unobfuscated operation. We proved an assertion (relating mapTree and flatten) for binary, ternary and rose trees. We saw the proofs for the obfuscated versions were more complicated. Further work is needed to fully compare the obfuscations according to the assertion definition of obfuscation. An area for future work is to explore what measures we could use for comparing the proof complexities — for example, we could use theorem provers such as HOL [12]. HOL allows the user to mechanically prove theorems by using a set of results that are already known, a *theory*, and a sequence of rules, the *tactics*, to solve a goal. For complexity measures we could compute the size of the theory or the number of tactics needed for a particular proof.

References

[1] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.

[2] Phillipe Biondi and Fabrice Desclaux. Silver needle in the Skype. Presentation at BlackHat Europe, March 2006. Available from URL: www.blackhat.com/html/bh-media-archives/bh-archives-2006.html.

[3] Richard Bird. *Introduction to Functional Programming in Haskell*. International Series in Computer Science. Prentice Hall, 1998.

[4] Christian Collberg, Clark D. Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[5] Oege de Moor and Ganesh Sittampalam. Generic program transformation. In *Third International Summer School on Advanced Functional Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[6] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[7] Stephen Drape. *Obfuscation of Abstract Data-Types*. DPhil thesis, Oxford University Computing Laboratory, 2004.

[8] Stephen Drape. An obfuscation for binary trees. In *To appear in the Proceedings of the IEEE Region 10 conference (TENCON 2006)*, Nov 2006.

[9] Stephen Drape. Generalising the array split obfuscation. *Information Sciences*, 177(1):202–219, January 2007.

[10] Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *Fun of Programming*, Cornerstones of Computing, pages 41–60. Palgrave, 2003.

[11] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 273–279. ACM Press, 1998.

[12] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[13] Erik. Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Verlag, 1991.

[14] Simon Peyton Jones. The Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), January 2003.

[15] Nuno Santos, Pedro Pereira, and Luís Moura e Silva. A Generic DRM Framework for J2ME Applications. In Olli Pitkänen, editor, *First International Mobile IPR Workshop: Rights Management of Information (MobileIPR)*, pages 53–66. Helsinki Institute for Information Tecnhology, August 2003.