Programming Research Group

USING HASKELL TO MODEL TREE OBFUSCATIONS

Stephen Drape

PRG-RR-04-17

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

An obfuscation is a program transformation whose aim is to make a program "harder to understand" so that reverse engineering of that program becomes difficult. Obfuscation is applied mainly to object-oriented programs.

This paper presents a fresh approach to obfuscation by considering obfuscation of objects, whose methods are modelled as functional programs. In that way we are able to obfuscate the objects knowing that it can be accessed only through the methods declared. We concentrate on an object of binary trees and find that our approach enables us to define obfuscations containing randomness designed to defeat an adversary hoping to reverse engineer the result.

To date obfuscation has been an area largely untouched by the formal method approach to program correctness. Formal methods allow us to establish a framework that provides support for the obfuscation of tree objects which exploit properties of trees. Establishing the correctness of imperative obfuscations can be a challenging task but our approach enables this to be achieved easily for *all* our obfuscations.

1 Introduction

An *obfuscation* is a program transformation with the aim to make a program "harder to understand" whilst preserving behaviour. Its purpose is to decrease the opportunities for a user to reverse engineer a commercially supplied program [2][6]. Currently obfuscation is mainly applied to programs written in object-oriented languages such as Java and C#. Two concerns about an obfuscation are whether it preserves behaviour and the degree to which it maintains efficiency.

To obfuscate a program you can either obfuscate the algorithm or obfuscate its data structures. In this paper, we concentrate on the latter and consider an object[1] for finite binary trees. The current view [6] of obfuscation concentrates on concrete data structures such as variables and arrays. With that view, we can represent trees imperatively using arrays (an example conversion is given in [7]). We could then use standard array obfuscations [6] to obfuscate our operations. Instead we will consider abstract data-types — a local state accessible by only declared operations. Why do we go to the effort of using data-types rather than using arrays directly? Apart from making proofs of correctness easy, using data-types gives us an extra level in which to add obfuscations. Going immediately to arrays forces us to think in array terms and we would have only array obfuscations at our disposal. In particular, since we will view a tree as an abstract data-type then the usual tree transformations (such as swapping two subtrees) are naturally available; they would be more difficult to conceive using arrays. As a result we have two opportunities for obfuscation; the first using our new data-types approach and the second using the standard imperative methods. The approach we use is built on the frameworks developed for sets [11] and matrices [10] in which we used a technique called *splitting*. However, we will see that splitting does not give a suitable obfuscation for trees and thus we develop a different technique. With the obfuscations that we propose we have a degree

[1] which we view (for the purpose of refinement) as a *data-type*, calling the methods *operations*

of flexibility which we exploit so that we can create *random obfuscations* which help to confuse attackers further.

We model our operations using the functional language Haskell to provide a framework for specifying data-types and obfuscations (note that we are not aiming to obfuscate Haskell code but to use Haskell as a modelling language). When creating obfuscations, we should ensure that they are correct (*i.e.* behaviour preserving) but in an imperative context, proofs of correctness are frequently hard, typically requiring language restrictions. This means that obfuscations are mostly stated without giving a proof of correctness. However, the use of Haskell allows us to reason about programs mathematically. We use data refinement techniques [9] which enables us to prove the correctness of *all* our obfuscations (and in some cases, *derive* obfuscations) easily.

Often, obfuscation is seen as applicable only to object-oriented languages (or the underlying bytecode) but the use of a more mathematical approach (by using standard refinements and derivation techniques) allows us to apply obfuscation to more general areas. Since we use Haskell for our (apparently new) approach we have the benefits of the elegance of the functional style and the consequent abstraction of side-effects. Thus our functional approach will provide support for purely imperative obfuscations.

Barak *et al.* propose a very formal definition of obfuscation and then show that obfuscation is impossible [2]. We use an informal notion of obfuscation — namely that a function is more obfuscated if it is "harder to understand". With abstract data-types, we may interpret "harder to understand" as, for instance, using an obscure data structure or having more complicated clauses in the definition of a function.

The rest of the paper is structured as follows. Section 2 introduces our data-type for binary trees and discusses how we can obfuscate binary trees. Section 3 develops some obfuscated operations. Section 4 discusses some changes that we can make to our representation and to our proofs. Some conclusions and areas for further research are given in Section 5.

2 Tree Operations

In this section, we state our binary tree data-type — many of the definitions of the tree operations are taken from [4]. The kind of binary tree that we consider contain a value at each node. Thus, the data-type for binary trees whose values are of type α is defined as follows:

$$\mathit{Tree}\ \alpha ::= \mathit{Null} \mid \mathit{Fork}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)$$

with operations:

$$\begin{aligned}
\mathsf{mktree} &:: \mathit{List}\ \alpha \to \mathit{Tree}\ \alpha \\
\mathsf{flatten} &:: \mathit{Tree}\ \alpha \to \mathit{List}\ \alpha \\
\mathsf{member} &:: \alpha \to \mathit{Tree}\ \alpha \to \mathbb{B} \\
\mathsf{height} &:: \mathit{Tree}\ \alpha \to \mathbb{N} \\
\mathsf{modify} &:: \alpha \to \alpha \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha
\end{aligned}$$

We assume that we deal with finite trees and all the values in the tree are well-defined (*i.e.* we do not allow $\perp$).

These operations satisfy:

$$\begin{array}{rcl} \mathsf{flatten} \cdot \mathsf{mktree} & = & id \\ \mathsf{member}\ p\ (\mathsf{mktree}\ xs) & = & \mathsf{elem}\ p\ xs \\ \mathsf{member}\ p\ t & \Rightarrow & \mathsf{member}\ q\ (\mathsf{modify}\ p\ q\ t) \end{array}$$

The flatten operation takes a tree and returns a list of values.

$$\begin{array}{l} \mathsf{flatten}\ Null \qquad\qquad = [\,] \\ \mathsf{flatten}\ (Fork\ lt\ v\ rt) = \mathsf{flatten}\ lt \mathbin{+\!\!+} [v] \mathbin{+\!\!+} \mathsf{flatten}\ rt \end{array}$$

The height operation measures how far away the furthest leaf is.

$$\begin{array}{l} \mathsf{height}\ Null \qquad\qquad = 0 \\ \mathsf{height}\ (Fork\ lt\ v\ rt) = 1 + (\mathsf{max}\ (\mathsf{height}\ lt)\ (\mathsf{height}\ rt)) \end{array}$$

For mktree, we have many ways of making a tree from a list. However, we impose the following requirement on such an operation:

$$\begin{array}{l} \mathsf{mktree} :: List\ \alpha \rightarrow Tree\ \alpha \bullet xt = \mathsf{mktree}\ ls \ \equiv \\ (\mathsf{flatten}\ xt = ls\ \wedge (\forall\ yt)\ (\mathsf{flatten}\ yt\ = ls \Rightarrow \mathsf{height}\ yt \geq \mathsf{height}\ xt)) \end{array}$$

We propose the following definition for mktree that builds a binary tree of minimum height:

$$\begin{array}{l} \mathsf{mktree}\ [\,]\ = Null \\ \mathsf{mktree}\ xs = Fork\ (\mathsf{mktree}\ ys)\ z\ (\mathsf{mktree}\ zs) \\ \quad \mathbf{where}\ (ys, (z : zs))\ =\ \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs \end{array}$$

and we can easily verify that the mktree requirement is satisfied.

We can check whether a particular value is contained within a tree using member.

$$\begin{array}{l} \mathsf{member}\ p\ Null \qquad\qquad = False \\ \mathsf{member}\ p\ (Fork\ lt\ v\ rt) = (v == p) \vee \mathsf{member}\ p\ lt \vee\ \mathsf{member}\ p\ rt \end{array}$$

Finally, modify $p\ q\ t$ should replace all occurrences of the value p in the tree with the value q.

$$\begin{array}{l} \mathsf{modify}\ p\ q\ Null = Null \\ \mathsf{modify}\ p\ q\ (Fork\ lt\ v\ rt) \\ \quad \mid\ p == v \quad\ = Fork\ (\mathsf{modify}\ p\ q\ lt)\ q\ (\mathsf{modify}\ p\ q\ rt) \\ \quad \mid\ \mathsf{otherwise} = Fork\ (\mathsf{modify}\ p\ q\ lt)\ v\ (\mathsf{modify}\ p\ q\ rt) \end{array}$$

2.1 Binary Search Trees

We now consider a special kind of binary tree — *binary search trees*. Binary search trees have the property that the value, of type α, of a node is greater than the values in the left subtree but less than the values in the right subtree (this assumes we have an total ordering on α). This property can be stated as:

$$\mathsf{inc}\ (\mathsf{flatten}\ t) \tag{1}$$

where inc is a Boolean function that tests whether a list is in strictly increasing order.

We define the data-type for binary search trees as follows: a binary tree t has type

$$(Ord\ \alpha) \Rightarrow BST\ \alpha ::= Null \mid Fork\ (BST\ \alpha)\ \alpha\ (BST\ \alpha)$$

satisfying the invariant (1) with the following operations:

$$\begin{array}{rcl}
\mathsf{mkBST} & :: & List\ \alpha \to BST\ \alpha \\
\mathsf{flatten} & :: & BST\ \alpha \to List\ \alpha \\
\mathsf{member} & :: & \alpha \to BST\ \alpha \to \mathbb{B} \\
\mathsf{height} & :: & BST\ \alpha \to \mathbb{N} \\
\mathsf{insert} & :: & \alpha \to BST\,\alpha \to BST\,\alpha \\
\mathsf{delete} & :: & \alpha \to BST\,\alpha \to BST\,\alpha
\end{array}$$

The last two operations satisfy

$$\begin{array}{rcl}
\mathsf{member}\ p\ (\mathsf{insert}\ p\ t) & = & True \\
\mathsf{member}\ p\ (\mathsf{delete}\ p\ t) & = & False
\end{array}$$

The definitions of flatten and height are the same as before — in fact, we could consider "importing" these operations from the binary tree data-type (with the appropriate change of signature).

For BSTs, member is implemented efficiently as follows:

$$\begin{array}{l}
\mathsf{member}\ p\ Null\ =\ False \\
\mathsf{member}\ p\ (Fork\ lt\ v\ rt) \\
\quad \left|\begin{array}{lcl}
p == v & = & True \\
p < v & = & \mathsf{member}\ p\ lt \\
p > v & = & \mathsf{member}\ p\ rt
\end{array}\right.
\end{array}$$

We cannot use the operation of mktree from before as we need to ensure that we create a BST. We could define

$$\mathsf{mkbst} = \mathsf{foldr}\ \mathsf{insert}\ Null$$

but this function does not necessarily build a BST of minimal height. Instead we define

$$\mathsf{mkbst} = \mathsf{mktree}.\mathsf{strict\_sort}$$

where the function $\mathsf{strict\_sort}$ sorts a list so that it is strictly-increasing (*i.e.* there are no duplicates). But does this operation actually build a binary tree? By definition, mktree splits a list in the form:

$$[\text{left subtree values}] +\!\!+ (\mathsf{value}) : [\mathsf{right\ subtree\ values}]$$

If the list is strictly-increasing then so is each sublist and thus invariant (1) is maintained.

We can define insert quite easily as follows

$$\begin{array}{l}
\mathsf{insert}\ x\ \mathit{Null} = \mathit{Fork}\ \mathit{Null}\ x\ \mathit{Null} \\
\mathsf{insert}\ x\ (\mathit{Fork}\ \mathit{lt}\ y\ \mathit{rt}) \\
\quad \left| \begin{array}{ll}
x < y & = \mathit{Fork}\ (\mathsf{insert}\ x\ \mathit{lt})\ y\ \mathit{rt} \\
x == y & = \mathit{Fork}\ \mathit{lt}\ y\ \mathit{rt} \\
x > y & = \mathit{Fork}\ \mathit{lt}\ y\ (\mathsf{insert}\ x\ \mathit{rt})
\end{array} \right.
\end{array}$$

However, the definition of delete is more complicated

$$\begin{array}{l}
\mathsf{delete}\ x\ \mathit{Null}\ =\ \mathit{Null} \\
\mathsf{delete}\ x\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) \\
\quad \left| \begin{array}{ll}
x < v & = \mathit{Fork}\ (\mathsf{delete}\ x\ \mathit{lt})\ v\ \mathit{rt} \\
x == v & = \mathsf{join}\ \mathit{lt}\ \mathit{rt} \\
x > v & = \mathit{Fork}\ \mathit{lt}\ v\ (\mathsf{delete}\ x\ \mathit{rt})
\end{array} \right.
\end{array}$$

where the function join satisfies the equation

$$\mathsf{flatten}\ (\mathsf{join}\ \mathit{xt}\ \mathit{yt}) = \mathsf{flatten}\ \mathit{xt} +\!\!+ \mathsf{flatten}\ \mathit{yt}$$

and has the effect of joining two trees together. We need to ensure that the resulting tree is of minimum height and so we define

$$\begin{array}{l}
\mathsf{join}\ \mathit{xt}\ \mathit{yt} \\
\quad \left| \begin{array}{ll}
\mathit{yt} == \mathit{Null} & = \mathit{xt} \\
\text{otherwise} & = \mathit{Fork}\ \mathit{xt}\ (\mathsf{headTree}\ \mathit{yt})\ (\mathsf{tailTree}\ \mathit{yt})
\end{array} \right.
\end{array}$$

The functions $\mathsf{headTree}$ and $\mathsf{tailTree}$ satisfy

$$\mathsf{headTree} = \mathsf{head} \cdot \mathsf{flatten} \tag{2}$$

$$\mathsf{flatten} \cdot \mathsf{tailTree} = \mathsf{tail} \cdot \mathsf{flatten} \tag{3}$$

and we define

$$\begin{array}{l}
\mathsf{headTree}\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) \\
\quad \left| \begin{array}{ll}
\mathit{lt} == \mathit{Null} & = v \\
\mathsf{otherwise} & = \mathsf{headTree}\ \mathit{lt}
\end{array} \right.
\end{array}$$

$$\begin{array}{l}
\mathsf{tailTree}\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) \\
\quad \left| \begin{array}{ll}
\mathit{lt} == \mathit{Null} & = \mathit{rt} \\
\mathsf{otherwise} & = \mathit{Fork}\ (\mathsf{tailTree}\ \mathit{lt})\ v\ \mathit{rt}
\end{array} \right.
\end{array}$$

Note that we have not included modify in our BST data-type. This is because for BSTs, modify is a "derived" operation as it can be written in terms of insert and delete:

$$\begin{array}{ll} \mathsf{modify}\ p\ q\ t = & \text{if}\ \mathsf{member}\ p\ t \\ & \text{then}\ \mathsf{insert}\ q\ (\mathsf{delete}\ p\ t) \\ & \text{else}\ t \end{array}$$

The membership test ensures that we do not add q to our tree when p does not occur in the tree.

2.2 Obfuscating Trees

Now that we have defined our binary tree data-type, we need to consider how we can obfuscate it. We need to view the obfuscation in the context of data-types, not just obfuscating some operations. The previous work of obfuscating data-types [10][11] considered a technique known as *splitting*.

An element e of type T can be represented by a tuple of elements of type T

$$e \leadsto \langle e_1, e_2, \ldots, e_n \rangle$$

and this tuple is called the *split* of e (where $\leadsto$ denotes data-refinement). The information contained in e is "split across" each of the e_is. The obfuscation was developed from work on array obfuscations [6][12].

We could consider splitting a binary tree at the root node and making the right subtree one component and the rest of the tree the other. Thus we have

$$\begin{array}{rcl} \mathit{Null} & \leadsto & \langle \mathit{Null}, \mathit{Null} \rangle \\ \mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt} & \leadsto & \langle \mathit{Fork}\ \mathit{lt}\ v\ \mathit{Null}, \mathit{rt} \rangle \end{array}$$

Using this representation, we could rewrite $\mathsf{flatten}$

$$\begin{array}{l} \mathsf{flatten}_{sp}\ \langle \mathit{Null}, \mathit{Null} \rangle = [\,] \\ \mathsf{flatten}_{sp}\ \langle \mathit{Fork}\ \mathit{lt}\ v\ \mathit{Null}, \mathit{rt} \rangle = \mathsf{flatten}\ \mathit{lt} + [v] + \mathsf{flatten}\ \mathit{rt} \end{array}$$

where the subscript $_{sp}$ denotes an operation for split trees. We can write operations for split binary search trees as well

$$\begin{array}{l} \mathsf{insert}_{sp}\ x\ \langle \mathit{Null}, \mathit{Null} \rangle = \langle \mathit{Fork}\ \mathit{Null}\ x\ \mathit{Null}, \mathit{Null} \rangle \\ \mathsf{insert}_{sp}\ x\ \langle \mathit{Fork}\ \mathit{lt}\ y\ \mathit{Null}, \mathit{rt} \rangle \\ \quad \left| \begin{array}{ll} x < y & = \langle \mathit{Fork}\ (\mathsf{insert}\ x\ \mathit{lt})\ y\ \mathit{Null}, \mathit{rt} \rangle \\ x == y & = \langle \mathit{Fork}\ \mathit{lt}\ y\ \mathit{Null}, \mathit{rt} \rangle \\ x > y & = \langle \mathit{Fork}\ \mathit{lt}\ y\ \mathit{Null}, \mathsf{insert}\ x\ \mathit{rt} \rangle \end{array} \right. \end{array}$$

There are two problems with this obfuscation. The first is that the definitions using split trees require the use of the functions for normal trees. Secondly, the definitions above are very similar to the original definitions and therefore not very well obfuscated.

Instead of splitting trees, we could consider changing the structure of the tree. For instance, we could flatten a tree to a list. However, to preserve correctness, we need to be able to recover the binary tree from the list. Since many trees flatten to the same list, this recovery will not be possible without introducing extra information about the shape of the tree.

We would like an obfuscation for binary trees that changes the structure whilst allowing us to recover both the information and the structure if we so wish, with minimum overhead. Tree transformations such as rotations and reflections are suitable obfuscations, but for this paper, we will consider converting a binary tree into a ternary tree. This conversion is chosen since it gives us the flexibility to add extra information as well as performing some tree transformations.

2.3 Ternary Trees

For the rest of the paper, we consider converting binary trees into ternary trees. Our data-type for ternary trees is as follows:

$$\mathit{Tree}_3\ \alpha ::= \mathit{Null}_3 \mid \mathit{Fork}_3\ \alpha\ (\mathit{Tree}_3\ \alpha)\ (\mathit{Tree}_3\ \alpha)\ (\mathit{Tree}_3\ \alpha)$$

with the operations mktree_3, $\mathsf{flatten}_3$, member_3, height_3 and modify_3 (with the appropriate signatures). If we want to represent search trees then we have the extra operations insert_3 and delete_3.

We convert a binary tree to a ternary tree by adding an extra subtree at every node of the binary tree. We consider this conversion to be a data-representation. Since we wish to preserve program correctness, if a binary tree t_2 is represented by a ternary tree t_3, then from [14], we have an abstraction function af and predicate dti — the data-type invariant — such that

$$t_2 = \mathsf{af}(t_3) \wedge \mathsf{dti}(t_3)$$

In Section 3, we define a function

$$\mathsf{to2} :: \mathit{Tree}_3\ \alpha \rightarrow \mathit{Tree}\ \alpha$$

which serves as an abstraction function. The data-type invariant for binary trees is *True* and for search trees is $\mathsf{inc}\ (\mathsf{flatten}_3\ t_3)$.

2.4 Proving Correctness

Each of the operations for the ternary trees stated in the last section corresponds to an operation for binary trees. As we are using ternary trees to represent binary trees, we must ensure that each of the ternary tree operations "behaves correctly". Suppose that we want to refine a binary tree operation op, which returns a binary tree as the result, with respect to an abstraction function af. We can construct the following commuting diagram [9]:

$$\begin{array}{ccc} t_2 & \xrightarrow{\mathsf{op}} & \mathsf{op}\ (t_2) \\ \uparrow{\scriptstyle \mathsf{af}} & & \uparrow{\scriptstyle \mathsf{af}} \\ t_3 & \xrightarrow{\mathsf{op}_3} & \mathsf{op}_3(t_3) \end{array}$$

This diagram gives the following equation

$$\mathsf{op} \cdot \mathsf{af} = \mathsf{af} \cdot \mathsf{op}_3 \tag{4}$$

Note that since that we have total, deterministic operations then we have equality. However general correctness follows if $=$ is replaced by $\sqsubseteq$ (refinement). Equation (4) provides a way of ensuring that if $t_2 \rightsquigarrow t_3$ then the result of $\mathsf{op}(t_2)$ is the same as finding the binary tree that is represented by $\mathsf{op}_3(t_3)$.

Note that, if we have an operation that does not return a binary tree but instead (say) a list or an integer then in the commuting diagram, we can replace the right-hand af by identity.

From a practical point of view, an obfuscator needs to keep the abstraction function secret as the function gives an attacker an indication of how to construct the unobfuscated operations.

3 Particular Representation

In this section, we give a particular ternary tree representation of binary trees which will be used throughout the rest of the paper. For demonstration purposes, we assume that α is the type of integers.

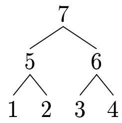
For a representation, we would like to convert the binary tree $Fork\ xt\ v\ yt$ into the ternary tree $Fork_3\ v\ lt\ ct\ rt$. The ternary trees lt, ct and rt will depend on v, xt and yt. Since a ternary tree can carry more information than a binary tree, we can add "junk" to the ternary tree which adds to the obfuscation. We then construct our abstraction function so that this junk is ignored — this gives us the opportunity to create random junk.

The conversion that we consider is as follows. The tree $Fork\ lt\ v\ rt$ is converted to

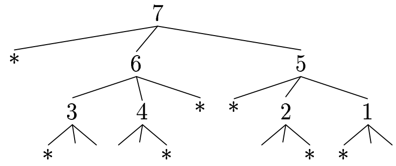
- $Fork_3\ v\ lt\ rt\ junk_1$ if v is even
- $Fork_3\ v\ junk_2\ rt\ lt$ if v is odd

and the tree $Null$ is converted to $Null_3$. This conversion assumes the odd and even numbers are uniformly distributed.

So, the binary tree, (say) et



is converted to a ternary tree of the form



where "*" stands for different arbitrary ternary trees.

For data refinement, we need to state an abstraction function that converts a ternary tree into a binary one. We can easily write the function in Haskell — for our refinement, we call this function $\mathsf{to2}$:

$$
\begin{array}{l}
\mathsf{to2}\ \mathit{Null}_3 = \mathit{Null} \\
\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) \\
\quad \left| \begin{array}{ll} \mathsf{even}\ v & = \mathit{Fork}\ (\mathsf{to2}\ \mathit{lt})\ v\ (\mathsf{to2}\ \mathit{ct}) \\ \mathrm{otherwise} & = \mathit{Fork}\ (\mathsf{to2}\ \mathit{rt})\ v\ (\mathsf{to2}\ \mathit{ct}) \end{array} \right.
\end{array}
$$

This means that for our representation, if $t_2 \rightsquigarrow t_3$ then $t_2 = \mathsf{to2}\ t_3$ with the data-type invariant *True*. We do not have a left inverse for this function as any binary tree can be represented by more than one ternary tree ($\mathsf{to2}$ is surjective but not injective). However, we can construct a right inverse, $\mathsf{to3}$, which satisfies the following equation:

$$
\mathsf{to2} \cdot \mathsf{to3} = \mathit{id} \tag{5}
$$

and so

$$
t_2 \rightsquigarrow \mathsf{to3}(t_2)
$$

We will call a function that converts binary trees to ternary trees a *conversion* function (and this is a conversion with respect to a particular refinement).

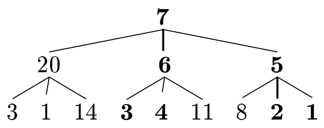
To convert a binary tree into a ternary tree, we have many choices of how to create the junk to be inserted into the tree at the appropriate places. Thus we can construct many functions that satisfy equation (5). The complexity of the binary tree operations defined in Section 2 depends on the number of nodes or on the height. Thus if we want our ternary tree operation to have a similar complexity then we should make sure that the height is similar and we do not introduce too many extra elements. As an example, we define:

$$
\begin{array}{l}
\mathsf{to3}\ \mathit{Null} = \mathit{Null}_3 \\
\mathsf{to3}\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) \\
\quad \left| \begin{array}{ll} \mathsf{even}\ v & = \mathit{Fork}_3\ v\ (\mathsf{to3}\ \mathit{lt})\ (\mathsf{to3}\ \mathit{rt})\ (\mathsf{to3'}\ \mathit{lt}) \\ \mathrm{otherwise} & = \mathit{Fork}_3\ v\ (\mathsf{to3'}\ \mathit{rt})\ (\mathsf{to3}\ \mathit{rt})\ (\mathsf{to3}\ \mathit{lt}) \end{array} \right.
\end{array}
$$

where

$$\mathsf{to3'}\ \mathit{Null} = \mathit{Null}_3$$
$$\mathsf{to3'}\ (\mathit{Fork}\ lt\ v\ rt) = \mathit{Fork}_3\ (3 * v + 2)\ (\mathsf{to3}\ lt)\ (\mathsf{to3'}\ lt)\ (\mathsf{to3'}\ rt)$$

This conversion keeps the height of the tree the same. We chose the function $\lambda v.3v + 2$ so that the odd and even numbers are still uniformly distributed and so that the junk values are not too large (and therefore do not "stand out" from the "real" values). As an example, the binary tree et above is converted to



(the numbers in **bold** denote the values from et). We can see that this ternary tree matches the general ternary form given earlier and has the same height as et. Although the height is kept the same, the number of nodes increases. Thus this obfuscation is unsuitable if the number of nodes is an important consideration and we should ensure that the conversion function restricts the number of new nodes.

In most of our examples, we state operations (and then prove correctness) rather than deriving them. We could use the function $\mathsf{to3}$ to derive obfuscated operations using the equation:

$$\mathsf{op}_3 = \mathsf{to3} \cdot \mathsf{op} \cdot \mathsf{to2} \tag{6}$$

However, the operations that we would obtain would be specialised to this particular conversion from binary to ternary since the junk information is built up in a specific way. We will see that we have a certain degree of freedom in how we deal with the junk information. Yet, we could use (6) to give us a guide in how an obfuscation might look and so aid us in giving the definition of a ternary tree operation.

3.1 Ternary Operations

Now, we need to define operations for our ternary tree representation. The definitions of height_3 and $\mathsf{flatten}_3$ are straightforward:

$$\begin{array}{l}
\mathsf{height}_3\ \mathit{Null}_3\ = 0\\
\mathsf{height}_3\ (\mathit{Fork}_3\ v\ lt\ ct\ rt)\\
\quad \left| \begin{array}{ll} \mathsf{even}\ v & = 1 + (\mathsf{max}\ (\mathsf{height}\ lt)\ (\mathsf{height}\ ct))\\ \text{otherwise} & = 1 + (\mathsf{max}\ (\mathsf{height}\ rt)\ (\mathsf{height}\ ct)) \end{array}\right.
\end{array}$$

$$\begin{array}{l}
\mathsf{flatten}_3\ \mathit{Null}_3\ = [\,]\\
\mathsf{flatten}_3\ (\mathit{Fork}_3\ v\ lt\ ct\ rt)\\
\quad \left| \begin{array}{ll} \mathsf{even}\ v & = (\mathsf{flatten}_3\ lt) +\!\!+ [v] +\!\!+ (\mathsf{flatten}_3\ ct)\\ \mathsf{otherwise} & = (\mathsf{flatten}_3\ rt) +\!\!+ [v] +\!\!+ (\mathsf{flatten}_3\ ct) \end{array}\right.
\end{array}$$

and we can easily show that our definitions satisfy

$$\mathsf{op}_3 = \mathsf{op} \cdot \mathsf{to2} \tag{7}$$

We can see that these operations have an extra test which adds an extra conditional and so makes the operation more obfuscated.

3.2 Membership

Let us consider how to define member_3 for ternary trees. The operation must satisfy equation (7) and so we can use this relationship to derive our operation using structural induction (we can use induction as we deal with finite, well-defined trees).

Base case Suppose that $t = \mathit{Null}_3$

$$
\begin{aligned}
& \mathsf{member}_3\ p\ \mathit{Null}_3 \\
= & \quad \{\text{equation (7)}\} \\
& \mathsf{member}\ p\ (\mathsf{to2}\ \mathit{Null}_3) \\
= & \quad \{\text{definition of } \mathsf{to2}\} \\
& \mathsf{member}\ p\ \mathit{Null} \\
= & \quad \{\text{definition of } \mathsf{member}\} \\
& \mathit{False}
\end{aligned}
$$

For the induction hypothesis, we suppose that

$$t = \mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}$$

and we suppose that the subtrees lt, ct and rt satisfy (7) for member. We have two subcases to consider.

Subcase 1 Suppose that v is even

$$
\begin{aligned}
& \mathsf{member}_3\ p\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) \\
= & \quad \{\text{equation (7)}\} \\
& \mathsf{member}\ p\ (\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
& \mathsf{member}\ p\ (\mathit{Fork}\ (\mathsf{to2}\ \mathit{lt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{definition of } \mathsf{member}\} \\
& v == p \vee \mathsf{member}\ p\ (\mathsf{to2}\ \mathit{lt}) \vee \\
& \qquad\qquad \mathsf{member}\ p\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\text{induction hypothesis}\} \\
& v == p \vee \mathsf{member}_3\ p\ \mathit{lt} \vee \mathsf{member}_3\ p\ \mathit{ct}
\end{aligned}
$$

Subcase 2 The case for v odd is similar to the case above and we obtain

$$v == p \vee \mathsf{member}_3\ p\ \mathit{rt} \vee \mathsf{member}_3\ p\ \mathit{ct}$$

This gives us the following definition:

$$\begin{array}{l}
\mathsf{member}_3\ p\ \mathit{Null}_3 = \mathit{False} \\
\mathsf{member}_3\ p\ (\mathit{Fork}_3\ v\ lt\ ct\ rt) = \\
\quad\quad v == p\ \lor\ (\mathsf{member}_3\ p\ ct) \lor \\
\quad\quad (\text{if } \mathsf{even}\ v \text{ then } (\mathsf{member}_3\ p\ lt) \\
\quad\quad\quad\quad\quad\quad \text{else } (\mathsf{member}_3\ p\ rt))
\end{array}$$

3.3 Making Ternary Trees

When using binary trees, we defined an operation mktree to convert a list into a binary tree. We can also define a corresponding operation mktree_3 that converts a list into a ternary tree and satisfies the relationship:

$$\mathsf{mktree} = \mathsf{to2} \cdot \mathsf{mktree}_3 \tag{8}$$

However as with the definition of a conversion function, we have many choices of mktree_3 that satisfy the above equation. The general form of a function that satisfies (8) is:

$$\begin{array}{l}
\mathsf{mktree}_3\ [\,] = \mathit{Null}_3 \\
\mathsf{mktree}_3\ xs \\
\quad \mid \mathsf{even}\ z \quad\quad\;\; = \mathit{Fork}_3\ z\ (\mathsf{mktree}_3\ ys)\ (\mathsf{mktree}_3\ zs)\ jt \\
\quad \mid \text{otherwise} = \mathit{Fork}_3\ z\ kt\ (\mathsf{mktree}_3\ zs)\ (\mathsf{mktree}_3\ ys) \\
\quad\;\; \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs
\end{array}$$

where the expressions jt and kt represent arbitrary ternary trees.

As an example, we define mktree_3 so that

$$\mathsf{mktree}_3 = \mathsf{to3} \cdot \mathsf{mktree}$$

where to3 as defined in section 3. This equation gives the following definition:

$$\begin{array}{l}
\mathsf{mktree}_3\ [\,] = \mathit{Null}_3 \\
\mathsf{mktree}_3\ xs \\
\quad \mid \mathsf{even}\ z \quad\quad\;\; = \mathit{Fork}_3\ z\ (\mathsf{mktree}_3\ ys)\ (\mathsf{mktree}_3\ zs)\ (\mathsf{mktree'}_3\ ys) \\
\quad \mid \text{otherwise} \;\; = \mathit{Fork}_3\ z\ (\mathsf{mktree'}_3\ zs)\ (\mathsf{mktree}_3\ zs)\ (\mathsf{mktree}_3\ ys) \\
\quad\;\; \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs
\end{array}$$

$$\begin{array}{l}
\mathsf{mktree'}_3\ [\,] = \mathit{Null}_3 \\
\mathsf{mktree'}_3\ xs = \mathit{Fork}_3\ (3 * z + 2)\ (\mathsf{mktree}_3\ ys)\ (\mathsf{mktree'}_3\ ys)\ (\mathsf{mktree'}_3\ zs) \\
\quad\quad \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ (\mathsf{length}\ xs)\ 2)\ xs
\end{array}$$

We can construct this function by using (6) and so this function is specialised to the choice of conversion of binary to ternary. Computing

$$\mathsf{mktree}_3\ [1, 5, 2, 7, 3, 6, 4]$$

produces the ternary tree given at the start of Section 3.

3.4 Modification

The modify operation can change the value of a node and thus the "parity" of the node (*i.e.* whether the value at the node is odd or even) may change. So when defining this operation for ternary trees, we may have to change the order of the subtrees (this reordering may also happen for other value types α).

We can define modify to operate on ternary trees as follows:

$$
\begin{array}{l}
\mathsf{modify}_3\ p\ q\ Null_3 = Null_3 \\
\mathsf{modify}_3\ p\ q\ (Fork_3\ v\ lt\ ct\ rt) \\
\quad \left| \begin{array}{ll}
v == p \wedge \mathsf{even}\ p \wedge \mathsf{even}\ q & = Fork_3\ q\ lt'\ ct'\ jt_1 \\
v == p \wedge \mathsf{odd}\ p \ \ \wedge \mathsf{odd}\ q & = Fork_3\ q\ jt_2\ ct'\ rt' \\
v == p \wedge \mathsf{even}\ p \wedge \mathsf{odd}\ q & = Fork_3\ q\ jt_3\ ct'\ lt' \\
v == p \wedge \mathsf{odd}\ p \ \ \wedge \mathsf{even}\ q & = Fork_3\ q\ rt'\ ct'\ jt_4 \\
v \neq p & = Fork_3\ v\ lt'\ ct'\ rt'
\end{array} \right. \\
\qquad \text{where } \begin{array}[t]{l}
lt' = \mathsf{modify}_3\ p\ q\ lt \\
rt' = \mathsf{modify}_3\ p\ q\ rt \\
ct' = \mathsf{modify}_3\ p\ q\ ct
\end{array}
\end{array}
$$

The four trees jt_i represent arbitrary ternary trees. Note that in the case for $v \neq p$ we do not check whether v is even. This can help disguise the fact that we need check the parity of the node in our representation.

To show that this operation is correct, we need to prove that it satisfies (4), *i.e.* that

$$\mathsf{modify}\ p\ q\ (\mathsf{to2}\ t) = \mathsf{to2}\ (\mathsf{modify}_3\ p\ q\ t)$$

and this can be proved by structural induction on t.

If we let $jt_1 = rt'$ and $jt_2 = lt'$ then we can collapse the first two conditions into one

$$|v == p \wedge (p - q)\ \text{`}\mathsf{mod}\text{`}\ 2\ ==\ 0 = Fork_3\ q\ lt'\ ct'\ rt'$$

Similarly by letting $jt_3 = rt'$ and $jt_4 = lt'$ we can collapse two more conditions. This leads us to the following optimisation of modify_3:

$$
\begin{array}{l}
\mathsf{modify}_3\ p\ q\ Null_3 = Null_3 \\
\mathsf{modify}_3\ p\ q\ (Fork_3\ v\ lt\ ct\ rt) \\
\quad \left| \begin{array}{ll}
v == p \wedge (p - q)\ \text{`}\mathsf{mod}\text{`}\ 2\ ==\ 0 & = Fork_3\ q\ lt'\ ct'\ rt' \\
v == p \wedge (p - q)\ \text{`}\mathsf{mod}\text{`}\ 2\ ==\ 1 & = Fork_3\ q\ rt'\ ct'\ lt' \\
\text{otherwise} & = Fork_3\ v\ lt'\ ct'\ rt'
\end{array} \right. \\
\qquad \text{where } \begin{array}[t]{l}
lt' = \mathsf{modify}_3\ p\ q\ lt \\
rt' = \mathsf{modify}_3\ p\ q\ rt \\
ct' = \mathsf{modify}_3\ p\ q\ ct
\end{array}
\end{array}
$$

With our representation we often have to test whether a value is even; this optimisation has the advantage that the test for even is hidden.

3.5 Operations for Binary Search Trees

Now let us consider representing of binary search trees with our specific abstraction function; for a binary search tree t_2, $t_2 \leadsto t_3$ if

$$t_2 = (\mathsf{to2}\ t_3) \wedge \mathsf{inc}\ (\mathsf{flatten}\ (\mathsf{to2}\ t_3))$$

We define operations corresponding to member and insert routinely:

$$\begin{array}{l}
\mathsf{member}_3\ p\ \mathit{Null}_3 = \mathit{False} \\
\mathsf{member}_3\ p\ (\mathit{Fork}_3\ v\ lt\ ct\ rt) \\
\quad \left| \begin{array}{lcl}
p == v & = & \mathit{True} \\
p < v \wedge \mathsf{even}\ v & = & \mathsf{member}_3\ p\ lt \\
p < v \wedge \mathsf{odd}\ v & = & \mathsf{member}_3\ p\ rt \\
p > v & = & \mathsf{member}_3\ p\ ct
\end{array} \right.
\end{array}$$

$$\begin{array}{l}
\mathsf{insert}_3\ x\ \mathit{Null}_3 = \mathit{Fork}_3\ x\ \mathit{Null}_3\ \mathit{Null}_3\ \mathit{Null}_3 \\
\mathsf{insert}_3\ x\ (\mathit{Fork}_3\ y\ lt\ ct\ rt) \\
\quad \left| \begin{array}{lcl}
x < y \wedge \mathsf{even}\ y & = & \mathit{Fork}_3\ y\ (\mathsf{insert}_3\ x\ lt)\ ct\ jt \\
x < y \wedge \mathsf{odd}\ y & = & \mathit{Fork}_3\ y\ kt\ ct\ (\mathsf{insert}_3\ x\ rt) \\
x == y & = & \mathsf{Fork}_3\ y\ lt\ ct\ rt \\
x > y & = & \mathit{Fork}_3\ y\ lt\ (\mathsf{insert}_3\ x\ ct)\ rt
\end{array} \right.
\end{array}$$

where jt and kt are arbitrary ternary trees. Note that the case $x > y$ does not require a test to see whether y is even. This is because we chose to always map the right subtree of a binary tree to the centre subtree of a ternary tree.

3.6 Deletion

To define delete for our ternary tree, we first need to define headTree, tailTree for ternary trees which satisfy analogues of (2) and (3):

$$\begin{array}{rcl}
\mathsf{headTree}_3 & = & \mathsf{head} \cdot \mathsf{flatten}_3 \\
\mathsf{flatten}_3 \cdot \mathsf{tailTree}_3 & = & \mathsf{tail} \cdot \mathsf{flatten}_3
\end{array}$$

Using these equations, we obtain

$$\begin{array}{l}
\mathsf{headTree}_3\ (\mathit{Fork}_3\ v\ lt\ ct\ rt) \\
\quad \left| \begin{array}{lcl}
\mathsf{even}\ v & = & \text{if } lt == \mathit{Null}_3 \text{ then } v \text{ else } \mathsf{headTree}_3\ lt \\
\text{otherwise} & = & \text{if } rt == \mathit{Null}_3 \text{ then } v \text{ else } \mathsf{headTree}_3\ rt
\end{array} \right.
\end{array}$$

$$\begin{array}{l}
\mathsf{tailTree}_3\ (\mathit{Fork}_3\ v\ lt\ ct\ rt) \\
\quad \left| \begin{array}{lcl}
\mathsf{even}\ v & = & \text{if } lt == \mathit{Null}_3 \text{ then } ct \\
 & & \quad \text{else } \mathit{Fork}_3\ v\ (\mathsf{tailTree}_3\ lt)\ ct\ rt \\
\text{otherwise} & = & \text{if } rt == \mathit{Null}_3 \text{ then } ct \\
 & & \quad \text{else } \mathit{Fork}_3\ v\ lt\ ct\ (\mathsf{tailTree}_3\ rt)
\end{array} \right.
\end{array}$$

and we can easily verify that

$$\begin{aligned}\mathsf{headTree}_3 &= \mathsf{headTree} \cdot \mathsf{to2}\\ \mathsf{to2} \cdot \mathsf{tailTree}_3 &= \mathsf{tailTree} \cdot \mathsf{to2}\end{aligned}$$

Using these we need to define a function join_3 that satisfies the equation

$$\mathsf{flatten}_3\ (\mathsf{join}_3\ xt\ yt) = \mathsf{flatten}_3\ xt \mathbin{+\!\!+} \mathsf{flatten}_3\ yt$$

We propose the following definition

$$\begin{array}{l}\mathsf{join}_3\ xt\ yt\\ \quad\left|\begin{array}{ll} yt == \mathit{Null}_3 & = xt\\ \text{otherwise} & = \mathit{Fork}_3\ (\mathsf{headTree}_3\ yt)\ xt\ (\mathsf{tailTree}_3\ yt)\ xt\end{array}\right.\end{array}$$

This definition can be shown to satisfy

$$\mathsf{join}\ (\mathsf{to2}\ xt)\ (\mathsf{to2}\ yt) = \mathsf{to2}\ (\mathsf{join}_3\ xt\ yt) \tag{9}$$

using two cases (depending on whether yt is equal to Null_3) and the properties of $\mathsf{headTree}_3$ and $\mathsf{tailTree}_3$ as stated above.

Now that we have given a correct definition of join_3, we propose the following definition for delete_3:

$$\begin{array}{l}\mathsf{delete}_3\ x\ \mathit{Null}_3 = \mathit{Null}_3\\ \mathsf{delete}_3\ x\ (\mathit{Fork}_3\ v\ lt\ ct\ rt)\\ \quad\left|\begin{array}{ll} x < v \wedge \mathsf{even}\ v & = \mathit{Fork}_3\ v\ (\mathsf{delete}_3\ x\ lt)\ ct\ jt\\ x < v \wedge \mathsf{odd}\ v & = \mathit{Fork}_3\ v\ kt\ ct\ (\mathsf{delete}_3\ x\ rt)\\ x == v \wedge \mathsf{even}\ v & = \mathsf{join}_3\ lt\ ct\\ x == v \wedge \mathsf{odd}\ v & = \mathsf{join}_3\ rt\ ct\\ x > v & = \mathit{Fork}_3\ v\ lt\ (\mathsf{delete}_3\ x\ ct)\ rt\end{array}\right.\end{array}$$

The values jt and kt represent ternary trees — we could in fact choose to write for example $\mathsf{delete}_3\ x\ rt$ or even $\mathsf{insert}_3\ x\ rt$ in place of these which would aid in making the function more obfuscated. We should take care that any changes we make should not adversely affect the complexity.

To show that this operation is correct, we need to prove that for a ternary tree t and a value x, delete satisfies (4):

$$\mathsf{delete}\ x\ (\mathsf{to2}\ t) = \mathsf{to2}\ (\mathsf{delete}_3\ x\ t) \tag{10}$$

We do this by structural induction on t.

Base Case Let us suppose that $t = \mathit{Null}_3$. Using the definitions of $\mathsf{to2}$ and the two deletion operations, we find that

$$\mathsf{delete}\ x\ (\mathsf{to2}\ \mathit{Null}_3) = \mathsf{to2}\ (\mathsf{delete}_3\ x\ \mathit{Null}_3)$$

Step Case For the step case, we suppose that the tree t is of the form $Fork_3\ v\ lt\ ct\ rt$ and for the induction hypothesis we suppose that each of these subtrees satisfies (10) — we have five subcases (representing the five guards) to prove.

Subcase 1 Suppose that $x < v$ and v is even.

$$
\begin{array}{ll}
 & \mathsf{delete}\ x\ (\mathsf{to2}\ (Fork_3\ v\ lt\ ct\ rt)) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
 & \mathsf{delete}\ x\ (Fork\ (\mathsf{to2}\ lt)\ v\ (\mathsf{to2}\ ct)) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x < v\} \\
 & Fork\ (\mathsf{delete}\ x\ (\mathsf{to2}\ lt))\ v\ (\mathsf{to2}\ ct) \\
= & \quad \{\text{induction hypothesis}\} \\
 & Fork\ (\mathsf{to2}\ (\mathsf{delete}_3\ x\ lt))\ v\ (\mathsf{to2}\ ct) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
 & \mathsf{to2}\ (Fork_3\ v\ (\mathsf{delete}_3\ x\ lt)\ ct\ jt) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ with } x < v \text{ and } v \text{ even}\} \\
 & \mathsf{to2}\ (\mathsf{delete}_3\ x\ (Fork_3\ v\ lt\ ct\ rt))
\end{array}
$$

Subcase 2 The argument for the case when $x < v$ and v is odd follows a similar pattern to the previous case.

Subcase 3 Suppose that $x = v$ and v is even.

$$
\begin{array}{ll}
 & \mathsf{delete}\ x\ (\mathsf{to2}\ (Fork_3\ v\ lt\ ct\ rt)) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
 & \mathsf{delete}\ x\ (Fork\ (\mathsf{to2}\ lt)\ v\ (\mathsf{to2}\ ct)) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x == v\} \\
 & \mathsf{join}\ (\mathsf{to2}\ lt)\ (\mathsf{to2}\ ct) \\
= & \quad \{\text{equation (9)}\} \\
 & \mathsf{to2}\ (\mathsf{join}_3\ lt\ ct) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ — } x == v \text{ and } v \text{ even}\} \\
 & \mathsf{to2}\ (\mathsf{delete}_3\ x\ (Fork_3\ v\ lt\ ct\ rt))
\end{array}
$$

Subcase 4 The argument for the case when $x = v$ and v is odd follows a similar pattern to the previous case.

Subcase 5 The last case is when $x > v$ — the proof is similar to the first case.

The full proof is given in Appendix A.

4 Other Methods

In this section we give some alternatives to the methods and conversions discussed earlier.

4.1 Abstraction Functions

Consider how we can design a more general abstraction function than $\mathsf{to2}$ given in Section 3. We need a function af that satisfies

$$t_2 \leadsto t_3 \Leftrightarrow t_2 = \mathsf{af}\ t_3$$

where t_2 is a binary tree and t_3 is a ternary tree.

We first define a partitioning function with type

$$p :: \alpha \rightarrow [0..n)$$

for some $n : \mathbb{N}$, where α is the value type of our binary trees. Then we could write the abstraction function as

$$\begin{array}{l} \mathsf{af}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) \\ \quad \left| \begin{array}{ll} p(v) == 0 & = \mathit{Fork}\ (\mathsf{af}\ (\mathit{xt}_0))\ v\ (\mathsf{af}\ (\mathit{yt}_0)) \\ p(v) == 1 & = \mathit{Fork}\ (\mathsf{af}\ (\mathit{xt}_1))\ v\ (\mathsf{af}\ (\mathit{yt}_1)) \\ \ldots & \\ \text{otherwise} & = \mathit{Fork}\ (\mathsf{af}\ (\mathit{xt}_{n-1}))\ v\ (\mathsf{af}\ (\mathit{yt}_{n-1})) \end{array} \right. \end{array}$$

We should ensure that p partitions α uniformly — thus a good choice for p would be a hash function. If we define $p = \lambda v.v \bmod 2$ and let $\mathit{xt}_0 = \mathit{lt}$, $\mathit{yt}_0 = \mathit{ct}$, $\mathit{xt}_1 = \mathit{rt}$ and $\mathit{yt}_1 = \mathit{ct}$ then we can write $\mathsf{to2}$ in this form.

Definitions of operations using this more general function will contain occurrence of the test for the value of p. This means we can supply the definition of p separately from the definition of an operation and so our operations are not do not dependent on the choice of p. Thus we can create a set of suitable partitions and choose one at random.

Up to now, we have not changed the value at each node. For instance, a more complicated representation is obtained by using the following conversion function (assuming that our value type is $\mathbb{Z}$):

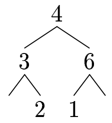
$$\begin{array}{l} \mathsf{toT}\ \mathit{Null} = \mathit{Null}_3 \\ \mathsf{toT}\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) \\ \quad \left| \begin{array}{ll} \mathsf{even}\ v & = \mathit{Fork}_3\ (v+2)\ (\mathsf{toT}\ \mathit{rt})\ \mathit{jt}\ (\mathsf{toT}\ \mathit{lt}) \\ \text{otherwise} & = \mathit{Fork}_3\ (v \times 3)\ \mathit{kt}\ (\mathsf{toT}\ \mathit{lt})\ (\mathsf{toT}\ \mathit{rt}) \end{array} \right. \end{array}$$

where jt and kt are arbitrary ternary trees.

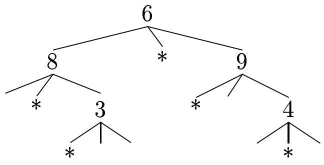
To create an abstraction function for this conversion, we need to invert the functions $\lambda\, i.(i+2)$ and $\lambda\, i.(3i)$. Thus we can define the abstraction function as follows:

$$\begin{array}{l} \mathsf{toB}\ \mathit{Null}_3 = \mathit{Null} \\ \mathsf{toB}\ (\mathit{Fork}_3\ w\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) \\ \quad \left| \begin{array}{ll} \mathsf{even}\ w & = \mathit{Fork}\ (\mathsf{toB}\ \mathit{rt})\ (w-2)\ (\mathsf{toB}\ \mathit{lt}) \\ \text{otherwise} & = \mathit{Fork}\ (\mathsf{toB}\ \mathit{ct})\ (\mathsf{div}\ w\ 3)\ (\mathsf{toB}\ \mathit{rt}) \end{array} \right. \end{array}$$

Using this abstraction function, the binary tree



can be represented by trees of the form



However, we must be careful when creating this kind of representation. For instance, if we had changed the function $v + 2$ in the definition of $\mathsf{to3}$ to $v + 1$ (and changed $v - 2$ to $v - 1$ in the abstraction function) then

$$\mathsf{toB}\ (\mathsf{toT}\ (Fork\ \ Null\ 2\ Null)) = (Fork\ \ (\mathsf{toB}\ jt)\ 1\ Null)$$

and it is not the case that $\mathsf{toB} \cdot \mathsf{toT} = id$.

4.2 Heuristics

Obfuscations are meant, by definition, to be obscure. We have given but one example here to indicate our method. Now, we give some points to consider when creating a representation of binary by using ternary trees.

- When creating a partitioning function for the definition of an abstraction function, we should ensure that each value in the range is equally likely. In particular, we should not construct a case in the definition of the abstraction function which is never considered or else it could be eliminated. We could use a hashing function for the partitioning function. In fact, for obfuscation purposes, we could have a set of hashing functions and randomly choose which hashing function we use (*cf.* Universal Hashing [5]).

- When defining operations for ternary trees, we need to make sure that some of these operations act on the "junk". This is so that it is not obvious that our junk is not really of interest.

- In the definition of insert_3, we saw that the last case did not have to test whether a value was even. This is because in our representation the centre subtree of a ternary tree corresponds to the right subtree of our binary tree. This was done to help obfuscate the definition so that it does not become obvious that we always check the parity of the value of each node. Therefore we should choose a ternary tree representation so that some of the cases in our functions simplify.

- When creating our ternary tree junk we could generate the junk randomly so that the same binary tree will be represented by different ternary trees in different executions.

- We must ensure that the junk we generate does not "stand out" — this is so that it is not obvious what is junk and what is "real" data. Thus we should the keep the junk values within the range of the values of our binary tree and we could possibly repeat values.

- We could decide to map the null binary tree to something other than $Null_3$ and we would need to change the abstraction function so that $Null$ is correctly recovered.

4.3 Using Fold Fusion

Many of our tree operations use structural recursion and so we could write them using tree folds and unfolds. As an example, we can define ternary tree fold as follows:

$$
\begin{aligned}
&\mathsf{foldTT}\ f\ e\ Null_3 = e\\
&\mathsf{foldTT}\ f\ e\ (Fork_3\ v\ lt\ ct\ rt) =\\
&\qquad f\ v\ (\mathsf{foldTT}\ f\ e\ lt)\ (\mathsf{foldTT}\ f\ e\ ct)\ (\mathsf{foldTT}\ f\ e\ rt)
\end{aligned}
$$

We can write all of the functions in Section 2 in terms of binary tree folds or unfolds (and some, such as modify, can be written as both a fold and an unfold). We can then use standard fold results to help prove the correctness of our operations.

For our proof of correctness, we require that our operations satisfy (4). Observing that we can write the abstraction function to2 as a binary tree unfold or a ternary tree fold gives us two cases using fusion [3] to help us establish correctness:

- If we can write to2 and op as binary tree unfolds then we can use unfold fusion to fuse in the functions from the right.

- If we can write to2 and op as ternary tree folds then we can use fold fusion to fuse in the functions from the left.

Let us consider the latter case. If we can write to2 and op_3 as ternary tree folds then to prove correctness we need to prove an equation of the form

$$f_1 \cdot (\mathsf{foldTT}\ g_1\ e_1) = f_2 \cdot (\mathsf{foldTT}\ g_2\ e_2)$$

for appropriate functions. We can prove this equation by fusing the left-hand functions into the fold to produce a new fold. Here is a theorem that allows us to do this.

Theorem 1. Ternary Tree Fold Fusion For a strict function f

$$f \cdot (\mathsf{foldTT}\ g\ a) = \mathsf{foldTT}\ h\ b$$

if $f\ a = b$ and h satisfies

$$f\ (g\ v\ xt\ yt\ zt) = h\ v\ (f\ xt)\ (f\ yt)\ (f\ zt)$$

This theorem is a special case of the general laws for fusion (see for example [3]). As an example of fold fusion, consider proving the correctness of a deletion operation (the full proof is contained in Appendix B). Using ternary fold fusion we can show that each side of (10) is equivalent to

$$
\begin{array}{l}
\mathsf{foldTT}\ h\ \mathit{Null} \\
\quad \text{where } h\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt} \\
\qquad \left| \begin{array}{ll}
x \neq v \wedge \mathsf{even}\ v & = \mathit{Fork}\ \mathit{lt}\ v\ \mathit{ct} \\
x \neq v \wedge \mathsf{odd}\ v & = \mathit{Fork}\ \mathit{rt}\ v\ \mathit{ct} \\
\mathsf{even}\ v & = \mathsf{join}\ \mathit{lt}\ \mathit{ct} \\
\text{otherwise} & = \mathsf{join}\ \mathit{rt}\ \mathit{ct}
\end{array} \right.
\end{array}
$$

To use fusion, we first define delete_3 in terms of ternary tree fold as follows:

$$
\begin{array}{l}
\mathsf{delete}_3\ x = \mathsf{foldTT}\ g\ \mathit{Null}_3 \\
\quad \text{where } g\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt} \\
\qquad \left| \begin{array}{ll}
x \neq v \wedge \mathsf{even}\ v & = \mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{jt} \\
x \neq v \wedge \mathsf{odd}\ v & = \mathit{Fork}_3\ v\ \mathit{kt}\ \mathit{ct}\ \mathit{rt} \\
\mathsf{even}\ v & = \mathsf{join}_3\ \mathit{lt}\ \mathit{ct} \\
\text{otherwise} & = \mathsf{join}_3\ \mathit{rt}\ \mathit{ct}
\end{array} \right.
\end{array}
$$

where jt and kt are arbitrary ternary trees. Note that to define delete_3 as a fold we do not have separate cases for $x < v$ and $x > v$.

As $\mathsf{to2}$ is strict and $\mathsf{to2}\ \mathit{Null}_3 = \mathit{Null}$, we can use ternary tree fusion and thus we need to find f and such that

$$\mathsf{to2} \cdot (\mathsf{foldTT}\ g\ \mathit{Null}_3) = \mathsf{foldTT}\ f\ \mathit{Null}$$

We propose that the definition of h above is a suitable candidate for the function f and so h must satisfy:

$$\mathsf{to2}\ (g\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) = h\ v\ (\mathsf{to2}\ \mathit{lt})\ (\mathsf{to2}\ \mathit{ct})\ (\mathsf{to2}\ \mathit{rt})$$

Hence, by ternary tree fold fusion $\mathsf{to2} \cdot (\mathsf{delete}_3\ x)$ can be written as $\mathsf{foldTT}\ h\ \mathit{Null}$ where h is defined as above.

Now, let us consider $\mathsf{delete} \cdot \mathsf{to2}$. We can write $\mathsf{to2}$ as follows:

$$
\begin{array}{l}
\mathsf{to2} = \mathsf{foldTT}\ g'\ \mathit{Null} \\
\quad \text{where } g'\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt} \\
\qquad \left| \begin{array}{ll}
\mathsf{even}\ v & = \mathit{Fork}\ \mathit{lt}\ v\ \mathit{ct} \\
\text{otherwise} & = \mathit{Fork}\ \mathit{rt}\ v\ \mathit{ct}
\end{array} \right.
\end{array}
$$

To enable us to perform ternary tree fold fusion, we write

$$
\begin{array}{l}
\mathsf{delete}\ x\ \mathit{Null} = \mathit{Null} \\
\mathsf{delete}\ x\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) \\
\quad \left| \begin{array}{ll}
x \neq v & = \mathit{Fork}\ (\mathsf{delete}\ x\ \mathit{lt})\ v\ (\mathsf{delete}\ x\ \mathit{rt}) \\
x == v & = \mathsf{join}\ (\mathsf{delete}\ x\ \mathit{lt})\ (\mathsf{delete}\ x\ \mathit{rt})
\end{array} \right.
\end{array}
$$

As delete is strict and $\mathsf{delete}\ x\ \mathit{Null} = \mathit{Null}$, we can apply ternary tree fold fusion to find a function f' such that

$$\mathsf{delete}\ x\ (\mathsf{foldTT}\ g'\ a) = \mathsf{foldTT}\ f'\ \mathit{Null}$$

Again, the function h above is suitable for the function f' and so satisfies:

$$\mathsf{delete}\ x\ (g'\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) = h\ v\ (\mathsf{delete}\ x\ \mathit{lt})\ (\mathsf{delete}\ x\ \mathit{ct})\ (\mathsf{delete}\ x\ \mathit{rt})$$

So by ternary tree fold fusion, $(\mathsf{delete}\ x) \cdot \mathsf{to2}$ is equivalent to $\mathsf{foldTT}\ h\ \mathit{Null}$ where h is defined as above. Thus we conclude that delete_3 satisfies (11). □

4.4 Comparing the Proof Methods

We have outlined two proof methods and shown how to prove the correctness of the deletion operation for ternary trees (the full proofs are given in Appendices A and B). In the earlier method, we used standard proof techniques including induction and in this case, these techniques yield shorter (and easier) proofs. We can always construct proofs no matter whether the function can be written as fold (or unfold).

Fold proofs require more inspiration as we not only have to write some functions as folds (or unfolds) but we have to "guess" how the fusion works (so that we can define a "h" function). We saw that we had to change our definitions of delete and delete_3 so that we could perform fusion and we may find that we have less opportunities for adding in extra obfuscations. The functions that we defined were less efficient as we had to traverse the whole tree rather than searching the appropriate subtree. The main advantage for using folds (and unfolds) is that the proofs can be automated (see, for example, [8]).

4.5 An object-oriented example

We now provide brief details about how our tree obfuscation could be applied to an object-oriented example. The example that we focus on is a method that deletes a value from a tree. The method was originally written in Oberon (from [15] — this book gives more details about how this operation works) and then converted to C# [1]. We first define a tree class as follows:

```
public class tree
{ public tree left;
  public tree right;
  public int val;  }
```

Then we need a method that removes a maximum from a subtree:

```
public static void remax (ref tree root, ref tree max)
{ if ((root.right) != null) {remax(ref root.right, ref max);}
  else {max = root; root=max.left;}
  return; }
```

Finally, a method for deleting an element:

```
public static void delete(ref tree root, int key)
{ tree node = new tree();
  if (root != null)
  { if (key < root.val) {delete(ref root.left,key);}
    else
    { if (key > root.val) {delete(ref root.right,key);}
      else
      { if (root.left == null) {root=root.right;}
        else
        { if (root.right == null) {root=root.left;}
          else
          { remax(ref root.left,ref node);
            node.left=root.left;
            node.right=root.right;
            root=node; } } } } } }
```

We use `to2` as our abstraction function for the obfuscation. We define the ternary tree class as follows:

```
public class ttree
{ public ttree left;
  public ttree centre;
  public ttree right;
  public int val; }
```

We need to state a method that serves a partition function (which we call `part`). If we wanted to use *even* then we would define:

```
public static bool part(int n)
{   return (n %2 == 0);}
```

Here is how we define `remax` for ternary trees:

```
public static void r3(ref ttree root, ref ttree x)
{ if ((root.centre) != null)
  {r3(ref root.centre, ref x); }
  else
  { x = root; if (part(root.val)) {root=x.left;}
              else {root=x.right;} }
    return; }
```

and Figure 4.5 shows how to define `delete`. Note that the methods `t1`, `t2`, `t3` and `t4` can be defined randomly. This method was obtained by repeatedly applying the rules for the ternary representation given in Section 3. Further work is needed to consider how to automate this transformation.

```
public static void d3(ref ttree root, int key)
{ ttree node = new ttree();
  if (root != null)
  { if (key < root.val)
    { if (part(root.val)) {d3(ref root.left, key);}
      else {d3(ref root.right,key);} }
    else
    { if (key > root.val) {d3(ref root.centre,key);}
      else
      { if (part(root.val))
        { if (root.left == null) {root=root.centre;}
          else
          { if (root.centre == null) {root=root.left;}
            else
            { r3(ref root.left,ref node);
              if (part(node.val))
              { node.left=root.left;
                t1(ref root.right, key);
                node.right = root.right; }
              else
              { node.right=root.left;
                t2(ref root.right, key);
                node.left = root.right; };
              node.centre=root.centre;
              root=node; } } }
        else
        { if (root.right == null) {root=root.centre;}
          else
          { if (root.centre == null) {root=root.right;}
            else
            { r3(ref root.right,ref node);
              if (part(node.val))
              { node.left=root.right;
                t3(ref root.left; key);
                node.right=root.left; }
              else
              { node.right=root.right;
                t4(ref root.left, key);
                node.left = root.left; };
              node.centre=root.centre;
              root=node; } } } } } } }
```

Figure 1: Delete method for ternary trees

5 Conclusions and Future Work

Using techniques developed for arrays, lists and matrices, we have created a new framework for obfuscating trees. The examples in this paper have been made simple in order to demonstrate the techniques involved. We have also given some heuristics (in Section 4.2) that need to be considered when producing obfuscations. This framework can be adapted to deal with more complicated obfuscations and also for obfuscating other kinds of trees such as rose trees or red-black trees. We have used Haskell as our modelling language which gives us an elegant yet simple way of specifying operations. Proving the correctness of an imperative obfuscation is usually hard to achieve; use of a functional language allows us to establish correctness easily.

We have demonstrated how to obfuscate binary trees by representing them as ternary trees. This representation has allowed us to add in bogus elements and so hide the "real" information. This extra information gives us further scope for obfuscation. For example, when trying to insert an element in a tree we could try to delete it from the junk part of the tree. We can actually specify a set of operations that represent the same function and we could choose one randomly. In fact, our approach gives plenty of scope for randomisation. In each execution run, we can have a different abstraction function (by choosing a suitable partition function), different junk values in a ternary tree representation and a different operation for each function. Each of these choices can be made randomly and so we create different program traces which compute the same value. This randomness provides additional confusion for an adversary and helps to keep the unobfuscated operations secret.

Once we have obfuscated operations using ternary trees, we can always add further obfuscations. A trivial obfuscation that can be used is *variable renaming* which helps to stop an attacker guessing the purpose of an operation from its name. For an operation, such as $\mathsf{flatten}$ that returns a list, we can apply list obfuscations such as splitting to the result.

Two concerns about obfuscation outlined in Section 1 were *correctness* and *complexity*. We have provided a way of proving the correctness of our obfuscated operations. Our proofs of correctness relied on only simple derivational techniques. This is a real strength of our approach as it means that our obfuscation methods are readily accessible. We need further exploration to determine whether these proofs can be automated. If we ensure that we keep the height and size of the ternary tree roughly the same as that of the binary tree it represents then our obfuscated operations should have the same complexity. Much of the extra work that the obfuscated operations do is to check the values of the partition function. When adding extra operations into our obfuscations (such as using insert_3 in the definition of delete_3) we must ensure that anything we add does not adversely change the complexity.

One area for future work is to explore how we can use these techniques to obfuscate imperative programs. Using imperative programs gives us different opportunities for obfuscation. For instance, trees can be implemented using pointers and we can exploit the difficulty in performing pointer analysis to construct obfuscations. We could then change binary trees into ternary trees and in the junk we could add in bogus pointers that point back up the tree.

It has not been mentioned *how* "obfuscated" our operations are. This is because there is no adequate definition for obfuscation that can be used with both functional and imperative programs. One possible definition is to consider how "difficult" it is to prove an assertion about an operation. We could, for instance, prove for the properties given in Section 2 for the original and obfuscated operations. We would find that the proofs for the obfuscated operations are more complicated. The advantage of considering the obfuscation of the operations of a data-type is that the axiomatic definition of a data-type [13] provides axioms (*i.e.* laws involving the operations) which are harder to prove for obfuscated programs. Thus, we could take the axioms of the data-type to be our assertions.

Acknowledgements

Thanks to my supervisor Jeff Sanders for the help and support he continually gives and to Rani Ettinger and Damien Sereni for useful comments and feedback on the paper.

References

[1] Tom Archer. *Inside C#*. Microsoft Press, 2001.

[2] 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.

[3] R. S. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1996.

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

[5] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

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

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Second Edition)*. The MIT Press, 2001.

[8] 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.

[9] 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.

[10] Stephen Drape. The matrix obfuscated. Technical Report PRG-RR-04-12, Programming Research Group, Oxford University Computing Laboratory, June 2004.

[11] Stephen Drape. Obfuscating set representations. Technical Report PRG-RR-04-09, Programming Research Group, Oxford University Computing Laboratory, May 2004.

[12] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .NET Intermediate Language using Path Logic Programming. In *Principles and Practice of Declarative Programming*, pages 133–144. ACM Press, 2002.

[13] Johannes J. Martin. *Data types and data structures*. Prentice Hall International (UK) Ltd., 1986.

[14] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, Inc., 1990.

[15] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula*. ACM Press, 1992.

Appendix

The appendix contains proofs of correctness for delete_3 using the standard method and the fold version, *i.e.*, we prove that for a ternary tree t and a value x

$$\mathsf{delete}\ x\ (\mathsf{to2}\ t) = \mathsf{to2}\ (\mathsf{delete}_3\ x\ t) \tag{11}$$

We will see that the proof using folds is more complicated.

A Standard Method

In this section, we use the definition of delete_3 given in Section 3.6 and we use structural induction to prove (11).

Base Case Let us suppose that $t = Null_3$.

$$\begin{array}{ll}
 & \mathsf{delete}\ x\ (\mathsf{to2}\ Null_3) \\
= & \quad \{\text{definition of } \mathsf{to2}\} \\
 & \mathsf{delete}\ x\ Null \\
= & \quad \{\text{definition of } \mathsf{delete}\} \\
 & Null \\
= & \quad \{\text{definition of } \mathsf{to2}\}
\end{array}$$

$$
\begin{array}{ll}
& \mathsf{to2}\ \mathit{Null}_3 \\
= & \quad \{\text{definition of } \mathsf{delete}_3\} \\
& \mathsf{to2}\ (\mathsf{delete}_3\ x\ \mathit{Null}_3)
\end{array}
$$

Step Case For the step case (which has five subcases), we suppose that the tree t is of the form $\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}$ and that the subtrees satisfy (11).

Subcase 1 Suppose that $x < v$ and v is even.

$$
\begin{array}{ll}
& \mathsf{delete}\ x\ (\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
& \mathsf{delete}\ x\ (\mathit{Fork}\ (\mathsf{to2}\ \mathit{lt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x < v\} \\
& \mathit{Fork}\ (\mathsf{delete}\ x\ (\mathsf{to2}\ \mathit{lt}))\ v\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\text{induction hypothesis}\} \\
& \mathit{Fork}\ (\mathsf{to2}\ (\mathsf{delete}_3\ x\ \mathit{lt}))\ v\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
& \mathsf{to2}\ (\mathit{Fork}_3\ v\ (\mathsf{delete}_3\ x\ \mathit{lt})\ \mathit{ct}\ \mathit{jt}) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ with } x < v \text{ and } v \text{ even}\} \\
& \mathsf{to2}\ (\mathsf{delete}_3\ x\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}))
\end{array}
$$

Subcase 2 Suppose that $x < v$ and v is odd

$$
\begin{array}{ll}
& \mathsf{delete}\ x\ (\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ odd}\} \\
& \mathsf{delete}\ x\ (\mathit{Fork}\ (\mathsf{to2}\ \mathit{rt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x < v\} \\
& \mathit{Fork}\ (\mathsf{delete}\ x\ (\mathsf{to2}\ \mathit{rt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{induction hypothesis}\} \\
& \mathit{Fork}\ (\mathsf{to2}\ (\mathsf{delete}_3\ x\ \mathit{rt}))\ v\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ odd}\} \\
& \mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{kt}\ \mathit{ct}\ (\mathsf{delete}_3\ x\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ with } x < v \text{ and } v \text{ odd}\} \\
& \mathsf{to2}\ (\mathsf{delete}_3\ x\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}))
\end{array}
$$

Subcase 3 Suppose that $x = v$ and v is even.

$$
\begin{array}{ll}
& \mathsf{delete}\ x\ (\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{delete}\ x\ (\mathit{Fork}\ (\mathsf{to2}\ \mathit{lt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x == v\} \\
 & \mathsf{join}\ (\mathsf{to2}\ \mathit{lt})\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\mathsf{join}\ (\mathsf{to2}\ \mathit{xt})\ (\mathsf{to2}\ \mathit{yt}) = \mathsf{to2}\ (\mathsf{join}_3\ \mathit{xt}\ \mathit{yt})\} \\
 & \mathsf{to2}\ (\mathsf{join}_3\ \mathit{lt}\ \mathit{ct}) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ --- } x == v \text{ and } v \text{ even}\} \\
 & \mathsf{to2}\ (\mathsf{delete}_3\ x\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}))
\end{array}
$$

Subcase 4 Suppose that $x = v$ and v is odd.

$$
\begin{array}{ll}
 & \mathsf{delete}\ x\ (\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ odd}\} \\
 & \mathsf{delete}\ x\ (\mathit{Fork}\ (\mathsf{to2}\ \mathit{rt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x == v\} \\
 & \mathsf{join}\ (\mathsf{to2}\ \mathit{rt})\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\mathsf{join}\ (\mathsf{to2}\ \mathit{xt})\ (\mathsf{to2}\ \mathit{yt}) = \mathsf{to2}\ (\mathsf{join}_3\ \mathit{xt}\ \mathit{yt})\} \\
 & \mathsf{to2}\ (\mathsf{join}_3\ \mathit{rt}\ \mathit{ct}) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ --- } x == v \text{ and } v \text{ odd}\} \\
 & \mathsf{to2}\ (\mathsf{delete}_3\ x\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}))
\end{array}
$$

Subcase 5 Suppose that $x > v$. For this case, let us define the following ternary tree

$$
\mathit{xt} = \begin{cases} \mathit{lt} & \text{if } v \text{ is even} \\ \mathit{rt} & \text{otherwise} \end{cases}
$$

Then

$$
\begin{array}{ll}
 & \mathsf{delete}\ x\ (\mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt})) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ and } \mathit{xt}\} \\
 & \mathsf{delete}\ x\ (\mathit{Fork}\ (\mathsf{to2}\ \mathit{xt})\ v\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x > v\} \\
 & \mathit{Fork}\ (\mathsf{to2}\ \mathit{xt})\ v\ (\mathsf{delete}\ x\ (\mathsf{to2}\ \mathit{ct})) \\
= & \quad \{\text{induction hypothesis}\} \\
 & \mathit{Fork}\ (\mathsf{to2}\ \mathit{xt})\ v\ (\mathsf{to2}\ (\mathsf{delete}_3\ x\ \mathit{ct})) \\
= & \quad \{\text{definitions of } \mathsf{to2} \text{ and } \mathit{xt}\} \\
 & \mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ (\mathsf{delete}_3\ x\ \mathit{ct})\ \mathit{rt}) \\
= & \quad \{\text{definition of } \mathsf{delete}_3 \text{ with } x > v\} \\
 & \mathsf{to2}\ (\mathsf{delete}_3\ x\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}))
\end{array}
$$

So, we can conclude that our definition of delete_3 satisfies (11). □

B Using folds

For this proof we use ternary tree fusion (Theorem 1) to show that each side of (11) is equivalent to

$$
\begin{array}{l}
\mathsf{foldTT}\ h\ \mathit{Null} \\
\quad \text{where } h\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt} \\
\qquad \left| \begin{array}{ll}
x \neq v \wedge \mathsf{even}\ v & = \mathit{Fork}\ \mathit{lt}\ v\ \mathit{ct} \\
x \neq v \wedge \mathsf{odd}\ v & = \mathit{Fork}\ \mathit{rt}\ v\ \mathit{ct} \\
\mathsf{even}\ v & = \mathsf{join}\ \mathit{lt}\ \mathit{ct} \\
\text{otherwise} & = \mathsf{join}\ \mathit{rt}\ \mathit{ct}
\end{array} \right.
\end{array}
\tag{12}
$$

Firstly, for the left-hand of (11), we use the definition of delete_3 given in Section 4.3 which is written in terms of ternary tree fold.

As $\mathsf{to2}$ is strict and $\mathsf{to2}\ \mathit{Null}_3 = \mathit{Null}$, we can use ternary tree fusion and thus we need to find f and such that

$$\mathsf{to2} \cdot (\mathsf{foldTT}\ g\ \mathit{Null}_3) = \mathsf{foldTT}\ f\ \mathit{Null}$$

We propose that the definition of h in (12) is a suitable candidate for the function f and we need h needs to satisfy:

$$\mathsf{to2}\ (g\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) = h\ v\ (\mathsf{to2}\ \mathit{lt})\ (\mathsf{to2}\ \mathit{ct})\ (\mathsf{to2}\ \mathit{rt})$$

We have four cases to prove

Case 1 $x \neq v \wedge \mathsf{even}\ v$

$$
\begin{array}{ll}
 & \mathsf{to2}\ (g\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) \\
= & \quad \{\text{first case for } g\} \\
 & \mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{jt}) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ even}\} \\
 & \mathit{Fork}\ (\mathsf{to2}\ \mathit{lt})\ v\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\text{first case for } h\} \\
 & h\ v\ (\mathsf{to2}\ \mathit{lt})\ (\mathsf{to2}\ \mathit{ct})\ (\mathsf{to2}\ \mathit{rt})
\end{array}
$$

Case 2 $x \neq v \wedge \mathsf{odd}\ v$

$$
\begin{array}{ll}
 & \mathsf{to2}\ (g\ v\ \mathit{lt}\ \mathit{ct}\ \mathit{rt}) \\
= & \quad \{\text{second case for } g\} \\
 & \mathsf{to2}\ (\mathit{Fork}_3\ v\ \mathit{kt}\ \mathit{ct}\ \mathit{rt}) \\
= & \quad \{\text{definition of } \mathsf{to2} \text{ with } v \text{ odd}\} \\
 & \mathit{Fork}\ (\mathsf{to2}\ \mathit{rt})\ v\ (\mathsf{to2}\ \mathit{ct}) \\
= & \quad \{\text{second case for } h\} \\
 & h\ v\ (\mathsf{to2}\ \mathit{lt})\ (\mathsf{to2}\ \mathit{ct})\ (\mathsf{to2}\ \mathit{rt})
\end{array}
$$

Case 3 $x = v \wedge \mathsf{even}\ v$

$$
\begin{array}{cl}
 & \mathsf{to2}\ (g\ v\ lt\ ct\ rt) \\
= & \quad \{\text{third case for } g\} \\
 & \mathsf{to2}\ (\mathsf{join}_3\ lt\ ct) \\
= & \quad \{\text{property of } \mathsf{join}_3\ (9)\} \\
 & \mathsf{join}\ (\mathsf{to2}\ lt)\ (\mathsf{to2}\ ct) \\
= & \quad \{\text{third case for } h\} \\
 & h\ v\ (\mathsf{to2}\ lt)\ (\mathsf{to2}\ ct)\ (\mathsf{to2}\ rt)
\end{array}
$$

Case 4 $x = v \wedge \mathsf{odd}\ v$

$$
\begin{array}{cl}
 & \mathsf{to2}\ (g\ v\ lt\ ct\ rt) \\
= & \quad \{\text{last case for } g\} \\
 & \mathsf{to2}\ (\mathsf{join}_3\ rt\ ct) \\
= & \quad \{\text{property of } \mathsf{join}_3\ (9)\} \\
 & \mathsf{join}\ (\mathsf{to2}\ rt)\ (\mathsf{to2}\ ct) \\
= & \quad \{\text{last case for } h\} \\
 & h\ v\ (\mathsf{to2}\ lt)\ (\mathsf{to2}\ ct)\ (\mathsf{to2}\ rt)
\end{array}
$$

Hence, by ternary tree fold fusion, $\mathsf{to2} \cdot (\mathsf{delete}_3\ x)$ can be written as the function (12).

Now for the right-hand side of (11), we use the ternary tree fold versions of delete and to2 (given in Section 4.3). As delete is strict and $\mathsf{delete}\ x\ Null = Null$, we can apply ternary tree fold fusion to find a function f' such that

$$\mathsf{delete}\ x\ (\mathsf{foldTT}\ g'\ a) = \mathsf{foldTT}\ f'\ Null$$

Again, the function h in (12) is suitable for the function f' and so we need to prove that

$$\mathsf{delete}\ x\ (g'\ v\ lt\ ct\ rt) = h\ v\ (\mathsf{delete}\ x\ lt)\ (\mathsf{delete}\ x\ ct)\ (\mathsf{delete}\ x\ rt)$$

We have four cases to prove.

Case 1 $x \neq v \wedge \mathsf{even}\ v$

$$
\begin{array}{cl}
 & \mathsf{delete}\ x\ (g'\ v\ lt\ ct\ rt) \\
= & \quad \{\text{first case for } g'\} \\
 & \mathsf{delete}\ x\ (Fork\ lt\ v\ ct) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x \neq v\} \\
 & Fork\ (\mathsf{delete}\ x\ lt)\ v\ (\mathsf{delete}\ x\ ct) \\
= & \quad \{\text{first case for } h\} \\
 & h\ v\ (\mathsf{delete}\ x\ lt)\ (\mathsf{delete}\ x\ ct)\ (\mathsf{delete}\ x\ rt)
\end{array}
$$

Case 2 $x \neq v \wedge \mathsf{odd}\ v$

$$
\begin{array}{cl}
 & \mathsf{delete}\ x\ (g'\ v\ lt\ ct\ rt) \\
= & \quad \{\text{second case for } g'\} \\
 & \mathsf{delete}\ x\ (\mathit{Fork}\ rt\ v\ ct) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x \neq v\} \\
 & \mathit{Fork}\ (\mathsf{delete}\ x\ rt)\ v\ (\mathsf{delete}\ x\ ct) \\
= & \quad \{\text{second case for } h\} \\
 & h\ v\ (\mathsf{delete}\ x\ lt)\ (\mathsf{delete}\ x\ ct)\ (\mathsf{delete}\ x\ rt)
\end{array}
$$

Case 3 $x = v \wedge \mathsf{even}\ v$

$$
\begin{array}{cl}
 & \mathsf{delete}\ x\ (g'\ v\ lt\ ct\ rt) \\
= & \quad \{\text{first case for } g'\} \\
 & \mathsf{delete}\ x\ (\mathit{Fork}\ lt\ v\ ct) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x = v\} \\
 & \mathsf{join}\ (\mathsf{delete}\ x\ lt)\ (\mathsf{delete}\ x\ ct) \\
= & \quad \{\text{third case for } h\} \\
 & h\ v\ (\mathsf{delete}\ x\ lt)\ (\mathsf{delete}\ x\ ct)\ (\mathsf{delete}\ x\ rt)
\end{array}
$$

Case 4 $x = v \wedge \mathsf{odd}\ v$

$$
\begin{array}{cl}
 & \mathsf{delete}\ x\ (g'\ v\ lt\ ct\ rt) \\
= & \quad \{\text{second case for } g'\} \\
 & \mathsf{delete}\ x\ (\mathit{Fork}\ rt\ v\ ct) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ with } x = v\} \\
 & \mathsf{join}\ (\mathsf{delete}\ x\ rt)\ (\mathsf{delete}\ x\ ct) \\
= & \quad \{\text{third case for } h\} \\
 & h\ v\ (\mathsf{delete}\ x\ lt)\ (\mathsf{delete}\ x\ ct)\ (\mathsf{delete}\ x\ rt)
\end{array}
$$

So by ternary tree fold fusion, $(\mathsf{delete}\ x) \cdot \mathsf{to2})$ is equivalent to the function (12). Thus we conclude that delete_3 satisfies (11) □