

# An Obfuscation for Binary Trees

---

Stephen Drape  
The University of Auckland  
New Zealand

With thanks to Jeff Sanders (Oxford University Computing Laboratory)

# What is obfuscation?

---

- An *obfuscation* is a behaviour preserving transformation which makes programs “hard to understand”.
- One key difficulty in obfuscation research is finding a suitable interpretation of “hard to understand”.
- Collberg et al “*A taxonomy of obfuscating transformations*”: data, layout and control-flow obfuscations with metric based definition.
- Barak et al “*On the (im)possibility of obfuscating programs*”: obfuscation is impossible for their definition.
- Aim for obfuscations which are **hard** but not necessarily **impossible** to undo.

# New approach to obfuscation

---

- Study **abstract data-types** by considering a “module” of operations and obfuscating the *whole* data-type rather than just individual methods
- Consider obfuscation to be **data refinement** allowing us to set us equations about our obfuscations.
- Use the functional language **Haskell** as a modelling language.
- Can easily prove that *all* our obfuscations are correct.

# Data refinement

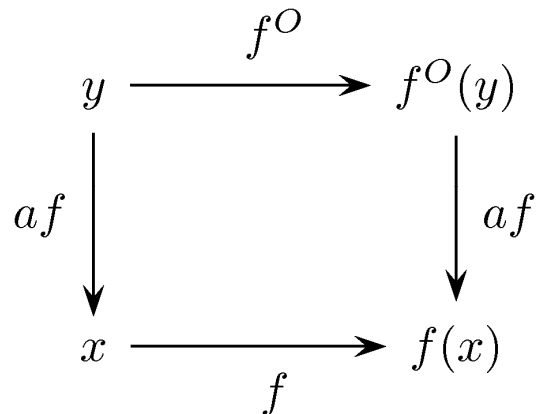
---

For refinement need an **abstraction function**  $af$  satisfying:

$$x \rightsquigarrow y \iff (x = af(y)) \wedge dti(y)$$

Suppose a function  $f :: D \rightarrow D$  is obfuscated to produce  $f^O$ . Then  $f^O$  is said to be **correct** wrt  $f$  if

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



# Correctness Equations

---

Using the abstraction function, we can say that  $f^O$  is correct if

$$f \cdot af = af \cdot f^O \quad (1)$$

If we have a function  $cf :: D \rightarrow O$ , which we call the *conversion function*, which satisfies  $of(x) = y \implies x \rightsquigarrow y$  then  $af \cdot cf = id$  and so we can rewrite Equation (1) to give

$$f = af \cdot f^O \cdot cf \quad (2)$$

Also, if we have that  $cf \cdot af = id$  then we can write

$$f^O = cf \cdot f \cdot af \quad (3)$$

which enables us to **derive** a definition for  $f^O$ .

# Binary Tree Data-Type

---

$Tree\ \alpha ::= Null \mid Fork\ (Tree\ \alpha)\ \alpha\ (Tree\ \alpha)$
---

$mkTree :: List\ \alpha \rightarrow Tree\ \alpha$
---

$flatten :: Tree\ \alpha \rightarrow List\ \alpha$
--

$member :: \alpha \rightarrow Tree\ \alpha \rightarrow \mathbb{B}$
--

$modify :: \alpha \rightarrow \alpha \rightarrow Tree\ \alpha \rightarrow Tree\ \alpha$
---

We can define `modify` as follows:

$$modify\ p\ q\ Null = Null$$
$$modify\ p\ q\ (Fork\ lt\ v\ rt)$$
$$\left| \begin{array}{l} v == p = Fork\ (modify\ p\ q\ lt)\ q\ (modify\ p\ q\ rt) \\ otherwise = Fork\ (modify\ p\ q\ lt)\ v\ (modify\ p\ q\ rt) \end{array} \right.$$

Later we'll see an obfuscated version of this operation.

# Conversion to Ternary Trees

---

The type definition for ternary trees is:

$$Tree_3 \alpha ::= Null_3 \mid Fork_3 \alpha (Tree_3 \alpha) (Tree_3 \alpha) (Tree_3 \alpha)$$

We need to state an abstraction function, which we will call `to2`, that converts a ternary tree into a binary one and satisfies:

$$t_2 \rightsquigarrow t_3 \iff t_2 = \text{to2} (t_3)$$

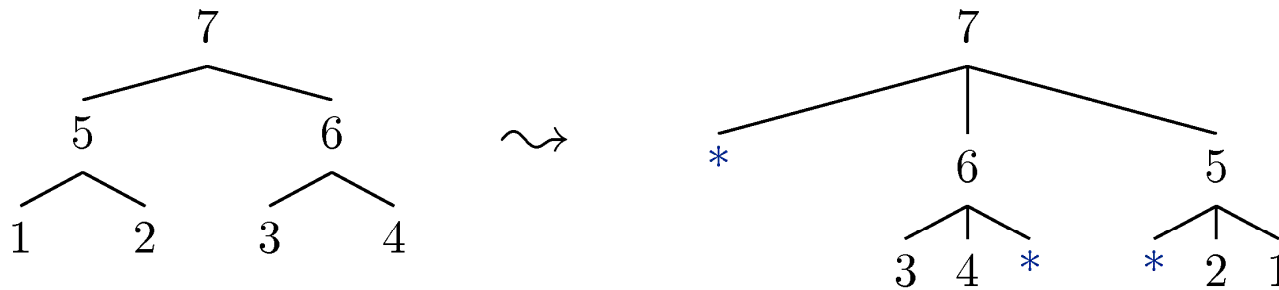
with the data-type invariant *True*.

We consider the following conversion:

$$Fork \ lt \ v \ rt \rightsquigarrow \begin{cases} Fork_3 \ v \ lt_3 \ rt_3 \ junk_a & \text{if } v \text{ is even} \\ Fork_3 \ v \ junk_b \ rt_3 \ lt_3 & \text{otherwise} \end{cases}$$

and the tree *Null* is converted to *Null<sub>3</sub>*.

# Abstraction Function



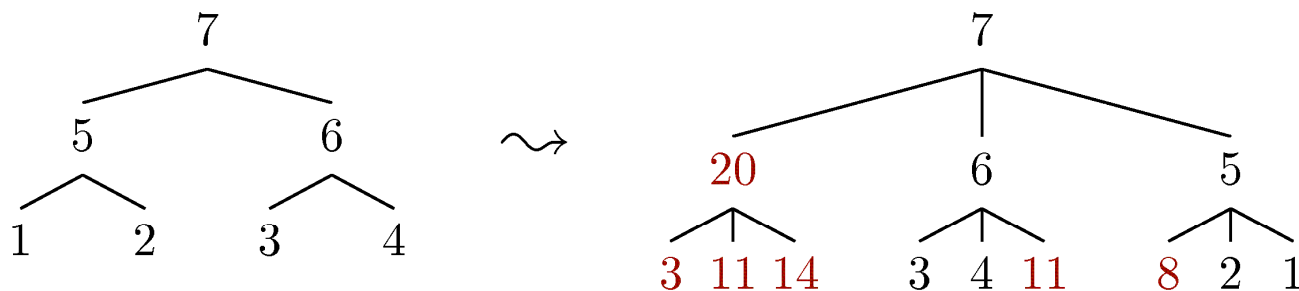
The symbol “\*” stands for different arbitrary ternary trees.

We define the *abstraction function* `to2` as follows:

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

We can also define a function `to3` that is a right inverse for `to2`.

# Conversion Function



$\text{to3 } \text{Null} = \text{Null}_3$

$\text{to3 } (\text{Fork } lt \ v \ rt)$

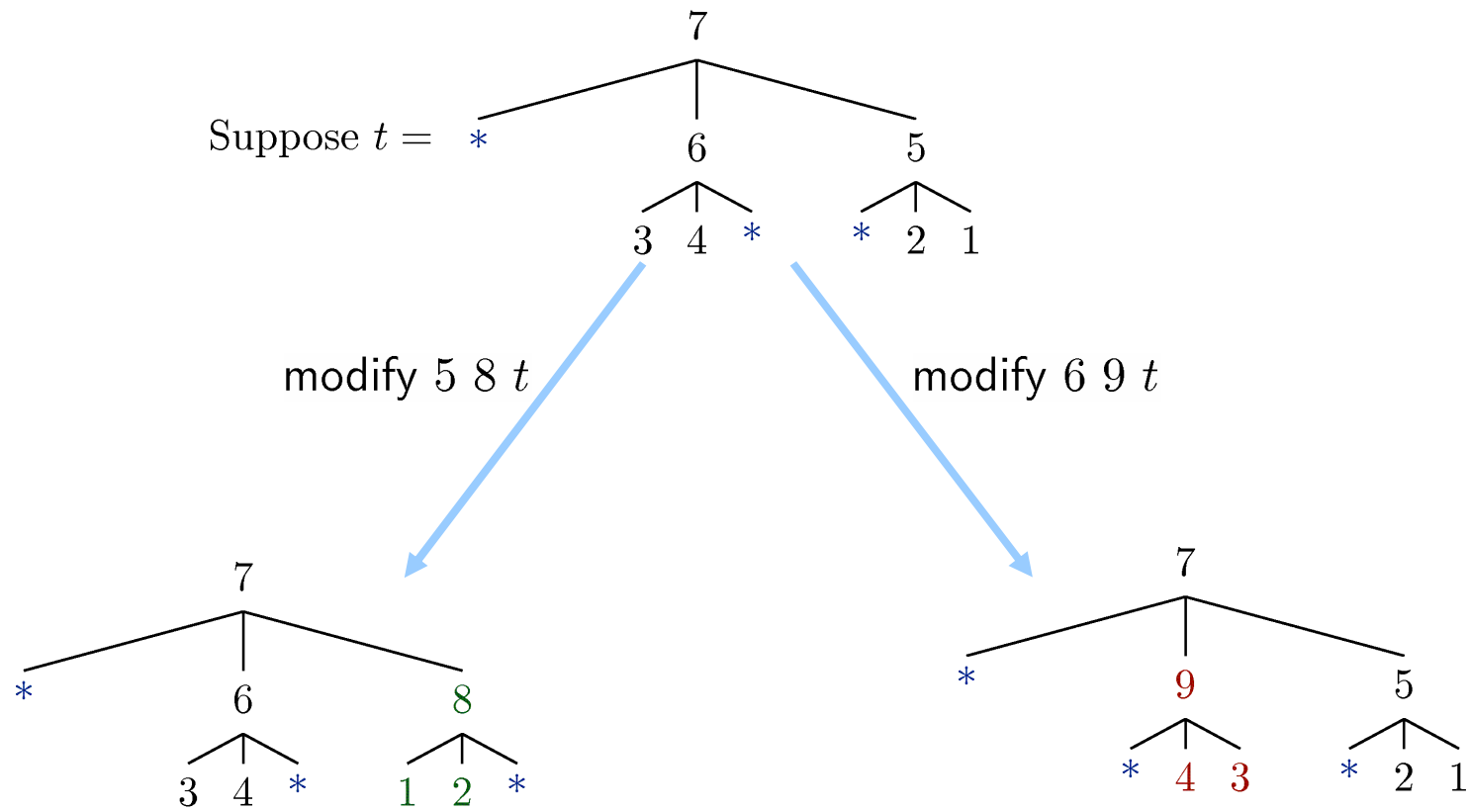
even $v$	$= \text{Fork}_3 \ v \ (\text{to3 } lt) \ (\text{to3 } rt) \ (\text{to3}' lt)$
otherwise	$= \text{Fork}_3 \ v \ (\text{to3}' rt) \ (\text{to3 } rt) \ (\text{to3 } lt)$

where

$\text{to3}' \ \text{Null} = \text{Null}_3$

$\text{to3}' \ (\text{Fork } lt \ v \ rt) = \text{Fork}_3 \ (3 \times v + 2) \ (\text{to3 } lt) \ (\text{to3}' lt) \ (\text{to3}' rt)$

# Modify for ternary trees



# Obfuscating modify

---

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

$$\begin{aligned} \text{modify}_3 p q \text{ Null}_3 &= \text{Null}_3 \\ \text{modify}_3 p q (\text{Fork}_3 v lt ct rt) & \\ \left| \begin{array}{l} v == p \wedge \text{even } p \wedge \text{even } q = \text{Fork}_3 q lt' ct' jt_1 \\ v == p \wedge \text{odd } p \wedge \text{odd } q = \text{Fork}_3 q jt_2 ct' rt' \\ v == p \wedge \text{even } p \wedge \text{odd } q = \text{Fork}_3 q jt_3 ct' lt' \\ v == p \wedge \text{odd } p \wedge \text{even } q = \text{Fork}_3 q rt' ct' jt_4 \\ \text{otherwise} = \text{Fork}_3 v lt' ct' rt' \end{array} \right. \\ \text{where } lt' &= \text{modify}_3 p q lt \\ rt' &= \text{modify}_3 p q rt \\ ct' &= \text{modify}_3 p q ct \end{aligned}$$

The four trees  $jt_i$  represent arbitrary ternary trees.

# An optimised version

---

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

$$| v == p \wedge \text{mod}(p-q) 2 == 0 = \text{Fork}_3 q lt' ct' rt'$$

Similarly by letting  $jt_3 = rt'$  and  $jt_4 = lt'$  we can collapse two more conditions. This leads us to the following optimisation of  $\text{modify}_3$ :

$\text{modify}_3 p q \text{Null}_3 = \text{Null}_3$

$\text{modify}_3 p q (\text{Fork}_3 v lt ct rt)$

$$\left| \begin{array}{l} v == p \wedge \text{mod}(p-q) 2 == 0 = \text{Fork}_3 q lt' ct' rt' \\ v == p \wedge \text{mod}(p-q) 2 == 1 = \text{Fork}_3 q rt' ct' lt' \\ \text{otherwise} = \text{Fork}_3 v lt' ct' rt' \end{array} \right.$$

where  $lt' = \text{modify}_3 p q lt$

$rt' = \text{modify}_3 p q rt$

$ct' = \text{modify}_3 p q ct$

# Proving correctness

---

Using

$$f \cdot af = af \cdot f^O \quad (1)$$

we can prove the operation  $\text{modify}_3$  is correct by showing that

$$(\forall t_3 :: \text{Tree}_3) \text{ modify } p \ q \ (\text{to2 } t_3) = \text{to2 } (\text{modify}_3 \ p \ q \ t_3)$$

We can prove this equation by using structural induction on  $t_3$ :

- **Base Case** We first suppose that  $t_3 = \text{Null}_3$  and this case follows immediately.
- **Step Case** We now suppose that  $t_3 = \text{Fork}_3 \ v \ lt \ ct \ rt$  and we have 6 subcases depending on the values of  $v$ ,  $p$  and  $q$ .

# Deriving obfuscations

---

The operation  $\text{flatten} :: \text{Tree } \alpha \rightarrow \text{List } \alpha$  converts a tree into a list. We can use the equation  $\text{flatten}_3 t_3 = \text{flatten} (\text{to2 } t_3)$  to derive a definition for  $\text{flatten}_3$ . We can use structural induction. So, suppose  $t_3 = \text{Fork}_3 v \text{ } lt \text{ } ct \text{ } rt$  and  $v$  is even then

$$\begin{aligned} & \text{flatten}_3 (\text{Fork}_3 v \text{ } lt \text{ } ct \text{ } rt) \\ = & \quad \{\text{definition}\} \\ & \text{flatten} (\text{to2} (\text{Fork}_3 v \text{ } lt \text{ } ct \text{ } rt)) \\ = & \quad \{\text{definition of to2 with } v \text{ even}\} \\ & \text{flatten} (\text{Fork} (\text{to2 } lt) v (\text{to2 } ct)) \\ = & \quad \{\text{definition of flatten}\} \\ & \text{flatten} (\text{to2 } lt) ++ [v] ++ \text{flatten} (\text{to2 } ct) \\ = & \quad \{\text{induction}\} \\ & \text{flatten}_3 lt ++ [v] ++ \text{flatten}_3 ct \end{aligned}$$

# Complete definition

---

We have shown the case for when  $v$  is even and the proof for when  $v$  is odd is similar. Finally we have to give a derivation for when  $t_3 = \text{Null}$  — this is straightforward.

Putting the cases together gives us the following definition:

$$\begin{aligned} \text{flatten}_3 \text{ Null}_3 &= [] \\ \text{flatten}_3 (\text{Fork}_3 v \text{ lt } \text{ ct } \text{ rt}) & \\ \left| \begin{array}{l} \text{even } v &= (\text{flatten}_3 \text{ lt}) ++ [v] ++ (\text{flatten}_3 \text{ ct}) \\ \text{otherwise} &= (\text{flatten}_3 \text{ rt}) ++ [v] ++ (\text{flatten}_3 \text{ ct}) \end{array} \right. \end{aligned}$$

Using similar equations we can derive the definitions of other obfuscated operations.

# An object-oriented example

---

A tree class:

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

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;}
```

A method for deleting an element from a binary search tree:

```
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; } } } } } }
```

Can we use ternary trees to obfuscate this operation?

# Using a ternary trees

---

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

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 rootcentre, key);}
      else {
        if (part(root.val)) {
          if (root.left == null) {root = rootcentre;}
          else {if (rootcentre == 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;};
              nodecentre = rootcentre; root = node;} } }
          else {if (root.right == null) {root = rootcentre;}
            else {
              if (rootcentre == 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;};
                nodecentre = rootcentre; root = node;} } } } } } }

public static bool part(int n) {return (n %2 == 0);} /* the partition function */
```

Note that the methods t1, t2, t3 and t4 can be defined arbitrarily.

# Conclusions

---

- We have used abstract data-types, refinement and functional programming to discuss a new approach to obfuscation.
- Obfuscations can be proved correct easily and, in some cases, we can derive obfuscations.
- This approach can be used with other data-types such as lists, sets and matrices.
- Can apply our methods to imperative programs — an area for future work is to automate this process.
- How “obfuscated” are our tree obfuscations?