Generalising the Array Split Obfuscation

Stephen Drape
Department of Computer Science,
University of Auckland

Draft Version

Abstract

An *obfuscation* is a behaviour-preserving program transformation whose aim is to make a program "harder to understand". Obfuscations are mainly applied to make reverse engineering of object-oriented programs more difficult. In this paper, we propose a fresh approach by obfuscating *abstract data-types* allowing us to develop structure-dependent obfuscations that would otherwise (traditionally) not be available. We regard obfuscation as data refinement enabling us to produce equations for proving correctness and we model the data-type operations as functional programs making our proofs easy to construct.

We show how we can generalise an imperative obfuscation — an *array split* — so that we can apply it to more general data-types and we give specific examples for lists and matrices. We develop a theorem which allows us, under certain conditions, to produce obfuscated operations directly. Our approach allows us to produce random obfuscations and we give an example for our list data-type.

1 Introduction

We will consider a particular imperative obfuscation and discuss how it can be generalised. To do this, we will study abstract data-types and propose a new approach to obfuscation. "Obfuscation" means "Concealment or obscuration of a concept, idea, expression". From a Computer Science perspective, an obfuscation is a behaviour-preserving program transformation whose aim is to make a program "harder to understand". Obfuscations are applied to a program to make reverse engineering of the program more difficult. Two concerns about an obfuscation are whether it preserves behaviour (*i.e.* it is *correct*) and the degree to which it maintains efficiency. Two current approaches to obfuscation are discussed in Collberg *et al.* [3] and Barak *et al.* [1]. Collberg *et al.* define metrics which try to qualify "harder to understand" and consider object-oriented obfuscations.

Barak *et al.* [1] take a more formal approach to obfuscation — their notion of obfuscation is as follows. An obfuscator $\mathcal{O}$ is a "compiler" which

takes as input a program P and produces a new program $\mathcal{O}(P)$ such that for every P:

- *Functionality* — $\mathcal{O}(P)$ computes the same function as P.
- *Polynomial Slowdown* — the description length and running time of $\mathcal{O}(P)$ are at most polynomially larger than that of P.
- *"Virtual black box" property* — "Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P" [1, Page 2].

With this definition, Barak *et al.* construct a family of functions which is unobfuscatable in the sense that there is no way of obfuscating programs that compute these functions. The main result of [1] is that their notion of obfuscation is impossible to achieve. This definition, in particular the "Virtual Black Box" property, is too strong for our purposes and so we consider a weaker notion. We do not consider our programs as being "black boxes" as we assume that any attacker can inspect and modify our code and we should aim for obfuscations that are *difficult* (but not necessarily *impossible*) to undo. In this paper, we will not explicitly give a definition pertinent to our approach — a definition is given in [6, Chapter 3] — but use the notion of obfuscation from Collberg *et al.* [3]. 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 current view of obfuscating programs (in particular [3]) often just focuses on code fragments and concrete data structures. Our contribution to the study of obfuscation will be to consider obfuscation as *data refinement* [5] and we propose a framework for objects which we view (for the purpose of refinement) as *data-types*, calling the methods *operations*. We consider abstract data-types and define obfuscations for the whole data-type. This means that rather than just obfuscating single operations we obfuscate *all* the operations in the data-type. We model our operations using the functional language Haskell [12] to provide a framework for specifying data-types and obfuscations.

A requirement of both of the definitions of Barak *et al.* [1] and Collberg *et al.* [3] is that an obfuscation is correct (*i.e.* behaviour preserving). Thus when creating obfuscations we should ensure that they are correct but in an imperative context, proofs of correctness for transformations are frequently hard, typically requiring language restrictions — see, for example, [9] in which a framework is provided for proving the correctness of compiler optimisations. This means that obfuscations are mostly stated without giving a proof of correctness. However, the use of data refinement techniques [5] and Haskell (which allows us to reason about programs mathematically) enable us to prove the correctness of *all* our 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 refinement and derivation techniques) allows us to apply obfuscation to more general areas. Since we use Haskell as a basis 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 approach will provide support for purely imperative obfuscations. Some aims of our proposal are to make the process of creating obfuscations easier and to have the ability to generalise our obfuscations. The techniques that we will use (such as data-types, refinement and functional programming) are well established in the programming research community and so they will be familiar to many computer scientists. Although the techniques of formal methods are well-known, the application of obfuscation using these techniques is not. To demonstrate the scope of this new proposal we show how to generalise some of the specific obfuscations discussed in [3]. We will discuss one particular array obfuscation and show how it can be applied to more abstract data-types.

1.1 Array Splitting

The particular obfuscation that we are going to study is called an *array split*. This obfuscation changes the structure of an array by placing the elements into two (or more) new arrays. Collberg *et al.* [3] gives this example of an array split:

$$\begin{array}{l} \textbf{int}\ [\,]\ A = \textbf{new int}\ [10]; \\ \\ \ldots \\ A[i] = \ldots; \end{array} \quad \Rightarrow \quad \begin{array}{l} \textbf{int}\ [\,]\ B1 = \textbf{new int}\ [5]; \\ \textbf{int}\ [\,]\ B2 = \textbf{new int}\ [5]; \\ \ldots \\ \textbf{if}\ ((i\,\%\,2) == 0)\ B1[i/2] = \ldots; \\ \qquad\qquad \textbf{else}\ B2[i/2] = \ldots; \end{array} \tag{1}$$

The first step to generalising this transformation is to specify functions which determine how an array is split. We need to define a split so that an array A of size n is broken up into two other arrays. To do this, we define three functions ch, f_1 and f_2 and two new arrays B_1 and B_2 of sizes m_1 and m_2 respectively (where $m_1 + m_2 \geq n$). Each array element needs to be placed in exactly one of the new arrays; this choice will be determined by ch (the *choice function*). Each f_i gives the position of the elements in each array.

The types of the functions are as follows:

$$\begin{aligned} ch &:: [0..n) \to \mathbb{B} \\ f_1 &:: [0..n) \to [0..m_1) \\ f_2 &:: [0..n) \to [0..m_2) \end{aligned}$$

Then the relationship between A and B_1 and B_2 is given by the following rule:

$$A[i] = \begin{cases} B_1[f_1(i)] & \text{if } ch(i) \\ B_2[f_2(i)] & \text{otherwise} \end{cases}$$

To ensure that there are no index clashes we require that f_1 is injective for the values for which ch is true (similarly for f_2). We can write the transformation described above as:

$$A[i] = \begin{cases} B_1[i \text{ div } 2] & \text{if } i \text{ is even} \\ B_2[i \text{ div } 2] & \text{if } i \text{ is odd} \end{cases} \tag{2}$$

This relationship can be generalised further so that the elements of A could be split between more than two arrays. Suppose that we want to split A into p arrays $B_0, \ldots, B_{p-1}$ with sizes $s_0, \ldots, s_{p-1}$ respectively. We can make the type of ch as follows:

$$ch :: [0..n) \to [0..p)$$

We define a family of functions $\mathcal{F}$ where $\mathcal{F} = \{f_0, \ldots, f_{p-1}\}$ where each function f_i has type

$$f_i :: ch^{-1}(i) \rightarrowtail [0..s_i)$$

The mapping for $A[i]$ is:

$$A[i] = B_k[f_k(i)] \quad \text{where } k = ch(i)$$

So that each f_i is a total function we use $ch^{-1}(i)$ for the domain rather than $[0..n)$. This generalisation is discussed further in Section 3 where we consider defining splits for more abstract data-types. We will also develop a theorem which, under certain conditions, gives definitions for obfuscated operations directly.

2 Data-types and Obfuscation

To obfuscate a program, you can either obfuscate its algorithms or obfuscate its data structures. We will concentrate on the latter and propose a framework for objects which we view (for the purpose of refinement) as *data-types*, calling the methods *operations*. We consider abstract data-types (*i.e.* a local state accessible only by declared operations) and define obfuscations for the whole data-type. This means that rather than just obfuscating single methods we are obfuscating *all* the methods of that data-type.

For our framework, we model data-type operations using a functional language and view obfuscation as data refinement [5]. These mathematical techniques allow us to prove the correctness of *all* our obfuscations and also for some obfuscations we are able to derive an obfuscated operation

from an unobfuscated one. When we define a data-type, we will insist that the operations are total and deterministic. This restriction ensures that we can use equality in our correctness proofs. We choose to use the style of the functional language Haskell [12] to specify operations and obfuscations (note that *we are not aiming to obfuscate Haskell code but to use Haskell as a modelling language*). Haskell is a suitable language as its mathematical flavour lends itself to derivational proofs thus allowing us to prove properties (such as correctness) of Haskell functions (see for example [2, Chapter 4] and [14, Chapter 14]). Since we are using Haskell as a modelling language we should ensure that we can convert our operations into other languages. Thus we will not exploit all the characteristics of Haskell and in particular, we will use finite, well-defined data-types and we will not use laziness. We could have chosen a strict language such as ML but the syntax of ML and the presence of reference cells means that it is not as elegant a setting for proofs as Haskell.

The view of Collberg *et al.* [3] concentrates on concrete data structures such as variables and arrays. All of the data-types that we will use could be implemented concretely using arrays — for example, we can use the standard "double, double plus one" conversion [4, Chapter 6] to represent a binary tree as an array. Why, therefore, do we obfuscate the abstract data-type rather than its concrete implementation? Apart from providing a simple way of proving correctness, 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. For instance, in [7], a data-type for trees is considered and so the usual tree transformations (such as swapping two subtrees) are naturally available; they would be more difficult to conceive using arrays. Also, converting a data-type to an array often loses information about the data-type (such as the structure) and so it would be difficult to perform operations that use or rely on knowledge of that structure. Some matrix operations rely on the 2-dimensional structure of matrices and so we would have difficulties defining such operations for matrices that have flattened to arrays. We may also gain new array obfuscations by obfuscating a data-type and then converting the data-type to an array. Thus, we have two opportunities for obfuscation; the first using our new data-type approach and the second using the standard imperative methods.

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 obfuscated operations) we view obfuscation as *data refinement* [5]. A refinement can be achieved by a relation R

between an abstract and a concrete state:

$$R :: A \leftrightarrow C$$

that satisfies a simulation condition [5, Section 2.1]. A refinement is called *functional* if and only if there exists a data-type invariant dti and a function $af :: C \to A$, called an *abstraction function*, such that R has the form

$$a\,R\,c \;\Leftrightarrow\; (af(c) = a) \wedge dti(c)$$

If we have a functional refinement then each instance of the concrete state satisfying the data-type invariant is related to at most one instance of the abstract state. That corresponds to the concrete state having more "structure" than the abstract state. In general, when obfuscating we aim to obscure the data-type by adding more structure and so we propose that the obfuscated data-type O will be no more abstract than the original data-type D. Thus the most general form of refinement for us is functional refinement. This formulation allows us to have many obfuscations which can be "undone" by the same abstraction function. We may have a situation where we obfuscate a data-type by first performing a (possibly non-functional) refinement and then obfuscating this refinement. As data refinement is a well-known technique, we will concentrate on just the obfuscation part of the 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 $y :: O$

$$x \rightsquigarrow y \Leftrightarrow (x = af(y)) \wedge dti(y) \tag{3}$$

The term $x \rightsquigarrow y$ is read as "x is data refined by y" (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 that if we have an obfuscation function $of :: D \to O$ that satisfies

$$of(x) = y \Rightarrow x \rightsquigarrow y$$

then

$$af \cdot of = id \tag{4}$$

Suppose that the operation $f :: D \to D$ is defined in D. Then to obfuscate f we want an operation f^O with type

$$f^O :: O \to O$$

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:

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

If f^O is a correct refinement (obfuscation) of f then we write $f \rightsquigarrow f^O$. Using Equation (3), we obtain

$$f \cdot af = af \cdot f^O \tag{5}$$

Thus we can prove that a definition of f^O is correct by using this equation. Note that since we have total, deterministic operations then we have equality in this equation. However general correctness follows if the equality is merely $\sqsubseteq$ (refinement). We can produce similar equations for operations with different types.

3 Generalising Splitting

We aim to generalise the array split obfuscation and we want to specify splitting as a refinement (Section 2.1). In this section we introduce some ideas and notation that we will use to define this generalisation. The aim of a split is to break up an object t of a particular data-type T into n smaller objects (which we normally refer to as the *split components*) $t_0, t_1, \ldots, t_{n-1}$ of type T by using a so-called split *sp*:

$$t \rightsquigarrow \langle t_0, t_1, \ldots, t_{n-1} \rangle_{sp}$$

The aim of this refinement is to spread the information contained in t across the split components. The obfuscator knows the relationship between an object and the components and should aim to hide this relationship.

3.1 Indexed Data-Types

What features do arrays have that allow us to define a split? First, the array data-type has an underlying type *e.g.* we can define an array of integers or an array of strings. Secondly, we can access any element of the array by providing an index (with arrays we usually use natural numbers).

We generalise these features as follows by defining an *Indexed Data-Type* (which we abbreviate to IDT). An indexed data-type T has the following properties:

(a) an underlying type called the *element type* — if T has an element type α then we write $T(\alpha)$

(b) a set of indexes I_T called the *indexer*

(c) a partial function $\mathbb{A}_T$ called the *access function* with type

$$\mathbb{A}_T :: T(\alpha) \times I_T \nrightarrow \alpha$$

If t has type $T(\alpha)$ (an IDT) then t consists of a collection of elements (multiple elements are allowed) of type α. Borrowing notation from Haskell list comprehension, we write $e \leftarrow t$ to denote that e is an element of t (this is analogous to set membership) and it satisfies:

$$e \leftarrow t \Leftrightarrow (\exists\, i \in I_T) \bullet \mathbb{A}_T(t, i) = e$$

We can define an indexer I_t for t as follows:

$$I_t = \{i \in I_T \mid \mathbb{A}_T(t, i) \leftarrow t\}$$

Note that $\mathbb{A}_T(t, i)$ will be undefined if and only if $i \in I_T \backslash I_t$.

Let us look at some example IDTs:

- For arrays, we can take $I_{array} = \mathbb{N}$ and $\mathbb{A}_{array}\,(A, I) = X \Leftrightarrow A[I] = X$.
- For finite Haskell lists, the indexer I_{list} is $\mathbb{N}$ and the access function is written !!.
- For sequences in Z [13, Section 4.5], the indexer is $\mathbb{N}$ and the access function is written as functional application, *i.e.* $\mathbb{A}_{seq}\,(s, n) = a \Leftrightarrow s\,n = a$
- If $\mathbf{M}^{r\times c}$ is the set of matrices with r rows and c columns, then $I_{\mathbf{M}^{r\times c}} = [0..r) \times [0..c)$ and we write $\mathbf{M}(i, j)$ to denote the access function.
- For binary trees which have type:

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

  we can define an indexer to be a string of the form $(L|R)^*$ where L stands for left subtree and R for right subtree (if we want to access the top element then we use the empty string ε).

3.2 Defining a Split

Now that we have a more general data-type, how can we define a split for IDTs? We will characterise a split of an IDT by a pair $(ch, \mathcal{F})$ — where ch is a function (called the *choice function*) and $\mathcal{F}$ is a family of functions. Suppose that we want to split $t :: T(\alpha)$ of an indexed data-type T using a split $sp = (ch, \mathcal{F})$. For data refinement we require a unique abstraction function that "recovers" t unambiguously from the split components. We insist that for every position $i \in I_t$ the element $\mathbb{A}_T(t, i)$ is mapped to exactly one split component. Thus we need the choice function to be a total function and all the functions $f_k \in \mathcal{F}$ to be injective. We also require that the domain of f_k is $ch^{-1}(k)$ so that each position of I_t is mapped to exactly one split component.

Hence

$$\begin{aligned} & t \rightsquigarrow \langle t_0, t_1, \ldots, t_{n-1} \rangle_{sp} \\ \Leftrightarrow & \\ & ch :: I_t \to [0..n) \\ \wedge & \\ & (\forall\ f_k \in \mathcal{F})\ \bullet\ f_k :: ch^{-1}(k) \rightarrowtail I_{t_k} \\ \wedge & \\ & \mathbb{A}_T(t_k, f_k(i)) = \mathbb{A}_T(t, i) \text{ where } ch(i) = k \end{aligned} \tag{6}$$

We will call the last equation — Equation (6) — the *split relationship*.

From this formulation, we can see that every element of t (counting multiplicities) must be contained in the split components. So we require that

$$(\forall\, a \leftarrow t)\ \bullet\ \mathsf{freq}(a, t) \leq \sum_{e \in [0..n)} \mathsf{freq}(a, t_e)$$

where $\mathsf{freq}(c, y)$ is the frequency of the occurrence of an element c where $c \leftarrow y$ and can be defined as follows:

$$\mathsf{freq}(c, y) = |\{i \in I_y \mid \mathbb{A}_T(y, i) = c\}|$$

3.3 Example Splits

We now give two examples and for these splits, we assume that we have an IDT $T(\alpha)$ which has an access function $\mathbb{A}_T$ and for each $t :: T(\alpha)$, the indexer $I_t \subseteq \mathbb{N}$.

The first split we shall look at is called the *alternating split*, which we denote by *asp*. This split has two components — the first contains the elements (in order) which have an even index and the second component contains the rest.

The choice function is defined as

$$ch(i) = i \bmod 2$$

and the family of functions $\mathcal{F} = \{f_0, f_1\}$ where

$$f_0 = f_1 = (\lambda i.i \text{ div } 2)$$

For an element $t :: T$, we write $t \rightsquigarrow \langle t_0, t_1 \rangle_{asp}$. We can easily see that if $dom(f_k) = ch^{-1}(k)$ then f_k is injective.

Using the definitions of ch and $\mathcal{F}$ we can define an access operation:

$$\mathbb{A}_{T_{asp}}(\langle t_0, t_1 \rangle_{asp}, i) = \begin{cases} \mathbb{A}_T(t_0, (i \text{ div } 2)) & \text{if } i \bmod 2 = 0 \\ \mathbb{A}_T(t_1, (i \text{ div } 2)) & \text{otherwise} \end{cases}$$

This split matches up to the example array split shown in Equation (2).

The second split that we will look at is called the *k-block split*, which we will write as $b(k)$ for a constant $k :: \mathbb{N}$. For this split, the first split component contains the elements which has an index less than k and the second contains the rest. The choice function is defined as

$$ch(i) = \begin{cases} 0 & \text{if } i < k \\ 1 & \text{otherwise} \end{cases}$$

and the family of functions $\mathcal{F} = \{f_0, f_1\}$ where

$$\begin{aligned} f_0 &= (\lambda i.i) \\ \text{and } \ f_1 &= (\lambda i.i - k) \end{aligned}$$

Note that we could write $\mathcal{F} = \{f_p = (\lambda i.i - k \times ((p+1) \bmod 2)) \,|\, p \in \{0,1\}\}$ and $ch(i) = sgn(i \text{ div } k)$ where sgn is the *signum* function, *i.e.*

$$sgn(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Since we assume that $i, k \in \mathbb{N}$ then $i \text{ div } k \geq 0$ and so $sgn(i \text{ div } k) \neq -1$. We can easily see that each f_i is injective. We can define an access function for this split as follows:

$$\mathbb{A}_{T_{b(k)}}(\langle t_0, t_1 \rangle_{b(k)}, i) = \begin{cases} \mathbb{A}_T(t_0, i) & \text{if } i < k \\ \mathbb{A}_T(t_1, (i-k)) & \text{otherwise} \end{cases}$$

A fuller discussion of these splits can be found in [6].

3.4 Splits and Operations

Suppose that we have an indexed data-type $D(X)$ and an operation g of arity p and elements $x^1, \ldots, x^p$ of type $D(X)$ which we split with respect to a split sp, so that:

$$x^e \quad \rightsquigarrow \quad \langle x^e_0, x^e_1, \ldots, x^e_{n-1} \rangle_{sp} \quad \text{for } e :: [1..p]$$

Suppose that we want to compute $g(x^1, \ldots, x^p)$. Is it possible to express each of the components of the split of this result in terms of exactly one of the split components from each of our p elements? That is, we would like

$$g(x^1, \ldots, x^p) \rightsquigarrow \langle g(x^1_{\theta^1(0)}, \ldots, x^p_{\theta^p(0)}), \ldots, g(x^1_{\theta^1(n-1)}, \ldots, x^p_{\theta^p(n-1)}) \rangle_{sp}$$

for some family of permutations on $[0..n)$, $\{\theta^e\}_{e::[1..p]}$. This can be achieved if we can find a function $h :: X^p \to Y$ and a family of functions $\{\phi^e\}_{e::[1..p]}$, where for each e

$$\phi_e :: I_{x_e} \to I_{x_e}$$

and for all $i \in I_x$

$$\mathbb{A}_D(y, i) = h(\mathbb{A}_D(x^1, \phi^1(i)), \ldots, \mathbb{A}_D(x^p, \phi^p(i))) \tag{7}$$

where $y = g(x^1, \ldots, x^p)$.

Theorem 1. ***Function Splitting Theorem***
Suppose that the elements $x^1, \ldots, x^p$ of a data-type $D(X)$ are split for some split $sp = (ch, \mathcal{F})$. Let g be an operation $g :: D(X)^p \to D(Y)$ with element y and functions h and $\{\phi^e\}_e$ as above in Equation (7). If there exists a family of functions $\{\theta^e\}_{e::[1..p]}$, such that for each e:

$$\theta^e \cdot ch = ch \cdot \phi^e \tag{8}$$

and if each ϕ^e satisfies:

$$\phi^e(f_k\,(i)) = f_{\theta^e(k)}\,(\phi^e(i)) \tag{9}$$

where $k = ch(i)$ and $f_k, f_{\theta(k)} \in \mathcal{F}$ and if

$$y \rightsquigarrow \langle y_0, y_1, \ldots, y_{n-1} \rangle_{sp}$$

then, for each split component of y (with respect to sp)

$$(\forall\ i)\ y_k = g(x^1_{\theta^1(k)}, \ldots, x^p_{\theta^p(k)}) \text{ where } k = ch(i)$$

Proof. Pick $i \in I_x$, let $k = ch(i)$ and then consider $\mathbb{A}_D(y^k, f_k(i))$.

$$\begin{array}{ll}
 & \mathbb{A}_D(y_k, f_k(i)) \\
= & \quad \{\text{split relationship (6)}\} \\
 & \mathbb{A}_D(y, i) \\
= & \quad \{\text{Equation (7)}\} \\
 & h\,(\mathbb{A}_D(x^1, \phi^1(i)), \ldots, \mathbb{A}_D(x^p, \phi^p(i))) \\
= & \quad \{\text{split relationship (6) with } k_e = ch(\phi^e(i))\} \\
 & h\,(\mathbb{A}_D(x^1_{k_1}, f_{k_1}(\phi^1(i))), \ldots, \mathbb{A}_D(x^p_{k_p}, f_{k_p}(\phi^p(i)))) \\
= & \quad \{\text{Property (8), } k_e = ch(\phi^e(i)) = \theta^e(ch(i)) = \theta^e(k)\} \\
 & h\,(\mathbb{A}_D(x^1_{\theta^1(k)}, f_{\theta^1(k)}\,(\phi^1(i))), \ldots, \mathbb{A}_D\ (x^p_{\theta^p(k)}, f_{\theta^p(k)}\,(\phi^p(i)))) \\
= & \quad \{\text{Property (9)}\} \\
 & h\,(\mathbb{A}_D(x^1_{\theta^1(k)}, \phi^1(f_k\ (i))), \ldots, \mathbb{A}_D(x^p_{\theta^p(k)}, \phi^p(f_k\ (i)))) \\
= & \quad \{\text{definition of } g \text{ in terms of } h \text{ and } \phi^e, \text{ Equation (7)}\} \\
 & \mathbb{A}_D(g(x^1_{\theta^p(k)}, \ldots, x^p_{\theta^p(k)}), f_k(i))
\end{array}$$

Thus

$$(\forall\ i)\ y_k = g(x^1_{\theta^1(k)}, \ldots, x^p_{\theta^p(k)}) \text{ where } k = ch(i)$$

□

$List\ \alpha ::= Empty \mid Cons\ \alpha\ (List\ \alpha)$
$(:) :: \alpha \to List\ \alpha \to List\ \alpha$
$\lvert\_\rvert :: List\ \alpha \to \mathbb{N}$
$\mathsf{head} :: List\ \alpha \to \alpha$
$\mathsf{tail} :: List\ \alpha \to List\ \alpha$
$(+\!\!+) :: List\ \alpha \to List\ \alpha \to List\ \alpha$
$\mathsf{map} :: (\alpha \to \beta) \to List\ \alpha \to List\ \beta$
$\mathsf{filter} :: (\alpha \to \mathbb{B}) \to List\ \alpha \to List\ \alpha$

Figure 1: Data-Type for finite lists

In Section 5.2 we show how this theorem can be used to define a transposition operation for a particular matrix split.

4 List Splitting

Our data-type for finite lists is given in Figure 1 This data-type is based on the list data-type in Haskell. We will use the normal Haskell shorthand for list viz. for $Empty$ we write [ ] and we write $Cons\ 1\ (Cons\ 2\ Empty)$ as $1 : (2 : [\,])$ or just $[1, 2]$ and so $x : xs \equiv Cons\ x\ xs$. We will write $|xs|$ instead of $\mathsf{length}\ xs$. Note that head and tail are defined for non-empty finite lists while the rest are defined for all finite lists. For the sake of conciseness in this paper we do not consider all of the operations in the data-type — just the ones that have some particular interest. A full discussion of this data-type can be found in [6, Chapter 4].

Let us suppose that we want to split the list xs into two lists l and r (the split components). As before, we write $xs \rightsquigarrow \langle l, r\rangle$ to denote that the list xs can be obfuscated by a split list $\langle l, r\rangle$. Suppose that we have a split $sp = (ch, \mathcal{F})$. As the list indexing operation (!!) is expensive to use, for Haskell list splitting we will define two functions split_{sp} (the *splitting function*) and $\mathsf{unsplit}_{sp}$ (which is the inverse of split_{sp}), such that

$$xs \rightsquigarrow \langle l, r\rangle_{sp} \quad \Leftrightarrow \quad \mathsf{split}_{sp}(xs) = \langle l, r\rangle_{sp} \ \wedge \ \mathsf{unsplit}_{sp}(\langle l, r\rangle_{sp}) = xs$$

and

$$\mathsf{unsplit}_{sp} \ \cdot \ \mathsf{split}_{sp} = id$$

These two functions correspond to the obfuscation and abstraction functions for this refinement. For any list split we must ensure that split_{sp} performs the same split as $(ch, \mathcal{F})$.

By the Function Splitting Theorem (Theorem 1), we can define $\mathsf{map}_{sp}\ f$ for any split sp. For map:

$$(\mathsf{map}\ f\ xs)\ !!\ n = f(xs\ !!\ n)$$

If we take $h = f$ and $\phi^1 = id$ then the conditions for the theorem are satisfied. So

$$\mathsf{map}_{sp}\ f\ \langle x_1, \ldots, x_n\rangle_{sp} = \langle \mathsf{map}\ f\ x_1, \ldots, \mathsf{map}\ f\ x_n\rangle_{sp}$$

4.1 The Alternating Split

We will now define the alternating split (defined in Section 3.3) for lists. Thus we would like

$$[4,8,3,1,8,7,6] \rightsquigarrow \langle [4,3,8,6], [8,1,7]\rangle_{asp}$$

The representation $xs \rightsquigarrow \langle l, r\rangle_{asp}$ satisfies the following invariant:

$$dti \equiv |r| \leq |l| \leq |r| + 1$$

We define a splitting function as follows:

$$\begin{aligned} \mathsf{split}_{asp}\,[\,] &= \langle [\,], [\,]\rangle_{asp} \\ \mathsf{split}_{asp}\,(a : xs) &= \langle a : r, l\rangle_{asp} \\ &\qquad \text{where } \langle l, r\rangle_{asp} = \mathsf{split}_{asp}\ xs \end{aligned}$$

and here is the abstraction function:

$$\begin{aligned} \mathsf{unsplit}_{asp}\,\langle [\,], [\,]\rangle_{asp} &= [\,] \\ \mathsf{unsplit}_{asp}\,\langle a : r, l\rangle_{asp} &= a : \mathsf{unsplit}_{asp}\,\langle l, r\rangle_{asp} \end{aligned}$$

These two functions must satisfy Equation (4) and so we require that

$$\mathsf{unsplit}_{asp} \cdot \mathsf{split}_{asp} = id$$

A proof can be found in [6, Chapter 4]. For this split, we can also prove that:

$$\mathsf{split}_{asp} \cdot \mathsf{unsplit}_{asp} = id$$

This means that $\mathsf{unsplit}_{asp}$ is a bijective abstraction function.

To ensure that our definition of split_{asp} matches the $(ch, \mathcal{F})$ formulation (stated in Section 3.3), we have to prove the following (a proof is given in [6]).

Property 1. *If* $\mathsf{split}_{asp}\ xs = \langle l, r\rangle_{asp}$ *then* $(\forall\ n :: \mathbb{N})\ \bullet\ n < |xs|$

$$xs\,!!\,n = \begin{cases} l\,!!\,(n\ \mathit{div}\ 2) & \textit{if } n \textit{ is even} \\ r\,!!\,(n\ \mathit{div}\ 2) & \textit{otherwise} \end{cases}$$

where xs *is a finite list.*

We define a function cons_{asp}

$$\mathsf{cons}_{asp}\ a\ \langle l, r\rangle_{asp} = \langle a : r, l\rangle_{asp}$$

which satisfies

$$\mathsf{split}_{asp}\ (a : xs) = \mathsf{cons}_{asp}\ a\ (\mathsf{split}_{asp}\ xs)$$

The function cons_{asp} for split lists corresponds to (:) for standard Haskell lists. We can prove this by using Equation (5), so for some element a:

$$((:)\ a) \cdot \mathsf{unsplit}_{asp} = \mathsf{unsplit}_{asp} \cdot (\mathsf{cons}_{asp}\ a)$$

Since split_{asp} is an inverse for $\mathsf{unsplit}_{asp}$ then we obtain:

$$\mathsf{cons}_{asp}\ a = \mathsf{split}_{asp} \cdot ((:)\ a) \cdot \mathsf{unsplit}_{asp}$$

We can prove this by induction and the details are omitted here.

As well as proving the correctness of split list operations, we can also derive definitions for split operations. For example, [6, Chapter 4] shows how to derive a concatenation operation $+\!\!+_{asp}$ for the alternating split using the equation:

$$xsp +\!\!+_{asp} ysp = \mathsf{split}_{asp}\ ((\mathsf{unsplit}_{asp}\ xsp)\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp))$$

Using this equation, we obtain the following definition:

$$\begin{aligned} \langle [\,], [\,]\rangle_{asp} +\!\!+_{asp}\ ysp &= ysp \\ \langle x : r_0, l_0\rangle_{asp} +\!\!+_{asp}\ ysp &= \mathsf{cons}_{asp}\ x\ (\langle l_0, r_0\rangle_{asp} +\!\!+_{asp}\ ysp) \end{aligned}$$

The operations for the alternating split have similar efficiencies to the unsplit list operations.

4.2 Block Split

The *k-block split* (written $b(k)$) — where $k \in \mathbb{N}$ is a constant — splits a list so that the first component contains the first k elements of the list and the second component contains the rest. For this split we must determine the value of k before the split is performed and we need to keep the value constant. The value of k determines how the list is split — we call such a value the *decision* value for a split. For instance,

$$[4, 3, 1, 1, 5, 8, 2] \rightsquigarrow \langle [4, 3, 1], [1, 5, 8, 2]\rangle_{b(3)}$$

The representation $xs \rightsquigarrow \langle l, r\rangle_{b(k)}$ satisfies the invariant:

$$dti \equiv (|r| = 0 \wedge |l| < k) \vee (|l| = k)$$

We can see that if the list xs has at most k elements then $xs \rightsquigarrow \langle xs, [\,]\rangle_{b(k)}$.

As with the alternating split, rather than using !!, we define a function that splits a list:

$$\begin{aligned}
&\mathsf{split}_{b(k)}\ [\,] &&= \langle [\,], [\,] \rangle_{b(k)} \\
&\mathsf{split}_{b(0)}\ xs &&= \langle [\,], xs \rangle_{b(0)} \\
&\mathsf{split}_{b(k)}\ (x : xs) &&= \langle x : l, r \rangle_{b(k)} \\
& && \quad \text{where } \langle l, r \rangle_{b(k-1)} = \mathsf{split}_{b(k-1)}\ xs
\end{aligned}$$

To ensure that our definition of $\mathsf{split}_{b(k)}$ matches up with the $(ch, \mathcal{F})$ formulation stated in Section 3.3, we have to prove the following (a proof is given in [6, Chapter 4]).

Property 2. *Let xs be a non-empty list and suppose $n :: \mathbb{N}$. If $\mathsf{split}_{b(k)}\ xs = \langle l, r \rangle_{b(k)}$ then $(\forall\ n :: \mathbb{N}) \bullet n < |xs|$*

$$xs\,!!\,n = \begin{cases} l\ !!\ n & \text{if } n < k \\ r\ !!\ (n-k) & \text{otherwise} \end{cases}$$

For the abstraction function, we can define:

$$\mathsf{unsplit}_{b(k)}\ \langle l, r \rangle_{b(k)} \quad = \quad l +\!\!+ r$$

Note that this definition is independent of k.

4.3 Arbitrary List Splitting

When a list is split using the alternating split or the block split it is always split in the same way. Suppose that we split up a finite list arbitrarily so that the list is split differently each time a program is executed. This means that even with the same input list, different program traces are produced on different executions and this helps to confuse an attacker further. So, for a list xs, we would like $xs \rightsquigarrow \langle l, r \rangle$. If we want to do this arbitrarily then an easy way to do so is to provide a random Boolean for each element of xs. We could say that if the value is $True$ for an element of xs then that element should be placed into the list r and if $False$ then into l. So when splitting each list, we need to provide a list of Booleans that tells us how to split the list — such a list can be a decision value for this split. For example, suppose we want to split up the list $[1, 2, 3, 4, 5, 6]$. Then using the list $[F, F, F, T, F, T]$

$$[1, 2, 3, 4, 5, 6] \rightsquigarrow \langle [1, 2, 3, 5], [4, 6] \rangle$$

Instead of providing a list of Booleans, we could use a natural number. If we let T have value 1 and F have value 0 then we can consider the list of Booleans as the binary representation of a natural number. For efficiency, we will consider the least significant bit to be the head of the list. For the example above, $[F, F, F, T, F, T]$ has the value 40.

To be able to use the split list, we will need to know how it has been split and so we have to carry around the decision value. To do this we create a new type called an *augmented split list* which contains a decision value (of type β) as well as the split lists:

$$ASpList\ \alpha ::= \langle \beta, List\ \alpha, List\ \alpha \rangle_A$$

For our examples, we take $\beta = \mathbb{N}$. So,

$$[1,2,3,4,5,6] \rightsquigarrow \langle 40, [1,2,3,5], [4,6] \rangle_A$$

We now need to consider how to implement a "cons" operation for this split. When adding a new element to the front of an augmented split list, we need to indicate whether the value should be added to the first or to the second list. So our cons operation will also need to take in a random bit which decides into which list we add to. If $m \in \{0,1\}$ then we can define:

$$\begin{aligned} \mathsf{cons}_A\ m\ x\ \langle d,l,r\rangle_A = \ & \text{if } m == 0 \text{ then } \langle n, (x:l), r\rangle_A \\ & \text{else } \ \langle n, l, (x:r)\rangle_A \\ & \text{where } n = (2 \times d) + m \end{aligned}$$

We would like two functions split_A and $\mathsf{unsplit}_A$ that satisfy the following property

$$\begin{aligned} xs \rightsquigarrow \langle n,l,r\rangle_A \Leftrightarrow\ & \mathsf{split}_A\ n\ xs = \langle n,l,r\rangle_A \ \ \wedge \\ & \mathsf{unsplit}_A\ \langle n,l,r\rangle_A = xs \end{aligned}$$

Note that for this split we will take $dti \equiv True$.

Using the definition of cons_A, we can easily define a splitting function, split_A:

$$\begin{aligned} \mathsf{split}_A\ n\ [\,] &= \langle n, [\,], [\,]\rangle_A \\ \mathsf{split}_A\ n\ (x:xs) &= \mathsf{cons}_A\ m\ x\ (\mathsf{split}_A\ d\ xs) \\ &\quad \text{where } (d,m) = \mathsf{divMod}\ n\ 2 \end{aligned}$$

Again we can define this split using $(ch, \mathcal{F})$ — however, we need to change the types of the functions so that they take in an extra parameter n. We can define a choice function as follows:

$$\begin{aligned} ch(0,n) &= n \bmod 2 \\ ch(i,n) &= ch(i-1, n \text{ div } 2) \end{aligned}$$

and we can define f_i as follows:

$$f_k(t,n) = |\{i \mid 0 \le i < t \wedge ch(i,n) == k\}|$$

Property 3. *Let xs be a non-empty list and suppose $n :: \mathbb{N}$. If $\mathsf{split}_A\ xs = \langle n,l,r\rangle_A$ then $(\forall\ s :: \mathbb{N})\ \bullet\ s < |xs|$*

$$xs\ !!\ s = \begin{cases} l\ !!\ f_0\,(s,n) & \text{if } ch(s,n) == 0 \\ r\ !!\ f_1\,(s,n) & \text{otherwise} \end{cases}$$

The proof that this formulation matches the split function is given in [6, Chapter 4]. The abstraction function can be defined as follows:

$$\begin{aligned}\mathsf{unsplit}_A\ \langle n, [\,], [\,]\rangle_A &= [\,]\\ \mathsf{unsplit}_A\ xsp &= h : (\mathsf{unsplit}_A\ t)\\ &\qquad \text{where } (b, h, t) = \mathsf{headTail}_A\ xsp\end{aligned}$$

where

$$\begin{aligned}\mathsf{headTail}_A\ \langle n, l, r\rangle_A = \text{if } m == 0 \text{ then } &(m, \mathsf{head}\ l, \langle d, \mathsf{tail}\ l, r\rangle_A)\\ \text{else } &(m, \mathsf{head}\ r, \langle d, l, \mathsf{tail}\ r\rangle_A)\\ &\text{where } (d, m) = \mathsf{divMod}\ n\ 2\end{aligned}$$

The operation headTail has type $ASpList\ \alpha \to (\mathbb{N}, \alpha, ASpList\ \alpha)$ and satisfies:

$$xsp = \mathsf{cons}_A\ b\ h\ t \ \Leftrightarrow\ (b, h, t) = \mathsf{headTail}_A\ xsp$$

5 Matrices

As an alternative to lists we consider *matrices* and we briefly develop some splits for this data-type. A matrix $\mathbf{M}$ which has r rows and c columns with elements of type α will be denoted by $\mathbf{M}^{r\times c}(\alpha)$. We write $\mathbf{M}(i, j)$ to denote the access function which denotes the element located at the ith row and the jth column. An indexer for a matrix $\mathbf{M}^{r\times c}$ is $[0..r) \times [0..c)$. Thus since matrices are examples of IDTs, we can perform splits. For simplicity, we assume the our element type is $\mathbb{Z}$ and so we simply write $\mathbf{M}$ instead of $\mathbf{M}(\mathbb{Z})$.

We can model matrices in Haskell using a list of lists. Not all lists of lists represent a matrix: mss represents a matrix if and only if all the members of mss are lists of the same length. We can define a function valid that checks whether a list of lists is a valid matrix representation.

$$\begin{aligned}\mathsf{valid}\ [mss] &= True\\ \mathsf{valid}\ (ms : ns : mss) &= (|ms| == |ns|)\ \wedge\ \mathsf{valid}\ (ns : mss)\end{aligned}$$

We represent $\mathbf{M}^{r\times c}$ by a list of lists mss where $|mss| = r$ and each list in mss has length c. Let us suppose that we want to define multiplication by a scalar (scale), addition ($+$), transposition (^{T}), multiplication ($\times$) and an indexer (!!!).

For addition, the two matrices must have the same size and for multiplication we need the matrices to be *conformable*, *i.e.* the number of columns of the first is equal to the number of rows in the second. We can specify the

operations point-wise as follows:

$$\begin{aligned}(\mathsf{scale}\ s\ \mathbf{M})(i,j) &= s \times \mathbf{M}(i,j)\\ (\mathbf{M}+\mathbf{N})(i,j) &= \mathbf{M}(i,j)+\mathbf{N}(i,j)\\ (\mathbf{M}^T)(i,j) &= \mathbf{M}(j,i)\\ (\mathbf{M}\times\mathbf{P})(i,k) &= \textstyle\sum_{j=1}^{c}(\mathbf{M}(i,j)\times\mathbf{P}(j,k))\\ \mathbf{M}\ !!!\ (i,j) &= \mathbf{M}(i,j)\end{aligned}$$

for matrices $\mathbf{M}^{r\times c}$, $\mathbf{N}^{r\times c}$ and $\mathbf{P}^{c\times d}$ with $i :: [0..r)$, $j :: [0..c)$ and $k :: [0..d)$. The Haskell definitions for these operations are given in [6, Chapter 6].

We assume that basic arithmetic operations take constant time and so the computational complexities of $\mathbf{M}+\mathbf{N}$, $\mathsf{scale}\ s\ \mathbf{M}$ and $\mathbf{M}^T$ are all $r\times c$ and the complexity of $\mathbf{M}\times\mathbf{P}$ is $r\times c\times d$. In fact, to reduce the number of multiplications in the calculation of $\mathbf{M}\times\mathbf{P}$, we could use *Strassen's algorithm* [4, Section 28.2] which performs matrix multiplication by establishing simultaneous equations. The algorithm requires starting with a $2n\times 2n$ matrix and splitting it into four $n\times n$ matrices but by padding a matrix with zeros, this method can be adapted for more general matrices. We should ensure that when we obfuscate these operations we do not change the complexity.

5.1 Splitting Matrices

Since matrices form an IDT we can use the Function Splitting Theorem (Theorem 1 from Section 3.4). Can we express our matrix operations using functions h and ϕ^e? For scale $(\times s)$, we can take $h = (\times s)$ and $\phi^1 = id$; for $(^T)$, $h = id$ and $\phi^1(i,j) = (j,i)$ and for $(+)$, we can take $h = (+)$ and $\phi^1 = id = \phi^2$. We cannot define $(\times)$ using h and ϕ^e — in the next section, we use a split in which the components of the split of $\mathbf{A}\times\mathbf{B}$ are calculated using two components from the split of $\mathbf{A}$ and two from the split of $\mathbf{B}$.

For $(+)$ and scale the ϕ functions are equal to id (as are the θ functions) and so Equation (9) is satisfied for any split. If, for some split sp,

$$\begin{aligned}\mathbf{A} &\rightsquigarrow \langle \mathbf{A_0},\ldots,\mathbf{A_{n-1}}\rangle_{sp}\\ \mathbf{B} &\rightsquigarrow \langle \mathbf{B_0},\ldots,\mathbf{B_{n-1}}\rangle_{sp}\end{aligned}$$

then

$$\begin{aligned}\mathbf{A}+_{sp}\mathbf{B} &= \langle \mathbf{A}_0+\mathbf{B}_0,\ldots,\mathbf{A}_{n-1}+\mathbf{B}_{n-1}\rangle_{sp}\\ \mathsf{scale}_{sp}\ s\ \mathbf{A} &= \langle \mathsf{scale}\ s\ \mathbf{A}_0,\ldots,\mathsf{scale}\ s\ \mathbf{A}_{n-1}\rangle_{sp}\end{aligned}$$

5.1.1 Splitting in squares

A simple matrix split is one which splits a square matrix into four matrices — two of which are square. Using this split we can give definitions of our operations for split matrices. Suppose that we have a square matrix $\mathbf{M}^{r\times r}$

and choose a positive integer k such that $k < n$. The choice function $ch(i,j)$ is defined as

$$ch(i,j) = \begin{cases} 0 & (0 \le i < k) \wedge (0 \le j < k) \\ 1 & (0 \le i < k) \wedge (k \le j < r) \\ 2 & (k \le i < n) \wedge (0 \le j < k) \\ 3 & (k \le i < n) \wedge (k \le j < r) \end{cases}$$

which can be written as a single formula

$$ch(i,j) = 2\, sgn\,(i \text{ div } k) + sgn\,(j \text{ div } k)$$

The family of functions $\mathcal{F}$ is $\{f_0, f_1, f_2, f_3\}$ where

$$\begin{array}{ll} & f_0 = (\lambda\,(i,j)\,.\,(i,j)) \\ & f_1 = (\lambda\,(i,j)\,.\,(i,j-k)) \\ & f_2 = (\lambda\,(i,j)\,.\,(i-k,j)) \\ \text{and} & f_3 = (\lambda\,(i,j)\,.\,(i-k,j-k)) \end{array}$$

Alternatively:

$$f_p = (\lambda\,(i,j)\,.\,(i - k \times (p \text{ div } 2), j - k \times (p \bmod 2))) \text{ where } p \in [0..3]$$

We call this split the (k,k)-*square split* since the first component of the split is a $k \times k$ square matrix. Pictorially, we split a matrix as follows:

$$\left(\begin{array}{ccc|ccc} a_{(0,0)} & \cdots & a_{(0,k-1)} & a_{(0,k)} & \cdots & a_{(0,n-1)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{(k-1,0)} & \cdots & a_{(k-1,k-1)} & a_{(k-1,k)} & \cdots & a_{(k-1,n-1)} \\ \hline a_{(k,0)} & \cdots & a_{(k,k-1)} & a_{(k,k)} & \cdots & a_{(k,n-1)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{(n-1,0)} & \cdots & a_{(n-1,k-1)} & a_{(n-1,k)} & \cdots & a_{(n-1,n-1)} \end{array}\right)$$

So if

$$\mathbf{M}(i,j) = \mathbf{M}_t(f_t(i,j)) \text{ where } t = ch(i,j)$$

then we can write

$$\mathbf{M}^{n\times n} \;\rightsquigarrow\; \langle \mathbf{M}_0^{k\times k}, \mathbf{M}_1^{k\times(n-k)}, \mathbf{M}_2^{(n-k)\times k}, \mathbf{M}_3^{(n-k)\times(n-k)} \rangle_{s(k)}$$

where the subscript $s(k)$ denotes the (k,k)-square split.

For this split

$$\mathbf{M}^T \rightsquigarrow \langle {\mathbf{M}_0}^T, {\mathbf{M}_2}^T, {\mathbf{M}_1}^T, {\mathbf{M}_3}^T \rangle_{s(k)}$$

or, pictorially,

$$\begin{pmatrix} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{pmatrix}^T = \begin{pmatrix} {\mathbf{M}_0}^T & {\mathbf{M}_2}^T \\ {\mathbf{M}_1}^T & {\mathbf{M}_3}^T \end{pmatrix}$$

This operation has complexity $n \times n$.

What is the definition of !!! for this split? To access an element of a split matrix, first we decide which component we need and then what position. We propose the following definition:

$$\begin{array}{l} \langle \mathbf{M}_0, \mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3 \rangle_{s(k)} \; !!!_{s(k)} \; (r, c) \\ \quad \left| \begin{array}{lcl} r < k \;\wedge\; c < k & = & \mathbf{M}_0 \; !!! \; (r, c) \\ r < k \;\wedge\; c \geq k & = & \mathbf{M}_1 \; !!! \; (r, c-k) \\ r \geq k \;\wedge\; c < k & = & \mathbf{M}_2 \; !!! \; (r-k, c) \\ r \geq k \;\wedge\; c \geq k & = & \mathbf{M}_3 \; !!! \; (r-k, c-k) \end{array} \right. \end{array}$$

Finally let us consider how we can multiply split matrices. Let

$$\begin{array}{rcl} \mathbf{M}^{n \times n} & \rightsquigarrow & \langle \mathbf{M}_0, \, \mathbf{M}_1, \, \mathbf{M}_2, \, \mathbf{M}_3 \rangle_{s(k)} \\ \mathbf{N}^{n \times n} & \rightsquigarrow & \langle \mathbf{N}_0, \;\; \mathbf{N}_1, \;\; \mathbf{N}_2, \;\; \mathbf{N}_3 \, \rangle_{s(k)} \end{array}$$

By considering the product

$$\begin{pmatrix} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{pmatrix} \times \begin{pmatrix} \mathbf{N}_0 & \mathbf{N}_1 \\ \mathbf{N}_2 & \mathbf{N}_3 \end{pmatrix}$$

we obtain the following result:

$$\begin{array}{rl} \mathbf{M} \times \mathbf{N} \rightsquigarrow & \langle (\mathbf{M}_0 \times \mathbf{N}_0) + (\mathbf{M}_1 \times \mathbf{N}_2), \;\; (\mathbf{M}_0 \times \mathbf{N}_1) + (\mathbf{M}_1 \times \mathbf{N}_3), \\ & (\mathbf{M}_2 \times \mathbf{N}_0) + (\mathbf{M}_3 \times \mathbf{N}_2), \;\; (\mathbf{M}_2 \times \mathbf{N}_1) + (\mathbf{M}_3 \times \mathbf{N}_3) \rangle_{s(k)} \end{array}$$

The computation of $\mathbf{M} \times \mathbf{N}$ using naive matrix multiplication needs n^3 integer multiplications. By adding up the number of multiplications for each of the components, we can see that split matrix multiplication also needs n^3 multiplications.

5.2 Generalising the square split

Suppose that we have a matrix $\mathbf{M}^{k \times k}$ which we want to split into n^2 blocks with the condition that the blocks down the main diagonal are square. We will call this the *n-square matrix split*, denoted by $sq(n)$. For this, we will need a set of numbers $S_0, S_1, \ldots, S_m$ such that $0 = S_0 < S_1 < S_2 < \ldots < S_{n-1} < S_n = k - 1$. We require strict inequality so that we have exactly n^2 blocks with both dimensions of each block at least 1.

The n-square matrix split is defined as follows: $sq(n) = (ch, \mathcal{F})$ such that

$$\begin{array}{rcl} ch & :: & [0..k) \times [0..k) \to [0..n^2) \\ ch(i,j) & = & p\,n + q \text{ where } (S_p \leq i < S_{p+1}) \wedge (S_q \leq j < S_{q+1}) \end{array}$$

and if $f_r \in \mathcal{F}$ then

$$\begin{array}{rcl} f_r & :: & [0..k) \times [0..k) \to [0..S_p - S_{p-1}) \times [0..S_q - S_{q-1}) \\ f_r(i,j) & = & (i - S_p, j - S_q) \text{ where } r = ch(i,j) = pn + q \end{array}$$

An alternative form for the choice function is

$$ch(i,j) = \sum_{t=1}^{n} \Big(n \times sgn(i \text{ div } S_t) + (j \text{ div } S_t) \Big)$$

Note that if $ch(i,j) = pn + q$ then $ch(j,i) = qn + p$. The matrices $\mathbf{M}$ and $\mathbf{M_r}$ are related by the formula

$$\mathbf{M_r}(f_r(i,j)) = \mathbf{M}(i,j) \quad \text{where } r = ch(i,j)$$

We can use the Function Splitting Theorem (Theorem 1) to define transposition. For transpose, $h = id$ and $\phi^1 = \lambda\ (i,j).(j,i)$. We define a permutation function as follows:

$$\theta^1 = \lambda\, s.(n \times (s \bmod n) + (s \text{ div } n))$$

Suppose that $t = ch(i,j) = pn + q$ then $\theta^1(t) = qn + p$. So,

$$\begin{aligned} \theta^1\left(ch(i,j)\right) &= \theta^1\left(pn+q\right) \\ &= qn + p \\ &= ch\left(j,i\right) \\ &= ch\left(\phi^1\left(i,j\right)\right) \end{aligned}$$

and thus Equation (8) is satisfied. Also,

$$\begin{aligned} \phi^1\left(f_t\left(i,j\right)\right) &= \phi^1\left(f_{pn+q}\left(i,j\right)\right) \\ &= \phi^1\left(i - S_p, j - S_q\right) \\ &= \left(j - S_q, i - S_p\right) \\ &= f_{qn+p}\left(j,i\right) \\ &= f_{qn+p}\left(\phi^1\left(i,j\right)\right) \\ &= f_{\theta^1(t)}\left(\phi^1\left(i,j\right)\right) \end{aligned}$$

and thus Equation (9) is satisfied. Hence Theorem 1 applies and so if

$$\mathbf{M} \rightsquigarrow \langle \mathbf{M_0}, \mathbf{M_1}, \ldots, \mathbf{M_n}, \mathbf{M_{n+1}}, \ldots, \mathbf{M_{n^2-1}} \rangle_{sq(n)}$$

then

$$\mathbf{M}^T \rightsquigarrow \langle \mathbf{M_0}^T, \mathbf{M_n}^T, \ldots, \mathbf{M_1}^T, \mathbf{M_{n+1}}^T, \ldots, \mathbf{M_{n^2-1}}^T \rangle_{sq(n)}$$

Without the theorem, the derivation of this obfuscation is much harder.

6 Conclusions and Future Work

The current view of obfuscation concentrates on obfuscating object-oriented languages (or the underlying intermediate representations) and the obfuscations given in [3] focus on concrete data-types such as variables and arrays. It is often difficult to produce more general obfuscations when using such concrete data-types. We have concentrated on a specific array obfuscation taken from Collberg *et al.* [3] called an array split and we have shown how this obfuscation could be generalised. Firstly, we stated that, using the $(ch, \mathcal{F})$ representation, we can specify how the elements of the original arrays are split into the two new arrays. We generalised this specification further by considering how to split the original array into more than two components. We developed an abstract data-type, called an *Indexed Data-Type*, which captured the properties of arrays that are needed for splitting. We then showed how to define "splits" for IDTs which reflect the generalisations developed for split arrays. The use of abstract data-types allowed us prove properties about the operations defined for splits. For list splits we gave the $(ch, \mathcal{F})$ specification and Haskell functions to perform the split (and unsplit) and we can verify that these two notions match up. For our data-types, we can prove that the split operations that we produce are correct and in some cases we can actually derive obfuscated operations. We also proved a theorem which, under certain conditions, provides definitions for split operations. In this paper, we have not stated how "obfuscated" our obfuscations are but a new definition of obfuscation for abstract data-types based upon proving assertions is given in the thesis [6]. In this thesis, different obfuscations of the concatenation operation ($+\!\!+$) are given and it is discussed which of these operations is "more obfuscated' according to this new definition.

Can we split binary trees? Yes, in fact trees turn out to be IDTs since, as we mentioned in Section 3.1, we can take the "index" of a node to be the path from the root. In [7] binary tree splitting is briefly discussed and the following example of a tree split is given. We can split a binary tree at the root node by making the right subtree one split component and the rest of the tree the other. Thus we would have

$$\begin{aligned} Null &\rightsquigarrow \langle Null, Null \rangle \\ Fork\ lt\ v\ rt &\rightsquigarrow \langle Fork\ lt\ v\ Null, rt \rangle \end{aligned}$$

The obfuscations produced using this split have comparable complexities but this refinement was rejected in [7] because the definitions of the obfuscated operations were too similar to the unobfuscated versions. Instead a refinement was explored which exploited the structure of binary trees.

Can other array obfuscations given in [3] be generalised? Another array obfuscation is *array merging* which takes two or more arrays and merges them into a single array. This obfuscation can be specified by considering the

inverse of split. In fact, for lists and matrices, the operation unsplit performs a merge. By considering IDTs, we can also specify and generalise arrays obfuscations such as increasing and decreasing the number of dimensions and permuting the order of the elements.

We have proposed a new approach to obfuscation by studying abstract data-types and considering obfuscation as functional refinement. A benefit of considering abstract data-types is that we have been able to introduce randomness in our obfuscations. We have given an example of an obfuscation that can split a list arbitrarily. In [6, Chapter 4] an example is given in which a list is split so that random elements can be placed in the split components and in [6, Chapter 7] random obfuscations for trees are discussed. Randomness can help confuse an attacker further by creating different program traces on different executions with the same input.

It is important to check that an obfuscation is correct — *i.e.* it preserves the functionality of an operation — but proving correctness is a challenging task. Considering obfuscation as refinement allows us to prove the correctness of *all* our obfuscations of data-type operations. Additionally if the abstraction function is injective then it has a unique inverse with which we can derive obfuscations. Note that the abstraction function acts as a deobfuscation function and so we must hide this function from an attacker to prevent the reconstruction of unobfuscated operations. Using simple derivational techniques and modelling our operations in Haskell has allowed us to establish correctness easily — this is a real strength of our approach. The example operations and obfuscations that we have given have been made simple in order to demonstrate our obfuscation techniques. It seems clear that more intricate, realistic obfuscations can be developed similarly for other abstract data-types.

We have seen that this our approach has allowed us to produce new obfuscations and we can easily prove properties (such as correctness) about our obfuscations. However, this proposal (and the array generalisation) needs further research. One area for future work is to find out how "strong" our obfuscations are and so the resilience of our obfuscations to reverse engineering and deobfuscators should be explored. One simple technique for reverse engineering is to execute the program and then study the program traces. If random obfuscations are used then different program traces can be created. Another technique which can be used is *refactoring* [8] which is the process of improving the design of existing code by performing behaviour-preserving transformations. One particular area to explore is the refactoring of functional programs — [10] gives a discussion of a tool for refactoring Haskell called *HaRe*. What happens to our obfuscations if they are refactored? Could HaRe be used to specify obfuscations and so can our obfuscations be automated easily?

In the Introduction (Section 1) we stated two aims which were to make the process of creating obfuscations easier and to have the ability to gener-

alise our obfuscations. The use of functional programming, data refinement and abstract data-types has meant that we are able to easily specify obfuscations but has our approach been general enough? For more generality *categories* could be obfuscated and concepts such as hylomorphisms [11] could be used. Barak *et al.* [1] provide a formal cryptographic treatment of obfuscation by obfuscating Turing machines. How does our approach compare with their definition?

Acknowledgements

Thanks to Dr Jeff Sanders at the Oxford University Computing Laboratory for his helpful comments on this paper.

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

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

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

[5] 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.

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

[7] Stephen Drape. Using Haskell to Model Tree Obfuscations. Technical Report PRG-RR-04-17, Programming Research Group, Oxford University Computing Laboratory, July 2004.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2000.

[9] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 283–294. ACM Press, 2002.

[10] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM Press, 2003.

[11] 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.

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

[13] J. M. Spivey. *The Z notation: a reference manual (Second Edition)*. Prentice Hall, 1992.

[14] Simon Thompson. *Haskell: the Craft of Functional Programming (Second Edition)*. International Computer Science Series. Addison-Wesley, March 1999.