Programming Research Group

THE MATRIX OBFUSCATED

Stephen Drape

PRG-RR-04-12

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 more difficult.

This paper considers a fresh approach to obfuscation by considering the operations of a data-type, which we model as functional programs. Obfuscation is mainly applied to object-oriented programs and we can view an *object* as a data-type and *methods* as the operations. We study matrices and the implementation of some operations using functional lists.

Constructing obfuscations for imperative programs usually requires expensive program analysis, but our method allows us to derive obfuscations directly. Whilst establishing the correctness of imperative obfuscations can be a challenging task, our approach enables this to be achieved easily. Our derivations are aided by taking advantage of standard functional programming results such as fold fusion.

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 matrix data-types which exploit properties of matrices.

1 Introduction

An *obfuscation* is a program transformation whose aim is 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 [1, 3]. Currently obfuscation is mainly applied to 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.

The focus of the paper consists of an object[1] for finite integer matrices having four operations: scalar multiplication, addition, transposition and multiplication, specified mathematically. The current view of obfuscation concentrates on concrete data structures such as variables and arrays. With that view, the process of obfuscating a matrix would require flattening the matrix and then obfuscating the resulting array. The extra information contained within the matrix — *i.e.* its structure — would be lost and so it would be difficult to perform operations that require knowledge of that structure; see Section 2.2 for an example.

We take the view that, instead, we should obfuscate abstract data-types. Thus when we use a data structure we state which operations we want to implement so that we obfuscate the data-type according to these operations. We show how to adapt an array obfuscation to work with matrices and we use data refinement [6] to represent a matrix as a "split matrix" — the source of obfuscation. Data refinement techniques then allow us to derive implementations of the operations that are correspondingly obfuscated. Thus we are guaranteed that our obfuscations are behaviour-preserving.

Barak *et al.* propose a very formal definition of obfuscation and then show that obfuscation is impossible [1]. We use an informal notion of obfuscation — namely that a

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

function is more obfuscated if it is "harder to understand". In functional programming, 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.

We use functional programs, written in Haskell style, to model our data-type and, in particular, a matrix is represented by a list of lists. We aim not to obfuscate functional programs but to use a functional language to model our operations. Our programs are all functional and so their derivations or verifications are affected in a functional formalism. We take the view that the derivation of imperative implementations is often, and certainly in the cases considered here, best achieved by initial high-level functional steps.

The benefits of our (apparently new) approach are the usual elegance of the functional style and the consequent abstraction of side-effects. Thus our functional approach will provide support for purely imperative obfuscations. In an imperative context, proofs of correctness are frequently difficult, typically requiring language restrictions. We therefore consider not only *derivations* of our obfuscated implementations but also an example of *verification*. By using functional programs, we can rely on established results to aid our derivations. In particular, we will write many of our functions using folds and so we are able to use properties relating to fold fusion.

The rest of the paper is structured as follows. Section 2 introduces the obfuscation that we are going to use and the operations that we are interested in implementing. Section 3 discusses how to represent matrices and their operations using a functional language. Section 4 develops obfuscated matrix operations from the corresponding unobfuscated ones. Section 5 shows some possible extensions of our results and some conclusions are given in Section 6.

2 Splitting Matrices

The matrix $\mathbf{M}$ which has r rows and c columns (for natural numbers r and c) will be denoted by $\mathbf{M}^{r \times c}$. The element of $\mathbf{M}$ that is located at row i and column j will be written as $\mathbf{M}(i,j)$, and, for simplicity, assumed to be an integer. The operation $\mathsf{dim}\,(\mathbf{M})$ returns the dimensions of $\mathbf{M}$. We often use intervals of integers; we write, for example, $[a..b)$ to indicate the set of integers greater than or equal to a and strictly less than b.

We would like to obfuscate matrices with the following matrix operations: *scalar multiplication*, *addition*, *transposition* and *multiplication*

$$
\begin{array}{rcl}
\mathsf{scale} & :: & Integer \rightarrow Matrix \rightarrow Matrix \\
\mathsf{add} & :: & (Matrix, Matrix) \rightarrow Matrix \\
\mathsf{transpose} & :: & Matrix \rightarrow Matrix \\
\mathsf{mult} & :: & (Matrix, Matrix) \rightarrow Matrix
\end{array}
$$

Note that 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 define the operations pointwise as follows:

$$\begin{array}{rcl}
(\mathsf{scale}\ s\ \mathbf{M})(i,j) & = & s \times \mathbf{M}(i,j) \\
(\mathsf{add}\ (\mathbf{M},\mathbf{N}))(i,j) & = & \mathbf{M}(i,j) + \mathbf{N}(i,j) \\
(\mathsf{transpose}\ \mathbf{M})(i,j) & = & \mathbf{M}(j,i) \\
(\mathsf{mult}\ (\mathbf{M},\mathbf{P}))(i,k) & = & \sum_{j=1}^{c} \mathbf{M}(i,j) \times \mathbf{P}(j,k)
\end{array}$$

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 $k : [0..d)$.

We assume that basic arithmetic operations take constant time and so the computational complexities of $\mathsf{add}\ \mathbf{M}\ \mathbf{N}$, $\mathsf{scale}\ s\ \mathbf{M}$ and $\mathsf{transpose}\ \mathbf{M}$ are all $r\times c$ and the complexity of $\mathsf{mult}\ \mathbf{M}\ \mathbf{P}$ is $r\times c\times d$. We must ensure that when we obfuscate these operations we do not change the complexity.

2.1 Defining a matrix split

In this section we develop an obfuscation for use with matrices. We achieve this by adapting an array obfuscation — *array splitting*. Examples of array splits are given in [3] and the general form for splitting an array into two arrays is given in [8]. As our aim is to exploit properties of the data-type, we must ensure that we can recover these properties, if we so wish, from the obfuscated data-type.

Suppose that we want to split a matrix $\mathbf{M}^{r\times c}$ into n matrices (called the *split components*)

$$\mathbf{M} \rightsquigarrow \langle \mathbf{M}_0, \ldots, \mathbf{M}_{n-1} \rangle_{sp}$$

where $\mathbf{M}_i$ has size $r_i \times c_i$ for $i : [0..n)$.

In that data representation, $\mathbf{M}$ is represented by n matrices using a split called sp. It consists of a *choice function*:

$$ch :: [0..r) \times [0..c) \to [0..n)$$

and a family $\mathcal{F}$ of injective functions where $\mathcal{F} = \{f_t\}_{t:[0..n)}$ such that for each t:

$$f_t :: ch^{-1}\{t\} \rightarrowtail [0..r_t) \times [0..c_t)$$

We define the relationship between $\mathbf{M}$ and the split components element-wise by using the choice function and the appropriate function from $\mathcal{F}$ to decide where an element is mapped to:

$$\mathbf{M}_t(f_t(i,j)) = \mathbf{M}(i,j) \text{ where } t = ch(i,j) \tag{1}$$

The requirement that we have a family of injective functions ensures that we can recover a matrix (and thus its properties) from the split components.

As an example, consider how we could define a split in which a matrix $\mathbf{M}^{r\times 2c}$ is split vertically into two matrices $\mathbf{M}_0^{r\times c}$ and $\mathbf{M}_1^{r\times c}$. The choice function is defined to be

$$ch(i,j) = j \text{ div } c$$

and the family of functions is:

$$\mathcal{F} = \{f_t = (\lambda\,(i,j)\,.\,(i, j \bmod c)) \mid t = 0 \vee t = 1\}$$

Suppose that we have a matrix operation g with arity p and matrices $\mathbf{A}^1, \ldots, \mathbf{A}^p$ that we split with respect to a split sp, so that:

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

Suppose that we want to compute $g(\mathbf{A}^1, \ldots, \mathbf{A}^p)$ — 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 matrices? That is, we would like

$$\begin{aligned} & g(\mathbf{A}^1, \ldots, \mathbf{A}^p) \rightsquigarrow \\ & \qquad \langle g(\mathbf{A}^1_{\theta^1(0)}, \ldots, \mathbf{A}^p_{\theta^p(0)}), \ldots,\; g(\mathbf{A}^1_{\theta^1(n-1)}, \ldots, \mathbf{A}^p_{\theta^p(n-1)})\rangle_{sp} \end{aligned}$$

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

$$h :: \mathbb{Z}^p \rightarrow \mathbb{Z}$$

and a family of functions $\{\phi^e\}_{e:[1..p]}$, where for each e

$$\phi_e :: [0..r) \times [0..c) \rightarrow [0..r) \times [0..c)$$

and for all (i,j)

$$\mathbf{C}(i,j) = h\,(\mathbf{A}^1(\phi^1(i,j)), \ldots, \mathbf{A}^p(\phi^p(i,j)))$$

where $\mathbf{C} = g(\mathbf{A}^1, \ldots, \mathbf{A}^p)$.

Theorem 1 (Function Splitting Theorem). Suppose that the matrices $\mathbf{A}^1, \ldots, \mathbf{A}^p$ are split for some split $sp = (ch, \mathcal{F})$. Let g be a function

$$g :: \mathit{Matrix}^p \rightarrow \mathit{Matrix}$$

with matrix $\mathbf{C}$ and functions h and $\{\phi^e\}_e$ as above. If there exists a family of functions $\{\theta^e\}_{e:[1..p]}$, such that for each e:

$$\theta^e \circ ch = ch \circ \phi^e \tag{2}$$

and if each ϕ^e satisfies:

$$\phi^e(f_t\,(i,j)) = f_{\theta^e(t)}\,(\phi^e(i,j)) \tag{3}$$

where $t = ch(i,j)$ and $f_t, f_{\theta(t)} \in \mathcal{F}$, then, for each split component of $\mathbf{C}$ (with respect to sp)

$$(\forall\ i,j)\ \mathbf{C}_t = g(\mathbf{A}^1_{\theta^1(t)}, \ldots, \mathbf{A}^p_{\theta^p(t)}) \text{ where } t = ch(i,j)$$

Proof. Pick i and j, let $t = ch(i,j)$ and then consider $\mathbf{C}_t\,(f_t\ (i,j))$.

$$
\begin{aligned}
& \mathbf{C}_t\,(f_t\,(i,j)) \\
= & \quad \{\text{split relationship (1)}\} \\
& \mathbf{C}(i,j) \\
= & \quad \{\text{definition of } \mathbf{C}\} \\
& h\,(\mathbf{A}^1(\phi^1(i,j)), \ldots, \mathbf{A}^p(\phi^p(i,j))) \\
= & \quad \{\text{split relationship (1) with } t_e = ch\,(\phi^e\,(i,j))\} \\
& h\,(\mathbf{A}^1_{t_1}(f_{t_1}(\phi^1(i,j))), \ldots, \mathbf{A}^p_{t_p}(f_{t_p}(\phi^p(i,j)))) \\
= & \quad \{\text{property (2), } t_e = ch(\phi^e(i,j)) = \theta^e(ch(i,j)) = \theta^e(t)\} \\
& h\,(\mathbf{A}^1_{\theta^1(t)}(f_{\theta^1(t)}\,(\phi^1(i,j))), \ldots, \mathbf{A}^p_{\theta^p(t)}(f_{\theta^p(t)}\,(\phi^p(i,j)))) \\
= & \quad \{\text{property (3)}\} \\
& h\,(\mathbf{A}^1_{\theta^1(t)}\,\phi^1(f_t\ (i,j)), \ldots, \mathbf{A}^p_{\theta^p(t)}\,\phi^p(f_t\ (i,j))) \\
= & \quad \{\text{definitions of } h \text{ and } \phi^e\} \\
& g(\mathbf{A}^1_{\theta^p(t)}\,(f_t(i,j)), \ldots, \mathbf{A}^p_{\theta^p(t)}\,(f_t(i,j)))
\end{aligned}
$$

Thus

$$(\forall\ i,j)\ \mathbf{C}_t = g(\mathbf{A}^1_{\theta^1(t)}, \ldots, \mathbf{A}^p_{\theta^p(t)}) \text{ where } t = ch(i,j)$$

□

In this paper, we consider only four matrix operations, although this theorem can be used for other operations, thus providing support for more general matrix data-types.

How can we express our four matrix operations using functions h and ϕ^e? For $\mathsf{scale}\ (\times s)$, we can take $h = (\times s)$ and $\phi^1 = id$; for $\mathsf{transposition}$, $h = id$ and $\phi^1(i,j) = (j,i)$ and for add, we can take $h = (+)$ and $\phi^1 = id = \phi^2$. We cannot define mult using h and ϕ^e — in the next section, we use a split in which the components of the split of $\mathsf{mult}\,(\mathbf{A},\mathbf{B})$ are calculated using two components from the split of $\mathbf{A}$ and two from the split of $\mathbf{B}$.

For add and scale the ϕ functions are equal to id (as are the θ functions) and so equation (3) 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

$$\mathsf{add}_{sp}\ (\mathbf{A},\mathbf{B}) \;=\; \langle \mathsf{add}\ (\mathbf{A}_0,\mathbf{B}_0), \ldots, \mathsf{add}\ (\mathbf{A}_{n-1},\mathbf{B}_{n-1}) \rangle_{sp}$$

and

$$\mathsf{scale}_{sp}\ s\ \mathbf{A} \;=\; \langle \mathsf{scale}\ s\ \mathbf{A}_0, \ldots, \mathsf{scale}\ s\ \mathbf{A}_{n-1} \rangle_{sp}$$

In the next section, we define a split in which we can write definitions of $\mathsf{transpose}$ and mult for split matrices.

2.2 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 four 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

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

which can be written as a single formula

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

where sgn is the signum function. The family of functions $\mathcal{F}$ is defined to be

$$\mathcal{F} = \begin{cases} f_0 & = & (\lambda\,(i,j)\,.\,(i,j)) \\ f_1 & = & (\lambda\,(i,j)\,.\,(i,j-k)) \\ f_2 & = & (\lambda\,(i,j)\,.\,(i-k,j)) \\ f_3 & = & (\lambda\,(i,j)\,.\,(i-k,j-k)) \end{cases}$$

Again, we can write this in a single formula:

$$\mathcal{F} = \{f_p = (\lambda\,(i,j)\,.\,(i - k\ (p \text{ div } 2),\ j - k\ (p \text{ mod } 2))) \mid p \in [0..3]\}$$

We call this split the $(k \times k)$-*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

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

where the subscript s_k denotes the $(k \times k)$-square split.

For example, let

$$\mathbf{E} = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix}$$

Then, using $k = 3$,

$$\mathbf{E} \rightsquigarrow \left\langle \begin{pmatrix} a & b & c \\ f & g & h \\ k & l & m \end{pmatrix}, \begin{pmatrix} d & e \\ i & j \\ n & o \end{pmatrix}, \begin{pmatrix} p & q & r \\ u & v & w \end{pmatrix}, \begin{pmatrix} s & t \\ x & y \end{pmatrix} \right\rangle_{s_3}$$

Using the $(k \times k)$-square split, we can transpose a matrix by considering transposing the individual components of the split. We use Theorem 1, with $g = \mathsf{transpose}$, $h = id$ and $\phi^1(i,j) = (j,i)$, to define transposition for our split. If we take

$$\theta^1(t) = 2\,(t \bmod 2) + (t \text{ div } 2)$$

then we can verify that equations (2) and (3) are satisfied. Hence, Theorem 1 gives us that

$$\mathbf{M}^T \rightsquigarrow \langle {\mathbf{M}_0}^T, {\mathbf{M}_2}^T, {\mathbf{M}_1}^T, {\mathbf{M}_3}^T \rangle_{s_k}$$

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 $n \times n$.

It was claimed in the introduction that the reason for performing splits on abstract data-types was that we can use information that might not be clear in the concrete representation. Let us consider how we could split and transpose a matrix after it had been flattened to an array. We will consider a flattening where we concatenate each row of the matrix. Consider the matrix $\mathbf{E}$ above and let A be an array which represents $\mathbf{E}$:

$$A \;=\; [a, b, c, d, e, f, \ldots, j, \ldots, u, v, w, x, y]$$

So if we split A to match how $\mathbf{E}$ was split then we obtain

$$A \rightsquigarrow \langle [a, b, c, f, g, h, k, l, m],\ [d, e, i, j, n, o], [p, q, r, u, v, w],\ [s, t, x, y] \rangle$$

Now we need to perform a matrix transposition on each of the split array components. For instance, we would like

$$\begin{aligned} \mathsf{transpose}\ ([\,d, e, i, j, n, o\,]) &= [\,d, i, n, e, j, o\,] \\ \mathsf{transpose}\ ([p, q, r, u, v, w]) &= [p, u, q, v, r, w] \end{aligned}$$

For these two cases, transpose will have to perform different permutations of the array elements despite the arrays having the same length. So transpose needs to know the dimensions of the split matrix components which have been lost by flattening.

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

$$\begin{array}{lcl} \mathbf{M}^{n\times n} & \rightsquigarrow & \langle \mathbf{M}_0, \mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_2 \rangle_{s_k} \\ \mathbf{N}^{n\times n} & \rightsquigarrow & \langle \mathbf{N}_0, \ \mathbf{N}_1, \ \mathbf{N}_2, \ \mathbf{N}_3 \rangle_{s_k} \end{array}$$

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:

$$\begin{array}{lcl} \mathbf{M} \times \mathbf{N} & \rightsquigarrow & \langle (\mathbf{M}_0 \times \mathbf{N}_0) + (\mathbf{M}_1 \times \mathbf{N}_2), \ (\mathbf{M}_0 \times \mathbf{N}_1) + (\mathbf{M}_1 \times \mathbf{N}_3), \\ & & \ (\mathbf{M}_2 \times \mathbf{N}_0) + (\mathbf{M}_3 \times \mathbf{N}_2), \ (\mathbf{M}_2 \times \mathbf{N}_1) + (\mathbf{M}_3 \times \mathbf{N}_3) \rangle_{s_k} \end{array}$$

The computation of $\mathbf{M}\times\mathbf{N}$ using normal matrix multiplication requires n^3 integer multiplications. If we multiply the split matrices, does this calculation require more multiplications? From the definition of split matrices, the number of multiplications required to compute each component is:

$$\begin{array}{l} \langle k^3 + k^2(n-k), \ k^2(n-k) + k(n-k)^2, \\ \ k^2(n-k) + k(n-k)^2, \ k(n-k)^2 + (n-k)^3 \ \rangle \end{array}$$

Adding these up gives

$$\begin{array}{rl} & k^3 + 3k^2(n-k) + 3k(n-k)^2 + (n-k)^3 \\ = & (k + (n-k))^3 \\ = & n^3 \end{array}$$

which is exactly the same number of multiplications as the unsplit version.

To reduce the number of multiplications, we could use *Strassen's algorithm* [4] which performs matrix multiplication by using seven multiplications instead of eight. However the algorithm requires starting with a $2n\times 2n$ matrix and splitting it into four $n\times n$ matrices and so would not be directly applicable for a general matrix.

3 Functional Programs

We can model matrices in Haskell using a list of lists, *i.e.*

type $Matrix\ \alpha = [[\alpha]]$

Not all lists of lists represent a matrix; for instance, the list $[[1, 2], [3]]$ does not represent a matrix. In fact, mss represents a matrix if and only if all the lists that are members of mss have the same length. We can define a function valid that checks whether a list of lists is a valid matrix representation.

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

(Note that $|ys|$ denotes the length of the list ys.) Thus, the representation $\mathbf{M}^{r\times c} \rightsquigarrow mss$ must satisfy the invariant valid mss. We can represent $\mathbf{M}^{r\times c}$ by a list of lists mss where $|mss| = r$ and each list in mss has length c.

3.1 Functional Matrix Operations

We want to implement the matrix operations stated in Section 2 — *i.e.* scale, add, transpose and mult.

Two of our definitions will use the function zipWith — the definition below contains a catchall to deal with the situation where one of the lists is empty:

$$\begin{aligned}\mathsf{zipWith}\ f\ (x : xs)\ (y : ys) &= f\ x\ y : (\mathsf{zipWith}\ f\ xs\ ys)\\ \mathsf{zipWith}\ f\ \_\ \_ &= [\,]\end{aligned}$$

For brevity, we define the following two infix operations:
$(\odot)$ denotes zipWith $(+)$ and $(\otimes)$ denotes zipWith $(+\!\!+)$.

We define the matrix operations functionally as follows:

$$\begin{aligned}\mathsf{scale}\ a\ mss\ &= \mathsf{map}\ (\mathsf{map}\ (a\times))\ mss\\ \mathsf{add}\,(mss, nss)\ &= \mathsf{zipWith}\ (\odot)\ mss\ nss\\ \mathsf{transpose}\,mss\ &= \mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\,(\mathsf{map}\ \mathsf{wrap})\ mss)\\ &\quad \text{where } \mathsf{wrap}\ x = [x]\\ \mathsf{mult}\ (xss, yss)\ &= \mathsf{map}\ (\mathsf{row}\ yss)\ xss\\ &\quad \text{where } \mathsf{row}\ yss\ xs = \mathsf{map}\ (\mathsf{dotp}\ xs)\ (\mathsf{transpose}\ yss)\\ &\qquad\quad \mathsf{dotp}\ ms\ ns = \mathsf{sum}\ (\mathsf{zipWith}\ (\times)\ ms\ ns)\end{aligned}$$

We can see that using lists of lists to model matrices allows us to write succinct definitions for our matrix operations.

3.2 Splitting

In this section, we show how to model the $(k\times k)$-square split with Haskell lists and so for this section, we deal with square matrices. We introduce the following type:

$$\mathit{SpMat}\ \alpha \ =\ \langle \mathit{Matrix}\ \alpha,\ \mathit{Matrix}\ \alpha,\ \mathit{Matrix}\ \alpha,\ \mathit{Matrix}\ \alpha\ \rangle$$

to describe split matrices.

Let us suppose that we have a square matrix mss with dim $mss = (n, n)$. Then the representation

$$mss \rightsquigarrow \langle as, bs, cd, ds\rangle_{s_k}$$

satisfies the invariant

$$\begin{array}{c} \mathsf{dim}\ as = (k, k) \wedge \mathsf{dim}\ bs = (k, n-k) \quad \wedge \\ \mathsf{dim}\ cs = (n-k, k) \wedge \mathsf{dim}\ ds = (n-k, n-k) \end{array} \tag{4}$$

for some $k : (0..n)$. The operation dim is defined to be:

$$\mathsf{dim}\ mss = (|mss|,\ |\mathsf{head}\ mss|)$$

If mss represents a matrix with dimensions $r \times c$, then the length of the list mss is r. Thus, we define

$$|\langle as, bs, cs, ds \rangle_{s_k}| = |as| + |cs|$$

Rather than using the choice function ch and the family of functions $\mathcal{F}$, we will define a splitting function directly for lists. We would like a function

$$\mathsf{split}_{s_k} :: Matrix\ \alpha \to SpMat\ \alpha$$

that operates on square matrices and returns the corresponding split matrix. Thus split_{s_k} satisfies the relationship:

$$mss \rightsquigarrow \langle as, bs, cs, ds \rangle_{s_k} \Leftrightarrow \mathsf{split}_{s_k}\ mss = \langle as, bs, cs, ds \rangle_{s_k}$$

For this split, we define

$$\begin{array}{l} \mathsf{split}_{s_k}\ mss = \langle as, bs, cs, ds \rangle_{s_k} \\ \qquad \text{where} \quad \begin{array}{l} (xss, yss) = \mathsf{splitAt}\ k\ mss \\ (as, bs) = \mathsf{unzip}\ (\mathsf{map}\ (\mathsf{splitAt}\ k)\ xss) \\ (cs, ds) = \mathsf{unzip}\ (\mathsf{map}\ (\mathsf{splitAt}\ k)\ yss) \end{array} \end{array}$$

Since we can undo $\mathsf{splitAt}$ by using $+\!\!+$, we define the (left and right) inverse of split_{s_k} to be

$$\mathsf{unsplit}_{s_k}\ \langle as, bs, cs, ds \rangle_{s_k} = (as \otimes bs) +\!\!+ (cs \otimes ds)$$

Note that this definition is independent of the choice of k.

We define

$$\mathsf{null}_{s_k} = \langle [\,], [\,], [\,], [\,] \rangle_{s_k}$$

which will be used in some of our definitions.

3.3 Derivations

We would now like to define our matrix operations for this split whilst ensuring that the operations are correct — *i.e.* their functionality is unchanged. Using the approach taken in [7], we can derive obfuscations operations or prove their correctness by using data refinement techniques.

$$
\begin{tikzcd}
(xss, yss) \arrow[r, "\mathsf{op2}"] \arrow[d, "\mathsf{cross(split, split)}"'] & \mathsf{op2}\ (xss, yss) \arrow[d, "\mathsf{split}"] \\
(\langle a, b, c, d\rangle, \langle e, f, g, h\rangle) \arrow[r, "\mathsf{sp2}"'] & \mathsf{sp2}\ (\langle a, b, c, d\rangle, \langle e, f, g, h\rangle)
\end{tikzcd}
$$

Figure 1: Commuting diagram for a binary operation

In [7], a unary operation for sets is considered. We can adapt this argument for an operation

$$\mathsf{op1} :: \mathit{Matrix} \to \mathit{Matrix}$$

so that we can derive an operation

$$\mathsf{sp1} :: \mathit{SpMat} \to \mathit{SpMat}$$

for split matrices. The derivation is achieved by constructing a commuting diagram, to find that

$$\mathsf{split} \circ \mathsf{op1} = \mathsf{sp1} \circ \mathsf{split}$$

Post-composing this equation by unsplit gives:

$$\mathsf{sp1} = \mathsf{split} \circ \mathsf{op1} \circ \mathsf{unsplit} \tag{5}$$

This equation allows us to derive a split operation from the corresponding unsplit version.

We also need to consider a binary operation

$$\mathsf{op2} :: (\mathit{Matrix}, \mathit{Matrix}) \to \mathit{Matrix}$$

and we would like to derive a split operation:

$$\mathsf{sp2} :: (\mathit{SpMat}, \mathit{SpMat}) \to \mathit{SpMat}$$

Figure 1 shows a commuting diagram [6] linking these two operations — the function cross is defined to be:

$$\mathsf{cross}\ (f,\ g)\ (x, y) = (f\ x,\ g\ y)$$

From the diagram, we observe that:

$$\mathsf{split} \circ \mathsf{op2} = \mathsf{sp2} \circ \mathsf{cross(split, split)} \tag{6}$$

Since the inverse of $\mathsf{cross(split, split)}$ is $\mathsf{cross(unsplit, unsplit)}$, post-composing equation (6) by this inverse gives:

$$\mathsf{sp2} = \mathsf{split} \circ \mathsf{op2} \circ \mathsf{cross}\ (\mathsf{unsplit}, \mathsf{unsplit}) \tag{7}$$

Again we can derive a split operation from the corresponding one for unsplit matrices. If we pre-compose (6) by $\mathsf{unsplit}$ then we obtain:

$$\mathsf{op2} = \mathsf{unsplit} \circ \mathsf{sp2} \circ \mathsf{cross}(\mathsf{split}, \mathsf{split})$$

This equation allows us to prove that $\mathsf{op2}$ is correct. Therefore from an obfuscation point of view, we must keep split and $\mathsf{unsplit}$ secret otherwise an adversary could reconstruct op by using this equation.

Now suppose that we have an operation

$$\mathsf{op3} :: \mathit{Matrix} \to S$$

where S is some set of values (for instance, Booleans or integers). If we define a corresponding operation for split lists

$$\mathsf{sp3} :: \mathit{SpMat} \to S$$

we can use the equation:

$$\mathsf{op3}\ \mathit{xss} = \mathsf{sp3}\ (\mathsf{split}\ \mathit{xss}) \tag{8}$$

to prove the correctness of the split operation.

3.4 Folds and Unfolds

In programming, we often identify common patterns so that we can develop general methods for these patterns. Two such patterns in functional programming are *fold* and *unfold* ([9], for example) — in this paper, we will only consider folding and unfolding from the right, thus we write fold and unfold instead of the usual foldr and $\mathsf{unfoldr}$. For unfold, we follow the definition given in [9]:

$$\mathsf{unfold}\ p\ g\ h\ b \;=\; \text{if } p\ b \text{ then } [\,] \text{ else } (g\ b) : (\mathsf{unfold}\ p\ g\ h\ (h\ b))$$

Many list operations can be written in terms of fold or unfold. For instance,

$$\begin{array}{l}
\mathsf{zipWith}\ (\bowtie)\ \mathit{xs}\ \mathit{ys} \;=\; \mathsf{unfold}\ p\ g\ h\ (\mathsf{pair}\ \mathit{xs}\ \mathit{ys}) \\
\quad \text{where} \;\; \begin{array}[t]{l}
p\ (\mathit{ms}, \mathit{ns}) = \mathit{ms} == [\,] \mid\mid \mathit{ns} = [\,] \\
g\ (m : \mathit{ms}, n : \mathit{ns}) = m \bowtie n \\
h\ (m : \mathit{ms}, n : \mathit{ns}) = (\mathit{ms}, \mathit{ns}) \\
\mathsf{pair}\ \mathit{ms}\ \mathit{ns} = (\mathit{ms}, \mathit{ns})
\end{array}
\end{array}$$

Our split_{s_k} operation consumes a list and produces a new data structure whilst $\mathsf{unsplit}_{s_k}$ takes the new data structure and builds a list. This means that we should be able to write split_{s_k} as a fold and $\mathsf{unsplit}_{s_k}$ as an unfold.

For split_{s_k}, if we use fold we have to build up the split matrix starting with the last row. So if $\mathit{mss} \rightsquigarrow \langle \mathit{as}, \mathit{bs}, \mathit{cs}, \mathit{ds} \rangle_{s_k}$, then we start building up the rows of cs and ds. Once we have built $n-k$ rows (where $n = |mss|$) then we build up the rows of as and bs. This gives us the following definition of split_{s_k}:

$$
\begin{array}{l}
\mathsf{split}_{s_k}\ mss\ = \mathsf{fold}\ f\ \mathsf{null}_{s_k}\ mss \\
\quad \text{where}\ \ f\ xss\ \langle es, fs, gs, hs\rangle_{s_k} \\
\qquad\qquad\qquad \left| \begin{array}{ll} |hs| < n-k & = \langle [\,], [\,], (y : gs), (z : hs)\rangle_{s_k} \\ \text{otherwise} & = \langle (y : es), (z : fs), gs, hs\rangle_{s_k} \end{array}\right. \\
\qquad\qquad\qquad\quad \text{where}\ (y, z) = \mathsf{splitAt}\ k\ xss \\
\qquad\qquad\qquad\qquad\qquad n = |mss|
\end{array}
$$

and this is equivalent to the previous definition of split_{s_k}.

We can write $\mathsf{unsplit}_{s_k}$ as follows:

$$
\begin{array}{l}
\mathsf{unsplit}_{s_k}\ \langle as, bs, cs, ds\rangle_{s_k} = \mathsf{unfold\ null}\ g\ \mathsf{tail}\ (\mathsf{zip}\ (as +\!\!+\ cs)\ (bs +\!\!+\ ds)) \\
\qquad\qquad\qquad\qquad \text{where}\ g\ ((y, z) : xs) = y +\!\!+\ z
\end{array}
$$

Since $|as| = |bs|$ then

$$\mathsf{zip}\ (as +\!\!+\ cs)\ (bs +\!\!+\ ds) = (\mathsf{zip}\ as\ bs) +\!\!+\ (\mathsf{zip}\ cs\ ds)$$

At each stage of the unfold we concatenate each of pair of lists together. So, this definition is equivalent to the previous definition of $\mathsf{unsplit}_{s_k}$.

The operations of fold and unfold satisfy various properties which will be needed later in this paper.

Property 1 (fold fusion). For a function g

$$g \circ \mathsf{fold}\ f\ a = \mathsf{fold}\ h\ b$$

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

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

Property 2 (unfold fusion). For a function f,

$$\mathsf{unfold}\ p\ g\ h \circ f = \mathsf{unfold}\ p'\ g'\ h'$$

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

Property 3 (map fusion). Two fusion rules for map:

$$
\begin{array}{rcl}
\mathsf{fold}\ f\ a \circ \mathsf{map}\ g & = & \mathsf{fold}\ (f \circ g)\ a \\
\mathsf{map}\ f \circ \mathsf{unfold}\ p\ g\ h & = & \mathsf{unfold}\ p\ (f \circ g)\ h
\end{array}
$$

Property 4 (folds and concat). For finite lists xs and ys

$$\mathsf{fold}\ f\ a\ (xs +\!\!+\ ys) = \mathsf{fold}\ f\ (\mathsf{fold}\ f\ a\ ys)\ xs$$

A fold followed by an unfold is called a *hylomorphism* [13]. In [9], a hylomorphism is defined to be:

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

which gives

$$\begin{aligned}\mathsf{hylo}\ f\ \ e\ \ p\ \ g\ \ h\ \ x = \ & \text{if } p\ \ x \text{ then } e \\ & \text{else } f\ \ (g\ \ x)(\mathsf{hylo}\ f\ \ e\ \ p\ \ g\ \ h\ \ (h\ \ x))\end{aligned} \tag{9}$$

As an example of a hylomorphism, we show that $\mathsf{unsplit}_{s_k}$ is a right inverse for split_{s_k}, *i.e.*

$$\mathsf{split}_{s_k}\ (\mathsf{unsplit}_{s_k}\ \langle as, bs, cs, ds\rangle_{s_k}) = \langle as, bs, cs, ds\rangle_{s_k}$$

Let us define functions fn and $\mathsf{fn'}$ where

$$\begin{aligned}&\mathsf{fn}\ \langle as, bs, cs, ds\rangle_{s_k} = \mathsf{fn'}\ (\mathsf{zip}\ (as +\!\!+\ cs)\ (bs +\!\!+\ ds))\\ &\mathsf{fn'}\ mss = \mathsf{split}_{s_k}\ (\mathsf{unfold}\ \mathsf{null}\ g\ \mathsf{tail}\ mss)\\ &\qquad\qquad \text{where } g\ ((y,z) : xs) = y +\!\!+\ z\end{aligned}$$

Using the fold definition of split and rewriting the zip, we obtain:

$$\begin{aligned}\mathsf{fn}\ \langle as, bs, cs, ds\rangle_{s_k} \ &=\ \mathsf{fn'}\ (\ (\mathsf{zip}\ as\ \ bs)\ +\!\!+\ (\mathsf{zip}\ cs\ \ ds))\\ \mathsf{fn'}\ mss \ &=\ \text{if } \mathsf{null}\ mss \text{ then } \mathsf{null}_{s_k}\\ &\qquad \text{else } f\ \ (g\ \ mss)(\mathsf{fn'}\ (\mathsf{tail}\ mss))\\ \text{where}\ \ & f\ \ xss\ \langle es, fs, gs, hs\rangle_{s_k}\\ &\quad \left|\ \begin{aligned}&|hs| < n-k \ = \langle [\ ], [\ ], (y : gs), (z : hs)\rangle_{s_k}\\ &\text{otherwise } \ = \langle (y : es), (z : fs), gs, hs\rangle_{s_k}\end{aligned}\right.\\ &\qquad \text{where } (y, z) = \mathsf{splitAt}\ k\ \ xss\\ & n = |mss|\\ & g\ \ ((y,z) : xs) = y +\!\!+\ z\end{aligned}$$

The function g acts on lists of pairs of the form (ps, qs), where ps and qs are themselves lists. Using the split invariant (4), the length of the first component is always k (as the lists are taken from either as or bs) and so

$$\mathsf{splitAt}\ k\ \ (g\ \ ((x,y) : xs)) = (x,y)$$

If we find a function f' such that

$$f \circ g = f' \circ \mathsf{head}$$

then we can write $\mathsf{fn'}$ in terms of fold. Thus

$$\begin{aligned}\mathsf{fn}\ \langle as, bs, cs, ds\rangle_{s_k} \ &=\ \mathsf{fn'}\ (\ (\mathsf{zip}\ as\ \ bs)\ +\!\!+\ (\mathsf{zip}\ cs\ \ ds))\\ \mathsf{fn'}\ mss \ &=\ \mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ mss\\ \text{where}\ \ & f'\ \ (y,z)\ \ \langle es, fs, gs, hs\rangle_k\\ &\quad \left|\ \begin{aligned}&|hs| < n-k \ = \langle [\ ], [\ ], (y : gs), (z : hs)\rangle_{s_k}\\ &\text{otherwise } = \langle (y : es), (z : fs), gs, hs\rangle_{s_k}\end{aligned}\right.\\ & n = |mss|\end{aligned}$$

By property 4,

$$\begin{array}{rl} & \mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ ((\mathsf{zip}\ as\ \ bs) \mathbin{+\!\!+} (\mathsf{zip}\ cs\ \ ds)) \\ = & \mathsf{fold}\ f'\ (\mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ (\mathsf{zip}\ cs\ \ ds))\ (\mathsf{zip}\ as\ \ bs) \end{array}$$

Consider $\mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ (\mathsf{zip}\ cs\ \ ds)$. The length of the last split component is zero initially and is increased by one (since we add an element to the front of it) at each stage of the fold until it has length $n-k$. Since $|\mathsf{zip}\ cs\ \ ds| = n-k$ (using invariant (4)) then the first case in the definition of f' always applies. So at each stage we add an element to the third and fourth components. Since the identity for lists can be written as $\mathsf{fold}\ (:)\ [\ ]$, we can see that

$$\mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ (\mathsf{zip}\ cs\ \ ds) \quad = \quad \langle [\ ], [\ ], cs, ds\rangle_{s_k}$$

Now consider $\mathsf{fold}\ f'\langle [\ ], [\ ], cs, ds\rangle_{s_k}\ (\mathsf{zip}\ as\ \ bs)$. As $|ds| = n-k$, the second condition in f' always applies. Again, we create the first two components by adding elements to the front of them and so

$$\mathsf{fold}\ f'\langle [\ ], [\ ], cs, ds\rangle_{s_k}\ (\mathsf{zip}\ as\ \ bs) = \langle as, bs, cs, ds\rangle_{s_k}$$

Hence, fn is the identity for split matrices.

We can use properties of fold and unfold to aid the derivation of a split operation sp from an operation op using the equation

$$\mathsf{sp} \ = \ \mathsf{split} \circ \mathsf{op} \circ \mathsf{unsplit}$$

We have three cases to consider:

- If op is a fold, we can fuse split into op and then we have a hylomorphism.
- If op is an unfold, we fuse $\mathsf{unsplit}$ into op and again we have a hylomorphism.
- If op is a map then we can fuse this into either split or $\mathsf{unsplit}$.

These cases will be used for the derivations in Section 4.

3.5 Properties

To make the derivations in Section 4 easier we will use some properties about map, $\mathbin{+\!\!+}$ and $\mathsf{zipWith}$. The proofs of some of these properties can in found in Appendix A.

First, two properties of $\mathbin{+\!\!+}$:

$$\begin{array}{rcl} \mathsf{map}\ f\ (xs \mathbin{+\!\!+} ys) & = & (\mathsf{map}\ f\ \ xs) \mathbin{+\!\!+} (\mathsf{map}\ f\ \ ys) \\ (ys, zs) = \mathsf{splitAt}\ k\ xs & \Rightarrow & xs = ys \mathbin{+\!\!+} zs \end{array}$$

Next, we will need a way of concatenating two $\mathsf{zipWith}$s together.

Property 5. Let $(\bowtie)$ be a function $\bowtie :: [\alpha] \to [\beta] \to [\gamma]$ and suppose that we have lists as and cs of type $[\alpha]$ and bs and ds of type $[\beta]$ such that $|as| = |bs|$. Then:

$$\begin{array}{l} (\mathsf{zipWith}\ (\bowtie)\ as\ \ bs) \mathbin{+\!\!+} (\mathsf{zipWith}\ (\bowtie)\ cs\ \ ds) \\ \quad = \mathsf{zipWith}(\bowtie)\ (as \mathbin{+\!\!+} cs)\ (bs \mathbin{+\!\!+} ds) \end{array}$$

Using this property, we can write an alternative definition for $\mathsf{unsplit}$:

$$\mathsf{unsplit}\ \langle as, bs, cs, ds\rangle_{s_k} = (as ⧺ cs)\ \otimes\ (bs ⧺ ds)$$

The next two properties link $\mathsf{transpose}$ and $\otimes$.

Property 6. For matrices mss and nss

$$\mathsf{transpose}\,(mss ⧺ nss) = (\mathsf{transpose}\ mss) \otimes (\mathsf{transpose}\ nss)$$

We can see this property pictorially,

$$\left(\frac{mss}{nss}\right)^T \quad = \quad \left(\begin{array}{c|c} mss^T & nss^T \end{array}\right)$$

Property 7. For matrices mss and nss where $|mss| = |nss|$ then

$$\mathsf{transpose}\,(mss\ \otimes\ nss) = \mathsf{transpose}\ mss\ ⧺\ \mathsf{transpose}\ nss$$

4 Split Matrix Operations

In this section, we develop split versions of the operations discussed in Section 3.1. To show the flexibility of our approach, we directly derive one of our operations and use fold fusion for the remainder.

4.1 Deriving a scalar operation using fusion

We now derive an operation that performs scalar multiplication on split matrices. From equation (5), we get

$$\mathsf{scale}_{s_k}\ s = \mathsf{split}_{s_k} \circ (\mathsf{scale}\ s) \circ \mathsf{unsplit}_{s_k}$$

As our scalar operation is a map, we can fuse it into either split_{s_k} or $\mathsf{unsplit}_{s_k}$ — we choose to fuse it into $\mathsf{unsplit}_{s_k}$.

$$
\begin{array}{cl}
 & ((\mathsf{scale}\ s) \circ \mathsf{unsplit}_{s_k})\ \langle a, b, c, d\rangle_{s_k} \\
= & \quad \{\text{definitions}\} \\
 & \mathsf{map}\ (\mathsf{map}\ (\times s))\ (\mathsf{unfold}\ \mathsf{null}\ g\ \mathsf{tail}\ (\mathsf{zip}\ (a ⧺ c)\ (b ⧺ d))) \\
 & \qquad \text{where } g\ ((x, y) : xs) = x ⧺ y \\
= & \quad \{\text{unfold map fusion — property 3}\} \\
 & \mathsf{unfold}\ \mathsf{null}\ (f \circ g)\ \mathsf{tail}\ (\mathsf{zip}\ (a ⧺ c)\ (b ⧺ d)) \\
 & \qquad \text{where } \ g\ ((x, y) : xs) = x ⧺ y \\
 & \qquad \qquad \quad f = \mathsf{map}\ (\times s) \\
= & \quad \{\text{functional composition}\} \\
 & \mathsf{unfold}\ \mathsf{null}\ g'\ \mathsf{tail}\ (\mathsf{zip}\ (a ⧺ c)\ (b ⧺ d)) \\
 & \qquad \text{where } g'\ ((x, y) : xs) = \mathsf{map}\ (\times s)\ (x ⧺ y)
\end{array}
$$

This function is written in terms of unfold whilst split can be written as a fold — thus we have a hylomorphism. Now define

$$\mathsf{scale'}_{s_k}\ s\ = \mathsf{split}_{s_k} \circ (\mathsf{unfold\ null}\ g'\ \mathsf{tail})$$

where g' is defined as above. Now we derive a definition for $\mathsf{scale'}_k$

$$\begin{array}{ll}
& \mathsf{scale'}_{s_k}\ s\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d)) \\
= & \quad \{\text{from above}\} \\
& \mathsf{split}_{s_k}\ (\mathsf{unfold\ null}\ g'\ \mathsf{tail}\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d))) \\
& \qquad \text{where } g'\ ((x, y) : xs) = \mathsf{map}\ (\times s)\ (x +\!\!+ y) \\
= & \quad \{\text{definition of } \mathsf{split}_{s_k}\} \\
& \mathsf{fold}\ f\ \mathsf{null}_{s_k}\ (\mathsf{unfold\ null}\ g'\ \mathsf{tail}(\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d))) \\
& \ \text{where}\ \ f\ xss\ \langle ms, ns, ps, qs\rangle_{s_k} \\
& \qquad\qquad \left|\ \begin{array}{l} |qs| < n - k\ \ = \langle [\,], [\,], (l : ps), (r : qs)\rangle_{s_k} \\ \text{otherwise}\ = \langle (l : ms), (r : ns), ps, qs\rangle_{s_k} \end{array}\right. \\
& \qquad\qquad\quad \text{where } (l, r) = \mathsf{splitAt}\ k\ xss \\
& \qquad\quad g'\ ((x, y) : xs) = \mathsf{map}\ (\times s)\ (x +\!\!+ y) \\
& \qquad\quad n = |a| + |c| \\
= & \quad \{\text{hylomorphism — (9)}\} \\
& \text{if } \mathsf{null}\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d)) \text{ then } \mathsf{null}_{s_k} \\
& \text{else } f\ \ (g'\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d))) \\
& \qquad (\mathsf{scale'}_{s_k}\ s\ \mathsf{tail}\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d))) \\
& \ \text{where}\ \ f\ xss\ \langle ms, ns, ps, qs\rangle_{s_k} \\
& \qquad\qquad \left|\ \begin{array}{l} |qs| < n - k\ \ = \langle [\,], [\,], (l : ps), (r : qs)\rangle_{s_k} \\ \text{otherwise}\ = \langle (l : ms), (r : ns), ps, qs\rangle_{s_k} \end{array}\right. \\
& \qquad\qquad\quad \text{where } (l, r) = \mathsf{splitAt}\ k\ xss \\
& \qquad\quad g'\ (x, y) : xs = \mathsf{map}\ (\times s)\ (x +\!\!+ y) \\
& \qquad\quad n = |a| + |c|
\end{array}$$

Since $|a| = k = |b|$ (from invariant (4)),

$$\mathsf{splitAt}\ k\ (g'\ ((x, y) : xs)) = (\mathsf{map}\ (\times s)\ x, \mathsf{map}\ (\times s)\ y)$$

and this helps us to compose f and g'. In fact, if we can find a function f' such that

$$f' \circ \mathsf{head} = f \circ g'$$

then we can write $\mathsf{scale'}_{s_k}$ as a fold and thus

$$\begin{array}{l}
\mathsf{scale}_{s_k}\ s\ \langle a, b, c, d\rangle_{s_k}\ = \mathsf{scale''}_{s_k}\ s\ (|a| + |c|)\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d))) \\
\mathsf{scale''}_{s_k}\ p\ n\ xs = \mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ xs \\
\quad \text{where}\ \ f'\ (x, y)\ \langle ms, ns, ps, qs\rangle_{s_k} \\
\qquad\qquad \left|\ \begin{array}{l} |qs| < n - k\ \ = \langle [\,], [\,], (l : ps), (r : qs)\rangle_{s_k} \\ \text{otherwise}\ = \langle (l : ms), (r : ns), ps, qs\rangle_{s_k} \end{array}\right. \\
\qquad\qquad\quad \text{where } (l, r) = (\mathsf{map}\ (\times s)\ x,\ \mathsf{map}\ (\times s)\ y)
\end{array}$$

Note that we have to pass in an extra argument, the size of the original matrix, into scale''_{s_k}. Following a similar argument to the one in Section 3.4 that showed $\mathsf{unsplit}$ was the inverse of split, we find that

$$\mathsf{scale}_{s_k}\ s\ \langle a, b, c, d\rangle_{s_k} = \langle \mathsf{scale}\ s\ a,\ \mathsf{scale}\ s\ b,\ \mathsf{scale}\ s\ c,\ \mathsf{scale}\ s\ d\rangle_{s_k}$$

which matches the operation given in Section 2.1. The direct derivation can be found in Appendix C.

4.2 Deriving addition using fusion

The previous derivation lead to an operation that matched our expectation. As a demonstration of unfold fusion, we show the derivation of an addition operation for split matrices. However, this method does not lead to a satisfactory operation.

Addition of two matrices was defined to be

$$\mathsf{add}\ mss\ nss = \mathsf{zipWith}\ (\odot)\ mss\ nss$$

As stated in Section 3.4, $\mathsf{zipWith}$ can be written using unfold and so we can define

$$\begin{array}{l} \mathsf{add}\ (mss, nss) = \mathsf{unfold}\ p\ g\ h\ (mss, nss) \\ \quad \text{where}\ \begin{array}[t]{l} p\ (xs, ys) = xs == [\,]\ ||\ ys == [\,] \\ g\ (xs, ys) = \mathsf{zipWith}\ (+)\ (\mathsf{head}\ xs)\ (\mathsf{head}\ ys) \\ h\ (xs, ys) = (\mathsf{tail}\ xs, \mathsf{tail}\ \ ys) \end{array} \end{array}$$

We need to derive an operation add_{s_k} and so by equation (7)

$$\mathsf{add}_{s_k}\ (mss, nss) = \mathsf{split}_{s_k}\ (\mathsf{add}\ (\mathsf{unsplit}_{s_k}\ mss,\ \mathsf{unsplit}_{s_k}\ nss))$$

As add is written in terms of unfold, we can perform unfold fusion.

Suppose that

$$mss \rightsquigarrow \langle a, b, c, d\rangle_{s_k}$$

so that

$$\begin{array}{rcl} mss & = & \mathsf{unsplit}_{s_k}\ \langle a, b, c, d\rangle_{s_k} \\ & = & (a \otimes b) +\!\!+ (c \otimes d) \end{array}$$

To perform the fusion, we need to define head and tail of $(a \otimes b) +\!\!+ (c \otimes d)$. By the properties of $\mathsf{zipWith}(+\!\!+)$, we can define

$$\begin{array}{l} (\mathsf{head}\ ((a \otimes b) +\!\!+ (c \otimes d)),\ \mathsf{tail}\ ((a \otimes b) +\!\!+ (c \otimes d))) \\ \quad \left| \begin{array}{ll} a == [\,] \wedge b == [\,] & = (\mathsf{head}\ c +\!\!+ \mathsf{head}\ d, \mathsf{tail}\ c \otimes \mathsf{tail}\ d) \\ \text{otherwise} & = (\mathsf{head}\ a +\!\!+ \mathsf{head}\ b,\ (\mathsf{tail}\ a \otimes \mathsf{tail}\ b) +\!\!+ (c \otimes d)) \end{array} \right. \end{array}$$

So, using fusion, we need to write

$$\mathsf{unfold}\ p'\ g'\ h'\ (xss, yss) = \mathsf{unfold}\ p\ g\ h\ (\mathsf{unsplit}_{s_k}\ (xss, yss))$$

where

$$\begin{aligned}
p' &= p \circ \mathsf{unsplit}_{s_k} \\
g' &= g \circ \mathsf{unsplit}_{s_k} \\
\mathsf{unsplit}_{s_k} \circ h' &= h \circ \mathsf{unsplit}_{s_k}
\end{aligned}$$

The last equation is equivalent to

$$h' = \mathsf{split}_{s_k} \circ h \circ \mathsf{unsplit}_{s_k}$$

Let us now define each of the functions

$$p'\ (xs, ys) = xs == \mathsf{null}_{s_k}\ ||\ ys == \mathsf{null}_{s_k}$$

Next,

$$g'\ (xss, yss) = \mathsf{zipWith}\ (+)\ (\mathsf{head}\ (\mathsf{unsplit}_{s_k}\ xss))\ (\mathsf{head}\ (\mathsf{unsplit}_{s_k}\ yss))$$

By using the result above for head, we can define

$$\begin{aligned}
&g'\ (\langle [\,], [\,], cs, ds\rangle_{s_k}, \langle [\,], [\,], gs, hs\rangle_{s_k}) = \\
&\qquad\qquad \mathsf{zipWith}\ (+)\ (\mathsf{head}\ cs \mathbin{+\!\!+} \mathsf{head}\ ds)\ (\mathsf{head}\ gs \mathbin{+\!\!+} \mathsf{head}\ hs) \\
&g'\ (\langle as, bs, cs, ds\rangle_{s_k}, \langle es, fs, gs, hs\rangle_{s_k}) = \\
&\qquad\qquad \mathsf{zipWith}\ (+)\ (\mathsf{head}\ as \mathbin{+\!\!+} \mathsf{head}\ bs)\ (\mathsf{head}\ es \mathbin{+\!\!+} \mathsf{head}\ fs)
\end{aligned}$$

Finally,

$$h'\ (xss, yss)\ = (\mathsf{split}_{s_k}\ (\mathsf{tail}\ (\mathsf{unsplit}_{s_k}\ xss)),\ \mathsf{split}_{s_k}\ (\mathsf{tail}\ (\mathsf{unsplit}_{s_k}\ yss)))$$

Using the result for tail, we obtain the following definition

$$\begin{aligned}
&h'\ (\langle [\,], [\,], cs, ds\rangle_{s_k}, \langle [\,], [\,], gs, hs\rangle_{s_k}) = \\
&\qquad\qquad (\langle [\,], [\,], \mathsf{tail}\ cs, \mathsf{tail}\ ds\rangle_{s_k},\ \langle [\,], [\,], \mathsf{tail}\ gs, \mathsf{tail}\ hs\rangle_{s_k}) \\
&h'\ (\langle as, bs, cs, ds\rangle_{s_k}, \langle es, fs, gs, hs\rangle_{s_k}) = \\
&\qquad\qquad (\langle \mathsf{tail}\ as, \mathsf{tail}\ bs, cs, ds\rangle_{s_k},\ \langle \mathsf{tail}\ es, \mathsf{tail}\ fs, gs, hs\rangle_{s_k})
\end{aligned}$$

Since we can write $\mathsf{add} \circ \mathsf{unsplit}$ in terms of unfold, we now use the hylomorphism rule, to write a definition for add_{s_k}.

$$
\begin{array}{l}
\mathsf{add}_{s_k}\ (xss, yss) \quad = \quad \text{if } p'\ (xss, yss) \text{ then } \mathsf{null}_{s_k} \\
\qquad\qquad\qquad\qquad\quad\ \ \text{else } f\ (g'\ (xss, yss))\ (\mathsf{add}_{s_k}\ h'\ (xss, yss)) \\
\quad \text{where} \quad p'\ (xss, yss) = xss == \mathsf{null}_{s_k}\ ||\ yss == \mathsf{null}_{s_k} \\
\qquad\qquad\quad f\ ts\ \langle ms, ns, ps, qs\rangle_{s_k} \\
\qquad\qquad\qquad \left|\begin{array}{ll} |qs| < |xss| - k & = \langle ms, ns, l : ps, r : qs\rangle_{s_k} \\ \text{otherwise} & = \langle l : ms, r : ns, ps, qs\rangle_{s_k} \end{array}\right. \\
\qquad\qquad\qquad \text{where } (l, r) = \mathsf{splitAt}\ k\ ts \\
\qquad\qquad\quad g'\ (\langle[\,], [\,], cs, ds\rangle_{s_k}, \langle[\,], [\,], gs, hs\rangle_{s_k}) = \\
\qquad\qquad\qquad \mathsf{zipWith}\ (+)\ (\mathsf{head}\ cs \mathbin{+\!\!+} \mathsf{head}\ ds)\ (\mathsf{head}\ gs \mathbin{+\!\!+} \mathsf{head}\ hs) \\
\qquad\qquad\quad g'\ (\langle as, bs, cs, ds\rangle_{s_k}, \langle es, fs, gs, hs\rangle_{s_k}) = \\
\qquad\qquad\qquad \mathsf{zipWith}\ (+)\ (\mathsf{head}\ as \mathbin{+\!\!+} \mathsf{head}\ bs)\ (\mathsf{head}\ es \mathbin{+\!\!+} \mathsf{head}\ fs) \\
\qquad\qquad\quad h'\ (\langle[\,], [\,], cs, ds\rangle_{s_k}, \langle[\,], [\,], gs, hs\rangle_{s_k}) = \\
\qquad\qquad\qquad (\langle[\,], [\,], \mathsf{tail}\ cs, \mathsf{tail}\ ds\rangle_{s_k},\ \langle[\,], [\,], \mathsf{tail}\ gs, \mathsf{tail}\ hs\rangle_{s_k}) \\
\qquad\qquad\quad h'\ (\langle as, bs, cs, ds\rangle_{s_k}, \langle es, fs, gs, hs\rangle_{s_k}) = \\
\qquad\qquad\qquad (\langle \mathsf{tail}\ as, \mathsf{tail}\ bs, cs, ds\rangle_{s_k},\ \langle \mathsf{tail}\ es, \mathsf{tail}\ fs, gs, hs\rangle_{s_k})
\end{array}
$$

This function contains the part of the definition for split and this is not ideal from an obfuscation point of view. It is difficult to compose together the functions so that the definition of split become less transparent. A direct derivation of add_{s_k} (found in Appendix B) yields the expected result that

$$
\mathsf{add}_{s_k}\ \langle a, b, c, d\rangle_{s_k}\ \langle e, f, g, h\rangle_{s_k} = \langle \mathsf{add}\ a\ e,\ \mathsf{add}\ b\ f,\ \mathsf{add}\ c\ g,\ \mathsf{add}\ d\ h\rangle_{s_k}
$$

4.3 Deriving transposition directly

We now show how to derive $\mathsf{transpose}_{s_k}$ using the properties from Section 3.5.

$$
\begin{array}{ll}
 & \mathsf{transpose}_{s_k}\ \langle a, b, c, d\rangle_{s_k} \\
= & \quad \{\text{using equation (5)}\} \\
 & \mathsf{split}_{s_k} (\mathsf{transpose}\ (\mathsf{unsplit}_{s_k}\ \langle a, b, c, d\rangle_{s_k})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{s_k}\} \\
 & \mathsf{split}_{s_k}\ (\mathsf{transpose}\ ((a\ \otimes\ b) \mathbin{+\!\!+} (c\ \otimes\ d))) \\
= & \quad \{\text{property 6}\} \\
 & \mathsf{split}_{s_k}\ ((\mathsf{transpose}\ (a\ \otimes\ b))\ \otimes\ (\mathsf{transpose}\ (c\ \otimes\ d))) \\
= & \quad \{\text{property 7}\} \\
 & \mathsf{split}_{s_k}\ ((\mathsf{transpose}\ a \mathbin{+\!\!+} \mathsf{transpose}\ b) \otimes (\mathsf{transpose}\ c \mathbin{+\!\!+} \mathsf{transpose}\ d)) \\
= & \quad \{\text{property 5, } (\otimes) = \mathsf{zipWith}(\mathbin{+\!\!+})\} \\
 & \mathsf{split}_{s_k}\ ((\mathsf{transpose}\ a \otimes \mathsf{transpose}\ c) \mathbin{+\!\!+} (\mathsf{transpose}\ b \otimes \mathsf{transpose}\ d)) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{s_k}\} \\
 & \mathsf{split}_{s_k}\ (\mathsf{unsplit}\ \langle \mathsf{transpose}\ a,\ \mathsf{transpose}\ c,\ \mathsf{transpose}\ b,\ \mathsf{transpose}\ d\rangle_{s_k}) \\
= & \quad \{\mathsf{split}_{s_k} \circ \mathsf{unsplit}_{s_k} = id\} \\
 & \langle \mathsf{transpose}\ a,\ \mathsf{transpose}\ c,\ \mathsf{transpose}\ b,\ \mathsf{transpose}\ d\,\rangle_{s_k}
\end{array}
$$

The derivation for $\mathsf{transpose}_{s_k}$ is quite short and matches the form given into Section 2.2. A derivation using folds is longer and more complicated.

4.4 Deriving multiplication using fusion

Next, we derive an operation for multiplying two split matrices

$$
\begin{array}{ll}
& \mathsf{mult}_{s_k}\ (xss, yss) \\
= & \quad \{\text{equation (7)}\} \\
& \mathsf{split}_{s_k}\ (\mathsf{mult}\ (\mathsf{unsplit}_{s_k}\ xss, \mathsf{unsplit}_{s_k}\ yss)) \\
= & \quad \{\text{definition of } \mathsf{mult}\} \\
& \mathsf{split}_{s_k}\ (\mathsf{map}\ (\mathsf{row}\ (\mathsf{transpose}\ (\mathsf{unsplit}_{s_k}\ yss)))\ (\mathsf{unsplit}_{s_k}\ xss)) \\
= & \quad \{\text{unfold definition of } \mathsf{unsplit}_{s_k}\text{, let } xss = \langle a, b, c, d\rangle_{s_k}\} \\
& \mathsf{split}_{s_k}\ (\mathsf{map}\ (\mathsf{row}\ (\mathsf{transpose}\ (\mathsf{unsplit}_{s_k}\ yss))) \\
& \qquad\quad (\mathsf{unfold}\ \mathsf{null}\ g\ \mathsf{tail}\ (\mathsf{zip}\ (a +\!\!+\ c)\ (b +\!\!+\ d)))) \\
& \qquad\qquad \text{where } g\ ((p, q) : xs) = p +\!\!+\ q \\
= & \quad \{\mathsf{unfold}\ \mathsf{map}\ \text{fusion — property 3}\} \\
& \mathsf{split}_{s_k}\ (\mathsf{unfold}\ \mathsf{null}\ g'\ \mathsf{tail}\ (\mathsf{zip}\ (a +\!\!+\ c)\ (b +\!\!+\ d))) \\
& \quad \text{where } g'\ ((p, q) : xs) = \mathsf{row}\ (\mathsf{transpose}\ (\mathsf{unsplit}_{s_k}\ yss))\ (p +\!\!+\ q)
\end{array}
$$

We now simplify the unfolding function g' to remove the use of $\mathsf{unsplit}_{s_k}$. Suppose that $yss = \langle s, t, u, v\rangle_{s_k}$ — note that $|s| = k$ and, with p as above, $|p| = |s|$

$$
\begin{array}{ll}
& \mathsf{row}\ (\mathsf{transpose}\ (\mathsf{unsplit}_{s_k}\ yss))\ (p +\!\!+\ q) \\
= & \quad \{\text{definition of } \mathsf{row}\} \\
& \mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ (\mathsf{transpose}\ (\mathsf{unsplit}_{s_k}\ yss)) \\
= & \quad \{\mathsf{unsplit}_{s_k} \circ \mathsf{transpose}_{s_k} = \mathsf{transpose} \circ \mathsf{unsplit}_{s_k}\} \\
& \mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ (\mathsf{unsplit}_{s_k}\ (\mathsf{transpose}_{s_k}\ yss)) \\
= & \quad \{yss = \langle s, t, u, v\rangle_{s_k}\} \\
& \mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ (\mathsf{unsplit}_{s_k}\ (\mathsf{transpose}_{s_k}\ \langle s, t, u, v\rangle_{s_k})) \\
= & \quad \{\text{definition of } \mathsf{transpose}_{s_k}\text{, write } \mathsf{transpose}\ s \text{ as } s^T\} \\
& \mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ (\mathsf{unsplit}_{s_k}\ \langle s^T, u^T, t^T, v^T\rangle_{s_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{s_k}\} \\
& \mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ ((s^T \otimes u^T) +\!\!+\ (t^T \otimes v^T)) \\
= & \quad \{\text{property of } \mathsf{map}\} \\
& (\mathsf{map}\,(\mathsf{dotp}\,(p +\!\!+\ q))\ (s^T \otimes u^T)) +\!\!+\ (\mathsf{map}\,(\mathsf{dotp}\,(p +\!\!+\ q))\ (t^T \otimes v^T)) \\
= & \quad \{\text{property 6}\} \\
& (\mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ (s +\!\!+\ u)^T) +\!\!+\ (\mathsf{map}\ (\mathsf{dotp}\ (p +\!\!+\ q))\ (t +\!\!+\ v)^T) \\
= & \quad \{\text{definition of } \mathsf{row}\} \\
& (\mathsf{row}\ (s +\!\!+\ u)^T\ (p +\!\!+\ q)) +\!\!+\ (\mathsf{row}\ (t +\!\!+\ v)^T\ (p +\!\!+\ q))
\end{array}
$$

Thus we can write

$$\begin{array}{l}
\mathsf{mult}\ (\mathsf{unsplit}_{s_k}\ \langle a,b,c,d\rangle_{s_k}, \mathsf{unsplit}_{s_k}\ \langle s,t,u,v\rangle_{s_k}) = \\
\quad \mathsf{unfold\ null}\ g'\ \mathsf{tail}\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d)) \\
\quad\quad \text{where } g'\ ((p,q):xs) = \\
\quad\quad\quad (\mathsf{row}\ (s +\!\!+ u)^T\ (p +\!\!+ q)) +\!\!+ (\mathsf{row}\ (t +\!\!+ v)^T\ (p +\!\!+ q))
\end{array}$$

Since split_{s_k} can be written in terms of fold, we can express mult_{s_k} as a hylomorphism:

$$\begin{array}{l}
\mathsf{mult}_{s_k}\ \langle a,b,c,d\rangle_{s_k} = \mathsf{mult'}_{s_k}\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d)) \\
\mathsf{mult'}_{s_k}\ xsp\ \langle s,t,u,v\rangle_{s_k} = \text{if } \mathsf{null}\ xsp \text{ then } \mathsf{null}_{s_k} \\
\qquad\qquad\qquad\qquad\qquad \text{else } f\ (g'\ xsp)\ (\mathsf{mult'}_{s_k}\ (\mathsf{tail}\ xsp)\ \langle s,t,u,v\rangle_{s_k}) \\
\quad \text{where } g'\ x = (\mathsf{row'}\ (s +\!\!+ u)\ (h\ xs)) +\!\!+ (\mathsf{row'}\ (t +\!\!+ v)\ (h\ xs)) \\
\qquad\quad h = (\mathsf{uncurry}\ (+\!\!+)).\mathsf{head} \\
\qquad\quad \mathsf{row'}\ a\ b = \mathsf{map}\ (\mathsf{dotp}\ b)\ (\mathsf{transpose}\ a) \\
\qquad\quad f\ xs\ \langle ms,ns,ps,qs\rangle_{s_k} \\
\qquad\qquad \left|\begin{array}{ll} |qs| < n-k & = \langle [\,],[\,],(l:ps),(r:qs)\rangle_{s_k} \\ \text{otherwise} & = \langle (l:ms),(r:ns),ps,qs\rangle_{s_k} \end{array}\right. \\
\qquad\qquad \text{where } (l,r) = \mathsf{splitAt}\ k\ xs \\
\qquad\quad n = |s| + |u|
\end{array}$$

Note that since we are multiplying square matrices, we can take $n = |s| + |u|$ and so we do not have to pass n as a parameter into $\mathsf{mult'}_{s_k}$.

To shorten this definition, we would like a function f' satisfying $f' \circ \mathsf{head} = f \circ g'$. Since $|(s +\!\!+ u)^T| = k$, then

$$\begin{array}{l}
\mathsf{splitAt}\ k\ ((\mathsf{row'}\ (s +\!\!+ u)\ (p +\!\!+ q)) +\!\!+ (\mathsf{row'}\ (t +\!\!+ v)\ (p +\!\!+ q))) \\
\quad = (\mathsf{row'}\ (s +\!\!+ u)\ (p +\!\!+ q),\ \mathsf{row'}\ (t +\!\!+ v)\ (p +\!\!+ q))
\end{array}$$

Thus we can define

$$\begin{array}{l}
f'\ (p,q)\ \langle e,f,g,h\rangle_{s_k} \\
\quad \left|\begin{array}{lcl} |h| < n-k & = & \langle [\,],[\,],(l:g),(r:h)\rangle_{s_k} \\ \text{otherwise} & = & \langle (l:e),(r:f),g,h\rangle_{s_k} \end{array}\right. \\
\quad \text{where } (l,r) = (\mathsf{row'}\ (s +\!\!+ u)\ (p +\!\!+ q),\ \mathsf{row'}\ (t +\!\!+ v)\ (p +\!\!+ q))
\end{array}$$

and so $\mathsf{mult'}_{s_k}$ can be written using fold:

$$\begin{array}{l}
\mathsf{mult}_{s_k}\ \langle a,b,c,d\rangle_{s_k}\ m = \mathsf{mult'}_{s_k}\ (\mathsf{zip}\ (a +\!\!+ c)\ (b +\!\!+ d))\ m \\
\mathsf{mult'}_{s_k}\ xsp\ \langle s,t,u,v\rangle_{s_k} = \mathsf{fold}\ f'\ \mathsf{null}_{s_k}\ xsp \\
\ \text{where } \mathsf{row'}\ a\ b = \mathsf{map}\ (\mathsf{dotp}\ b)\ (\mathsf{transpose}\ a) \\
\qquad\quad f'\ (p,q)\ \langle e,f,g,h\rangle_{s_k} \\
\qquad\qquad \left|\begin{array}{ll} |h| < n-k & = \langle [\,],[\,],(l:g),(r:h)\rangle_{s_k} \\ \text{otherwise} & = \langle (l:e),(r:f),g,h\rangle_{s_k} \end{array}\right. \\
\qquad\qquad \text{where } (l,r) = (\mathsf{row'}\ (s +\!\!+ u)\ (p +\!\!+ q),\ \mathsf{row'}\ (t +\!\!+ v)\ (p +\!\!+ q)) \\
\qquad\quad n = |s| + |u|
\end{array}$$

Again, this does not quite match up to the operation in Section 2.2 but a direct derivation does generate the expected result.

4.5 An accessing operation

When using matrices we take it for granted that we can access any element of a matrix. However, when using functional lists to represent matrices we cannot access an element so readily. For lists we have an operation !! which finds an element of a list located at a specific position — xs !! n returns the element of xs located at position n (counting from 0). We can extend this function to work with matrices and define

$$mss \;!!!\; (r, c) = (mss \;!!\; r) \;!!\; c$$

so that if $\mathbf{M} \rightsquigarrow mss$ then $mss \;!!!\; (i, j) = \mathbf{M}(i, j)$.

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

$$\begin{array}{l} \langle mss, nss, pss, qss \rangle_{s_k} \;!!!_{s_k}\; (r, c) \\ \quad \left| \begin{array}{lcl} r < k \;\wedge\; c < k & = & mss \;!!!\; (r, c) \\ r < k \;\wedge\; c \geq k & = & nss \;!!!\; (r, c - k) \\ r \geq k \;\wedge\; c < k & = & pss \;!!!\; (r - k, c) \\ r \geq k \;\wedge\; c \geq k & = & qss \;!!!\; (r - k, c - k) \end{array} \right. \end{array}$$

We now prove that the last case is correct. For the proof we will use the following property of !! from [2]:

Property 8 (List indexing).

$$\begin{array}{lcl} (xs +\!\!+\, ys) \;!!\; k & = & \text{if } k < n \text{ then } xs \;!!\; k \text{else } ys \;!!\; (k - n) \\ & & \qquad \text{where } n = |xs| \end{array}$$

From equation (8), we need to show that

$$xss \;!!!\; (r, c) = (\mathsf{split}\; xss) \;!!!_{s_k}\; (r, c)$$

Suppose that $xss \rightsquigarrow \langle mss, nss, pss, qss \rangle_{s_k}$ and so

$$xss = (mss \otimes nss) +\!\!+\, (pss \otimes qss)$$

and that $k \leq r < n$ and $k \leq c < n$, then

$$
\begin{array}{ll}
& (\mathsf{split}\ xss)\ !!!_{s_k}\ (r, c) \\
= & \quad \{\text{definition of xss}\} \\
& \langle mss, nss, pss, qss \rangle_{s_k}\ !!!_{s_k}\ (r, c) \\
= & \quad \{\text{definition of } !!!_{s_k}, \text{ with } r \geq k \wedge c \geq k\} \\
& qss\ !!!\ (r - k, c - k) \\
= & \quad \{\text{definition of } !!!\} \\
& (qss\ !!\ (r - k))\ !!\ (c - k) \\
= & \quad \{|pss\ !!\ (r - k)| = k,\ c \geq k \text{ and property 8}\} \\
& ((pss\ !!\ (r - k)) +\!\!+ (qss\ !!\ (r - k)))\ !!\ c \\
= & \quad \{\text{definitions of } \otimes \text{ and } !!\} \\
& ((pss \otimes qss)\ !!\ (r - k))\ !!\ c \\
= & \quad \{|mss \otimes nss| = k,\ r \geq k \text{ and property 8}\} \\
& (((mss \otimes nss) +\!\!+ (pss \otimes qss))\ !!\ r)\ !!\ c \\
= & \quad \{\text{definition of } xss\} \\
& (xss\ !!\ r)\ !!\ c \\
= & \quad \{\text{definition of } !!!\} \\
& xss\ !!!\ (r, c)
\end{array}
$$

The proofs of other three cases are similar.

5 Other splits and operations

This section briefly considers possible extensions to the ideas discussed in the previous sections.

5.1 Other Splits

Let us now consider splitting non-square matrices. We can define a (k, l) split which will produce four split components so that the first split component is a matrix of size (k, l).

The choice function is

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

and the family of functions is

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

The representation

$$mss \rightsquigarrow \langle as, bs, cd, ds \rangle_{s_{(k,l)}}$$

with $\mathsf{dim}\ mss = (r, c)$ satisfies the invariant

$$\begin{array}{r} \mathsf{dim}\ as = (k, l) \wedge \mathsf{dim}\ bs = (k, c-l)\ \wedge \\ \mathsf{dim}\ cs = (r-k, l) \wedge \mathsf{dim}\ ds = (r-k, c-l) \end{array} \tag{10}$$

for some $k : (0..r)$ and $l : (0..c)$.

The definition of $!!!_{s_{(k,l)}}$ is similar to the definition of $!!!_k$:

$$\begin{array}{l} \langle mss, nss, pss, qss\rangle_{(k,l)}\ !!!_{s_{(k,l)}}\ (i,j) \\ \left| \begin{array}{lcl} i < k\ \wedge\ j < l & = & mss\ !!!\ (i,j) \\ i < k\ \wedge\ j \geq l & = & nss\ !!!\ (i, j-l) \\ i \geq k\ \wedge\ j < l & = & pss\ !!!\ (i-k, j) \\ i \geq k\ \wedge\ j \geq l & = & qss\ !!!\ (i-k, j-l) \end{array} \right. \end{array}$$

Theorem 1 guarantees that the definitions of addition and scalar multiplication are the same for this split.

We are unable give a simple definition for $\mathsf{transpose}_{s_{(k,l)}}$ as we cannot apply Theorem 1. As an example, suppose that $k \neq l$ and $k < r$ and $l < c$. For $\mathsf{transpose}$, $h = id$ and $\phi^1(i,j) = (j,i)$. Consider the matrix positions $(0,0)$ and $(k{-}1, l{-}1)$. Then $ch(0,0) = 0$ and so, by equation (2)

$$\theta^1(ch(0,0)) = ch(\phi^1(0,0)) \Rightarrow \theta^1(0) = 0$$

But, if $k \neq l$, then $ch(l{-}1, k{-}1) \neq 0$ and so,

$$ch(\phi^1(k-1, l-1)) \neq 0$$

and thus $\theta^1(0) \neq 0$. So, we cannot find a function θ^1 that satisfies equation (2). We can easily show that if $k = l$ then we need to have that $r = c$ so that we can find a suitable θ^1 for rectangular matrices.

Also, we have difficulties defining a multiplication operation for this split; we cannot use our the definition for the $(k{\times}k)$-split because the matrices that we would try to multiply could be non-conformable.

5.2 n square split

For the split in the last section, we had problems defining a transposition operation. We will now consider a more general split than the $(k \times k)$ split.

Suppose that we have a matrix $\mathbf{M}^{k\times k}$ which we want to split into n^2 blocks. We want to ensure that the blocks down the main diagonal are square. We will call this the *n-square matrix split*, denoted by nsq. 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: $nsq = (ch, \mathcal{F})$ such that

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

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

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

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,

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

and thus equation (2) is satisfied. Also,

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

and thus equation (3) is satisfied. Hence the Function Splitting Theorem applies and so if

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

then

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

If we want to split and transpose a rectangular matrix then we could pad out the matrix (with zeros, for instance) to make it square and then split this new matrix.

5.3 Matrix forms

Can we use different matrix forms, such as diagonal matrices and triangular matrices, with splits?

When performing operations on split matrices, we need to ensure that the split invariant is preserved. Suppose that we consider converting matrices into tri-diagonal matrices and then splitting up the matrix according to the elements on the diagonals. Then we would not be able to perform multiplication since the product of two tri-diagonal matrices is not necessarily tri-diagonal.

We would like a form that can be used with all matrices. For example, using diagonal matrices would allow us to split the matrix into components containing the diagonal elements and components containing zeros. However, it is not possible to represent every matrix in diagonal form.

The most natural way to split a matrix is to split it up into rectangular components. Thus it would be difficult to split up upper (or lower) triangular matrices in such a way to partition zeros and non-zero elements.

Block diagonal matrices satisfy the requirements above. A *block diagonal matrix* [10] is a square matrix $\mathbf{M}$ which can be written in the form

$$\begin{pmatrix} \mathbf{A}_0 & & 0 \\ & \ddots & \\ 0 & & \mathbf{A}_{n-1} \end{pmatrix}$$

where each $\mathbf{A}_i$ is a square matrix.

We can write $\mathbf{M}$ as a direct sum:

$$\mathbf{M} = \mathbf{A}_0 \oplus \ldots \oplus \mathbf{A}_{n-1}$$

and so this gives us a convenient way of splitting a matrix. But this method is only applicable if we start with a block diagonal matrix. However, we can use canonical forms, such as *Jordan Form* [10], to provide a way of applying the direct sum split.

5.4 Other operations

We may wish to compute the inverse of a square matrix — *i.e.*, given a matrix $\mathbf{M}$ we would like a matrix $\mathbf{N}$ such that

$$\mathbf{M} \times \mathbf{N} = \mathbf{I} = \mathbf{N} \times \mathbf{M}$$

We can derive an inverse for matrices which have been split by the $(k \times k)$-square split by using the equation for multiplying two split matrices. This then leads to solving four simultaneous equations. In fact, if

$$\mathbf{M} = \begin{pmatrix} \mathbf{M_0} & \mathbf{M_1} \\ \mathbf{M_2} & \mathbf{M_3} \end{pmatrix}$$

then $\mathbf{M^{-1}}$ is

$$\begin{pmatrix} (\mathbf{M_0} - (\mathbf{M_1}\ \mathbf{M_3}^{-1}\ \mathbf{M_2}))^{-1} & ((\mathbf{M_2}\ \mathbf{M_0}^{-1}\ \mathbf{M_1}) - \mathbf{M_3})^{-1}\,\mathbf{M_0}^{-1}\,\mathbf{M_1} \\ \mathbf{M_3}^{-1}\,\mathbf{M_2}(\mathbf{M_1}\,\mathbf{M_3}^{-1}\,\mathbf{M_2} - \mathbf{M_0})^{-1} & (\mathbf{M_3} - \mathbf{M_2}\,\mathbf{M_0}^{-1}\,\mathbf{M_1})^{-1} \end{pmatrix}$$

If we wish to extend the data-type of matrices to support the *determinant* operation then splitting this data-type proves to very difficult for dense matrices as the computation of a determinant usually requires knowledge of the entire matrix. We can however change a matrix into Jordan form (which preserves the value of the determinant) in which case the determinant is just the product of the diagonal values. However, this method is likely to be more computationally expensive.

6 Conclusions

An important concern for applying an obfuscation to an operation is how much the obfuscation will affect the complexity of the operation. In our example split in Section 2.2, we have seen that the computational complexities of the operations were unchanged. Further work needs to be carried out to see how other data-types and splits would be affected.

Another concern is that an obfuscation does not change the functionality of an operation — the *correctness* of the obfuscation. In this paper, we have shown that when we use functional programming, it is easy to check that our obfuscations are correct. Moreover, we have shown how we can construct obfuscated operations from unobfuscated ones. Further work is needed to see how these methods can be adapted for object-oriented languages. One drawback of the derivational approach is that it is easy to "unobfuscate" the operations — the proof of correctness can be viewed as unobfuscating the operations. To prevent this, we must ensure that we keep the split function secret.

We have used previous work on arrays [3] and sets [7] to provide a framework for producing obfuscated matrix operations. The examples that we showed were straightforward so that the techniques could be clearly demonstrated — these techniques are applicable for more complicated splits and for other matrix operations. The framework can be adapted for other data-types and obfuscations. In particular, we have used this framework to obfuscate binary trees by changing their structure.

Deriving split operations using the properties of fold gives a structured method for the derivation. However this approach can only be used when the operation is either a fold or an unfold and often leaves traces of the splitting or unsplitting functions. Although a direct derivation is less prescriptive, it often yields a shorter and more elegant result

It has not been mentioned how "obfuscated" our derived 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. The advantage of considering the obfuscation of the operations of a data-type is that the axiomatic definition of a data-type [12] 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.

Generally, obfuscations are applied to programs by using a transformation toolkit. So, another area for further research is to explore whether our derivations can be automated. A related area is the refactoring of Haskell [11]: refactoring is the process of improving the design of existing code by behaviour-preserving program transformation. Obfuscation can be viewed as the opposite of refactoring. Another related area is the mechanisation of fusion [5]. Could these techniques for automation be used to automate our obfuscations?

Acknowledgements

Thanks to my supervisor Jeff Sanders for the help and support he continually gives. Thanks also to Rani Ettinger, Damien Sereni and Yorck Hünke for useful comments and feedback on the work and to Jeremy Gibbons and Richard Bird for their suggestions on folding and unfolding.

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

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

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

[8] 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.

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

[10] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[11] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *ACM SIGPLAN 2003 Haskell Workshop*. Association for Computing Machinery, August 2003.

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

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

A Proofs of properties

We prove some of the properties from Section 3.5.

Property 5 For a function $\bowtie :: [\alpha] \to [\beta] \to [\gamma]$ and lists as and cs of type $[\alpha]$ and bs and ds of type $[\beta]$ with $|as| = |bs|$:

$$\begin{array}{l}(\mathsf{zipWith}\ (\bowtie)\ as\ bs) +\!\!\!+ (\mathsf{zipWith}\ (\bowtie)\ cs\ ds) = \\ \qquad \mathsf{zipWith}(\bowtie)\ (as +\!\!\!+ cs)\ (bs +\!\!\!+ ds)\end{array}$$

Proof. Suppose that

$$\begin{array}{rcl} as & = & [a_0, a_1, a_2, \ldots, a_n] \\ bs & = & [b_0, b_1, b_2, \ldots, b_n] \\ cs & = & [c_0, c_1, \ldots] \\ \text{and } ds & = & [d_0, d_1, \ldots] \end{array}$$

Then

$$
\begin{array}{ll}
& (\mathsf{zipWith}\ (\bowtie)\ as\ bs) \mathbin{+\!\!+} (\mathsf{zipWith}\ (\bowtie)\ cs\ ds) \\
= & \quad \{\text{definition of } \mathsf{zipWith}\} \\
& [a_0 \bowtie b_0, a_1 \bowtie b_1, \ldots, a_n \bowtie b_n] \mathbin{+\!\!+} [c_0 \bowtie d_0, \ldots] \\
= & \quad \{\text{definition of } \mathbin{+\!\!+}\} \\
& [a_0 \bowtie b_0, a_1 \bowtie b_1, \ldots, a_n \bowtie b_n, c_0 \bowtie d_0, \ldots] \\
= & \quad \{\text{definition of } \mathsf{zipWith}\} \\
& \mathsf{zipWith}\ (\bowtie)\ [a_0, a_1, a_2, \ldots, a_n, c_0, \ldots]\ [b_0, b_1, b_2, \ldots, b_n, d_0, \ldots] \\
= & \quad \{\text{definition of } \mathbin{+\!\!+}\ \} \\
& \mathsf{zipWith}\ (\bowtie)\ (as \mathbin{+\!\!+} cs)\ (bs \mathbin{+\!\!+} ds)
\end{array}
$$

□

Property 6 For matrices mss and nss

$$\mathsf{transpose}\ (mss \mathbin{+\!\!+} nss) = (\mathsf{transpose}\ mss) \otimes (\mathsf{transpose}\ nss)$$

Proof. Let

$$
\begin{array}{lccccc}
 & as & = & \mathsf{map}\ (\mathsf{map}\ \mathsf{wrap})\ mss & = & [a_0, \ldots, a_m] \\
\text{and} & bs & = & \mathsf{map}\ (\mathsf{map}\ \mathsf{wrap})\ nss & = & [b_0, \ldots, b_n]
\end{array}
$$

then

$$
\begin{array}{ll}
& \mathsf{transpose}\ (mss \mathbin{+\!\!+} nss) \\
= & \quad \{\text{definition of } \mathsf{transpose}\} \\
& \mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\ (\mathsf{map\ wrap})\ (mss \mathbin{+\!\!+} nss)) \\
= & \quad \{\text{property of } \mathsf{map}\} \\
& \mathsf{foldr1}\ (\otimes)\ ((\mathsf{map}\ (\mathsf{map\ wrap})\ mss) \mathbin{+\!\!+} (\mathsf{map}\ (\mathsf{map\ wrap})\ nss)) \\
= & \quad \{\text{definitions of } as \text{ and } bs\} \\
& \mathsf{foldr1}\ (\otimes)\ (as \mathbin{+\!\!+} bs) \\
= & \quad \{\text{definition of } \mathsf{foldr1}\} \\
& a_0 \otimes (a_1 \otimes (\ldots \otimes (a_m \otimes (b_0 \otimes (\ldots \otimes (b_{n-1} \otimes b_n) \ldots))) \ldots)) \\
= & \quad \{\otimes \text{ is associative}\} \\
& (a_0 \otimes \ldots \otimes a_m) \otimes (b_0 \otimes \ldots \otimes b_n) \\
= & \quad \{\text{definition of } \mathsf{foldr1}\} \\
& (\mathsf{foldr1}\ (\otimes)\ as) \otimes (\mathsf{foldr1}\ (\otimes)\ bs) \\
= & \quad \{\text{definitions of } as \text{ and } bs\} \\
& (\mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\ (\mathsf{map\ wrap})\ mss)) \otimes \\
& \quad (\mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\ (\mathsf{map\ wrap})\ nss)) \\
= & \quad \{\text{definition of } \mathsf{transpose}\} \\
& (\mathsf{transpose}\ mss) \otimes (\mathsf{transpose}\ nss)
\end{array}
$$

□

Property 7 For matrices mss and nss where $|mss| = |nss|$

$$\mathsf{transpose}(mss \ \otimes \ nss) = \mathsf{transpose}\ mss \ \mathbin{+\!\!+}\ \mathsf{transpose}\ nss$$

Proof. Let

$$
\begin{array}{rcl}
mss & = & [ms_0, ms_1, \ldots, ms_t] \\
nss & = & [ns_0, ns_1, \ldots, ns_t]
\end{array}
$$

and

$$
\begin{array}{rrcccl}
& as & = & \mathsf{map}\ (\mathsf{map\ wrap})\ mss & = & [a_0, \ldots, a_m] \\
\text{and} & bs & = & \mathsf{map}\ (\mathsf{map\ wrap})\ nss & = & [b_0, \ldots, b_n]
\end{array}
$$

So, $a_i = \mathsf{map\ wrap}\ ms_i$ and $b_i = \mathsf{map\ wrap}\ ns_i$.

$$
\begin{array}{ll}
& \mathsf{transpose}(mss \otimes nss) \\
= & \quad \{\text{definition of } \otimes \text{ and } \otimes \text{ is associative}\} \\
& \mathsf{transpose}\ [ms_0 +\!\!+\ ns_0, ms_1 +\!\!+\ ns_1, \ldots , ms_t +\!\!+\ ns_t] \\
= & \quad \{\text{definition of } \mathsf{transpose}\} \\
& \mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\ (\mathsf{map\ wrap})\ [ms_0 +\!\!+\ ns_0, \ldots , ms_t +\!\!+\ ns_t]) \\
= & \quad \{\text{definition of } \mathsf{map}\} \\
& \mathsf{foldr1}\ (\otimes)\ [\mathsf{map\ wrap}\ (ms_0 +\!\!+\ ns_0), \ldots , \mathsf{map\ wrap}\ (ms_t +\!\!+\ ns_t)] \\
= & \quad \{\text{properties of } \mathsf{map}\text{, definitions of } as \text{ and } bs\} \\
& \mathsf{foldr1}\ (\otimes)\ [a_0 +\!\!+\ b_0, \ldots , a_t +\!\!+\ b_t)] \\
= & \quad \{\text{definition of } \mathsf{foldr1}\} \\
& (a_0 +\!\!+\ b_0)\ \otimes\ ((a_1 +\!\!+\ b_1)\ \otimes\ (\ldots\ \otimes\ (a_t +\!\!+\ b_t) \ldots)) \\
= & \quad \{(\otimes) \text{ is associative}\} \\
& ((\ldots(((a_0 +\!\!+\ b_0) \otimes (a_1 +\!\!+\ b_1)) \otimes (a_2 +\!\!+\ b_2)) \ldots) \otimes (a_t +\!\!+\ b_t)) \\
= & \quad \{\text{property 5}\} \\
& ((\ldots(((a_0 \otimes a_1) +\!\!+\ (b_0 \otimes b_1)) \otimes (a_2 +\!\!+\ b_2)) \ldots) \otimes (a_t +\!\!+\ b_t)) \\
= & \quad \{\text{induction, } \otimes \text{ is associative}\} \\
& (a_0 \otimes a_1 \otimes \ldots \otimes a_t) +\!\!+\ (b_0 \otimes b_1 \otimes \ldots \otimes b_t) \\
= & \quad \{\text{definition of } \mathsf{foldr1}\} \\
& (\mathsf{foldr1}\ (\otimes)\ as) +\!\!+\ (\mathsf{foldr1}\ (\otimes)\ bs) \\
= & \quad \{\text{definition of } as \text{ and } bs\} \\
& (\mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\ (\mathsf{map\ wrap})\ mss)) +\!\!+ \\
& \quad (\mathsf{foldr1}\ (\otimes)\ (\mathsf{map}\ (\mathsf{map\ wrap})\ nss)) \\
= & \quad \{\text{definition of } \mathsf{transpose}\} \\
& (\mathsf{transpose}\ mss) +\!\!+\ (\mathsf{transpose}\ nss)
\end{array}
$$

□

B Addition

By the Function Splitting Theorem, we know that under any matrix split, addition can be performed by adding together the corresponding split components. We now show that this is true for the $(k \times k)$-square split:

$$
\begin{array}{cl}
& \mathsf{add}_k\ \langle a,b,c,d\rangle_k\ \langle e,f,g,h\rangle_k \\
= & \quad \{\text{using equation (7)}\} \\
& \mathsf{split}_k(\mathsf{add}\ (\mathsf{unsplit}\ \langle a,b,c,d\rangle_k)\ (\mathsf{unsplit}\ \langle e,f,g,h\rangle_k)) \\
= & \quad \{\text{definition of } \mathsf{unsplit}\} \\
& \mathsf{split}_k(\mathsf{add}\ ((a \otimes b) +\!\!+ (c \otimes d))\ ((e \otimes f) +\!\!+ (g \otimes h))) \\
= & \quad \{\text{definition of } \mathsf{add}\} \\
& \mathsf{split}_k(\mathsf{zipWith}\ (\odot)\ ((a \otimes b) +\!\!+ (c \otimes d))\ ((e \otimes f) +\!\!+ (g \otimes h))) \\
= & \quad \{\text{property 5}\} \\
& \mathsf{split}_k((\mathsf{zipWith}\ (\odot)\ (a \otimes b)\ (e \otimes f)) +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{definition of } \otimes\} \\
& \mathsf{split}_k((\mathsf{zipWith}\ (\odot)\ [a_0 +\!\!+ b_0, \ldots]\ [e_0 +\!\!+ f_0, \ldots]) +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{definition of } \mathsf{zipWith}\} \\
& \mathsf{split}_k([(a_0 +\!\!+ b_0) \odot (e_0 +\!\!+ f_0), \ldots] +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{property 5}\} \\
& \mathsf{split}_k([(a_0 \odot e_0) +\!\!+ (b_0 \odot f_0), \ldots] +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{definition of } \otimes\} \\
& \mathsf{split}_k(([a_0 \odot e_0, \ldots] \otimes [b_0 \odot f_0, \ldots]) +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{definition of } \mathsf{zipWith}\} \\
& \mathsf{split}_k(((\mathsf{zipWith}\ (\odot)\ a\ e) \otimes (\mathsf{zipWith}\ (\odot)\ b\ d)) +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{definition of } \mathsf{add}\} \\
& \mathsf{split}_k(((\mathsf{add}\ a\ e) \otimes (\mathsf{add}\ b\ f)) +\!\!+ \\
& \qquad (\mathsf{zipWith}\ (\odot)\ (c \otimes d)\ (g \otimes h))) \\
= & \quad \{\text{similarly}\} \\
& \mathsf{split}_k(((\mathsf{add}\ a\ e) \otimes (\mathsf{add}\ b\ f)) +\!\!+ ((\mathsf{add}\ c\ g) \otimes (\mathsf{add}\ d\ h))) \\
= & \quad \{\text{definition of } \mathsf{unsplit}\} \\
& \mathsf{split}_k(\mathsf{unsplit}\ \langle \mathsf{add}\ a\ e,\ \mathsf{add}\ b\ f,\ \mathsf{add}\ c\ g,\ \mathsf{add}\ d\ h\rangle_k) \\
= & \quad \{\mathsf{split}_k \circ \mathsf{unsplit} = id\} \\
& \langle \mathsf{add}\ a\ e,\ \mathsf{add}\ b\ f,\ \mathsf{add}\ c\ g,\ \mathsf{add}\ d\ h\rangle_k
\end{array}
$$

Thus, for this split, we perform addition by adding the corresponding split components.

C Scalar Multiplication

By the Function Splitting Theorem, we know that to multiply a split matrix by a scalar p we have to multiply each component by p. We now show this by deriving the operation scale_k:

$$
\begin{aligned}
& \mathsf{scale}_k\ p\ \langle as, bs, cs, ds\rangle_k \\
= & \quad \{\text{using equation (5)}\} \\
& \mathsf{split}_k(\mathsf{scale}\ p\ (\mathsf{unsplit}\ \langle as, bs, cs, ds\rangle_k)) \\
= & \quad \{\text{definition of } \mathsf{unsplit}\} \\
& \mathsf{split}_k(\mathsf{scale}\ p\ ((as \otimes bs) +\!\!+ (cs \otimes ds))) \\
= & \quad \{\text{definition of } \mathsf{scale}\} \\
& \mathsf{split}_k(\mathsf{map}\ (\mathsf{map}\ (\times p))((as \otimes bs) +\!\!+ (cs \otimes ds))) \\
= & \quad \{\text{property of } \mathsf{map}\} \\
& \mathsf{split}_k((\mathsf{map}\ (\mathsf{map}\ (\times p))\ (as \otimes bs)) +\!\!+\ (\mathsf{map}\ (\mathsf{map}\ (\times p))\ (cs \otimes ds))) \\
= & \quad \{\text{definition of } \otimes\} \\
& \mathsf{split}_k((\mathsf{map}\ (\mathsf{map}\ (\times p))\ [a_0 +\!\!+ b_0, a_1 +\!\!+ b_1, \ldots]) +\!\!+ \\
& \qquad (\mathsf{map}\ (\mathsf{map}\ (\times p))\ (cs \otimes ds))) \\
= & \quad \{\text{definition of } \mathsf{map}\} \\
& \mathsf{split}_k\ (([\mathsf{map}\ (\times p)\ (a_0 +\!\!+ b_0),\ \mathsf{map}\ (\times p)\ (a_1 +\!\!+ b_1), \ldots]) +\!\!+ \\
& \qquad (\mathsf{map}\ (\mathsf{map}\ (\times p))\ (cs \otimes ds))) \\
= & \quad \{\text{property of } \mathsf{map}\} \\
& \mathsf{split}_k\ (([(\mathsf{map}\ (\times p)\ a_0) +\!\!+ (\mathsf{map}\ (\times p)\ b_0), \ldots]) +\!\!+ \\
& \qquad (\mathsf{map}\ (\mathsf{map}\ (\times p))\ (cs \otimes ds))) \\
= & \quad \{\text{definition of } \otimes\} \\
& \mathsf{split}_k\ (([\mathsf{map}\ (\times p)\ a_0, \ldots] \otimes [\mathsf{map}\ (\times p)\ b_0, \ldots]) +\!\!+ \\
& \qquad (\mathsf{map}\ (\mathsf{map}\ (\times p))\ (cs \otimes ds))) \\
= & \quad \{\text{definition of } \mathsf{map} \text{ and } \mathsf{scale}\} \\
& \mathsf{split}_k\ (((\mathsf{scale}\ p\ as) \otimes (\mathsf{scale}\ p\ bs)) +\!\!+\ (\mathsf{map}\ (\mathsf{map}\ (\times p))\ (cs \otimes ds))) \\
= & \quad \{\text{similarly}\} \\
& \mathsf{split}_k\ (((\mathsf{scale}\ p\ as) \otimes (\mathsf{scale}\ p\ bs)) +\!\!+\ ((\mathsf{scale}\ p\ cs) \otimes (\mathsf{scale}\ p\ ds))) \\
= & \quad \{\text{definition of } \mathsf{unsplit}\} \\
& \mathsf{split}_k(\mathsf{unsplit}\ \langle \mathsf{scale}\ p\ as,\ \mathsf{scale}\ p\ bs,\ \mathsf{scale}\ p\ cs,\ \mathsf{scale}\ p\ ds\rangle_k) \\
= & \quad \{\mathsf{split}_k \circ \mathsf{unsplit} = id\} \\
& \langle \mathsf{scale}\ p\ as,\ \mathsf{scale}\ p\ bs,\ \mathsf{scale}\ p\ cs,\ \mathsf{scale}\ p\ ds\rangle_k
\end{aligned}
$$

Hence, we have shown that scalar multiplication can be performed by multiplying each split component.