Programming Research Group

OBFUSCATING SET REPRESENTATIONS

Stephen Drape

PRG-RR-04-09

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. Two concerns about an obfuscation are whether it preserves behaviour and how much it changes efficiency.

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 consider and study a type of sets and implementation of its operations using functional lists. An imperative obfuscation — array splitting — is adapted and applied to the set operations. We will see that these particular obfuscations make little difference to efficiency of the operations.

Constructing obfuscations for imperative programs usually requires expensive program analysis, but our method allows us to derive functional obfuscations directly. Whilst establishing the correctness of imperative obfuscations can be a challenging task, our approach enables this to be achieved easily.

To date obfuscation has been an area largely untouched by the formal method approach to program correctness. We have begun to evaluate how a more mathematical view contributes to the study of obfuscating imperative programs.

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 [2, 4]. Currently, obfuscation is mainly applied to object-oriented languages such as Java [7] and C# [1], and achieved using transformation toolkits, (for example [11]). A major concern is the degree to which an obfuscation maintains efficiency.

In this paper, we consider obfuscating programs written in a purely functional language by adapting a known imperative obfuscation. We then apply this obfuscation to functions written in a style based on Haskell [10].

The focus of the paper consists of an object[1] for finite sets having three operations: insertion, deletion and test for membership, specified mathematically. To show the generality of our approach, we consider two obfuscated implementations of each operation; some derived and some verified. We use data refinement [5] to represent a set as a list and then a list as a "split list" — the source of obfuscation. Data refinement techniques then allow us to derive implementations that are correspondingly obfuscated.

We consider the list representations for sets discussed in [3]: unordered lists with duplicates; unordered lists without duplicates and lists in strictly increasing order. For each representation, we define the three operations mentioned above. We show that we can either derive obfuscated operations by using functional composition or prove the correctness of obfuscated operations. Both of these approaches ensure that our obfuscations do not change the functionality of the basic set operations. We also give the computa-

[1] whose methods we tend to call *operations*

tional complexity of each operation and see that our obfuscations make little difference in efficiency.

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 final step to imperative code consists, for our examples, of a standard data refinement consisting of a linked list, for instance.

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 provides 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 some examples of verification.

The rest of the paper is structured as follows. Section 2 introduces the obfuscation that we are going to use and the set operations that we are interested in implementing. Sections 3 and 4 consider the first two set representations and show how we can derive obfuscated operations from unobfuscated ones. Section 5 states some obfuscated operations for ordered lists and the correctness of these operations is established in Section 6. We give some conclusions and areas for future research in Section 7.

2 Preliminaries

2.1 List Splitting

The obfuscation that we consider is list splitting — this is based on *array splitting* [4, 6]. Barak et al. propose a definition of obfuscation and show that it is impossible to obfuscate a program [2]. We use an informal notion of obfuscation — namely that a function is more obfuscated if it is "harder to understand". In functional programming, we may interpret "harder to understand" as, for instance, using a complicated data structure or having more clauses in the definition of a function. We return to the subject of a definition for obfuscation in Section 7.

Let us now consider how to define a *list split*. Suppose that we want to split a list xs into two lists l and r — called the *split components*. As we will be using different splits, we will need to give each split a name. We write

$$xs \sim \langle l, r \rangle_{sp}$$

to denote that we can represent xs by two lists l and r by a split called sp — so xs is data refined by $\langle l, r \rangle_{sp}$. For every split sp, we define a *splitting function* — called split_{sp} — and we require that the splitting function has an inverse — called $\mathsf{unsplit}_{sp}$. These two functions determine the relationship between l, r and xs:

$$\begin{aligned} xs \sim \langle l, r \rangle_{sp} \quad \Leftrightarrow \quad & \mathsf{split}_{sp}(xs) = \langle l, r \rangle_{sp} \ \wedge \\ & \mathsf{unsplit}_{sp}(\langle l, r \rangle_{sp}) = xs \end{aligned}$$

Thus,

$$\mathsf{unsplit}_{sp} \circ \mathsf{split}_{sp} = id \quad (1)$$

$$\mathsf{split}_{sp} \circ \mathsf{unsplit}_{sp} = id \quad (2)$$

where $\circ$ is functional composition.

Suppose that we have a list xs which contains elements of type T, then if $\mathsf{split}_{sp}(xs) = \langle l, r\rangle_{sp}$ we require that

$$(\forall x \in T)\ \mathsf{freq}(x, xs) = \mathsf{freq}(x, l) + \mathsf{freq}(x, r) \quad (3)$$

where $\mathsf{freq}(x, xs)$ is the frequency of the occurrence of an element x in the list xs. From equation (3), it is immediate that

$$|xs| = |l| + |r|$$

where $|ys|$ denotes the length of ys.

For our splits, we will need to know whether they preserve ordering. So, if we split a list that is increasing (decreasing), are both of the split components increasing (decreasing)? To help us answer this question, we define the notion of a *sublist* (if we think of lists as sequences then sublists are analogous to subsequences).

We write $ls \trianglelefteq xs$ if ls is a sublist of xs. The operator $\trianglelefteq$ can be defined as follows:

$$\begin{array}{rcl} [\,] \trianglelefteq xs & = & \mathsf{True} \\ (l : ls) \trianglelefteq [\,] & = & \mathsf{False} \\ (l : ls) \trianglelefteq (x : xs) & = & \mathsf{if}\ l == x\ \mathsf{then}\ ls \trianglelefteq xs\ \mathsf{else}\ (l : ls) \trianglelefteq xs \end{array}$$

From this, we can see that if $ls \trianglelefteq xs$ and xs is increasing (decreasing) then ls is also increasing (decreasing). So, when we split a list xs, if the split components are both sublists of xs then we know that the split has preserved the ordering.

When obfuscating the set operations, we will use two specific splits described in the next two subsections.

2.1.1 Alternating Split

The *Alternating Split* (written aSp) splits a list into two by placing the even placed elements (counting from zero) in the first component and the remaining elements into the second component. So, for instance

$$[4, 3, 1, 1, 5, 8, 2] \sim \langle [4, 1, 5, 2], [3, 1, 8]\rangle_{aSp}$$

The representation $xs \sim \langle l, r\rangle_{aSp}$ satisfies the invariant:

$$|r| \leq |l| \leq |r| + 1 \quad (4)$$

We will strengthen this invariant in Section 5.1 when we work with ordered lists.

Now, we define the splitting function:

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

We need split_{aSp} to be invertible — here is the inverse function:

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

Both of these operations take time proportional to $|l| + |r|$ $(=|xs|)$. We can easily check that equations (1) to (4) hold.

As a split list represents a list, we should ensure that we can use familiar list operations with splits. As the components of a split are lists, we can define the following:

$$\begin{aligned}
a : \langle l, r \rangle_{aSp} &= \langle a : r, l \rangle_{aSp} \\
\mathsf{head}\ \langle l, r \rangle_{aSp} &= \mathsf{head}\ l \\
\mathsf{tail}\ \langle l, r \rangle_{aSp} &= \langle r, \mathsf{tail}\ l \rangle_{aSp} \\
\langle l_0, r_0 \rangle_{aSp} +\!\!+ \langle l_1, r_1 \rangle_{aSp} &= \begin{cases} \langle l_0 +\!\!+ l_1, r_0 +\!\!+ r_1 \rangle_{aSp} & \text{if } |l_0| = |r_0| \\ \langle l_0 +\!\!+ r_1, r_0 +\!\!+ l_1 \rangle_{aSp} & \text{otherwise} \end{cases}
\end{aligned}$$

These operations can be verified using the definition of split_{aSp} and will be useful in our derivations and proofs. From these definitions, we can show that $:$ and $+\!\!+$ distribute over split_{aSp} and $\mathsf{unsplit}_{aSp}$. Thus:

$$\begin{aligned}
\mathsf{split}_{aSp}(x : xs) &= x : \mathsf{split}_{aSp}(xs) \\
\mathsf{unsplit}_{aSp}(x : xs) &= x : \mathsf{unsplit}_{aSp}(xs) \\
\mathsf{split}_{aSp}(xs +\!\!+ ys) &= \mathsf{split}_{aSp}(xs) +\!\!+ \mathsf{split}_{aSp}(ys) \\
\mathsf{unsplit}_{aSp}(xs +\!\!+ ys) &= \mathsf{unsplit}_{aSp}(xs) +\!\!+ \mathsf{unsplit}_{aSp}(ys)
\end{aligned}$$

These equations can be proved using structural induction.

The implementation of filter is a little more subtle. We might expect to define

$$\mathsf{filter}\ p\ \langle l, r \rangle_{aSp} = \langle \mathsf{filter}\ p\ l, \mathsf{filter}\ p\ r \rangle_{aSp}$$

But, for example,

$$\mathsf{filter}\ (\mathsf{even})\ \langle [1, 4, 6, 8], [2, 5, 7] \rangle_{aSp}$$

would give

$$\langle [1], [5, 7] \rangle_{aSp}$$

which violates the invariant (4). Performing a filter requires rebalancing the split components, making sure we preserve the ordering of the elements. Thus we are not able to give a concise definition for filter.

2.1.2 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 instance,

$$[4,3,1,1,5,8,2] \sim \langle [4,3,1],[1,5,8,2] \rangle_{b_3}$$

If the list xs has at most k elements then,

$$xs \sim \langle xs, [\,] \rangle_{b_k}$$

The representation $xs \sim \langle l, r \rangle_{b_k}$ satisfies the invariant:

$$(|r| = 0 \wedge |l| < k) \vee (|l| = k) \tag{5}$$

This invariant will be strengthened in Section 5.2.

The splitting operation is defined to be:

$$\begin{aligned} \mathsf{split}_{b_k}\ xs &= \langle l, r \rangle_{b_k} \\ &\quad \text{where } (l, r) = \mathsf{splitAt}\ k\ xs \end{aligned}$$

We need split_{b_k} to be invertible and so by using the properties of $\mathsf{splitAt}$, we define:

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

These are constant time operations (as $|l| \leq k$ and k is constant) and we can easily show that equations (1), (2), (3) and (5) are satisfied.

As with the alternating split, we can use common list operations with the k-block split:

$$\begin{aligned}
a : \langle l, r \rangle_{b_k} &= \begin{cases} \langle a : l, r \rangle_{b_k} & \text{if } |l| < k \\ \langle a : (\mathsf{init}\ l), (\mathsf{last}\ l) : r \rangle_{b_k} & \text{otherwise} \end{cases} \\
\mathsf{head}\ \langle l, r \rangle_{b_k} &= \mathsf{head}\ l \\
\mathsf{tail}\ \langle l, r \rangle_{b_k} &= \begin{cases} \langle \mathsf{tail}\ l, r \rangle_{b_k} & \text{if } |r| = [\,] \\ \langle \mathsf{tail}\ l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r \rangle_{b_k} & \text{otherwise} \end{cases} \\
\langle l_0, r_0 \rangle_{b_k} +\!\!+ \langle l_1, r_1 \rangle_{b_k} &= \begin{cases} \langle l_0, r_0 +\!\!+ l_1 +\!\!+ r_1 \rangle_{b_k} & \text{if } |l_0| = k \\ \langle l_0 +\!\!+ l', r' \rangle_{b_k} & \text{otherwise} \end{cases} \\
&\quad (l', r') = \mathsf{splitAt}\ (k - |l_0|)(r_0 +\!\!+ l_1 +\!\!+ r_1)
\end{aligned}$$

We can show that $:$ and $+\!\!+$ distribute over split_{b_k} and $\mathsf{unsplit}_{b_k}$ and, as before, we do not have a succinct definition for filter.

2.2 Set operations

The focus for this paper is an object for sets with the following three basic operations:

$$\begin{array}{rl} x \in S & \mathsf{member}\ x\ S \\ S \cup \{x\} & \mathsf{insert}\ x\ S \\ S \backslash \{x\} & \mathsf{delete}\ x\ S \end{array}$$

We will assume that the sets are finite and the elements of the set can be ordered. We will use three functional list representation to implement these operations.

When using sets, we commonly use union $(\cup)$, $\mathsf{intersection}$ $(\cap)$ and minus $(-)$. However, since we are using lists to represent the sets, we can define union and minus using insert and delete and $\mathsf{intersection}$ using minus. In fact,

$$\begin{array}{rcl} \mathsf{union}\ A\ B & = & \mathsf{flip}\ (\mathsf{foldr}\ \mathsf{insert1})\ A\ B \\ \mathsf{minus}\ A\ B & = & \mathsf{foldr}\ \mathsf{delete}\ A\ B \\ \mathsf{intersection}\ A\ B & = & \mathsf{minus}\ A\ (\mathsf{minus}\ A\ B) \end{array}$$

We will look at the definition of these operations again when we use ordered lists.

For each list representation, we will define the three basic operations above; but how can we define the corresponding operations for split lists? Defining member is straightforward.

Suppose that $xs \sim \langle l, r \rangle_{sp}$. Then from equation (3), we know that each member of xs must be a member of l or r, therefore we can define:

$$\mathsf{member}_{sp}\ x\ \langle l, r \rangle_{sp} = (\mathsf{member}\ x\ l) \vee (\mathsf{member}\ x\ r) \tag{6}$$

Defining the other two basic operations requires more work.

Suppose that we have a list operation:

$$\mathsf{op} : List \rightarrow List$$

and we want to define the corresponding split list operation:

$$\mathsf{split\_op} : \langle List, List \rangle \rightarrow \langle List, List \rangle$$

We would like $\mathsf{split\_op}$ to satisfy the following commuting diagram [5]:

$$\begin{tikzcd} xs \arrow[r, "\mathsf{op}"] \arrow[d, "\mathsf{split}"'] & \mathsf{op}\ (xs) \arrow[d, "\mathsf{split}"'] \\ \langle l, r \rangle \arrow[r, "\mathsf{split\_op}"] & \mathsf{split\_op}(\langle l, r \rangle) \end{tikzcd}$$

As all our operations are functions, we require that:

$$\mathsf{split} \circ \mathsf{op} = \mathsf{split\_op} \circ \mathsf{split} \tag{7}$$

Since $\mathsf{unsplit}$ is the inverse for split, we can pre-compose (7) by $\mathsf{unsplit}$ to obtain:

$$\mathsf{op} = \mathsf{unsplit} \circ \mathsf{split\_op} \circ \mathsf{split} \tag{8}$$

Similarly, we can post-compose (7) by $\mathsf{unsplit}$ to give:

$$\mathsf{split\_op} = \mathsf{split} \circ \mathsf{op} \circ \mathsf{unsplit} \tag{9}$$

Thus, we can use equation (9) to derive a split operation from the corresponding list operation or we can state a split operation and then use equation (8) to prove it is correct.

From a practical point of view, the obfuscator needs to keep split and $\mathsf{unsplit}$ secret, otherwise an attacker could reconstruct op from $\mathsf{split\_op}$ by using equation (8).

2.3 Using Lists

To obfuscate set operations, we have to do the following: represent sets and set operations using lists, split the lists and then derive the obfuscated operations. Can we just split the sets themselves and then derive the obfuscated set operations directly?

Suppose that we had started with sets in Section 2.1 and had wanted to split the set $\{1,2,3,4\}$ using the "alternating split". Since we can write the elements of $\{1,2,3,4\}$ in any order, the following splits are all valid:

$$\langle\{1,3\},\{2,4\}\rangle_{aSp}\,, \quad \langle\{1,2\},\{3,4\}\rangle_{aSp}\,, \quad \langle\{4,3\},\{2,1\}\rangle_{aSp} \quad \ldots$$

This means that split_{aSp} is now a relation but not a function and so inverting it, as we did in the previous section, may not produce functional implementations directly.

3 Unordered Lists with duplicates

Now, we consider our first list representation of sets. Then we will implement some set operations and derive obfuscated operations for our two splits.

If we represent a set as an unordered list with duplicates (i.e. there is no particular data-type invariant on the list) then the implementations for the basic operations are:

$$\begin{aligned} \mathsf{member}\ a\ xs &= \mathsf{or}\ (\mathsf{map}\ (== a)\ xs) \\ \mathsf{insert}\ a\ xs &= a : xs \\ \mathsf{delete}\ a\ xs &= \mathsf{filter}\ (\neq a)\ xs \end{aligned}$$

The advantage of this representation is that insert is a constant time operation but, in the worst case, member and delete have complexity $O(|xs|)$.

3.1 Alternating split

Let us now derive the three basic operations for the alternating split in terms of the operations defined above — we use the subscript $_{aSp}$ to denote operations for this split.

Using equation (6), we define

$$\mathsf{member}_{aSp}\ a\ \langle l, r\rangle_{aSp} = \mathsf{member}\ a\ l \vee \mathsf{member}\ a\ r$$

This operation has complexity $O(|l| + |r|)$. Since $|xs| = |l| + |r|$ then the complexity is the same as for member.

The other two operations are derived by using equation (9), i.e.

$$\mathsf{op}_{aSp}\ \langle l, r\rangle_{aSp} = \mathsf{split}_{aSp}(\mathsf{op}\ (\mathsf{unsplit}_{aSp}\ \langle l, r\rangle_{aSp}))$$

We have difficulties deriving delete_{aSp} as we do not have a simple way of defining filter for this split. Thus we define

$$\mathsf{delete}_{aSp}\ a\ \langle l, r\rangle_{aSp} = \mathsf{split}_{aSp}(\mathsf{delete}\ a\ (\mathsf{unsplit}_{aSp}(\langle l, r\rangle_{aSp})))$$

Doing the unsplit/split combination will usually be easier than removing all the occurrences of a from the split components and then re-balancing them to maintain the invariant (4). As all of the operations in the definition are linear, this operation takes time proportional to $|l| + |r|$ — the same complexity as delete. This definition is undesirable from an obfuscation point of view as we have to use split_{aSp} and $\mathsf{unsplit}_{aSp}$ in our definition. However, we will see that with the next two set representations, the definitions of delete_{aSp} do not contain uses of split_{aSp} or $\mathsf{unsplit}_{aSp}$.

We now derive insert_{aSp}. Suppose that

$$\langle l, r\rangle_{aSp} = \langle [l_0, l_1, \ldots], [r_0, r_1, \ldots]\rangle_{aSp}$$

Then

$$\begin{array}{cl}
 & \mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp} \\
= & \quad \{\text{using equation (9)}\} \\
 & \mathsf{split}_{aSp}(\mathsf{insert}\ a\ (\mathsf{unsplit}_{aSp}\ \langle l, r\rangle_{aSp})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{aSp}\} \\
 & \mathsf{split}_{aSp}(\mathsf{insert}\ a\ [l_0, r_0, l_1, r_1, \ldots]) \\
= & \quad \{\text{definition of } \mathsf{insert}\} \\
 & \mathsf{split}_{aSp}\ [a, l_0, r_0, l_1, r_1, \ldots] \\
= & \quad \{\text{definition of } \mathsf{split}_{aSp}\} \\
 & \langle [a, r_0, r_1, \ldots], [l_0, l_1, \ldots]\rangle_{aSp} \\
= & \quad \{\text{definition of } l \text{ and } r\} \\
 & \langle a : r, l\rangle_{aSp} \\
= & \quad \{\text{definition of } \mathsf{insert}\} \\
 & \langle \mathsf{insert}\ a\ r, l\rangle_{aSp}
\end{array}$$

So, we define

$$\mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp} = \langle \mathsf{insert}\ a\ r, l\rangle_{aSp}$$

We can see that insert_{aSp} is a constant time operation — the same as the unsplit version.

3.2 Block Split

As with the alternating split, we derive the basic set operations for the k-block split — denoted by a subscript ${}_{b_k}$. As we do not have a simple definition for filter, we define delete_{b_k} in terms of split_{b_k} and $\mathsf{unsplit}_{b_k}$:

$$\mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k} = \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (\mathsf{unsplit}_{b_k}\langle l, r\rangle_{b_k}))$$

and, using equation (6), we define:

$$\mathsf{member}_{b_k}\ a\ \langle l, r\rangle_{b_k} = \mathsf{member}\ a\ l \vee \mathsf{member}\ a\ r$$

Both these operations take time proportional to $|l| + |r|$.

Let us now consider how to derive the insert operation.

$$\begin{array}{cl}
 & \mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
= & \quad \{\text{using equation (9)}\} \\
 & \mathsf{split}_{b_k}\ (\mathsf{insert}\ a\ (\mathsf{unsplit}_{b_k}\ \langle l, r\rangle_{b_k})) \\
= & \quad \{\text{definition of } \mathsf{insert}\} \\
 & \mathsf{split}_{b_k}\ (a : (\mathsf{unsplit}_{b_k}\ \langle l, r\rangle_{b_k})) \\
= & \quad \{\text{properties of } :\} \\
 & a : (\mathsf{split}_{b_k}\ (\mathsf{unsplit}_{b_k}\ \langle l, r\rangle_{b_k})) \\
= & \quad \{\mathsf{unsplit}_{b_k} \text{ is the inverse of } \mathsf{split}_{b_k}\} \\
 & a : \langle l, r\rangle_{b_k} \\
= & \quad \{\text{definition of : for } b_k\} \\
 & \ \mathsf{if}\ |l| < k\ \ \mathsf{then}\ \langle a : l, r\rangle_{b_k} \\
 & \qquad\qquad\quad \mathsf{else}\ \langle a : (\mathsf{init}\ l), (\mathsf{last\ l}) : r\rangle_{b_k}
\end{array}$$

So, we define

$$\begin{array}{lcl}
\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k} & = & \mathsf{if}\ |l| < k \\
 & & \qquad \mathsf{then}\ \langle \mathsf{insert}\ a\ l, r\rangle_{b_k} \\
 & & \qquad \mathsf{else}\ \langle \mathsf{insert}\ a\ (\mathsf{init}\ l), \mathsf{insert}\ (\mathsf{last}\ l)\ r\rangle_{b_k}
\end{array}$$

This definition uses the operations init and last which are usually quite expensive. We can obtain a more efficient version by exploiting the fact that the split sets are unordered. Having unordered lists means that we can insert the new element into any place in either of the split components. If the length of l is k then we insert the new element into r, otherwise we insert it into l. This gives

$$\begin{array}{lcl}
\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k} & = & \mathsf{if}\ |l| < k\ \mathsf{then}\ \langle \mathsf{insert}\ a\ l, r\rangle_{b_k} \\
 & & \qquad\qquad\quad \mathsf{else}\ \langle l, \mathsf{insert}\ a\ r\rangle_{b_k}
\end{array}$$

As l has length at most k (and k is constant), the test $|l| < k$ takes constant time. Thus, we conclude that insert_{b_k} is a constant time operation — the unsplit version was also a constant time operation.

4 Unordered Lists without duplicates

Let us now consider unordered lists without duplicates. If xs is such a list then it has to satisfy the following invariant:

$$(\forall\, 0 \le m < n < |xs|) \quad xs \;!!\; m \ne xs \;!!\; n \tag{10}$$

where !! is the list indexer.

For this representation, we can define the three basic operations as follows:

$$\begin{aligned}
\mathsf{member}\ a\ xs &= \mathsf{or}\ (\mathsf{map}\ (== a)\ xs)\\
\mathsf{insert}\ a\ xs &= \mathsf{if\ member}\ a\ xs\ \mathsf{then}\ xs\ \mathsf{else}\ a : xs\\
\mathsf{delete}\ a\ xs &= ys +\!\!+ (\mathsf{if\ null}\ zs\ \mathsf{then}\ zs\ \mathsf{else\ tail}\ zs)\\
&\quad \text{where } (ys, zs) = \mathsf{span}\ (\ne a)\ xs
\end{aligned}$$

All three operations now have complexity $O(|xs|)$, but delete should be more efficient than before as we should only have to remove one occurrence of an element.

4.1 Alternating split

The definition of member_{aSp} is the same as before:

$$\mathsf{member}_{aSp}\ a\ \langle l, r\rangle_{aSp} = \mathsf{member}\ a\ l \vee \mathsf{member}\ a\ r$$

The insert operations for unordered lists without duplicates is the same as the operation for lists with duplicates except that we have an extra test for membership. Thus we define

$$\begin{aligned}
\mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp} &= \mathsf{if\ member}_{aSp}\ a\ \langle l, r\rangle_{aSp}\\
&\quad\ \mathsf{then}\ \langle l, r\rangle_{aSp}\\
&\quad\ \mathsf{else}\ \langle a : r, l\rangle_{aSp}
\end{aligned}$$

Note that we have written $a : r$ instead of insert $a\ r$. This is because insert contains a membership test — which has already been performed. This gives us an operation which has complexity $O(|l| + |r|)$ — which is the same as the unsplit version.

Let us now turn our attention to delete_{aSp}. Suppose that $l = [l_0, l_1, \ldots]$ and $r = [r_0, r_1, \ldots]$. We have three cases to consider.

(i) If $a \in l$ then since we do not have any duplicates, there must be exactly one element of l, say l_j, such that $l_j = a$ and also we know that $a \notin r$. We define

$$\begin{aligned}
(ly, lz) &= \mathsf{span}\ (\ne a)\ l\\
(ry, rz) &= \mathsf{splitAt}\ |ly|\ r
\end{aligned}$$

As $l_j = a$, we find that

$$\begin{aligned}
ly &= [l_0, l_1, \ldots, l_{j-1}]\\
ry &= [r_0, r_1, \ldots, r_{j-1}]\\
lz &= [l_j, l_{j+1}, \ldots]\\
rz &= [r_j, r_{j+1}, \ldots]
\end{aligned}$$

Then,

$$
\begin{array}{cl}
& \mathsf{delete}_{aSp}\ a\ \langle l, r\rangle_{aSp} \\
= & \quad \{\text{using equation } (9)\} \\
& \mathsf{split}_{aSp}(\mathsf{delete}\ a\ (\mathsf{unsplit}_{aSp}\langle l, r\rangle_{aSp})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{aSp}\} \\
& \mathsf{split}_{aSp}(\mathsf{delete}\ a\ [l_0, r_0, l_1, r_1, \ldots, l_{j-1}, r_{j-1}, l_j, r_j, l_{j+1} \ldots]) \\
= & \quad \{\text{definition of } \mathsf{delete} \text{ and } l_j = a\} \\
& \mathsf{split}_{aSp}([l_0, r_0, l_1, r_1, \ldots, l_{j-1}, r_{j-1}] \mathbin{+\!\!+} [r_j, l_{j+1} \ldots]) \\
= & \quad \{\mathsf{split}_{aSp} \text{ distributes over } \mathbin{+\!\!+}\} \\
& \mathsf{split}_{aSp}([l_0, r_0, l_1, r_1, \ldots, l_{j-1}, r_{j-1}]) \mathbin{+\!\!+} \mathsf{split}_{aSp}([r_j, l_{j+1} \ldots]) \\
= & \quad \{\text{definition of } \mathsf{split}_{aSp}\} \\
& \langle [l_0, l_1 \ldots, l_{j-1}], [r_0, r_1, \ldots, r_{j-1}]\rangle_{aSp} \mathbin{+\!\!+} \langle [r_j, \ldots], [l_{j+1}, \ldots]\rangle_{aSp} \\
= & \quad \{\text{definitions above}\} \\
& \langle ly, ry\rangle_{aSp} \mathbin{+\!\!+} \langle rz, \mathsf{tail}\ lz\rangle_{aSp}
\end{array}
$$

If we apply the definition of $\mathbin{+\!\!+}$ for aSp, knowing that $|ly| = |ry|$, we would obtain

$$\langle ly \mathbin{+\!\!+} rz, ry \mathbin{+\!\!+} (\mathsf{tail}\ lz)\rangle_{aSp}$$

We cannot simplify this expression any further. However, we can use the fact that the lists are unordered so that, as $|ly| = |ry|$, the sets represented by $\langle ly, ry\rangle_{aSp}$ and $\langle ry, ly\rangle_{aSp}$ are the same. From above, we have:

$$
\begin{array}{cl}
& \langle ly, ry\rangle_{aSp} \mathbin{+\!\!+} \langle rz, \mathsf{tail}\ lz\rangle_{aSp} \\
= & \quad \{\text{the lists are unordered and } |ly| = |ry|\} \\
& \langle ry, ly\rangle_{aSp} \mathbin{+\!\!+} \langle rz, \mathsf{tail}\ lz\rangle_{aSp} \\
= & \quad \{|ry| = |ly|, \text{ definition of } \mathbin{+\!\!+}\} \\
& \langle ry \mathbin{+\!\!+} rz, ly \mathbin{+\!\!+} \mathsf{tail}\ lz\rangle_{aSp} \\
= & \quad \{\text{definitions of } ry \text{ and } rz\} \\
& \langle r, ly \mathbin{+\!\!+} \mathsf{tail}\ lz\rangle_{aSp} \\
= & \quad \{\text{definition of } \mathsf{delete}\} \\
& \langle r, \mathsf{delete}\ a\ l\rangle_{aSp}
\end{array}
$$

(ii) Now suppose that $a \in r$ and so we know that $l \neq [\,]$. Then

$$
\begin{array}{ll}
& \mathsf{delete}_{aSp}\ a\ \langle l, r\rangle_{aSp} \\
= & \quad \{\mathsf{head}\ l = l_0\} \\
& \mathsf{delete}_{aSp}\ a\ \langle l_0 : (\mathsf{tail}\ l), r\rangle_{aSp} \\
= & \quad \{\text{definition of : for } aSp\} \\
& \mathsf{delete}_{aSp}\ a\ (l_0 : \langle r, \mathsf{tail}\ l\rangle_{aSp}) \\
= & \quad \{\text{as } a \neq l_0\} \\
& l_0 : (\mathsf{delete}_{aSp}\ a\ \langle r, \mathsf{tail}\ l\rangle_{aSp}) \\
= & \quad \{\text{using previous definition of } \mathsf{delete}_{aSp}\} \\
& l_0 : \langle \mathsf{tail}\ l, \mathsf{delete}\ a\ r\rangle_{aSp} \\
= & \quad \{l_0 = \mathsf{head}\ l, \text{ definition of : for } aSp\} \\
& \langle (\mathsf{head}\ l) : (\mathsf{delete}\ a\ r), \mathsf{tail}\ l\rangle_{aSp}
\end{array}
$$

(iii) The final case is when $a \notin l \wedge a \notin r$ and so

$$\mathsf{delete}_{aSp}\ a\ \langle l, r\rangle_{aSp} = \langle l, r\rangle_{aSp}$$

Thus the three cases give the following function which again is linear time:

$$
\begin{array}{l}
\mathsf{delete}_{aSp}\ a\ \langle l, r\rangle_{aSp} \\
\quad \left| \begin{array}{lcl}
\mathsf{member}\ a\ l & = & \langle r, \mathsf{delete}\ a\ l\rangle_{aSp} \\
\mathsf{member}\ a\ r & = & \langle (\mathsf{head}\ l) : (\mathsf{delete}\ a\ r), \mathsf{tail}\ l\rangle_{aSp} \\
\text{otherwise} & = & \langle l, r\rangle_{aSp}
\end{array} \right.
\end{array}
$$

4.2 Block split

As before, for the block split:

$$\mathsf{member}_{b_k}\ a\ \langle l, r\rangle_{b_k} = \mathsf{member}\ a\ l\ \vee \mathsf{member}\ a\ r$$

As with the alternating split, we can define insert_{b_k} by adding a membership test to the definition for unordered lists with duplicates. So,

$$
\begin{array}{lcl}
\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k} & = & \mathsf{if}\ \mathsf{member}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
& & \mathsf{then}\ \langle l, r\rangle_{b_k} \\
& & \mathsf{else}\quad \mathsf{if}\ |l| < k \quad \mathsf{then}\ \langle a : l, r\rangle_{b_k} \\
& & \qquad\qquad\qquad\quad\ \mathsf{else}\ \langle l, a : r\rangle_{b_k}
\end{array}
$$

Again, as we have a membership test at the beginning, we use the : operation instead of using insert — this gives us a linear operation.

To define $\mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k}$, we have four cases.

(i) Suppose that $r = [\,]$. Then

$$\begin{array}{ll}
& \mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
= & \quad \{\text{using equation (9)}\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (\mathsf{unsplit}_{b_k}\ \langle l, r\rangle_{b_k})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (l +\!\!+ r)) \\
= & \quad \{r = [\,]\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ l) \\
= & \quad \{\text{definition of } \mathsf{split}_{b_k} \text{ and } |\mathsf{delete}\ a\ l| \leq k\} \\
& \langle \mathsf{delete}\ a\ l, [\,]\rangle_{b_k}
\end{array}$$

(ii) Suppose that $a \in l$ and $r \neq [\,]$ — since there are no duplicates, $a \notin r$. Then

$$\begin{array}{ll}
& \mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
= & \quad \{\text{using equation (9)}\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (\mathsf{unsplit}_{b_k}\ \langle l, r\rangle_{b_k})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (l +\!\!+ r)) \\
= & \quad \{\text{definition of } \mathsf{delete},\ r = (\mathsf{head}\ r) : (\mathsf{tail}\ r)\} \\
& \mathsf{split}_{b_k}((\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r] +\!\!+ (\mathsf{tail}\ r)) \\
= & \quad \{\text{definition of } \mathsf{split}_{b_k},\ |(\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r]| = k\} \\
& \langle (\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r\rangle_{b_k} \\
= & \quad \{\text{lists are unordered}\} \\
& \langle \mathsf{head}\ r : (\mathsf{delete}\ a\ l), \mathsf{tail}\ r\rangle_{b_k}
\end{array}$$

(iii) Suppose that $a \in r$. Then

$$\begin{array}{ll}
& \mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
= & \quad \{\text{using equation (9)}\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (\mathsf{unsplit}_{b_k}\ \langle l, r\rangle_{b_k})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& \mathsf{split}_{b_k}(\mathsf{delete}\ a\ (l +\!\!+ r)) \\
= & \quad \{\text{definition of } \mathsf{delete},\ a \notin l\} \\
& \mathsf{split}_{b_k}(l +\!\!+ (\mathsf{delete}\ a\ r)) \\
= & \quad \{\text{definition of } \mathsf{split}_{b_k}\} \\
& \langle l, \mathsf{delete}\ a\ r\rangle_{b_k}
\end{array}$$

(iv) If $a \notin \langle l, r\rangle_{b_k}$ then $\mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k}$ should leave the split unchanged. Note that if $a \notin r$ then $\mathsf{delete}\ a\ r = r$, so we can combine the last two cases.

Putting all the cases together gives

$$
\begin{array}{l}
\mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
\quad \left|\begin{array}{lcl}
r == [\,] & = & \langle \mathsf{delete}\ a\ l, r\rangle_{b_k} \\
\mathsf{member}\ a\ l & = & \langle (\mathsf{head}\ r) : (\mathsf{delete}\ a\ l), \mathsf{tail}\ r\rangle_{b_k} \\
\text{otherwise} & = & \langle l, \mathsf{delete}\ a\ r\rangle_{b_k}
\end{array}\right.
\end{array}
$$

In the worst case, as $|l| \leq k$, this will take time proportional to $|r|$.

5 Strictly Ordered Lists

In this section, we represent sets by lists which are in strictly-increasing order. If xs is a strictly-increasing ordered list then it satisfies the following invariant:

$$
(\forall\, 0 \leq m < n < |xs|) \quad xs\ !!\ m < xs\ !!\ n \tag{11}
$$

The three basic operations are now defined as:

$$
\begin{array}{rcl}
\mathsf{member}\ a\ xs & = & \mathsf{if\ null}\ ys\ \mathsf{then}\ \mathit{False}\ \mathsf{else}\ (a == \mathsf{head}\ ys) \\
 & & \text{where } ys = \mathsf{dropWhile}\ (< a)\ xs \\
\mathsf{insert}\ a\ xs & = & ys\ +\!\!+\ (\mathsf{if\ null}\ zs \vee \mathsf{head}\ zs \neq a\ \mathsf{then}\ a : zs\ \mathsf{else}\ zs) \\
 & & \text{where } (ys, zs) = \mathsf{span}\ (< a)\ xs \\
\mathsf{delete}\ a\ xs & = & ys\ +\!\!+\ (\mathsf{if\ null}\ zs \vee \mathsf{head}\ zs \neq a\ \mathsf{then}\ zs\ \mathsf{else\ tail}\ zs) \\
 & & \text{where } (ys, zs) = \mathsf{span}\ (< a)\ xs
\end{array}
$$

These operations all have complexity $O(|xs|)$.

In [3], the union operation is defined for this representation:

$$
\begin{array}{l}
\begin{array}{lcl}
\mathsf{union}\ [\,]\ ys & = & ys \\
\mathsf{union}\ xs\ [\,] & = & xs
\end{array} \\
\mathsf{union}\ (x : xs)\ (y : ys) \\
\quad \left|\begin{array}{lcl}
x < y & = & x : \mathsf{union}\ xs\ (y : ys) \\
x == y & = & x : \mathsf{union}\ xs\ ys \\
\text{otherwise} & = & y : \mathsf{union}\ (x : xs)\ ys
\end{array}\right.
\end{array}
$$

The definitions of minus and intersection follow a similar pattern:

$$
\begin{array}{l}
\begin{array}{lcl}
\mathsf{minus}\ [\,]\ ys & = & [\,] \\
\mathsf{minus}\ xs\ [\,] & = & xs
\end{array} \\
\mathsf{minus}\ (x : xs)\ (y : ys) \\
\quad \left|\begin{array}{lcl}
x == y & = & \mathsf{minus}\ xs\ ys \\
x < y & = & x : \mathsf{minus}\ xs\ (y : ys) \\
\text{otherwise} & = & \mathsf{minus}\ (x : xs)\ ys
\end{array}\right.
\end{array}
$$

$$
\begin{array}{l}
\mathsf{intersection}\ [\,]\ ys \quad = \ [\,] \\
\mathsf{intersection}\ xs\ [\,] \quad = \ [\,] \\
\mathsf{intersection}\ (x : xs)\ (y : ys) \\
\quad \left|\ \begin{array}{lcl} x == y & = & x : \mathsf{intersection}\ xs\ ys \\ x < y & = & \mathsf{intersection}\ xs\ (y : ys) \\ \text{otherwise} & = & \mathsf{intersection}\ (x : xs)\ ys \end{array}\right.
\end{array}
$$

Since the lists are ordered (invariant (11)), we can walk through both lists in order, taking the appropriate action at each stage. We could have defined the insert and delete operations in a similar way — these operations would be slightly more efficient (but with the same complexity). We chose to define insert and delete in the way that we have as they produce more interesting obfuscated versions. Comparing the definitions of union to union_{aSp} in Section 5.1 and intersection to $\mathsf{intersection}_{b_k}$ in Section 5.2, we can see that the obfuscated operations follow a very similar pattern to the unobfuscated ones.

5.1 Alternating Split

If we wish to use the alternating split with ordered lists then we need to strengthen the invariant (4). The representation $xs \sim \langle l, r\rangle_{aSp}$ satisfies:

$$(|r| \leq |l| \leq |r| + 1) \ \wedge \ (l \trianglelefteq xs) \ \wedge \ (r \trianglelefteq xs) \tag{12}$$

Since we require that the split components are sublists, we know that the alternating split preserves ordering. Using the definition of split_{aSp}, we can easily check that this new invariant holds.

As usual, member_{aSp} is defined to be:

$$\mathsf{member}_{aSp}\ a\ \langle l, r\rangle_{aSp} \quad = \quad \mathsf{member}\ a\ l \vee \mathsf{member}\ a\ r$$

As proofs of correctness are important when obfuscating, in this section, we will not derive insert_{aSp} and delete_{aSp}. Instead we will state the operations and then prove that they are correct (see Section 6 for the proofs).

The definition of delete_{aSp} is:

$$
\begin{array}{rcl}
\mathsf{delete}_{aSp}\ a\ \langle l, r\rangle_{aSp} & = & \mathsf{if}\ \mathsf{member}\ a\ lz \\
& & \mathsf{then}\ \langle ly +\!\!+ rz, ry +\!\!+ \mathsf{tail}\ lz\rangle_{aSp} \\
& & \mathsf{else\ if}\ \mathsf{member}\ a\ rz \\
& & \quad \mathsf{then}\ \langle ly +\!\!+ \mathsf{tail}\ rz, ry +\!\!+ lz\rangle_{aSp} \\
& & \quad \mathsf{else}\ \langle l, r\rangle_{aSp} \\
\text{where}\ (ly, lz) & = & \mathsf{span}\ (< a)\ l \\
(ry, rz) & = & \mathsf{span}\ (< a)\ r
\end{array}
$$

Note that in the definitions of delete_{aSp} and insert_{aSp}, we use $\mathsf{member}\ a\ lz$ instead of $\mathsf{member}\ a\ l$. By the definition of lz, $\mathsf{member}\ a\ lz$ reduces to the check $\mathsf{head}\ lz == a$.

This means that the number of steps for computing delete is proportional to $|ly| + |ry|$ — so it has linear complexity. The definition of insert_{aSp} is:

$$
\begin{array}{rcl}
\mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp} & = & \mathsf{if}\ (\mathsf{member}\ a\ lz) \vee (\mathsf{member}\ a\ rz) \\
& & \mathsf{then}\ \langle l, r\rangle_{aSp} \\
& & \mathsf{else\ if}\ |ly| == |ry| \\
& & \qquad \mathsf{then}\ \langle ly +\!\!+ (a : rz), ry +\!\!+ lz\rangle_{aSp} \\
& & \qquad \mathsf{else}\ \langle ly +\!\!+ rz, ry +\!\!+ (a : lz)\rangle_{aSp} \\
\text{where } (ly, lz) & = & \mathsf{span}\ (< a)\ l \\
(ry, rz) & = & \mathsf{span}\ (< a)\ r
\end{array}
$$

This version of insert_{aSp} is still linear time — the only extra work is a check that $|ly| == |ry|$.

The definitions of the other set operations follow the same pattern to the definitions at the start of Section 5. As an example, here is union_{aSp}:

$$
\begin{array}{llll}
\mathsf{union}_{aSp} & \langle [\,], [\,]\rangle_{aSp} & ys & = ys \\
\mathsf{union}_{aSp} & xs & \langle [\,], [\,]\rangle_{aSp} & = xs \\
\mathsf{union}_{aSp} & \langle x : l_0, r_0\rangle_{aSp} & \langle y : l_1, r_1\rangle_{aSp} &
\end{array}
$$
$$
\left|\begin{array}{lclll}
x < y & = & x : (\mathsf{union}_{aSp} & \langle r_0, l_0\rangle_{aSp} & \langle y : l_1, r_1\rangle_{aSp}) \\
x == y & = & x : (\mathsf{union}_{aSp} & \langle r_0, l_0\rangle_{aSp} & \langle r_1, l_1\rangle_{aSp}) \\
\text{otherwise} & = & y : (\mathsf{union}_{aSp} & \langle x : l_0, r_0\rangle_{aSp} & \langle r_1, l_1\rangle_{aSp})
\end{array}\right.
$$

This operation is not interesting from an obfuscation point of view because it is very similar to the unobfuscated operation — except for a reversal in the order of the split operations.

5.2 Block split

As we are now working with ordered lists, we need to strengthen the invariant (5). The representation $xs \sim \langle l, r\rangle_{b_k}$ satisfies:

$$((|r| = 0 \wedge |l| < k) \vee (|l| = k)) \ \wedge\ (l \trianglelefteq xs) \ \wedge\ (r \trianglelefteq xs) \tag{13}$$

This ensures that the block split preserves ordering. Using the definition of split_{b_k}, we can easily check that this invariant holds. So, we can conclude that this split preserves ordering and so we can use it on ordered lists.

As with the alternating split, we will state the operations for ordered lists and then later prove that these operations are correct. The member_{b_k} operation is the same as usual:

$$\mathsf{member}_{b_k}\ a\ \langle l, r\rangle_{b_k} \ = \ \mathsf{member}\ a\ l \vee \mathsf{member}\ a\ r$$

For insert_{b_k}, we have to break the list l into $ls +\!\!+ [l']$, where l' is the last element of l. Note that since $|l| \leq k$ breaking l into ls and l' is a constant operation.

$$
\begin{array}{lcl}
\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k} & = & \mathsf{if\ member}_{b_k}\ a\ \langle l, r\rangle_{b_k} \\
& & \mathsf{then}\ \langle l, r\rangle_{b_k} \\
& & \mathsf{else\ if}\ |l| < k \\
& & \quad \mathsf{then}\ \langle \mathsf{insert}\ a\ l, r\rangle_{b_k} \\
& & \quad \mathsf{else\ if}\ l' < a \\
& & \quad\quad \mathsf{then}\ \langle l, \mathsf{insert}\ a\ r\rangle_{b_k} \\
& & \quad\quad \mathsf{else}\ \langle \mathsf{insert}\ a\ ls, l' : r\rangle_{b_k} \\
& & \ \text{where}\ \ ls\ =\ \mathsf{init}\ l \\
& & \qquad\qquad l'\ =\ \mathsf{last}\ l
\end{array}
$$

As $|l| \leq k$, this means the insert still has linear complexity in the worst case.

The delete operation follows a similar pattern to the unordered version, except that we have to take care where we place $\mathsf{head}\ r$.

$$
\begin{array}{lcl}
\mathsf{delete}_{b_k}\ a\ \langle l, r\rangle_{b_k} & = & \mathsf{if\ member}\ a\ l \\
& & \mathsf{then\ if}\ r == [\,] \\
& & \quad \mathsf{then}\ \langle \mathsf{delete}\ a\ l, r\rangle_{b_k} \\
& & \quad \mathsf{else}\ \langle (\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r], \\
& & \quad\quad \mathsf{tail}\ r\rangle_{b_k} \\
& & \mathsf{else}\ \langle l, \mathsf{delete}\ a\ r\rangle_{b_k}
\end{array}
$$

Again, in the worst case, delete has linear complexity.

Here is the definition of $\mathsf{intersection}_{b_k}$

$$
\begin{array}{llll}
\mathsf{intersection}_{b_k} & \langle [\,], [\,]\rangle_{b_k} & ys & = \langle [\,], [\,]\rangle_{b_k} \\
\mathsf{intersection}_{b_k} & xs & \langle [\,], [\,]\rangle_{b_k} & = \langle [\,], [\,]\rangle_{b_k} \\
\mathsf{intersection}_{b_k} & (x : \langle l_0, r_0\rangle_{b_k}) & (y : \langle l_1, r_1\rangle_{b_k}) &
\end{array}
$$
$$
\begin{array}{|lcl}
x == y & = & x : (\mathsf{intersection}_{b_k}\ \langle l_0, r_0\rangle_{b_k}\ \langle l_1, r_1\rangle_{b_k}) \\
x < y & = & \mathsf{intersection}_{b_k}\ \langle l_0, r_0\rangle_{b_k}\ (y : \langle l_1, r_1\rangle_{b_k}) \\
\text{otherwise} & = & \mathsf{intersection}_{b_k}\ (x : \langle l_0, r_0\rangle_{b_k})\ \langle l_1, r_1\rangle_{b_k}
\end{array}
$$

This definition is similar to the definition stated at the start of Section 5.

6 Proofs of Correctness

We have stated the operations for split ordered lists and so we need to ensure that they are correct.

6.1 Span and the alternating split

Before we show proofs of correctness, we will need a result about span. Suppose that $xs \sim \langle l, r\rangle_{aSp}$, where xs is a strictly-increasing list and

$$
\begin{array}{lcl}
l & = & [l_0, l_1, \ldots] \\
r & = & [r_0, r_1, \ldots]
\end{array}
$$

Then unsplitting $\langle l, r\rangle_{aSp}$, we can see that $xs = [l_0, r_0, l_1, r_1, \ldots]$. Now, let

$$\begin{array}{rcl} (ly, lz) & = & \mathsf{span}\ (< a)\ l \\ (ry, rz) & = & \mathsf{span}\ (< a)\ r \end{array}$$

for some a, and

$$(ys, zs) = \mathsf{span}\ (< a)\ xs$$

Then

$$\begin{array}{llllll} \text{(i)} & \text{if } |ly| = |ry| & \text{then} & ys & = & \mathsf{unsplit}_{aSp}\ (\langle ly, ry\rangle_{aSp}) \quad \wedge \\ & & & zs & = & \mathsf{unsplit}_{aSp}\ (\langle lz, rz\rangle_{aSp}) \\ \text{(ii)} & \text{if } |ly| \neq |ry| & \text{then} & ys & = & \mathsf{unsplit}_{aSp}\ (\langle ly, ry\rangle_{aSp}) \quad \wedge \\ & & & zs & = & \mathsf{unsplit}_{aSp}\ (\langle rz, lz\rangle_{aSp}) \end{array} \tag{14}$$

Proof of (i) If $(\forall x \in xs)\ a < x$ then $ly = [\,] = ry$ and so the result is trivial. Similarly, if $(\forall x \in xs)\ x < a$ then $lz = [\,] = rz$ — another trivial result. So, suppose $\exists j$ such that $r_{j-1} < a \leq l_j$. Then we have

$$\begin{array}{rcllrcl} ys & = & [l_0, r_0, l_1, r_1, \ldots, l_{j-1}, r_{j-1}] & \wedge & zs & = & [l_j, r_j, \ldots] \\ ly & = & [l_0, l_1, \ldots, l_{j-1}] & \wedge & lz & = & [l_j, \ldots] \\ ry & = & [r_0, r_1, \ldots, r_{j-1}] & \wedge & rz & = & [r_j, \ldots] \end{array}$$

We can easily see that $|ly| = |ry|$ and

$$ys = \mathsf{unsplit}_{aSp}\ (\langle ly, ry\rangle_{aSp}) \ \wedge\ zs = \mathsf{unsplit}_{aSp}\ (\langle lz, rz\rangle_{aSp})$$

Proof of (ii) Now suppose $\exists j$ such that $l_j < a \leq r_j$ and so

$$\begin{array}{rcllrcl} ys & = & [l_0, r_0, l_1, r_1, \ldots, r_{j-1}, l_j] & \wedge & zs & = & [r_j, l_{j+1} \ldots] \\ ly & = & [l_0, l_1, \ldots, l_j] & \wedge & lz & = & [l_{j+1}, \ldots] \\ ry & = & [r_0, r_1, \ldots, r_{j-1}] & \wedge & rz & = & [r_j, \ldots] \end{array}$$

This time $|ly| = |ry| + 1$ and

$$ys = \mathsf{unsplit}_{aSp}\ (\langle ly, ry\rangle_{aSp}) \ \wedge\ zs = \mathsf{unsplit}_{aSp}\ (\langle rz, lz\rangle_{aSp})$$

From this result, we can see that

$$\begin{array}{lcl} a \in l & \Rightarrow & |ly| = |ry| \\ a \in r & \Rightarrow & |ly| \neq |ry| \end{array} \tag{15}$$

6.2 Correctness of the Alternating Split Operations

We need to show that equation (8) holds for both insert_{aSp} and delete_{aSp}. Therefore, we would like to prove the following results:

(a) $\mathsf{unsplit}_{aSp}(\mathsf{insert}_{aSp}\ a\ (\mathsf{split}_{aSp}\ xs)) = \mathsf{insert}\ a\ xs$

(b) $\mathsf{unsplit}_{aSp}(\mathsf{delete}_{aSp}\ a\ (\mathsf{split}_{aSp}\ xs)) = \mathsf{delete}\ a\ xs$

We will need the properties from Sections 2.1.1 and 6.1.

Proof of (a) Suppose that $xs \sim \langle l, r\rangle_{aSp}$ and consider $\mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp}$. Let $(ly, lz) = \mathsf{span}\ (< a)\ l$ and $(ry, rz) = \mathsf{span}\ (< a)\ r$.

We have three cases:

(i) If $a \in xs$ then nothing changes so the result is (vacuously) true.

(ii) Suppose that $|ly| = |ry|$.

$$
\begin{array}{ll}
& \mathsf{unsplit}_{aSp}(\mathsf{insert}_{aSp}\ a\ (\mathsf{split}_{aSp}\ xs)) \\
= & \quad \{xs \sim \langle l, r\rangle_{aSp}\} \\
& \mathsf{unsplit}_{aSp}(\mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{aSp} \text{ with } |ly| = |ry| \text{ and } a \notin xs\} \\
& \mathsf{unsplit}_{aSp}(\langle ly +\!\!+ (a : rz), ry +\!\!+ lz\rangle_{aSp}) \\
= & \quad \{\text{first } +\!\!+ \text{ rule as } |ly| = |ry|\} \\
& \mathsf{unsplit}_{aSp}(\langle ly, ry\rangle_{aSp} +\!\!+ \langle (a : rz), lz\rangle_{aSp}) \\
= & \quad \{\text{rule for } :\} \\
& \mathsf{unsplit}_{aSp}(\langle ly, ry\rangle_{aSp} +\!\!+ (a : \langle lz, rz\rangle_{aSp})) \\
= & \quad \{\text{distributing } \mathsf{unsplit}_{aSp} \text{ over } +\!\!+ \text{ and } :\} \\
& \mathsf{unsplit}_{aSp}(\langle ly, ry\rangle_{aSp}) +\!\!+ (a : (\mathsf{unsplit}_{aSp}(\langle lz, rz\rangle_{aSp}))) \\
= & \quad \{\text{span property (14) with } |ly| = |ry|\} \\
& ys +\!\!+ (a : zs) \text{ where } (ys, zs) = \mathsf{span}\ (< a)\ xs \\
= & \quad \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ xs
\end{array}
$$

(iii) Now suppose that $|ly| \neq |ry|$.

$$
\begin{array}{ll}
& \mathsf{unsplit}_{aSp}(\mathsf{insert}_{aSp}\ a\ (\mathsf{split}_{aSp}\ xs)) \\
= & \quad \{xs \sim \langle l, r\rangle_{aSp}\} \\
& \mathsf{unsplit}_{aSp}(\mathsf{insert}_{aSp}\ a\ \langle l, r\rangle_{aSp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{aSp}\} \\
& \mathsf{unsplit}_{aSp}(\langle ly +\!\!+ rz, ry +\!\!+ (a : lz)\rangle_{aSp}) \\
= & \quad \{\text{second } +\!\!+ \text{ rule}\} \\
& \mathsf{unsplit}_{aSp}(\langle ly, ry\rangle_{aSp} +\!\!+ \langle (a : lz), rz\rangle_{aSp}) \\
= & \quad \{\text{rule for } :\} \\
& \mathsf{unsplit}_{aSp}(\langle ly, ry\rangle_{aSp} +\!\!+ a : \langle rz, lz\rangle_{aSp}) \\
= & \quad \{\text{distributing } \mathsf{unsplit}_{aSp} \text{ over } +\!\!+ \text{ and } :\} \\
& \mathsf{unsplit}_{aSp}(\langle ly, ry\rangle_{aSp}) +\!\!+ (a : (\mathsf{unsplit}_{aSp}(\langle rz, lz\rangle_{aSp}))) \\
= & \quad \{\text{span property (14) with } |ly| \neq |ry|\} \\
& ys +\!\!+ a : zs \text{ where } (ys, zs) = \mathsf{span}\ (< a)\ xs \\
= & \quad \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ xs
\end{array}
$$

Proof of (b) Suppose that $xs \sim \langle l, r \rangle_{aSp}$ and then consider $\mathsf{delete}_{aSp}\ a\ \langle l, r \rangle_{aSp}$. We have three cases:

(i) If $a \notin xs$ then nothing changes so the result is (vacuously) true.

(ii) Now suppose that $a \in l$ and define $(ly, lz) = \mathsf{span}\ (< a)\ l$ and $(ry, rz) = \mathsf{span}\ (< a)\ r$. Hence, so $a \in lz$. From equation (15), we know that $|ly| = |ry|$ and so

$$
\begin{array}{cl}
 & \mathsf{unsplit}_{aSp}(\mathsf{delete}_{aSp}\ a\ (\mathsf{split}_{aSp}\ xs)) \\
= & \quad \{xs \sim \langle l, r \rangle_{aSp}\} \\
 & \mathsf{unsplit}_{aSp}(\mathsf{delete}_{aSp}\ a\ \langle l, r \rangle_{aSp}) \\
= & \quad \{\text{definition of } \mathsf{delete}_{aSp}\} \\
 & \mathsf{unsplit}_{aSp}(\langle ly +\!\!+\ rz, ry +\!\!+\ \mathsf{tail}\ lz \rangle_{aSp}) \\
= & \quad \{\text{first rule for } +\!\!+\} \\
 & \mathsf{unsplit}_{aSp}(\langle ly, ry \rangle_{aSp} +\!\!+\ \langle rz, \mathsf{tail}\ lz \rangle_{aSp}) \\
= & \quad \{\text{definition of } \mathsf{tail}\} \\
 & \mathsf{unsplit}_{aSp}(\langle ly, ry \rangle_{aSp} +\!\!+\ \mathsf{tail}\ (\langle lz, rz \rangle_{aSp})) \\
= & \quad \{\text{span property (14) with } |ly| = |ry|, \text{ unsplitting}\} \\
 & ys +\!\!+\ \mathsf{tail}\ zs \text{ where } (ys, zs) = \mathsf{span}\ (< a)\ xs \\
= & \quad \{\text{definition of } \mathsf{delete}\} \\
 & \mathsf{delete}\ a\ xs
\end{array}
$$

(iii) Now suppose that $a \in r$ and define ly, ry, lz and rz as before (so $a \in rz$). From equation (15), we know that $|ly| \neq |ry|$ and so

$$
\begin{array}{cl}
 & \mathsf{unsplit}_{aSp}(\mathsf{delete}_{aSp}\ a\ (\mathsf{split}_{aSp}\ xs)) \\
= & \quad \{xs \sim \langle l, r \rangle_{aSp}\} \\
 & \mathsf{unsplit}_{aSp}(\mathsf{delete}_{aSp}\ a\ \langle l, r \rangle_{aSp}) \\
= & \quad \{\text{definition of } \mathsf{delete}_{aSp}\} \\
 & \mathsf{unsplit}_{aSp}(\langle ly +\!\!+\ \mathsf{tail}\ rz, ry +\!\!+\ lz \rangle_{aSp}) \\
= & \quad \{\text{second rule for } +\!\!+\} \\
 & \mathsf{unsplit}_{aSp}(\langle ly, ry \rangle_{aSp} +\!\!+\ \langle lz, \mathsf{tail}\ rz \rangle_{aSp}) \\
= & \quad \{\text{definition of } \mathsf{tail}\} \\
 & \mathsf{unsplit}_{aSp}(\langle ly, ry \rangle_{aSp} +\!\!+\ \mathsf{tail}\ (\langle rz, lz \rangle_{aSp})) \\
= & \quad \{\text{span property (14) with } |ly| \neq |ry|, \text{ unsplitting}\} \\
 & ys +\!\!+\ \mathsf{tail}\ zs \text{ where } (ys, zs) = \mathsf{span}\ (< a)\ xs \\
= & \quad \{\text{definition of } \mathsf{delete}\} \\
 & \mathsf{delete}\ a\ xs
\end{array}
$$

6.3 Correctness of the Block Split Operations

Again, we need to prove that equation (8) holds for both insert_{b_k} and delete_{b_k}. Therefore, we would like to prove that:

(a) $\mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) = \mathsf{insert}\ a\ xs$

(b) $\mathsf{unsplit}_{b_k}(\mathsf{delete}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) = \mathsf{delete}\ a\ xs$

Proof of (a) Suppose that $xs \sim \langle l, r\rangle_{b_k}$ and if $l \neq [\,]$ then we let $ls +\!\!+ [l'] = l$. If $a \in xs$ then insert leaves xs unchanged. Let us now suppose that $a \notin xs$ and we have three cases

(i) Suppose that $|l| < k$, so $r = [\,]$ and then

$$\begin{array}{cl}
 & \mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) \\
= & \quad \{xs \sim \langle l, r\rangle_{b_k}\} \\
 & \mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{b_k}\} \\
 & \mathsf{unsplit}_{b_k}(\langle \mathsf{insert}\ a\ l, [\,]\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
 & (\mathsf{insert}_{b_k}\ a\ l) +\!\!+ [\,] \\
= & \quad \{ys +\!\!+ [\,] = ys \text{ for all lists } ys\} \\
 & \mathsf{insert}_{b_k}\ a\ (l +\!\!+ [\,]) \\
= & \quad \{xs = l +\!\!+ [\,]\} \\
 & \mathsf{insert}\ a\ xs
\end{array}$$

(ii) Suppose that $|l| = k$ and $l' < a$, so

$$\begin{array}{cl}
 & \mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) \\
= & \quad \{xs \sim \langle l, r\rangle_{b_k}\} \\
 & \mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{b_k}\} \\
 & \mathsf{unsplit}_{b_k}(\langle l, \mathsf{insert}\ a\ r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
 & l +\!\!+ (\mathsf{insert}\ a\ r) \\
= & \quad \{\text{let } (ry, rz) = \mathsf{span}\ (< a)\ r \text{ and definition of } \mathsf{insert}\} \\
 & l +\!\!+ ry +\!\!+ (a : rz) \\
= & \quad \{\mathsf{span}\ (< a)\ xs = (l +\!\!+ ry, rz) \text{ and definition of } \mathsf{insert}\} \\
 & \mathsf{insert}\ a\ xs
\end{array}$$

(iii) Suppose that $|l| = k$ and $l' > a$ (we cannot have $l' = a$ as $a \notin xs$), then

$$
\begin{aligned}
& \mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) \\
= & \quad \{xs \sim \langle l, r\rangle_{b_k}\} \\
& \mathsf{unsplit}_{b_k}(\mathsf{insert}_{b_k}\ a\ \langle l, r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{b_k}\} \\
& \mathsf{unsplit}_{b_k}(\langle \mathsf{insert}\ a\ ls, l' : r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& (\mathsf{insert}\ a\ ls) +\!\!+ (l' : r) \\
= & \quad \{\text{let } (ly, lz) = \mathsf{span}\ (< a)\ ls \text{ and definition of } \mathsf{insert}\} \\
& ly +\!\!+ (a : lz) +\!\!+ (l' : r) \\
= & \quad \{\text{properties of : and } +\!\!+\} \\
& ly +\!\!+ a : (lz +\!\!+ [l'] +\!\!+ r) \\
= & \quad \{\mathsf{span}\ (< a)\ xs = (ly, lz +\!\!+ [l'] +\!\!+ r) \text{ and definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ xs
\end{aligned}
$$

Proof of (b) If $a \notin xs$ then delete leaves xs unchanged. So suppose that $a \in xs$ and $xs \sim \langle l, r\rangle_{b_k}$. There are three cases:

(i) if $a \in l$ and $r = [\,]$ then

$$
\begin{aligned}
& \mathsf{unsplit}_{b_k}(\mathsf{delete}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) \\
= & \quad \{\text{definition of } \mathsf{delete}_{b_k}\} \\
& \mathsf{unsplit}_{b_k}(\langle \mathsf{delete}\ a\ l, [\,]\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& (\mathsf{delete}_{b_k}\ a\ l) +\!\!+ [\,] \\
= & \quad \{ys +\!\!+ [\,] = ys \text{ for all lists } ys\} \\
& \mathsf{delete}_{b_k}\ a\ (l +\!\!+ [\,]) \\
= & \quad \{xs = l +\!\!+ [\,]\} \\
& \mathsf{delete}\ a\ xs
\end{aligned}
$$

(ii) if $a \in l$ and $r \neq [\,]$ then

$$
\begin{aligned}
& \mathsf{unsplit}_{b_k}(\mathsf{delete}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) \\
= & \quad \{\text{definition of } \mathsf{delete}_{b_k}\} \\
& \mathsf{unsplit}_{b_k}(\langle(\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& (\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r] +\!\!+ (\mathsf{tail}\ r) \\
= & \quad \{\text{associativity of } +\!\!+\text{, definitions of } \mathsf{head} \text{ and } \mathsf{tail}\} \\
& (\mathsf{delete}\ a\ l) +\!\!+ r \\
= & \quad \{\text{let } (ly, lz) = \mathsf{span}\ (<a)\ l\text{, definition of } \mathsf{delete}\} \\
& ly +\!\!+ (\mathsf{tail}\ lz) +\!\!+ r \\
= & \quad \{\text{properties of } \mathsf{tail} \text{ and } +\!\!+\} \\
& ly +\!\!+ \mathsf{tail}\ (lz +\!\!+ r) \\
= & \quad \{\mathsf{span}\ (<a)\ xs = (ly, lz +\!\!+ r)\text{, definition of } \mathsf{delete}\} \\
& \mathsf{delete}\ a\ xs
\end{aligned}
$$

(iii) if $a \in r$ then

$$
\begin{aligned}
& \mathsf{unsplit}_{b_k}(\mathsf{delete}_{b_k}\ a\ (\mathsf{split}_{b_k}\ xs)) \\
= & \quad \{\text{definition of } \mathsf{delete}_{b_k}\} \\
& \mathsf{unsplit}_{b_k}(\langle l, \mathsf{delete}\ a\ r\rangle_{b_k}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b_k}\} \\
& l +\!\!+ (\mathsf{delete}\ a\ r) \\
= & \quad \{\text{let } (ry, rz) = \mathsf{span}\ a\ r\text{, definition of } \mathsf{delete}\} \\
& l +\!\!+ ry +\!\!+ (\mathsf{tail}\ rz) \\
= & \quad \{\mathsf{span}\ (<a)\ xs = (l +\!\!+ ry, rz)\text{, definition of } \mathsf{delete}\} \\
& \mathsf{delete}\ a\ xs
\end{aligned}
$$

7 Conclusions and Future Work

An important concern for applying an obfuscation to an operation is how much the obfuscation will affect the complexity of the operation. In our particular examples we have seen that, surprisingly, our obfuscations have little effect on the complexity of the operations. Further work needs to be carried out to see how other data-types 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. The derivation techniques involved are simple and only require a basic understanding of functional programming. 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 also need to consider different data-types and obfuscations to see how our methods can be extended.

It has not been mentioned how "obfuscated" our derived operations are. This is because there is not an adequate definition for obfuscation that can be used with both functional and imperative programs. One possible definition is that we 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 [9] 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. Many of our operations can be written in terms of foldr. Using fold definitions does not necessarily lead to shorter proofs but the proofs can be automated. A related area is the refactoring of Haskell [8]: 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. Could the techniques used to automate refactoring be used to automate obfuscation?

Acknowledgements

Thanks to Jeff Sanders for being a very supportive supervisor who has given constant help and encouragement throughout the process of writing of this paper. Thanks also to Rani Ettinger, Yorck Hünke, Oege de Moor, Damien Sereni and Barney Stratford for their constructive comments on the paper.

References

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

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

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

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

[5] Wilem-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, 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.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley, 2000.

[8] 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.

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

[10] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.

[11] PreEmptive Solutions. Java and .NET obfuscators. Available from URL: `http://www.preemptive.com`.