Obfuscation of Abstract Data-Types

Stephen Drape
St John's College

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford

Trinity Term 2004

Obfuscation of Abstract Data Types

Stephen Drape
St John's College
University of Oxford

Thesis submitted for the degree of Doctor of Philosophy
Trinity Term 2004

Abstract

An *obfuscation* is a behaviour-preserving program transformation whose aim is to make a program "harder to understand". Obfuscations are applied to make reverse engineering of a program more difficult. Two concerns about an obfuscation are whether it preserves behaviour (*i.e.* it is *correct*) and the degree to which it maintains efficiency. Obfuscations are applied mainly to object-oriented programs but constructing proofs of correctness for such obfuscations is a challenging task. It is desirable to have a workable definition of obfuscation which is more rigorous than the metric-based definition of Collberg *et al.* and overcomes the impossibility result of Barak *et al.* for their strong cryptographic definition.

We present a fresh approach to obfuscation by obfuscating *abstract data-types* allowing us to develop structure-dependent obfuscations that would otherwise (traditionally) not be available. We regard obfuscation as data refinement enabling us to produce equations for proving correctness and we model the data-type operations as functional programs making our proofs easy to construct. For case studies, we examine different data-types exploring different areas of computer science. We consider *lists* letting us to capture array-based obfuscations, *sets* reflecting specification based software engineering, *trees* demonstrating standard programming techniques and as an example of numerical methods we consider *matrices*.

Our approach has the following benefits: obfuscations can be proved correct; obfuscations for some operations can be *derived* and random obfuscations can be produced (so that different program executions give rise to different obfuscations). Accompanying the approach is a new definition of obfuscation based on measuring the effectiveness of an obfuscation. Furthermore in our case studies the computational complexity of each obfuscated operations is comparable with the complexity of the unobfuscated version. Also, we give an example of how our approach can be applied to implement imperative obfuscations.

Contents

I The Current View of Obfuscation **11**

1 Obfuscation **12**
1.1 Protection 12
1.2 Obfuscation 13
1.3 Examples of obfuscations 14
1.3.1 Commercial Obfuscators 15
1.3.2 Opaque predicates 16
1.3.3 Variable transformations 16
1.3.4 Loops 17
1.3.5 Array transformations 18
1.3.6 Array Splitting 18
1.3.7 Program transformation 19
1.3.8 Array Merging 20
1.3.9 Other obfuscations 20
1.4 Conclusions 21

2 Obfuscations for Intermediate Language **22**
2.1 .NET 22
2.1.1 IL Instructions 22
2.2 IL obfuscations 25
2.2.1 Variable Transformation 25
2.2.2 Jumping into while loops 26
2.3 Transformation Toolkit 27
2.3.1 Path Logic Programming 28
2.3.2 Expression IL 29
2.3.3 Predicates for EIL 31
2.3.4 Modifying the graph 32
2.3.5 Applying transformations 33
2.3.6 Variable Transformation 33
2.3.7 Array Splitting 34
2.3.8 Creating an irreducible flow graph 37
2.4 Related Work 40
2.4.1 Forms of IL 40
2.4.2 Transformation Systems 41
2.4.3 Correctness 41
2.5 Conclusions 42

II An Alternative View 43

3 Techniques for Obfuscation 44
3.1 Data-types 44
3.1.1 Notation 45
3.1.2 Modelling in Haskell 46
3.2 Obfuscation as data refinement 48
3.2.1 Homogeneous operations 49
3.2.2 Non-homogeneous operations 50
3.2.3 Operations using tuples 51
3.2.4 Refinement Notation 51
3.3 What does it mean to be obfuscated? 52
3.4 Proof Trees 54
3.4.1 Measuring 56
3.4.2 Comparing proofs 56
3.4.3 Changing the measurements 59
3.4.4 Comments on the definition 60
3.5 Folds 60
3.6 Example Data-Types 61

III Data-Type Case Studies 62

4 Splitting Headaches 63
4.1 Generalising Splitting 63
4.1.1 Indexed Data-Types 63
4.1.2 Defining a split 64
4.1.3 Example Splits 65
4.1.4 Splits and Operations 66
4.2 Arrays 68
4.2.1 Array-Split properties 69
4.3 Lists 71
4.3.1 List Splitting 72
4.3.2 Alternating Split 73
4.3.3 Block Split 77
4.4 Random List Splitting 82
4.4.1 Augmented Block Split 87
4.4.2 Padded Block Split 89
4.5 Conclusions 90

5 Sets and the Splitting 92
5.1 A Data-Type for Sets 92
5.2 Unordered Lists with duplicates 92
5.2.1 Alternating Split 94
5.3 Strictly Ordered Lists 95
5.3.1 Alternating Split 96
5.3.2 Block Split 98
5.4 Comparing different definitions 99
5.4.1 First Definition 100
5.4.2 Second Definition 101

5.4.3 Third Definition 102
5.4.4 Last Definition 104
5.5 Conclusions 106

6 The Matrix Obfuscated **107**
6.1 Example 107
6.2 Matrix data-type 108
6.2.1 Definitions of the operations 109
6.3 Splitting Matrices 110
6.3.1 Splitting in squares 110
6.3.2 Modelling the split in Haskell 112
6.3.3 Properties 113
6.3.4 Deriving transposition 113
6.3.5 Problems with arrays 115
6.4 Other splits and operations 116
6.4.1 Other Splits 116
6.4.2 A more general square splitting 117
6.4.3 Extending the data-type of matrices 118
6.5 Conclusions 118

7 Cultivating Tree Obfuscations **119**
7.1 Binary Trees 119
7.1.1 Binary Search Trees 120
7.2 Obfuscating Trees 123
7.2.1 Ternary Trees 123
7.2.2 Particular Representation 124
7.2.3 Ternary Operations 126
7.2.4 Making Ternary Trees 127
7.2.5 Operations for Binary Search Trees 128
7.2.6 Deletion 128
7.3 Other Methods 129
7.3.1 Abstraction Functions 130
7.3.2 Heuristics 131
7.4 Conclusions 132

IV Conclusions **133**

8 Conclusions and Further Work **134**
8.1 Discussion of our approach to Obfuscation 134
8.1.1 Meeting the Objectives 134
8.1.2 Complexity 135
8.1.3 Definition 136
8.1.4 Randomness 136
8.1.5 Contributions 137
8.2 Further Work 137
8.2.1 Deobfuscation 138
8.2.2 Other techniques 138

V Appendices 143

A List assertion 144
A.1 Unobfuscated version . . . 144
A.2 Alternating Split for Lists . . . 145
A.2.1 Version 1 . . . 145
A.2.2 Version 2 . . . 147
A.3 Block Split for Lists . . . 149
A.4 Augmented Split . . . 152
A.5 Augmented Block Split . . . 154
A.6 Padded Block Split . . . 156
A.7 Comparing the proofs . . . 158

B Set operations 159
B.1 Span Properties . . . 159
B.2 Insertion for the alternating split . . . 164

C Matrices 166
C.1 Rewriting the definition . . . 166
C.2 Using fold fusion . . . 167

D Proving a tree assertion 169
D.1 Producing a binary tree from a list . . . 169
D.1.1 Operations . . . 169
D.1.2 Proving Properties . . . 170
D.1.3 Proving the assertion . . . 172
D.2 Producing a binary tree from a split list . . . 174
D.2.1 Operations . . . 174
D.2.2 Preliminaries . . . 174
D.2.3 Proof of the assertion . . . 179
D.3 Producing a ternary tree from a list . . . 180
D.3.1 Operations . . . 180
D.3.2 Proof . . . 180
D.4 Producing a ternary tree from a split list . . . 182
D.4.1 Proof . . . 182
D.5 Comparing the proofs . . . 184

E Obfuscation Example 185

List of Figures

1 An example of obfuscation 8

1.1 Fibonacci program after an array split 20

2.1 Some common IL instructions 23
2.2 IL for a GCD method 24
2.3 Output from Salamander 27
2.4 Syntax for a simple subset of EIL 30
2.5 Some common predicates 32
2.6 Predicates for replacing assignments 35
2.7 Definition of *replace_array* for assignments 36
2.8 Flow graph to build a conditional expression 37
2.9 Flow graph to create jump into a loop 38
2.10 Code for the *irred_jump* method 39

3.1 Data-type for Arrays 47

4.1 Data-type for Lists 70
4.2 Data-type for Split Lists 72

5.1 Data-type for Sets 93

6.1 Data-type for Matrices 109

7.1 Data-Type for Binary Trees 120
7.2 Data-Type for Binary Search Trees 121
7.3 Data-Type for Ternary Trees 124

E.1 Original Program 186

Preliminaries

Homer: "All right brain, you don't like me and I don't like you. But let's just do this and I can get back to killing you with beer."

The Simpsons — *The Front* (1993)

In this thesis, we consider the **obfuscation of programs**. "To Obfuscate" means "To cast into darkness or shadow; to cloud, obscure". From a Computer Science perspective, an obfuscation is a behaviour-preserving program transformation whose aim is to make a program "harder to understand". Obfuscations are applied to a program to make reverse engineering of the program more difficult. Two concerns about an obfuscation are whether it preserves behaviour (*i.e.* it is *correct*) and the degree to which it maintains efficiency.

Example

As an example of obfuscation, consider the program in Figure 1. The method *start* takes an integer value array as input and the array is returned from the method. But what does this method do to the elements of the array? The obfuscation was achieved by using some of the methods outlined in this thesis.

Aims of the thesis

The current view of obfuscation (in particular, the paper by Collberg *et al.* [10]) concentrates on object-oriented programs. In [10], obfuscations for constructs such as loops, arrays and methods are stated informally: without proofs of correctness. Constructing proofs of correctness for imperative obfuscations is a challenging task.

We offer an alternative approach based on data refinement and functional programming. We want our approach to have the following objectives:

- to yield proofs of correctness (or even yield derivations) of all our obfuscations
- to use simple, established refinement techniques, leaving the ingenuity for obfuscation
- to generalise obfuscations to make obfuscations more applicable.

```
public class c
{   d t1  = new d();
    public int [] b;
    int r;

    public class d
    {   public d a1;
        public d a2;
        public d a3;
        public int z;
    }

    public int [] start(int [] b)
    {   t1 = null;
        r = 0;
        while (r < b.Length) { i(ref t1, b[r]); r ++; }
        r = 0;
        s(t1);
        return b;
    }

    void s(d t2)
    {   if (t2  ==  null) return;
        int k  =  t2.z;
        if (p(k)) s(t2.a1); else s(t2.a2);
        b[r] = g(k);
        r ++;
        s(t2.a3);
    }

    void i(ref d t2, int k)
    {   int j  = f(k);
        if (t2  ==  null) { t2 = new d();
                            t2.z = j; }
        else { if (j < t2.z)
                 { if (p(t2.z)) { i(ref t2.a1, g(j)); i(ref t2.a2, h(j)); }
                   else { i(ref t2.a1, f(j)); i(ref t2.a2, g(j)); }
                 else i(ref t2.a3, g(j)); }
    }

    bool p(int n) { return (n % 2 == 0); }
    int f(int n)   { return (3 * n + 1); }
    int g(int n)   { return ((n - 1)/3); }
    int h(int n)   { return ((4 * n + 5)/3); }
}
```

Figure 1: An example of obfuscation

We study abstract data-types (consisting of a local state accessible only by declared operations) and define obfuscations for the whole data-type. In other words, we obfuscate the state of the data-type under the assumption that the only way it is being accessed is via the operations of the type. Different operations (on a given state) may require different obfuscations.

To date, obfuscation has been an area largely untouched by the formal method approach to program correctness. We regard obfuscation as data refinement allowing us to produce equations for proving correctness. We model the data-type operations as functional programs. That enables us to establish correctness easily as well as providing us with an elegant style in which to write definitions of our operations. Two benefits of using abstract data-types are that we can specify obfuscations which exploit structural properties inherent in the data-type; and the ability to create *random* obfuscations. We also provide a new definition of obfuscation that avoids the impossibility problem considered by Barak *et al.* [6] and is appropriate for our data-type approach.

Structure of the thesis

The thesis is structured as follows:

- In Chapters 1 and 2 we consider the current view of obfuscation. In Chapter 1 we discuss the need for obfuscation and summarise some of the obfuscations from [10]. Also we evaluate the definitions for obfuscation given in [6, 10]. In Chapter 2 we look at the .NET Intermediate Language [23] and discuss joint work with Oege de Moor and Ganesh Sittampalam that allows us to write some specifications of obfuscations for Intermediate Language.

- In Chapter 3 we give an alternative view of obfuscation by concentrating on abstract data-types. We use data refinement and functional programming to produce a framework that allows us to prove the correctness of obfuscations (or even to derive them) and we give a definition of obfuscation pertinent to our approach.

- In Chapter 4 we use our approach to generalise an obfuscation called *array splitting* and we show how to split more general data-types.

- The next three chapters concentrate on specific case studies for different data-types. In Chapters 5 and 6 we use the results on data-type splitting to show how to construct obfuscations for sets and matrices. In Chapter 7 we give a transformation suitable for obfuscating binary trees.

- Finally, in Chapter 8, we summarise our results and discuss possible areas for future work.

Contributions

The thesis provides the following contributions.

Using established work on refinement, abstract data-types and functional programming, a new approach to obfuscation is developed. This approach has the following benefits:

- Obfuscations can be proved correct.
- The obfuscations of some operations can be *derived* from their unobfuscated versions.
- Random obfuscations can be produced so that different program executions give rise to different obfuscations, making them particularly obscure.

The approach is accompanied by a new definition, based on measuring the effectiveness of an obfuscation. Furthermore, in the case studies included here, our obfuscations are (close to) optimal in the sense that the computational complexity of each obfuscated operation is comparable with the complexity of the unobfuscated version.

Notation

We will use various programming languages in this thesis and so we adopt the following conventions to differentiate between the different languages:

- C#: We use **bold sans serif** for standard keywords and *italics* for variables.
- IL: We use `typewriter` font for the whole language.
- Prolog: We write all predicates in *italics*.
- Haskell: Operations are written in sans serif font, variables in *italics* and keywords in normal font.

Type declarations are of the form $f :: T$ — we use this notation instead of the more usual notation $f : T$ so that type statements and list constructions are not confused.

We denote lists (and occasionally arrays) by $[_]$, sets by $\{_\}$, sequences by $⟪_⟫$ and a split data-type by $\langle _ \rangle$ (see Chapter 4 for the definition of split data-types). We assume the following types: $\mathbb{N}$ (natural numbers), $\mathbb{Z}$ (integers) and $\mathbb{B}$ (Booleans). Also, we write $[a..b]$ to denote $\{i :: \mathbb{Z} \mid a \leq i \leq b\}$ and similarly $[a..b)$ to denote $\{i :: \mathbb{Z} \mid a \leq i < b\}$.

Acknowledgements

Thanks to Jeff Sanders for his continual support and encouragement during his time as my supervisor. This thesis would have been impossible to complete without Jeff's constant help and advice.

Various members of the Programming Research Group have given their time to provide comments and advice, in particular: Richard Brent, Rani Ettinger, Yorck Hünke, Damien Sereni, Barney Stratford and Irina Voiculescu.

Thanks also to Geraint Jones and Simon Thompson for providing useful feedback during the viva.

Part I

The Current View of Obfuscation

Chapter 1

Obfuscation

Scully: "Not everything is a labyrinth of dark conspiracy.
And not everyone is plotting to deceive, inveigle
and obfuscate"

The X-Files — *Teliko* (1996)

This chapter introduces the notion of obfuscation, summarises specific examples given in the original paper [10] and discusses some related work.

1.1 Protection

Suppose that a programmer develops a new piece of software which contains a unique innovative algorithm. If the programmer wants to sell this software, it is usually in the programmer's interest that the algorithm should remain secret from purchasers. What can the programmer do to stop the algorithm from being seen or modified by a client? A similar issue arises when a piece of software contains a date stamp that allows it to be used for a certain length of time. How can we stop people from identifying (and changing) the date stamp?

We will assume that it is possible for an executable file to be decompiled to a high-level language and so the source code of an executable can be studied. In fact, this is the situation with Java Bytecode and the .NET framework (see Chapter 2 for more details). Therefore, we need to consider ways of protecting software so that it is hard to understand a program after decompilation. Here are some of the protections that are identified in [10]:

- *Server-side execution* — this involves clients remotely accessing the software from a programmer's site. The downside of this approach is that the application may become slow and hard to use due to limits of network capacity.

- *Watermarking* — this is useful only for "stamping" a product. Stamps are used for copyright protection — but they still do not prevent people from stealing parts of the program, although there are methods for embedding a watermark in a program [9, 42].

- *Encryption* — an encryption algorithm could be used to encrypt the entire code or parts of the code. However, the decryption process must be in the executable and so a client could intercept the code after decryption has taken place. The only really viable option is for the encryption and decryption processes take place in hardware [30].

1.2 Obfuscation

We now consider a different technique for software protection: *code obfuscation*. An obfuscation is a behaviour-preserving transformation whose aim is to make a program "harder to understand".

Collberg *et al.* [10] do not *define* obfuscation but instead qualify "hard to understand" by using various metrics which measure the complexity of code. For example:

- *Cyclomatic Complexity* [37] — the complexity of a function increases with the number of predicates in the function.
- *Nesting Complexity* [29] — the complexity of a function increases with the nesting level of conditionals in the function.
- *Data-Structure Complexity* [40] — the complexity increases with the complexity of the static data structures declared in a program. For example, the complexity of an array increases with the number of dimensions and with the complexity of the element type.

Using such metrics Collberg *et al.* [10] measure the *potency* of an obfuscation as follows. Let $\mathcal{T}$ be a transformation which maps a program P to a program P'. The potency of a transformation $\mathcal{T}$ with respect to the program P is defined to be:

$$\mathcal{T}_{pot}(P) = \frac{E(P')}{E(P)} - 1$$

where $E(P)$ is the complexity of P (using an appropriate metric). $\mathcal{T}$ is said to be a *potent obfuscating transformation* if $\mathcal{T}_{pot}(P) > 0$ (*i.e.* if $E(P') > E(P)$). In [10], P and P' are not required to be equally efficient — it is stated that many of the transformations given will result in P' being slower or using more memory than P.

Other properties Collberg *et al.* [10] measure are:

- *Resilience* — this measures how well a transformation survives an attack from a deobfuscator. Resilience takes into account the amount of time required to construct a deobfuscator and the execution time and space actually required by the deobfuscator.
- *Execution Cost* — this measures the extra execution time and space of an obfuscated program P' compared with the original program P.
- *Quality* — this combines potency, resilience and execution cost to give an overall measure.

These three properties are measured informally on a non-numerical scale (*e.g.* for resilience, the scale is *trivial*, *weak*, *strong*, *full*, *one-way*).

Another useful measure is the *stealth* [12] of an obfuscation. An obfuscation is stealthy if it does not "stand out" from the rest of the program, *i.e.* it resembles the original code as much as possible. Stealth is context-sensitive — what is stealthy in one program may not be in another one and so it is difficult to quantify (as it depends on the whole program and also the experience of the reader).

The metrics mentioned above are not always suitable to measure the degree of obfuscation. Consider these two code fragments:

$$\textsf{\textbf{if}}\ (p)\ \{A;\}\ \textsf{\textbf{else}}\ \{\textsf{\textbf{if}}\ (q)\ \{B;\}\ \textsf{\textbf{else}}\ \{C;\}\} \tag{1.1}$$

$$\begin{array}{l} \textsf{\textbf{if}}\ (p)\ \{A;\}; \\ \textsf{\textbf{if}}\ (\neg p \wedge q)\ \{B;\}; \\ \textsf{\textbf{if}}\ (\neg p \wedge \neg q)\ \{C;\} \end{array} \tag{1.2}$$

These two fragments are equivalent if A leaves the value of p unchanged and B leaves p and q unchanged. If we transform (1.1) to (1.2) then the cyclomatic complexity is increased but the nesting complexity is decreased. Which fragment is more obfuscated?

Barak *et al.* [6] takes a more formal approach to obfuscation — their notion of obfuscation is as follows. An obfuscator $\mathcal{O}$ is a "compiler" which takes as input a program P and produces a new program $\mathcal{O}(P)$ such that for every P:

- *Functionality* — $\mathcal{O}(P)$ computes the same function as P.
- *Polynomial Slowdown* — the description length and running time of $\mathcal{O}(P)$ are at most polynomially larger than that of P.
- *"Virtual black box" property* — "Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P" [6, Page 2].

With this definition, Barak *et al.* construct a family of functions which is unobfuscatable in the sense that there is no way of obfuscating programs that compute these functions. The main result of [6] is that their notion of obfuscation is impossible to achieve.

This definition of obfuscation, in particular the "Virtual Black Box" property, is evidently too strong for our purposes and so we consider a weaker notion. We do not consider our programs as being "black boxes" as we assume that any attacker can inspect and modify our code. Also we would like an indication of how "good" an obfuscation is. In Section 3.3 we will define what we mean by "harder to understand", however for the rest of the chapter (and for Chapter 2) we will use the notion of obfuscation from Collberg *et al.* [10].

1.3 Examples of obfuscations

This section summarises some of the major obfuscations published to date [10, 11, 12]. First, we consider some of the commercial obfuscators available and then discuss some data structure and control flow obfuscations that are not commonly implemented by commercial obfuscators.

1.3.1 Commercial Obfuscators

We briefly describe a few of the many commercial obfuscators that are available. Many commercial obfuscators employ two common obfuscations: *renaming* and *string encryption*. Renaming is a simple obfuscation which consists of taking any objects (such as variables, methods or classes) that have a "helpful" name (such as "`total`" or "`counter`") and renaming them to a less useful name.

Strings are often used to output information to the user and so some strings can give information to an attacker. For example, if we saw the string "`Enter the password`" in a method, we could infer that this method is likely to require the input of passwords! From this, we may be able to see what the correct passwords are or even bypass this particular verification. The idea of string encryption is to encrypt all the strings in a program to make them unreadable so that we can no longer obtain information about code fragments simply by studying the strings. However, string encryption has a major drawback. All of the output strings in a program must be displayed properly on the screen during the execution of a program. Thus the strings must first be decrypted and so the decryption routine will be contained in the program itself, as will the key for the decryption. So by looking in the source code at how strings are processed before being output we should be able to see how the strings are decrypted. Thus we could easily see the decrypted strings by passing them through the decryption method. Thus string encryption should not be considered as strong protection.

Here are some obfuscators which are commercially available:

- **JCloak** [26] This works on all classes in a program by looking at the symbolic references in the class file and generating a new unique and unintelligible name for each symbol name.

- **Dotfuscator** [49] This uses a system of renaming classes, fields and methods called "Overload-Induction". This system induces method overloading as much as possible and it tries to rename methods to a small name (*e.g.* a single character). It also applies string encryptions and some control-flow obfuscations.

- **Zelix** [57] This obfuscator uses a variety of different methods. One method is the usual name obfuscation and another is string encryption which encrypts strings such as error messages which are decrypted at runtime. It also uses a flow obfuscation that tries to change the structure of loops by using **goto**s.

- **Salamander** [46] Variable renaming is used to try to convert all names to "A" — overloading is then used to distinguish between different methods and fields. Any unneccesary meta-data (such as debug information and parameter names) is removed.

- **Smokescreen** [48] This uses some control-flow obfuscations. One obfuscation shuffles stack operations so that popping a stack value into a local variable is delayed. The aim of this transformation is to make it more difficult for decompilers to determine where stack values come from. Another obfuscation adds fake exceptions so that the exception block partly overlaps with an existing block of code. The aim is to make control flow analysis more difficult. Another transformation to the control flow is made

by changing some switch statements by attempting to make the control flow appear to bypass a switch statement and go straight to a case statement.

1.3.2 Opaque predicates

One of the most valuable obfuscation techniques is the use of *opaque predicates* [10, 12]. An opaque predicate P is a predicate whose value is known at obfuscation time — P^T denotes a predicate which is always *True* (similarly for P^F) and $P^?$ denotes a predicate which sometimes evaluates to *True* and sometimes to *False*.

Here are some example predicates which are always *True* (supposing that x and y are integers):

$$\begin{array}{rcl} x^2 & \geq & 0 \\ x^2(x+1)^2 & \equiv & 0 \ (\text{mod} 4) \\ x^2 & \neq & 7y^2 - 1 \end{array}$$

Opaque predicates can be used to transform a program block B as follows:

- **if** (P^T) $\{B;\}$
  This hides the fact that B will always be executed.
- **if** (P^F) $\{B';\}$ **else** $\{B;\}$
  Since the predicate is always false we can make the block B' to be a copy of B which may contain errors.
- **if** $(P^?)$ $\{B;\}$ **else** $\{B';\}$
  In this case, we can have two copies of B each with the same functionality.

We can also use opaque predicates to manufacture bogus jumps. Section 2.2.2 shows how to use an opaque predicate to create a jump into a **while** loop so that an irreducible flow graph is produced.

When creating opaque predicates, we must ensure that they are stealthy so that it is not obvious to an attacker that a predicate is in fact bogus. So we must choose predicates that match the "style" of the program and if possible use expressions that are already used within the program. A suitable source of opaque predicates can be obtained from considering watermarks, such as the ones discussed in [9, 42].

1.3.3 Variable transformations

In this section, we show how to transform an integer variable i within a method. To do this, we define two functions f and g:

$$f :: X \rightarrow Y$$
$$g :: Y \rightarrow X$$

where $X \subseteq \mathbb{Z}$ — this represents the set of values that i takes. We require that g is a left inverse of f (and so f needs to be injective). To replace the variable i with a new variable, j say, of type Y we need to perform two kinds of replacement depending on whether we have an assignment to i or use of i. An *assignment* to i is a statement of the form $i = V$ and a *use* of i is an occurrence of i which is not an assignment. The two replacements are:

- Any assignments of i of the form $i = V$ are replaced by $j = f(V)$.
- Any uses of i are replaced by $g(j)$.

These replacements can be used to obfuscate a **while** loop.

1.3.4 Loops

In this section, we show some possible obfuscations of this simple **while** loop:

```
i = 1;
while (i < 100)
{
  ...
  i++;
}
```

We can obfuscate the loop counter i — one possible way is to use a variable transformation. We define functions f and g to be:

$$f = \lambda\ i.(2i + 3)$$
$$g = \lambda\ i.(i - 3) \text{ div } 2$$

and we can verify that $g \cdot f = id$.

Using the rules for variable transformation (and noting that the statement $i++$; corresponds to a use and an assignment), we obtain:

```
j = 5;
while ((j - 3)/2 < 100)
{
  ...
  j = (2 * (((j - 3)/2) + 1)) + 3;
}
```

With some simplifications, the loop becomes:

```
j = 5;
while (j < 203)
{
  ...
  j = j + 2;
}
```

Another method we could use is to introduce a new variable, k say, into the loop and put an opaque predicate (depending on k) into the guard. The variable k performs no function in the loop, so we can make any assignment to k. As an example, our loop could be transformed to something of the form:

```
i = 1;
k = 20;
while (i < 100 && (k * k * (k + 1) * (k + 1) % 4 == 0))
{
  ...
  i++;
  k = k * (i + 3);
}
```

We could also put a false predicate into the middle of the loop that attempts to jump out of the loop.

Our last example changes the original single loop into two loops:

```
j = 0;
k = 1;
while (j < 10)
{
   while (k < 10)
   {
      <replace uses of i by (10 * j) + k >
      k++;
   }
   k = 0;
   j++;
}
```

1.3.5 Array transformations

There are many ways in which arrays can be obfuscated. One of the simplest ways is to change the array indices. Such a change could be achieved either by a variable transformation (such as in Section 1.3.3) or by defining a permutation. Here is an example permutation for an array of size n:

$$p = \lambda i.(a \times i + b \pmod{n}) \text{ where } \gcd(a, n) = 1$$

Other array transformations involve changing the structure of an array. One way of changing the structure is by choosing different array dimensions. We could fold a 1-dimensional array of size $m \times n$ into a 2-dimensional array of size $[m, n]$. Similarly we could flatten an n-dimensional array into a 1-dimensional array.

Before performing array transformations, we must ensure that the arrays are safe to transform. For example, we may require that a whole array is not passed to another method or that elements of the array do not throw exceptions.

1.3.6 Array Splitting

Collberg *et al.* [10] gives an example of a structural change called an *array split*:

```
int [] A = new int [10];          int [] A1 = new int [5];
                                  int [] A2 = new int [5];
...                        ⇒      ...                                    (1.3)
A[i] = ...;                       if ((i % 2) == 0) A1[i/2] = ...;
                                                    else A2[i/2] = ...;
```

How can we generalise this transformation? We need to define a split so that an array A of size n is broken up into two other arrays. To do this, we define three functions ch, f_1 and f_2 and two new arrays B_1 and B_2 of sizes m_1 and m_2 respectively (where $m_1 + m_2 \geqslant n$).

The types of the functions are as follows:

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

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

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

To ensure that there are no index clashes we require that f_1 is injective for the values for which ch is true (similarly for f_2).

This relationship can be generalised so that A could be split between more than two arrays. For this, ch should be regarded as a choice function — this will determine which array each element should be transformed to.

We can write the transformation described in (1.3) as:

$$A[i] = \begin{cases} B_1[i \text{ div } 2] & \text{if } i \text{ is even} \\ B_2[i \text{ div } 2] & \text{if } i \text{ is odd} \end{cases} \tag{1.4}$$

We will consider this obfuscation in much greater detail: in Section 2.3.7 we give a specification of this transformation and in Chapter 4 we apply splits to more general data-types.

1.3.7 Program transformation

We now show how we can transform a method using example (1.4) above. The statement $A[i] = V$ is transformed to:

$$\textbf{if } ((i\%2) == 0)\ \{B_1[i/2] = V;\}\ \textbf{else } \{B_2[i/2] = V;\} \tag{1.5}$$

and an occurrence of $A[i]$ on the right hand side of an assignment can be dealt with in a similar manner by substituting either $B_1[i/2]$ or $B_2[i/2]$ for $A[i]$.

As a simple example, we present an imperative program for finding the first n Fibonacci numbers:

```
A[0] = 0;
A[1] = 1;
i = 2;
while (i ≤ n)
{ A[i] = A[i−1] + A[i−2];
  i++;
}
```

In this program, we can easily spot the Fibonacci identity:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \qquad (\forall n \geq 2) \end{aligned}$$

Let us now obfuscate the array by using the rules above. We obtain the program shown in Figure 1.1 which, after some simplifications, becomes:

```
B1[0] = 0;
B2[0] = 1;
i = 2;
while (i ≤ n)
{ if (i % 2 == 0) B1[i/2] = B2[(i − 1)/2] + B1[(i − 2)/2];
               else B2[i/2] = B1[(i − 1)/2] + B2[(i − 2)/2];
  i++;
}
```

```
B1[0] = 0;
B2[0] = 1;
i = 2;
while (i ≤ n)
{  if (i % 2 == 0)
   { if ((i−1) % 2 == 0)
     {  if ((i−2) % 2 == 0) B1[i/2] = B1[(i−1)/2] + B1[(i−2)/2];
                       else B1[i/2] = B1[(i−1)/2] + B2[(i−2)/2];
     } else {  if ((i−2) % 2 == 0) B1[i/2] = B2[(i−1)/2] + B1[(i−2)/2];
                              else B1[i/2] = B2[(i−1)/2] + B2[(i−2)/2];
            }
   } else
   {  if ((i−1) % 2 == 0)
      {  if ((i−2) % 2 == 0) B2[i/2] = B1[(i−1)/2] + B1[(i−2)/2];
                        else B2[i/2] = B1[(i−1)/2] + B2[(i−2)/2];

      } else { if ((i − 2) % 2 == 0) B2[i/2] = B2[(i−1)/2] + B1[(i−2)/2];
                                else B2[i/2] = B2[(i−1)/2] + B2[(i−2)/2];
             }
   }
 i + +;
}
```

Figure 1.1: Fibonacci program after an array split

Simplification was achieved by removing infeasible paths. For instance, if we had

$$\textbf{if } (i \,\%\, 2 == 0) \ \{ \ \textbf{if } ((i{-}1) \,\%\, 2 == 0 \ \{X;\} \ \textbf{else } \{Y;\} \ \}$$

then we would know that the block of statements X could never be executed, since if $i \,\%\, 2 = 0$ is true then $(i-1) \,\%\, 2 = 0$ is false. So, out of the eight possible assignments, we can eliminate six of them.

1.3.8 Array Merging

For another obfuscation, we can the reverse the process of splitting an array by merging two (or more) arrays into one larger array. As with a split, we will need to determine the order of the elements in the new array.

Let us give a simple example of a merge. Suppose we have arrays B_1 of size m_1 and B_2 of size m_2 and a new array A of size $m_1 + m_2$. We can define a relationship between the arrays as follows:

$$A[i] = \begin{cases} B_1[i] & \text{if } i < m_1 \\ B_2[i{-}m_1] & \text{if } i \geq m_1 \end{cases}$$

This transformation is analogous to the concatenation of two sequences (lists).

1.3.9 Other obfuscations

All of the obfuscations that we have discussed so far only consider transformations within methods. Below are some transformations that deal with methods themselves:

- **Inlining** A method call is replaced with the body of the method.

- **Outlining** A sequence of statements within a method is made into a separate method.
- **Interleaving** Two methods are merged; to distinguish between the original methods, an extra parameter could be passed.
- **Cloning** Many copies of the same method are created by applying different transformations.

1.4 Conclusions

We have seen there is a need for software protection, and code obfuscation is one method for making reverse engineering harder. We have summarised some current obfuscation techniques and highlighted a few of the many commercial obfuscators available. In Section 1.2 we reviewed two definitions of obfuscation: the definition of Collberg *et al.* [10] uses various complexity metrics and Barak *et al.* [6] prove that obfuscation is impossible using their definition. We have seen how to apply obfuscations to variables and arrays. In Chapter 3 we shall develop a different approach to obfuscation which enables us to apply obfuscations to more abstract data structures. As a consequence we will give a new definition for obfuscation and also discuss the efficiency of obfuscated operations.

In the next chapter, we give a case study of a situation where there is a particular need for obfuscation and give specifications for some obfuscations.

Chapter 2

Obfuscations for Intermediate Language

> ...if you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language
>
> *The Hitchhiker's Guide to the Galaxy* (1979)

Before we look at a new approach to obfuscation, we present a case study in which we explore a potential application of obfuscation. We look at the intermediate language that is part of Microsoft's .NET platform and we give some obfuscations for this language. In particular, we specify some generalisations of some of the obfuscations given in Section 1.3.

2.1 .NET

The heart of Microsoft's .NET framework is the *Common Language Infrastructure* [23] which consists of an intermediate language (IL) which has been designed to be the compilation target for various source languages. IL is a typed stack based language and since it is at quite a high level, it is fairly easy to decompile IL to C# [3]. Two example decompilers are *Anakrino* [2] and *Salamander* [45]. IL is similar to Java bytecode but more complicated as it is the compilation target for various languages.

Programs written using .NET are distributed as portable executables (PEs) which are compact representations of IL. These executables are then just-in-time compiled on the target platform. We can translate a PE to IL by using a *disassembler* and we can convert back by using an *assembler*. This means that from a PE we can disassemble back to IL and then decompile back to C#. Since we can convert a PE to C#, there is a need for obfuscation.

2.1.1 IL Instructions

Before we describe some IL obfuscations, we give a brief overview of IL. We concentrate on a fragment of IL — the instructions of interest are shown in

`add`	Adds the top two elements of the stack
`beq.s` l	Branches to l on equality
`bge.s` l	Branches to l on greater than or equal to
`bgt.s` l	Branches to l on greater than
`ble.s` l	Branches to l on less than or equal to
`blt.s` l	Branches to l on less than
`bne.s` l	Branches to l on not equal
`br.s` l	Unconditional branch to l
`call`	Calls a method
`div`	Divides the top two elements of the stack
`dup`	Duplicates the top value of the stack
`ldarg.s` v	Pushes value of argument v onto the stack
`ldc.`t v	Pushes a constant v with type t onto the stack
`ldloc.s` v	Pushes value of variable v onto the stack
`mul`	Multiplies the top two elements of the stack
`rem`	Calculates the remainder when dividing
`ret`	Returns from a method
`sub`	Subtracts the top two elements of the stack
`starg.s` v	Pops a value and stores it in argument v
`stloc.s` v	Pops a value and stores it into v

Figure 2.1: Some common IL instructions

Figure 2.1. As an example, consider the following C# method which computes the GCD of two integers:

```
public static int gcd(int a, int b)
{
  int x = a;
  int y = b;
  while (x != y)
    if (x < y) y = y - x; else x = x - y;
  return x;
}
```

After compiling and disassembling this method, we obtain the IL code shown in Figure 2.2.

As IL is a typed stack language, each value handled by the stack must have a type associated with it. For example, `int32` is the type for 4-byte (32-bit) integers and `float64` for 8-byte real numbers — number types can also be signed or unsigned. Non numeric types include `string` and `bool`.

At the start of a method, various properties of the method are stated. First, the signature of the method is given. In the GCD example, we can see that the method expects two integers and returns one integer as the result. Next, the maximum stack depth in the method is specified by using the `.maxstack` keyword. If we make any changes to the IL code then we should check whether the maximum stack depth has increased and so must change the value of `.maxstack` accordingly. Finally the names and types of any local variables for this method are stated using the `.locals` keyword.

```
.method public static int32  gcd(int32 a, int32 b)
  {
    .maxstack  2
    .locals (int32 V_0, int32 V_1, int32 V_2)

    IL0000:  ldarg.0
    IL0001:  stloc.0                 IL0010:  ldloc.0
    IL0002:  ldarg.1                 IL0011:  ldloc.1
    IL0003:  stloc.1                 IL0012:  sub
    IL0004:  br.s     IL0014         IL0013:  stloc.0
                                     IL0014:  ldloc.0
    IL0006:  ldloc.0                 IL0015:  ldloc.1
    IL0007:  ldloc.1                 IL0016:  bne.un.s IL0006
    IL0008:  bge.s    IL0010
                                     IL0018:  ldloc.0
    IL000a:  ldloc.1                 IL0019:  stloc.2
    IL000b:  ldloc.0                 IL001a:  br.s     IL001c
    IL000c:  sub
    IL000d:  stloc.1                 IL001c:  ldloc.2
    IL000e:  br.s     IL0014         IL001d:  ret
}
```

Figure 2.2: IL for a GCD method

In the main body of an IL method, instructions can be preceded by a unique label — typically when using ILDASM (the disassembler distributed with .NET) each instruction has a label of the form `IL****`. Labels are needed for branch instructions.

A value from the stack can be stored in a local variable, V, by using the instruction `stloc.s` V and similarly, a value stored in a local variable V can be pushed on the stack using `ldloc.s` V. If the local variable that is being accessed is one of the first four variables that was declared in this method, then there is a shorter form of `stloc` and `ldloc`. Suppose the local variable were `.locals(int32 V, int32 W)`. Then we could write `ldloc.0` or `stloc.1` instead of `ldloc.s V` or `stloc.s W`, respectively. Values passed to the method are accessed using `ldarg` and `starg`. So in the GCD method, `ldarg.0` loads the value of "a" onto the stack. We use the command `ldc` to load constants (with the appropriate type) onto the stack. As with `ldloc`, there is another form of the instruction: we can write `ldc.i4.4` instead of `ldc.i4 4` for the integers 0..8 (note that `i4` means a 4-byte integer).

Arithmetic operations, such as `add` and `sub`, take values off the stack and then put the result back onto the stack. For instance, in the following sequence:

```
ldc.i4.7
ldc.i4.2
sub
```

`sub` will pop 7 and 2 off the stack and push 5 back.

Branches can either be unconditional (of the form `br`) or conditional (such as `bne` — "jump if not equal to"), which compares values from the stack, and requires a target (corresponding to a label). For a conditional jump, if the condition is true, then the next instruction to be executed will be the one at

the target label, otherwise the next instruction following the branch will be executed.

In this example,

```
IL0001:  ldloc.s V
IL0002:  ldc.i4.1
IL0003:  bge.s IL0020
IL0004:  ...
...
IL0020:  ...
```

if $V \geq 1$ then `IL0020` will be executed next — otherwise, `IL0004` will be executed.

When writing IL methods, we require that the code that we produce is *verifiable* — here are some conditions that must be met for verifiability:

- Stacks must have the same height and contain the same types when control flow paths meet.
- Operations must have the correct number of items on the stack (*e.g.* for a binary operation there must be at least two elements on the stack).
- Operations must receive the type that they expect off the stack

If we have verified code then we can be sure that the code will run safely (*e.g.* the code will not access memory locations that it not permitted to) and so we must ensure that any obfuscations that we apply produce verifiable code.

2.2 IL obfuscations

Now, we will look at how to perform some obfuscations on IL by manually editing an IL file and assembling this file to make a PE. We look at some of the obfuscations given in [10] and we show how to write them in IL. The aim of performing obfuscations on IL is to make it hard for a decompiler to take a PE and produce C#. Ideally, we would like to stop the decompilation process altogether but at the very least, we should make the resulting code harder to understand.

2.2.1 Variable Transformation

For the first example of obfuscating IL, we show how to perform a simple variable transformation (as outlined in Section 1.3.3). The functions we will use to perform the transformations are:

$$\begin{aligned} f &= \lambda i.(2i - 1) \\ g &= \lambda j.((j + 1)/2) \end{aligned}$$

Assignment of a variable corresponds to `stloc` and use corresponds to `ldloc`. Using the GCD example given in Figure 2.2, we aim to transform the local

variable `V_0`. So any occurrences of `stloc.0` will need to be replaced by:

```
ldc.i4.2
mul
ldc.i4.1
sub
stloc.0
```

We replace the instruction `ldloc.0` by:

```
ldloc.0
ldc.i4.1
add
ldc.i4.2
div
```

Also, we need to change the value of `.maxstack` to 4 as there are more items to be stored on the stack. After assembling and decompiling, we obtain the following program:

```
private static int gcd(int a, int b)
{
  int x = a * 2 - 1;
  int y = b;
  while ((x + 1)/2 != y)
  { if ((x + 1)/2 < y) {y = y - (x + 1)/2; }
    else {x = ((x + 1)/2 - y) * 2 - 1; }
  }
  return (i + 1)/2;
}
```

For this trivial example, only a handful of changes need to be made in the IL. For a larger method which has more uses and definitions or for a complicated transformation, manually editing would be time-consuming. So it would be desirable to automate this process.

2.2.2 Jumping into while loops

C# contains a jump instruction **goto** which allows a program to jump to a statement marked by the appropriate label. The **goto** statement must be within the scope of the labelled statement. This means that we can jump out of a loop (or a conditional) but not into a loop (or conditional). This ensures that a loop has exactly one entry point (but the loop can have more than one exit by using **break** statements). This restriction on the use of **goto** ensures that all the flow graphs are *reducible* [1, Section 10.4]. Thus, the following program fragment would be illegal in C#:

```
if (P) goto S1;
...
  while (G)
  {
      ...
  S1: ...
      ...
  }
```

```
gcd(int a, int b)
{        int i;
         int j;
         i = a;
         j = b;
         if (i * i > 0) {goto IL001a;}
         else {goto IL0016;}
IL000c: if (i < j) {j -= i; continue;}
IL0016: i -= j;
IL001a: if (i == j) {return i;}
        else {goto IL000c;}
}
```

Figure 2.3: Output from Salamander

However in IL, we are allowed to use branches to jump into loops (**while** loops do not, of course, occur in IL — they are achieved by using conditional branches). So, if we insert this kind of jump in IL, we will have a control flow graph which is irreducible and a naive decompiler could produce incorrect C# code. A smarter decompiler could change the **while** into an **if** statement that uses **goto** jumps. As we do not actually want this jump to happen, we use an opaque predicate that is always false.

Let us look at the GCD example in Section 2.1.1 again. Suppose that we want to insert the jump:

if $((x * x) < 0)$ **goto** L;

before the **while** loop where L is a statement in the loop. So, we need to put instructions in the IL file to create this:

```
IL0100: ldloc.0
IL0101: ldloc.0
IL0102: mul
IL0103: ldc.i4.0
IL0104: blt.s  IL0010
```

The place that we jump to in the IL needs to be chosen carefully — a suitable place in the GCD example would be between the instructions `IL0003` and `IL0004`. We must (obviously) ensure that it does actually jump to a place inside the **while** loop. Also, we must ensure that we do not interfere with the depth of the stack (so that we can still verify the program). Figure 2.3 shows the result of decompiling the resulting executable using the Salamander decompiler [45]. We can see that the **while** statement has been removed and in its place is a more complicated arrangement of **if**s and **goto**s.

This obfuscation as it stands is not very resilient. It is obvious that the conditional $x * x < 0$ can never be true and so the jump into the loop never happens.

2.3 Transformation Toolkit

In the last section we saw writing obfuscations for IL involved finding appropriate instructions and replacing all occurrences of these instructions by a set of

new instructions. This replacement is an example of a *rewrite rule* [32]:

$$L \Rightarrow R \text{ if } \; c$$

which says that if the condition c is true then we replace each occurrence of L with R. This form of rewrite rule can be automated and we summarise the transformation toolkit described in [22] to demonstrate one way of specifying obfuscations for IL. This toolkit consists of three main components:

- A *representation of IL*, called *EIL*, which makes specifying transformations easier.
- A new *specification language*, called *Path Logic Programming*, which allows us to specify program transformations on the control flow graph.
- A *strategy language* with which we can control how the transformations are applied.

We briefly describe these components (more details can be found in [22]) before specifying some of the generalised obfuscations given in Section 1.3.

2.3.1 Path Logic Programming

Path Logic Programming (PLP) is a new language developed for the specification of program transformations. PLP extends Prolog [51] with new primitives to help express the side conditions of transformations. The Prolog program will be interpreted relative to the flow graph of the object program that is being transformed. One new primitive is

$$\mathit{all}\ Q\ (N, M)$$

which is true if N and M are nodes in the flow graph, and all paths from N to M are of the form specified by the pattern Q. Furthermore, there should be at least one path that satisfies the pattern Q. This is to stop a situation where we do not have a path between N and M and so the predicate $\mathit{all}\ Q\ (N, M)$ will be vacuously true. Similarly, the predicate

$$\mathit{exists}\ Q\ (N, M)$$

is true if there exists a path from N to M which satisfies the pattern Q.

As an example of *all*, consider:

$$\begin{array}{l} \mathit{all}\ (\ \{\,\}*; \\ \qquad \{\ '\mathit{set}(X, A), \\ \qquad\ \ \mathit{local}(X)\,\}; \\ \qquad \{\ \mathit{not}\,('\mathit{def}(X))\,\}*; \\ \qquad \{\ '\mathit{use}(X)\,\} \\ \quad)\ (\mathit{entry}, N) \end{array}$$

This definition says that all paths from the program entry to node N should satisfy a particular pattern. A *path* is a sequence of edges in the flow graph. The pattern for a path is a regular expression and in the above example the regular expression consists of four components:

- Initially we have zero or more edges that we do not particularly care about which is indicated by $\{\}*$ — the interpretation of $\{\}$ is a predicate that is always true.
- Next, we require an edge whose target node is an expression of the form $X := A$ where X is local variable.
- Then we want zero or more edges to nodes that do not re-define X.
- Finally we reach an edge pointing to node N which uses variable X.

A pattern, which denotes an property on the control flow graph, is a regular expression whose alphabet is given by temporal goals — the operator ; represents sequential composition, + represents choice, $*$ is zero or more occurrences and ϵ an empty path. A *temporal goal* is a list of temporal predicates, enclosed in curly brackets. A temporal predicate is either an ordinary predicate (like *local* in the example we just examined), or a ticked predicate (like *use*). Ticked predicates are properties involving edges and are denoted by the use of a tick mark ($'$) in front of the predicate. For example, $def(X, E)$ is a predicate that takes two arguments: a variable X and an edge E, and it holds true when the edge points at a node where X is assigned. Similarly, $use(X, E)$ is true when the target of E is a node that is labelled with a statement that makes use of X. When we place a tick mark in front of a predicate inside a path pattern, the current edge is added as a final parameter when the predicate is called.

We can think of the path patterns in the usual way as automata, where the edges are labelled with temporal goals. In turn, a temporal goal is interpreted as a property of an edge in the flow graph. The pattern

$$\{p_0, p_1, \ldots, p_{k-1}\}$$

holds at edge e if each of its constituents holds at edge e. To check whether a ticked predicate holds at e, we simply add e as a parameter to the given predicate and non-ticked predicates ignore e. A syntax for this language and a more detail discussion of *all* and *exists* is given in [22].

2.3.2 Expression IL

Since IL is a stack based language, performing a simple variable transformation described in Section 2.2.1 leads to performing quite a complicated replacement. To make specifying transformation easier we work with a representation of IL which replaces stack-based computations with expressions — this representation is called *Expression IL* (EIL). To convert from IL to EIL, we introduce a new local variable for each stack location and replace each IL instruction with an assignment. As we use only verifiable IL, this translation is possible.

EIL is analogous to both the Jimple and Grimp languages from the SOOT framework [53, 54] — the initial translation produces code similar to the three-address code of Jimple, and assignment merging leaves us with proper expressions like those of Grimp.

$$
\begin{aligned}
\mathit{stmt} &::= \texttt{nop} \mid \mathit{var} := \mathit{exp} \mid \texttt{br}\ \ \mathit{target} \mid \texttt{brif}\ \ \mathit{cnd}\ \mathit{target} \\
\mathit{exp} &::= \mathit{var} \mid \mathit{const} \mid \mathit{monop}(\mathit{exp}) \mid \mathit{binop}(\mathit{exp}, \mathit{exp}) \\
\mathit{monop} &::= \texttt{neg} \\
\mathit{binop} &::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{rem} \\
\mathit{var} &::= \texttt{local}(\mathit{name}) \mid \texttt{arg}(\mathit{name}) \mid \texttt{local}(\mathit{name})[\mathit{exp}] \\
\mathit{const} &::= \texttt{ldc}.\mathit{type}\ \ \mathit{val} \\
\mathit{cnd} &::= \texttt{and}(\mathit{cnd}, \mathit{cnd}) \mid \texttt{or}(\mathit{cnd}, \mathit{cnd}) \mid \texttt{not}(\mathit{cnd}) \mid \texttt{ceq}(\mathit{exp}, \mathit{exp}) \mid \texttt{cne}(\mathit{exp}, \mathit{exp}) \\
&\quad\ \mid \texttt{clt}(\mathit{exp}, \mathit{exp}) \mid \texttt{cle}(\mathit{exp}, \mathit{exp}) \mid \texttt{cgt}(\mathit{exp}, \mathit{exp}) \mid \texttt{cge}(\mathit{exp}, \mathit{exp}) \\
\mathit{target} &::= \mathit{string} \\
\mathit{name} &::= \mathit{string} \\
\mathit{type} &::= \texttt{i4} \mid \texttt{i8} \\
\mathit{instr} &::= [\mathit{target}\ :]\mathit{stmt} \\
\mathit{prog} &::= \mathit{instr}\ \ \mathit{list}
\end{aligned}
$$

Figure 2.4: Syntax for a simple subset of EIL

Thus, the following set of IL instructions:

```
ldc.i4.2
stloc x
ldc.i4.3
ldloc x
add
stloc y
```

is converted to something like:

```
local(v1) := ldc.i4.2
local(x)  := local(v1)
local(v2) := ldc.i4.3
local(v3) := local(x)
local(v4) := add(local(v2), local(v3))
local(y)  := local(v4)
```

We can then perform standard compiler optimisations (such as constant propagation and dead code elimination) to simplify this set of instructions further. We concentrate on only a simple fragment of EIL and a concrete syntax for this fragment is given in Figure 2.4.

This concrete syntax omits many significant details of EIL; for example, all expressions are typed and arithmetic operators have multiple versions with different overflow handling. This detail is reflected in the representation of these expressions as logic terms. For example, the integer 5 becomes the logic term

$$\mathit{expr_type}(\mathit{ldc}(\mathit{int}(\mathit{true}, b32), 5), \mathit{int}(\mathit{true}, b32))$$

The first parameter of *expr_type* is the expression and the second is the type — this constructor reflects the fact that all expressions are typed. The type $\mathit{int}(\mathit{true}, b32)$ is a 32-bit signed integer (the *true* would become *false* if we wanted an unsigned one). To construct a constant literal, the constructor *ldc* is used — it takes a type parameter, which is redundant but simplifies the processing of EIL in other parts of the transformation system, and the literal value.

For a slightly more complicated example, the expression $x + 5$ (where x is a local variable) is represented by

$$\begin{aligned}&\mathit{expr_type}(\mathit{applyatom}(\mathit{add}(\mathit{false}, \mathit{true}),\\ &\qquad\qquad\qquad \mathit{expr_type}(\mathit{localvar}(\mathit{sname}(``x")),\\ &\qquad\qquad\qquad\qquad \mathit{int}(\mathit{true}, b32)),\\ &\qquad\qquad\qquad \mathit{expr_type}(\mathit{ldc}(\mathit{int}(\mathit{true}, b32), 5),\\ &\qquad\qquad\qquad\qquad \mathit{int}(\mathit{true}, b32))),\\ &\qquad \mathit{int}(\mathit{true}, b32))\end{aligned}$$

The term $\mathit{localvar}(\mathit{sname}(``x"))$ refers to the local variable x — the seemingly redundant constructor *sname* reflects the fact that it is also possible to use a different constructor to refer to local variables by their position in the method's declaration list, although this facility is not used.

The constructor *applyatom* exists to simplify the relationship between IL and EIL — the term $\mathit{add}(\mathit{false}, \mathit{true})$ directly corresponds to the IL instruction *add*, which adds the top two items on the stack as signed values without overflow. Thus, the meaning of *applyatom* can be summarised as: "apply the IL instruction in the first parameter to the rest of the parameters, as if they were on the stack".

Finally, it remains to explain how EIL *instructions* are defined. It is these that will be used to label the edges and nodes of flow graphs. An instruction is either an expression, a branch or a return statement, combined with a list of labels for that statement using the constructor *instr_label*. For example, the following defines a conditional branch to the label *target*:

$$\begin{aligned}&\mathit{instr_label}(``\mathit{conditional}" : \mathit{nil},\\ &\qquad \mathit{branch}(\mathit{cond}(\ldots), ``\mathit{target}"))\end{aligned}$$

Note that we borrow the notation for lists from functional programming, writing $X : Xs$ instead of $[X|Xs]$. If the current instruction is an expression, then *exp* enclosing an expression would be used in place of *branch*, and similarly *return* is used in the case of a return statement.

2.3.3 Predicates for EIL

The nodes of the flow graph are labelled with the logic term corresponding to the EIL instruction at that node. In addition, each edge is labelled with the term of the EIL instruction at the node that the edge points to; it is these labels that are used to solve the existential and universal queries.

The logic language provides primitives to access the relevant label given a node or an edge — $@\mathit{elabel}(E, I)$ holds if I is the instruction at edge E, and $@\mathit{vlabel}(V, I)$ holds if I is the instruction at node V (where @ denotes a primitive).

We can define the *set* predicate used in Section 2.3.1 as follows:

$$\begin{aligned}&\mathit{set}(X, A, E) :-\\ &\qquad @\mathit{elabel}(E, \mathit{instr_label}(_, \mathit{exp}(\mathit{expr_type}(\mathit{assign}(X, A), _)))).\end{aligned}$$

Note that we denote unbound variables by using an underscore. It is straightforward to define *def* in terms of *set*:

$$\mathit{def}(X, E) \;:-\; \mathit{set}(X, _, E)$$

$\mathit{build}(N, V, X, M)$	builds M by replacing uses of X in N with V
$\mathit{def}(X, E)$	holds if X is defined at edge E
$@\mathit{elabel}(E, I)$	holds if I is the instruction at edge E
$@\mathit{new_vertex}(L, \mathit{Es}, M)$	builds a vertex M with edges Es and label L
$\mathit{occurs}(R, X)$	holds if X occurs in R
$\mathit{set}(X, A, E)$	holds if X is assigned the value A at E
$\mathit{source}(N, E)$	holds if the vertex N is the source of the edge E
$\mathit{subst}(X, V, M, N)$	builds N from M by replacing uses of X with V
$\mathit{use}(X, E)$	holds if X is used at E
$@\mathit{vlabel}(V, I)$	holds if I is the instruction at node V

Figure 2.5: Some common predicates

The definition of *use* is based on the predicate $\mathit{occurs}(R, X)$, which checks whether X occurs in R (by the obvious recursive traversal). When defining $\mathit{use}(X, E)$, we want to distinguish *uses* of X from *definitions* of X, whilst still finding the uses of the variable x in expressions such as $a[x] := 5$ and $x := x + 1$:

$$\begin{aligned} \mathit{use}(X, E) &:- \ @\mathit{elabel}(E, S), \mathit{occurs}(S, X), \\ &\quad\ \ \mathit{not}(\mathit{def}(X, E)). \\ \mathit{use}(X, E) &:- \ \mathit{set}(_, R, E), \mathit{occurs}(R, X). \end{aligned}$$

The common predicates that we will use are summarised in Figure 2.5.

2.3.4 Modifying the graph

When creating new vertices the predicate *build* will often be used. The expression $\mathit{build}(N, V, X, M)$ creates a new vertex M, by copying the old vertex N, replacing uses of X with V:

$$\begin{aligned} \mathit{build}(N, V, X, M) :- \ & @\mathit{vlabel}(N, \mathit{Old}), \\ & \mathit{subst}(V, X, \mathit{Old}, \mathit{New}), \\ & \mathit{listof}(E, \mathit{source}(N, E), \mathit{Es}), \\ & @\mathit{new_vertex}(\mathit{New}, \mathit{Es}, M). \end{aligned}$$

The predicate:

$$\mathit{subst}(V, X, \mathit{Old}, \mathit{New})$$

constructs the term *New* from *Old*, replacing uses of V with X. As with *use*, it is defined so as not to apply this to *definitions* of X — if we are replacing x with 0 in $x := x + 1$ we want to end up with $x := 0 + 1$, not $0 := 0 + 1$.

New vertices are constructed by using *@new_vertex*. This primitive takes a vertex label and a list of outgoing edges and binds the new vertex to its final parameter. For the definition of *build* we use the same list of edges as the old vertex, since all we wish to do is to replace the label.

The predicate $\mathit{source}(N, E)$ is true if the vertex N is the source of the edge E, whilst the *listof* predicate is the standard Prolog predicate which takes three parameters: a term T, a predicate involving the free variables of T, and a third parameter which will be bound to a list of all instantiations of T that solve the predicate. Thus the overall effect of $\mathit{listof}(E, \mathit{source}(N, E), \mathit{Es})$ is to bind *Es* to the outgoing edges from node N, as required.

2.3.5 Applying transformations

Although the logic language we have described makes it convenient to define side conditions for program transformation, it would be rather difficult to use this language to apply these transformations, since that would require the program flow graph to be represented as a logic term. The approach that is taken is that a successful logic query should also bind its parameter to a list of symbolic "actions" which define a correct transformation on the flow graph. A high-level strategy language, which is similar to Stratego [55], is responsible for directing in what order logic queries should be tried and for applying the resulting transformations.

An action is a term, which can be either of the form $\mathit{replace_vertex}(V, W)$ or $\mathit{new_local}(T, N)$. The former replaces the vertex V with the vertex W, while the latter introduces a new local variable named N of type T. We write a list of actions as the last parameter of predicate.

Suppose that we have a predicate P specified. If we want to apply this predicate once then we write $\mathit{apply}(P)$ in the strategy language. If we want to exhaustively apply this predicate (*i.e.* keep applying the predicate while it is true) then we write $\mathit{exhaustively}(\mathit{apply}(P))$. An example of a strategy is given at the end of the next section.

2.3.6 Variable Transformation

For our first example, let us consider how to define a variable transformation (Section 1.3.3). The first part of the transformation involves finding a suitable variable and for simplicity, we require that the variable is local and has integer type. We also require that it is assigned somewhere (otherwise there would be no point in performing the transformation). Once we find a suitable variable (which we call OldVar), we generate a new name using @*fresh_name* which takes a type as a parameter. We can write the search for a suitable variable as follows:

$$
\begin{array}{l}
\mathit{find_local}\;(\mathit{OldVar},\\
\qquad\qquad \mathit{NewVar},\\
\qquad\qquad \mathit{new_local}(\mathit{int}(\mathit{true}, b32), \mathit{NewVarName}) : \mathit{nil}\\
\qquad\qquad)\ :-\\
\quad \mathit{exists}\;(\;\{\,\}*\,;\\
\qquad\qquad \{\;'\mathit{set}(\mathit{OldVar}, V),\\
\qquad\qquad\quad \mathit{OldVar} = \mathit{expr_type}(\mathit{localvar}(_), \mathit{int}(\mathit{true}, b32))\}\\
\qquad\qquad)\quad (\mathit{entry}, \mathit{OldVarVert}),\\
\quad @\mathit{fresh_name}\;(\mathit{int}(\mathit{true}, b32), \mathit{NewVarName}),\\
\quad \mathit{NewVar} = \mathit{expr_type}(\mathit{localvar}\;(\mathit{sname}(\mathit{NewVarName})),\\
\qquad\qquad\qquad\qquad \mathit{int}(\mathit{true}, b32)).
\end{array}
$$

The next step of the transformation replaces uses and assignments of OldVar exhaustively. To do this, we need to define predicates which allow us to build expressions corresponding to f and g (the functions used for variable transformation). The predicate $\mathit{use_fn}(A, B)$ binds B to a representation of $g(A)$ and similarly, $\mathit{assign_fn}(C, D)$ binds D to $f(C)$. We can specify any variable transformation by changing these two predicates. As an example, let us suppose

that

$$
\begin{aligned}
f &= \lambda i.(2i) \\
g &= \lambda j.(j/2)
\end{aligned}
$$

Then we would define

$$
\begin{aligned}
&\mathit{use_fn}(A, \\
&\qquad \mathit{expr_type}(\mathit{applyatom}(\mathit{cdiv}(\mathit{true}), \\
&\qquad\qquad\qquad A, \\
&\qquad\qquad\qquad \mathit{expr_type}(\mathit{applyatom}(\mathit{ldc}(\mathit{int}(\mathit{true}, b32), 2)), \\
&\qquad\qquad\qquad\qquad\qquad \mathit{int}(\mathit{true}, b32))), \\
&\qquad \mathit{int}(\mathit{true}, b32))).
\end{aligned}
$$

and we can define *assign_fn* similarly.

We define a predicate *replace_var* which replaces occurrences of *OldVar* with *NewVar* with the appropriate transformation. We need to deal with assignment and uses separately. We can easily write a predicate to replace uses of *OldVar* as follows:

$$
\begin{aligned}
&\mathit{replace_var}\ (\mathit{OldVar}, \\
&\qquad\qquad \mathit{NewVar}, \\
&\qquad\qquad \mathit{replace_vertex}(\mathit{OldVert}, \mathit{NewVert}) : \mathit{nil} \\
&\qquad\qquad)\ :- \\
&\quad \mathit{exists}\ (\{\ \}*; \\
&\qquad\quad \{\ '\mathit{use}(\mathit{OldVar})\} \\
&\qquad\quad)\ \ (\mathit{entry}, \mathit{OldVert}), \\
&\quad \mathit{use_fn}\ (\mathit{NewVar}, \mathit{NewUse}), \\
&\quad \mathit{build}\ (\mathit{OldVert}, \mathit{NewUse}, \mathit{OldVar}, \mathit{NewVert}).
\end{aligned}
$$

To replace assignments to *OldVar* we cannot use the *build* and instead have to create a new vertex manually — the definition is shown in Figure 2.6. In this definition, we use the predicate *set* to find a vertex where there is an assignment of the form $\mathit{OldVar} := \mathit{OldVal}$. At this vertex, we find a list of edges from this vertex and the list of labels. We then create the expression $f(\mathit{OldVal})$ and bind it to *NewVal*. Finally, we use the predicate *build_assign* to create a new vertex that contains the instruction $\mathit{NewVar} := \mathit{NewVal}$ and which has the same labels and edges as *OldVar*.

To apply this transformation, we write the following two lines in the strategy language:

$$
\begin{aligned}
&\mathit{apply}(\mathit{find_local}(\mathit{OldVar}, \mathit{NewVar})); \\
&\mathit{exhaustively}(\mathit{apply}(\mathit{replace_var}(\mathit{OldVar}, \mathit{NewVar})))
\end{aligned}
$$

The first line applies the search to find a suitable variable and the second exhaustively replaces all occurrences of the old variable with the new variable.

2.3.7 Array Splitting

For our second example, we consider splitting an array using the method stated in Section 1.3.6. First, we briefly describe how to find a suitable array — more details can be found in [22].

```
replace_var (OldVar,
             NewVar,
             replace_vertex(OldVert, NewVert) : nil
             ) :−
    exists ({ }∗ ;
            { 'set(OldVar, OldVal)}
           ) (entry, OldVert),
    listof (OldEdge, source(OldVert, OldEdge), OldEdges),
    @vlabel(OldVert, instr_label(Labels, _)),
    assign_fn(OldVal, NewVal),
    build_assign(Labels, NewVar, NewVal, OldEdges, NewVert).

build_assign(Labs, L, R, Edges, Vert) :−
    @new_vertex(instr_label(Labs, exp(expr_type(assign(L, R),
                                                 int(true, b32)))),
                Edges, Vert).
```

Figure 2.6: Predicates for replacing assignments

Finding a suitable array

We need to find a vertex *InitVert* which has an array initialisation of the form:

$$OldArray := newarr(Type)[Size]$$

where *OldArray* is a local variable of array type.

For this transformation to be applicable, we ensure that every path through our method goes through *InitVert* and that the array is always used with its index (except at the initialisation) — so for example, we cannot pass the whole array to another method. We can write this safety condition as follows:

```
safe_to_transform(InitVert, OldArray) :−
  all ({ }∗ ;
       { 'isnode(InitVert) } ;
       { not('unindexed(OldArray)) }∗
       ) (entry, exit).
```

The predicate *unindexed*(*OldArray*, E) holds if *OldArray* is used without an index at the node pointed to by edge E. We define *unindexed* by pattern matching to EIL expressions. So, for example:

```
                        unindexed(A, A).
            unindexed(exp(E), A)  :−  unindexed(E, A).
    unindexed(expr_type(E, T), A)  :−  unindexed(E, A).
  unindexed(arrayindex(L, R), A)  :−  not(L = A), unindexed(L, A).
  unindexed(arrayindex(L, R), A)  :−  unindexed(R, A).
```

After a suitable array has been found we need to create initialisations for our two new arrays with the correct sizes.

```
replace_array(A, B1, B2, replace_vertex(N, M) : nil) :−
    exists ({ } ∗;
            { 'set(X, V),
              X = expr_type(arrayindex(A, I), _) }
           )  (entry, N),
    listof(E, source(N, E), Es),
    @vlabel(N, instr_label(L, _)),
    is_even(I, C),
    index(I, J),
    @fresh_label(ThenLab),
    build_assign(ThenLab : nil,
                 expr_type(arrayindex(B1, J), int(true, b32)),
                 V, Es, ThenVert),
    new_edge(ThenVert, ThenEdge),
    @fresh_label(ElseLab),
    build_assign(ElseLab : nil,
                 expr_type(arrayindex(B2, J), int(true, b32)),
                 V, Es, ElseVert),
    new_edge(ElseVert, ElseEdge),
    @new_vertex(instr_label(L, branch(cond(C), ThenLab)),
                 ThenEdge : ElseEdge : nil, M).
```

Figure 2.7: Definition of *replace_array* for assignments

Replacing the array

Once we have identified which array we want to split (and checked that it is safe to do so) then we need to replace exhaustively the original array with our two new ones. As with variable transformation, we have to treat uses and assignments of array values separately. But instead of a straightforward substitution, we have to insert a conditional expression at every occurrence of the original array.

Let us consider how we can replace an assignment of an array value. Suppose that we want to split an array A into two arrays B_1 and B_2 using the split (Equation (1.4)) defined in Section 1.3.6. For an assignment of A, we look for a statement of the form:

$$A[I] = V$$

and, from Equation (1.5), we need to replace it with a statement of the form:

$$\textbf{if } ((i\%2) == 0)\ \{B_1[i/2] = V;\}\ \textbf{else } \{B_2[i/2] = V;\}$$

The predicate for this transformation is given in Figure 2.7. The first step in the transformation is to find an assignment to A by using the predicate *set*. We want to match the left-hand side of an assignment to

$$expr_type(arrayindex(A, I), T)$$

where I is the index of the array (and T is the type). Once we find the node, N, of such an assignment, we need to find the list of outgoing edges from N and the label L for N.

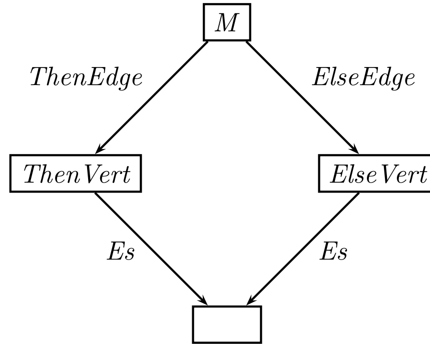


Figure 2.8: Flow graph to build a conditional expression

Next, we create a test C corresponding to $i \,\%\, 2 == 0$ in the the predicate $is_even(I, C)$ and we create an expression J for the indices of the new arrays, using the predicate $index(I, J)$ (we omit the details of these predicates).

Now, we need to create vertices corresponding to the branches of the conditional — the flow graph we want to create is shown in Figure 2.8. To create the "then" branch, we first obtain a fresh label *ThenLab* with which we create a new vertex *ThenVert*. This vertex needs to contain an instruction $B_1[J] = V$ and have the same outgoing edges as N. Then we create a new incoming edge, *ThenEdge*, for *ThenVert*. The construction of the "else" branch is similar (we build the instruction $B_2[J] = V$ instead).

Finally, we are ready to build a new vertex M which replaces N. This vertex contains an instruction for the conditional which has label L. The outgoing edges for this vertex are *ThenEdge* and *ElseEdge*.

Replacing uses of the original array is slightly easier. Instead of manually building *ThenVert* and *ElseVert* we can use the predicate *subst* to replace $A[I]$ with $B_1[J]$ or $B_2[J]$ as appropriate.

We can adapt this specification for other array splits. In Section 1.3.6, we defined an array split in terms of three functions ch, f_1 and f_2. The predicate *is_even* corresponds to the function ch and *index* corresponds to f_1 and f_2 (which are equal for our example). So by defining a predicate that represents ch and two predicates for f_1 and f_2, we can write a specification for any split that creates two new arrays.

2.3.8 Creating an irreducible flow graph

In Section 2.2.2, we saw how by placing an **if** statement before a loop we could create a method with an irreducible control flow graph. Could we specify this transformation in PLP? The control flow graph that we want to search for is shown in Figure 2.9 and we would like to place a jump from the node *Jump* to the node *InLoop*. However, in EIL (and IL) we do not have an instruction corresponding to **while**; instead we have to identify a loop from the flow graph.

Figure 2.10 contains the logic program that performs this transformation. The code has two parts: the first finds a suitable loop and the second builds the jump.

Let us look at the conditions needed to find a loop. First we want to find a

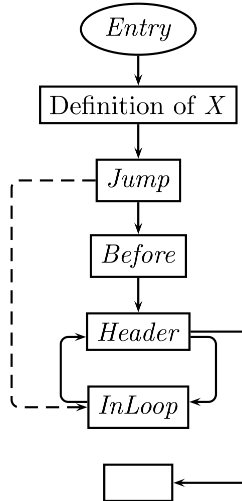


Figure 2.9: Flow graph to create jump into a loop

node such that there is a path from this node to itself. The conditions

$$
\begin{array}{l}
\mathit{exists}\,(\{\,\};\\
\qquad\{\,\}*\\
\qquad)\quad(\mathit{entry}, \mathit{Header}),\\
\mathit{exists}\,(\{\,\};\\
\qquad\{\,\}*\\
\qquad)\quad(\mathit{Header}, \mathit{Header})
\end{array}
$$

find such a node which we call *Header*. We want to make sure that this is the "first" node in the loop (*i.e.* the *header* of the loop) and so there must be a node immediately before *Header*:

$$
\begin{array}{l}
\mathit{exists}\,(\{\,\};\\
\qquad\{\,\}*\\
\qquad)\quad(\mathit{entry}, \mathit{Before}),\\
\mathit{exists}\,(\{\,\}\\
\qquad)\quad(\mathit{Before}, \mathit{Header})
\end{array}
$$

and this node, *Before*, is not in the loop (*i.e.* there is not a path from *Loop* to *Before*):

$$
\begin{array}{l}
\mathit{not}\,(\mathit{exists}\,(\{\,\};\\
\qquad\qquad\{\,\}*\\
\qquad\qquad)\quad(\mathit{Header}, \mathit{Before}))
\end{array}
$$

We want to jump to a node in the loop that is not *Header* so that we can create an irreducible jump. To find another node in the loop, we require that a node has a path to itself and this node is not *Header*:

$$
\begin{array}{l}
\mathit{exists}\,(\{\,\};\\
\qquad\{\,\}*\\
\qquad)\quad(\mathit{Header}, \mathit{InLoop}),\\
\mathit{exists}\,(\{\,\}*;\\
\qquad\{\,\mathit{not}\,({}'\mathit{is_node}\,(\mathit{Header}))\,\}\\
\qquad)\quad(\mathit{InLoop}, \mathit{InLoop})
\end{array}
$$

```
irred_jump(replace_vertex(Jump, Branch) : nil):−
    exists ({ };
            { }∗
            )   (entry, Header),
    exists ({ };
            { }∗
            )   (Header, Header),
    exists ({ };
            { }∗
            )   (entry, Before),
    exists ({ }
            )   (Before, Header),
    not (exists ({ };
                  { }∗
                  )   (Header, Before)),
    exists ({ };
            { }∗
            )   (Header, InLoop),
    exists ({ }∗;
            { not ('is_node (Header)) }
            )   (InLoop, InLoop),
    all ({ }∗;
         {'def(X),
          X = expr_type(localvar(_), int(true, b32))};
         { }
         )   (entry, Jump),
    not (exists ({ };
                  { }∗
                  )   (Header, Jump)),

    jump_cond(X, C),
    @vlabel(Jump, instr_label(Label, Inst)),
    outedges(Jump, Edges),
    @fresh_label(NewLabel),
    @new_vertex(instr_label(NewLabel : nil, Inst),
                Edges, Then),
    new_edge(Then, ThenEdge),
    new_edge(InLoop, InLoopEdge),
    @new_vertex(instr_label(Label, branch(cond(C), NewLabel)),
                ThenEdge : InLoopEdge : nil, Branch).
```

Figure 2.10: Code for the *irred_jump* method

Finally, we need to find the declaration of a local variable (in our case, we look for an integer variable) that occurs before the loop. We bind the node *Jump* to the node after the declaration.

$$
\begin{array}{l}
all\,(\{\,\}*;\\
\quad\{\,'def(X),\\
\quad\;\; X = expr_type(localvar(_), int(true, b32))\};\\
\quad\{\,\}\\
\quad)\quad(entry, Jump),\\
not\,(exists\,(\{\,\};\\
\qquad\qquad\{\,\}*\\
\qquad\qquad)\quad(Header, Jump))
\end{array}
$$

Now that we have found a loop and a suitable variable, we are ready to create the jump. We aim to replace the node *Jump* with a node *Branch* which is a conditional expression. If the test of the conditional is *True* then we perform the instructions that were at *Jump* but if the test is *False* then we jump to *InLoop*. We create an expression for the test using the predicate *jump_cond* — so $jump_cond(X, C)$ uses the variable X to bind C to a representation of the condition. For example, here is a predicate that creates $X \times X \geq 0$:

$$
\begin{array}{l}
jump_cond(X,\\
\quad expr_type(applyatom(\\
\qquad\qquad compare(geq(true)),\\
\qquad\qquad expr_type(applyatom(mul(false, true), X, X),\\
\qquad\qquad\qquad int(true, b32)),\\
\qquad\qquad expr_type(applyatom(ldc(int(true, b32), 0)),\\
\qquad\qquad\qquad int(true, b32))),\\
\qquad bool)).
\end{array}
$$

We must ensure that the predicate that we build evaluates to *True*.

Since we are replacing the node *Jump*, we need to know what label *Label*, instruction *Inst* and outgoing edges *Edges* are at *Jump*. We then create a new node *Then* (for when the condition is *True*) which has the label *NewLabel* and contains the instruction *Inst* and has outgoing edges *Edges*. Next we create two new incoming edges: *ThenEdge* and *InLoopEdge*. Finally, we can build the node *Branch*. This node has the same label as *Jump* and contains a conditional jump — if *True* we go to the node *Then* and if *False* we go to *InLoop*.

If we had nested loops then this specification would be able to produce only a jump from outside the outermost loop. We have to write many conditions just to find a loop — it would be convenient to use a language that includes loops.

2.4 Related Work

We briefly summarise some work related to the specifications of IL transformations.

2.4.1 Forms of IL

In Section 2.3.2 we outlined a form of IL that replaced stack expressions with assignments which we called EIL. As we discussed in Section 2.3.8, it would

be useful to have loops in our language and so we need to recover loops as well as expressions from IL. Baker [5] gives an algorithm for transforming a flow graph into a program which contains constructs such as **if**/**then**/**else** and **repeat** and Ramshaw [44] shows how many **goto** statements can be eliminated. An algorithm is given in [24] which eliminates **goto** statements by applying a sequence of transformations to move a **goto**. This is followed by a **goto**-eliminating transformation which introduces extra variables and predicates.

Extensions of EIL and PLP are discussed in [47] in which the specification language has a new primitive:

$$paths(B, P)$$

which states that all paths through B satisfy the pattern P and the language can match expressions of the form:

$$while(Cond, Body)$$

These extensions should allow the specification in Section 2.3.8 to be written more succinctly.

2.4.2 Transformation Systems

PLP uses the idea of *Universal Regular Path Queries* [14] in which transformations are expressed as logic predicates using regular expressions. The APTS system of Paige [41] was also a major source of inspiration. In APTS, program transformations are expressed as rewrite rules, with side conditions expressed as Boolean functions on the abstract syntax tree, and data obtained by program analyses. As mentioned in Section 2.3.5, the strategy language used in the toolkit was based on Stratego [55].

Some other transformation systems that provided inspiration for PLP are briefly described below.

- TyRuBa [17] is based on representing programs as logic propositions. The system is not really used for program transformations; instead it is used for the implementation of classes and interfaces.

- Gospel [56] is a specification language to express program-improving transformations. This needs a declaration of variables (which might be statements), a precondition consisting of a code pattern and dependencies and an action.

- TRANS [31] is a language in which temporal logic is used to specify transformations with rewrite rules. The specification language has constructors such as E (there exist a path) and A (for all paths) for expressing paths in a control flow graph.

2.4.3 Correctness

Throughout this chapter we have not been concerned with the correctness of the transformations, *i.e.* whether a transformation is behaviour-preserving. Proving the correctness of a transformation is a challenging task. In [33] a framework is provided for proving the correctness of compiler optimisations that are expressed

in temporal logic. A simple language consisting of read, write, assignments, conditionals and jumps (but not procedures or exceptions) is considered and a semantics for this language is given. A method for showing semantic equivalence is stated and used to prove the correctness of some optimisations. This method is quite complicated but seems suited to automation — a technique for automatically proving the soundness of optimisations is given in [34].

The proofs given in [33] are difficult since they require setting up complicated semantics and a proof is too detailed to be stated here. For instance, the proof of a code motion transformation takes one whole double column page. This method could be adapted for our obfuscations but would be a very demanding task.

2.5 Conclusions

We have seen that there is a need for obfuscating the .NET Intermediate Language and we have given a way of specifying obfuscations for IL. We have seen that by changing certain predicates in the specification we can specify more general obfuscations.

Proving the correctness of transformations requires considerable effort and theoretical techniques (such as developing semantics) which are detailed and complicated. The proofs also require simplifying the source language. Thus proving the correctness of our obfuscations will be a challenging task. Using IL also makes proofs difficult as it is a low-level language.

In the rest of the thesis, we will consider obfuscating abstract data-types. We will model operations as functional programs and we consider obfuscation as data refinement. We will find that using abstract data-types makes proving correctness easier and in some cases we will be able to derive obfuscated operations.

Part II

An Alternative View

Chapter 3

Techniques for Obfuscation

Klieg: "How did you know in the first place?"
Doctor: "Oh, I used my own special technique."
Klieg: "Oh really, Doctor? And may we know what that is?"
Doctor: "Keeping my eyes open and my mouth shut!"

Doctor Who — *The Tomb of the Cybermen* (1967)

3.1 Data-types

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

When creating obfuscations it is important to know various properties about our obfuscations. Most importantly we need to know that an obfuscation is correct thus ensuring that the obfuscation does not change the behaviour of a program. In imperative languages, proofs of correctness are frequently hard, typically requiring language restrictions. As a result, obfuscations are mostly stated without giving a proof of correctness. We also might want to know whether an obfuscation can be generalised. For instance can we apply the technique of splitting to data-types other than arrays? The conditions for the application of an imperative obfuscation can be quite complicated as we have to deal with features such as side-effects, exceptions and pointers.

For our framework, we model data-type operations using a functional language (see Section 3.1.2 for more details) and view obfuscation as data refinement [16]. Using these mathematical techniques allow us to prove the correctness of *all* our obfuscations and also for some obfuscations we will be able to derive an obfuscated operation from an unobfuscated one. Our framework also provides scope for generalisation of some of the standard obfuscations mentioned in Chapter 1. A benefit of generalisation is that we can introduce *randomness* into our obfuscations. Having random obfuscations means that different program executions produce different program traces (even with the same input)

which helps to confuse an attacker further. Random obfuscations appear to have received little attention to date.

The current view of obfuscation [10] (and Chapters 1 and 2) concentrates on concrete data structures such as variables and arrays. All of the data-types that we will use could be implemented concretely using arrays — for example, we can use the standard "double, double plus one" conversion [13, Chapter 6] to represent a binary tree as an array. Why, therefore, do we obfuscate the abstract data-type rather than its concrete implementation? Apart from providing a simple way of proving correctness, using data-types gives us an extra "level" in which to add obfuscations. Going immediately to arrays forces us to think in array terms and we would have only array obfuscations at our disposal. For instance, in Chapter 7, we consider a data-type for trees and so the usual tree transformations (such as swapping two subtrees) are naturally available; they would be more difficult to conceive using arrays. Also, converting a data-type to an array often loses information about the data-type (such as the structure) and so it would be difficult to perform operations that use or rely on knowledge of that structure. Some matrix operations rely on the 2-dimensional structure of matrices and so we would have difficulties defining such operations for matrices that have flattened to arrays — Section 6.3.5 gives an example considering the transposition operation. We may also gain new array obfuscations by obfuscating a data-type and then converting the data-type to an array. Thus, we have two opportunities for obfuscation; the first using our new data-type approach and the second using the standard imperative methods.

When we define a data-type, we will insist that the operations are total and deterministic. This restriction ensures that we can use equality in our correctness proofs. As the aim of obfuscation is to obscure the meaning of an operation we should give an indication of what an operation should do so that we can assess whether an obfuscation is suitable. Therefore when defining a data-type we require that we give assertions for the operations that we declare in our data-type. But we do not insist that we give a full axiomatisation of the data-type — we can just consider assertions that we are interested in. This requirement will aid us in producing our own definition of obfuscation. We could give an axiomatic definition of a data-type [36, Chapter 4] which provides axioms (*i.e.* laws involving the operations) but we choose to give a combination of both definitions and axioms.

One aim of our approach is to make the process of creating obfuscations more accessible and applicable. The techniques that we will use (such as data-types, refinement and functional programming) are well established in the programming research community and so they will be familiar to many computer scientists. Although the techniques of formal methods are well-known, the application of obfuscation using these techniques is not.

3.1.1 Notation

When defining a data-type, we want to specify the following things:

- a name for the data-type
- a declaration of the types and constructors (which can be given as a schema)

- data-type invariants (dti)
- a list of operations with their types
- a list of assertions that these operations satisfy

and we use the following notation when declaring a data-type:

name
declaration dti
operations: $f_1 :: T_1$ $f_2 :: T_2$ ... $f_n :: T_n$
assertions

which borrows features from modules in [39, Chapter 16] and Z schemas [50].

This notation acts as a "module" for the data-type D which means that only the operations defined in this data-type are "visible" to other data-types. We may define other operations that act on D but these cannot be used by another data-type. In we wish to use another data-type then we write the word "USING" in the name part. When defining a data-type, we insist that we state a list of assertions which are only in terms of the "visible" operations. If we have no particular invariant for the data-type then we omit this from the data-type declaration.

As an example, we show an array data-type for arrays of length N whose elements have type α in Figure 3.1. The operation $access(A, i)$ is usually written as $A[i]$ and $update(i, x, A)$ is written as $A[i] = x$. We can make some of the array values $Null$ (where $Null \notin \alpha$) which ensures that we can perform the operation $A[i]$ for all i in the range $[0..N)$. All the values of the array new are $Null$ (corresponding to **new** in object-oriented languages).

3.1.2 Modelling in Haskell

We now need to choose a language to model data-type operations. Since we aim for a more mathematical approach in which we want to prove correctness easily, we should use a language that reflects this approach. We choose to use the functional language Haskell [43] to specify operations and obfuscations (note that *we are not aiming to obfuscate Haskell code but to use Haskell as a modelling language*). Haskell is a suitable language as its mathematical flavour lends itself to derivational proofs thus allowing us to prove properties (such as correctness) of Haskell functions (see for example [7, Chapter 4] and [52, Chapter

Array (α)
$Array\ \alpha :: [0..N) \rightarrow \alpha \cup \{Null\}$
$access :: Array\ \alpha \times [0..N) \rightarrow \alpha \cup \{Null\}$ $update :: [0..N) \times (\alpha \cup \{Null\}) \times Array\ \alpha \rightarrow Array\ \alpha$ $new :: Array\ \alpha$
$access(update(i, x, A), i) = x$ $i \neq j \Rightarrow access(update(i, x, A), j) = access(A, j)$ $(\forall\ i :: [0..N)) \bullet access(new, i) = Null$

Figure 3.1: Data-type for Arrays

14]). As Haskell is a purely functional language there are no side-effects, which can complicate proofs. Since we using Haskell as a modelling language we should ensure that we can convert our operations in other languages. Thus we will not exploit all the characteristics of Haskell and in particular, we will use finite, well-defined data-types and we will not use laziness. We could have chosen a strict language such as ML but the syntax of ML and the presence of reference cells means that it is not as elegant a setting for proofs as Haskell.

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

When defining data-types, we usually give operations which act on the local state of the data-type and so they have type $D \rightarrow D$. We will insist that all operations are deterministic and are functions rather than relations (which mirrors the use of functional programming as our modelling language). This means that we can take a more mathematical view and define functions for abstract data-types. Also we will use functional composition $(\cdot)$ rather than sequential composition $(;)$.

Can we match this mathematical view up with the traditional data-type view? Consider the `length` function for lists which takes a list as input and returns a natural number as the output. Imperatively, using linked lists we

could define length as follows:

```
public static int length(list l)
{
  int s = 0;
  while (l != null)
  {   s = s + 1;
      l = l.next; }
  return s;
}
```

We can consider this to be an operation on the data-type of lists which leaves a list unchanged but gives a natural number as an output. This conversion from "function" to "operation" is valid for all our definitions and so, in our cases, these two notions are equivalent. Thus we will write *operation* or *function* to denote the actions that we allow on our data-types.

3.2 Obfuscation as data refinement

Suppose that we have a data-type D and we want to obfuscate it to obtain the data-type O. Obfuscating D involves giving definitions for the obfuscation of each of the operations defined in D and ensuring that they are correct. What does it mean to be correct and how can we prove correctness?

To provide a framework for obfuscating data-types (and establishing the correctness of the obfuscated operations) we view obfuscation as *data refinement* [16]. A refinement can be achieved by a relation R between an abstract and a concrete state:

$$R :: A \leftrightarrow C$$

that satisfies a simulation condition [16, Section 2.1]. A refinement is called *functional* if and only if there exists a data-type invariant dti and a function af called an *abstraction function* with type:

$$af :: C \rightarrow A$$

such that R has the form

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

If we have a functional refinement then each instance of the concrete state satisfying the data-type invariant is related to at most one instance of the abstract state. That corresponds to the concrete state having more "structure" than the abstract state. In general, when obfuscating we aim to obscure the data-type by adding more structure and so we propose that the obfuscated data-type O will be no more abstract than the original data-type D. Thus the most general form of refinement for us is functional refinement. This formulation allows us to have many obfuscations which can be "undone" by the same abstraction function (see Section 7.2.2 for an example using trees). We may have a situation where we obfuscate a data-type by first performing a (possibly non-functional) refinement and then obfuscating this refinement. As data refinement is a well-known technique, we will concentrate on just the obfuscation part of the refinement.

So, for obfuscation we require an abstraction function af with type

$$af :: O \rightarrow D$$

and a data-type invariant dti such that for elements $x :: D$ and $y :: O$

$$x \rightsquigarrow y \Leftrightarrow x = af(y) \wedge dti(y) \tag{3.1}$$

The term $x \rightsquigarrow y$ is read as "x is data refined by y" (or in our case, "...is obfuscated by...") which expresses how data-types are related.

For obfuscation the abstraction function acts as a "deobfuscation" and therefore it is important to keep this function secret from an attacker. In our situation, it turns out that af is a surjective function so that if we have an obfuscation function of

$$of :: D \rightarrow O$$

that satisfies

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

then

$$af \cdot of = id \tag{3.2}$$

Thus, of is a right-inverse for af. Note that it is not necessarily the case that

$$of \cdot af = id \tag{3.3}$$

since we could have another obfuscation function of' such that $of'(x) = y'$ and $x \rightsquigarrow y'$ and so we have that

$$of\ (af\ (y')) = of\ (x) = y$$

The abstraction function will have a left-inverse only if it is injective. In that case, each object of D will be refined by exactly one of O and we will call the inverse (which is both left and right) of af the *conversion function* (cf) for the obfuscation.

3.2.1 Homogeneous operations

Suppose that the operation f with type

$$f :: D \rightarrow D$$

is defined in D. Then to obfuscate f we want a operation f^O with type

$$f^O :: O \rightarrow O$$

which preserves the correctness of f. In terms of data refinement, we say that f^O is *correct* if it is satisfies:

$$(\forall x :: D; y :: O) \bullet x \rightsquigarrow y \Rightarrow f(x) \rightsquigarrow f^O(y) \tag{3.4}$$

If f^O is a correct refinement (obfuscation) of f then we write $f \leadsto f^O$. Using Equation (3.1), we can draw the following commuting diagram [16]:

$$\begin{array}{ccc} x & \xrightarrow{f} & f(x) \\ \uparrow{\scriptstyle af} & & \uparrow{\scriptstyle af} \\ y & \xrightarrow[f^O]{} & f^O(y) \end{array}$$

From this diagram, we have the following equation:

$$f \cdot af = af \cdot f^O \tag{3.5}$$

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

Now, suppose that af is a bijection with conversion function cf

$$cf :: D \to O$$

Then af and cf satisfy (3.2) and (3.3) and so $cf = af^{-1}$. Thus pre-composing Equation (3.5) by cf gives us

$$f^O = cf \cdot f \cdot af \tag{3.6}$$

which provides a way of deriving the operation f^O from f.

3.2.2 Non-homogeneous operations

Suppose that we have a operation

$$f :: D \to E$$

where D and E are the state spaces of two data-types. Let af_D and af_E be abstraction functions for some obfuscations of D and E. How do we define an obfuscation f^O of f? Using the abstraction functions, we can construct the following commuting diagram:

$$\begin{array}{ccc} x & \xrightarrow{f} & f(x) \\ \uparrow{\scriptstyle af_D} & & \uparrow{\scriptstyle af_E} \\ y & \xrightarrow[f^O]{} & f^O(y) \end{array}$$

and from this, we obtain

$$f \cdot af_D = af_E \cdot f^O \tag{3.7}$$

In Section 7.2.4, we define a operation `mktree` which takes a list and produces a tree. We can obfuscate this operation twice by obfuscating trees and obfuscating lists — see Appendix D for more details.

3.2.3 Operations using tuples

Let us define a function *cross* that applies a tuple of functions to a tuple of arguments

$$cross\ (f_1, f_2, \ldots, f_n)\ (x_1, x_2, \ldots, x_n) = (f_1\ x_1, f_2\ x_2, \ldots, f_n\ x_n)$$

Note that we cannot write this function in Haskell without writing different definition for each n since we are not able to define arbitrary tuples (also we cannot use a list as the functions may have different types). If we want to implement this function then we could use a dependently typed language such as Cayenne [4].

So, for an operation

$$f :: S_1 \times S_2 \times \ldots \times S_n \to T$$

an obfuscation f^O of f (with respect to the relevant abstraction functions) satisfies:

$$f \cdot cross\ (af_{S_1}, af_{S_2}, \ldots, af_{S_n}) = af_T \cdot f^O \tag{3.8}$$

If we have a conversion function cf_T for the data-type T such that:

$$cf_T \cdot af_T = id$$

then we can pre-compose (3.8) by cf_T to give

$$f^O = cf_T \cdot f \cdot cross\ (af_{S_1}, af_{S_2}, \ldots, af_{S_n}) \tag{3.9}$$

3.2.4 Refinement Notation

If the data-type D is refined by a data-type O then we write

New-Name *REFINEMENT* $[af]$: D
declaration dti
$\langle\!\lbrack f_1, f_2, \ldots, f_n \rbrack\!\rangle \leadsto \langle\!\lbrack g_1, g_2, \ldots, g_n \rbrack\!\rangle$ [additional operations]
[additional assertions]

The notation

$$\langle\!\lbrack f_1, f_2, \ldots, f_n \rbrack\!\rangle \leadsto \langle\!\lbrack g_1, g_2, \ldots, g_n \rbrack\!\rangle$$

is shorthand for:

$$f_1 \leadsto g_1 \wedge f_2 \leadsto g_2 \wedge \ldots \wedge f_n \leadsto g_n$$

When we state "REFINEMENT" in a data-type we assume that we have a functional refinement and so we give the abstraction function (af) for the refinement. We consider this refinement to be an *implementation* (analogous to the implementation of interfaces in object-oriented programming) rather than an *overriding*. This means that O still has access to the operations $f_1, \ldots, f_n$ as well as the refined operations $g_1, g_2, \ldots, g_n$. The ability to access the original operations will prove useful when defining obfuscations. The type of the refined operations can be inferred from the original operations and the set of assertions for O contains the set of assertions for $f_1, \ldots, f_n$ with the appropriate name changes. We can add extra operations to the refined data-type if we wish and if we do so, then we must give assertions for these extra operations. Section 4.3 gives an example of a data-type for lists and Section 4.3.1 shows how we can refine it. Note that as the list of assertions give away knowledge about a data-type we should ensure that we hide the assertions when implementing an obfuscated data-type.

3.3 What does it mean to be obfuscated?

In Section 1.2, we saw that if P' is an obfuscation of P then

- P' should have the same functionality as P.
- The running time and length of P' should be at most polynomially larger than that of P.
- P' should be "harder to understand" than P.

Since we are going to model our operations in Haskell many of the metrics for object-oriented programs mentioned in [10] are not directly applicable. Thus we must state when we consider an operation to be obfuscated (*i.e.* when an operation is "harder to understand").

Suppose that we have an operation g and we obfuscate it to obtain an operation g^O. Using cyclomatic complexity [37] and data-structure complexity [40] for inspiration, we could consider g^O to be more obfuscated than g if:

- g^O uses a more complicated data structure than g
- g^O has more guards in the definition than g

When considering what an operation does we often see what properties the operation has. If an operation is obfuscated then it should be harder to prove what properties it has. For our data-types we give a list of *assertions* — properties that the operations of the data-type satisfy. We should make sure that the definitions of each operation satisfy the assertions and so we will have to construct a proof for each assertion. If an operation is obfuscated then it should be harder to prove that the assertions are true — we will have to qualify what it means for a proof to be "harder".

Definition 1 (Assertion Obfuscation). Let g be an operation and $\mathcal{A}$ be an assertion that g satisfies. We obfuscate g to obtain g^O and let $\mathcal{A}^O$ be the assertion corresponding to $\mathcal{A}$ which g^O satisfies. The obfuscation O is said to be an *assertion obfuscation* if the proof that g^O satisfies $\mathcal{A}^O$ is more complicated than the proof that g satisfies $\mathcal{A}$.

Note that $\mathcal{A}^O$ is well-defined as we have restricted ourselves to functional refinements.

How can we compare proofs and decide whether one is more complicated than another? One way to have a fair comparison is to ensure that our proofs are minimal. This is not practical as we can never be certain if we have a minimal proof. Instead we should ensure that we prove assertions *consistently*. We will split the proof of an assertion into different cases (often determined by the guards in a definition). For example, for induction proofs, the base steps and the induction steps all constitute cases.

We will prove each case using the derivational style [18, Chapter 4]:

$$\begin{array}{ll} & \mathit{LHS} \\ = & \quad \{\text{result}\} \\ & \ldots \\ = & \quad \{\text{result}\} \\ & \mathit{RHS} \end{array}$$

We can replace "=" by other connectives such as "⇒" or "⇐".

Each stage of the derivation should use one *result*, where we consider a result to be

- the definition of an operation
- an arithmetic or Boolean law
- a property of an operation

If we want to use a property of an operation then we also need to also prove this property. We will take standard results from set theory and logic (such as natural number arithmetic and predicate calculus) as our laws.

How do we measure whether one proof is more complicated than another? One way could be to count the total number of results (including multiplicities) for all the cases of a proof. If an obfuscation increases the number of guards in the definition of an operation then a proof for this obfuscation may be longer as we will have more cases to prove. Therefore, to ensure that an operation is an assertion obfuscation we can just add in extra guards. So, if the definition of a operation $f :: \alpha \to \beta$ is

$$f\ x\ = g$$

then by using a predicate $p :: \alpha \to \mathbb{B}$, we could define

$$\begin{array}{l} f^O\ x \\ \quad \left| \begin{array}{ll} p(x) & = g \\ \text{otherwise} & = g \end{array} \right. \end{array}$$

The proof of an assertion for f^O will require two cases (one for $p(x)$ and one for $\neg p(x)$) and so to make an assertion obfuscation, we just have to insert a predicate. To prevent this, we require that each case in a definition gives rise to a different expression.

At each stage of a proof, we aim to use definitions and properties associated with the innermost operation. For instance, if we had an assertion

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

Then we first deal with h before we deal with g. This method will not always produce a minimal length proof but will help in making our proofs consistent. Where possible, we aim to use only one result at each stage so that we do not make proofs shorter by using many results in the same stage. We should also ensure that at each stage, we make progress towards our goal. In particular, we should not have any cycles in our proofs and so we insist that at each stage we do not have an expression that we have seen before (although we may use the same result many times).

For our definition of obfuscation, we will not be concerned with syntactic properties such as the name of operations, the layout of operations and whether a definition uses standard operations (for example, head or foldr). For example, we do not make distinctions between guarded equations and (nested) conditionals. So we consider the following two expressions to be equivalent:

$$\begin{array}{l} f\ x \\ \left|\begin{array}{ll} g_1 & = s_1 \\ g_2 & = s_2 \\ \ldots & \\ \text{otherwise} & = s_n \end{array}\right. \end{array} \quad \equiv \quad \begin{array}{l} f\ x = \text{if } g_1 \text{ then } s_1 \\ \qquad\qquad \text{else (if } g_2 \text{ then } s_2 \\ \qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad \text{else } s_n \cdots) \end{array}$$

The total number of results that a proof takes may not be a detailed enough measure. For instance, suppose we have two proofs: the first has one case and uses n results and the second has n cases each of which uses one result. Both of these proofs use n results in total but is one proof more complicated than the other? We propose that the first is more complicated as we claim that it is harder to do one long proof than many short proofs. Thus the number of results used is not a satisfactory measure — we need to consider the "shape" of a proof.

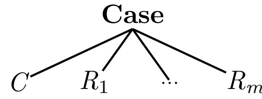
3.4 Proof Trees

How do we measure the shape of a proof? We will compare proofs by drawing proof trees — using trees gives an indication of the "shape" of a proof. Note that our style of proof lends itself to producing proof trees.

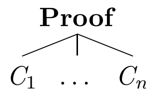
From the last section, we consider a proof to be a series of cases. Each case is proved using the derivational style where each stage is either a definition, a law or a property of an operation. This gives us a way of defining a grammar for proof trees:

$$\begin{array}{rcl} \textbf{Proof} & = & \textit{Seq}\ \textbf{Case} \\ \textbf{Case} & = & \textit{Stage}\ \textbf{Case}\ (\textit{Seq}\ \textbf{Result}) \mid \textit{End} \\ \textbf{Result} & = & \textit{Definition} \mid \textit{Law} \mid \textbf{Proof} \end{array}$$

Rather than use this grammar explicitly, we will draw out the proof trees instead. We draw *Stage* C $\langle\!\langle R_1, \ldots, R_m \rangle\!\rangle$ (where C is a case and R_i are results):



and $\langle\!\langle C_1, \ldots, C_n \rangle\!\rangle$ (where C_i are cases) as



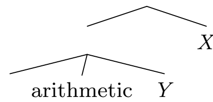
Let us now see how a derivation can be drawn as a proof tree. The derivation

$$\begin{array}{ll} & e_0 \\ = & \{\text{definition of } X\} \\ & e_1 \\ = & \{\text{arithmetic and definition of } Y\} \\ & e_2 \end{array}$$

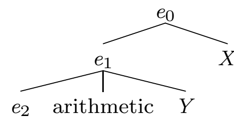
corresponds to the following expression:

$$\langle\!\langle \textit{Stage } (\textit{Stage End } \langle\!\langle \text{arithmetic}, \text{definition of } Y \rangle\!\rangle)\ \langle\!\langle \text{definition of } X \rangle\!\rangle \rangle\!\rangle$$

and can be drawn as follows:



To improve the readability of the trees, we can annotate each stage with a label if we wish, and so for the above example, we could draw



Let us now consider how to draw a tree for an induction proof. An induction consists of some base cases and some step cases. For our example, let us suppose that we have one base and two step cases.

Base Case

$$\begin{array}{ll} & b_0 \\ = & \{X_1\} \\ & \ldots \\ = & \{X_n\} \\ & b_n \end{array}$$

Step Case We set up an induction hypothesis (IH) and have two subcases.

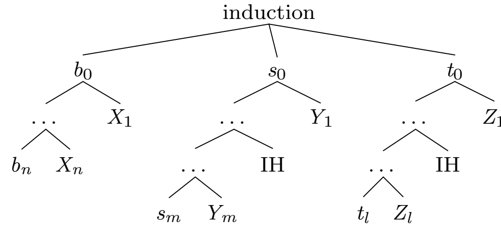
Subcase 1

$$
\begin{array}{ll}
 & s_0 \\
= & \{Y_1\} \\
 & \ldots \\
= & \{\text{IH}\} \\
 & .. \\
= & \{Y_m\} \\
 & s_m
\end{array}
$$

Subcase 2

$$
\begin{array}{ll}
 & t_0 \\
= & \{Z_1\} \\
 & \ldots \\
= & \{\text{IH}\} \\
 & .. \\
= & \{Z_l\} \\
 & t_l
\end{array}
$$

Note that we treat the use of IH as a result. This proof can be drawn as follows:



3.4.1 Measuring

For our comparison of proofs we will measure two things: *number of results* and *height*.

Since we are using sequences and not sets, we do not use $\in$ to define membership. Instead we write $x \leftarrow sq$ to mean the element x is "drawn from" the sequence sq (this notation matches list comprehension in Haskell).

We define $\mathcal{R}$, the number of results, as follows:

$$
\begin{array}{lr}
\text{[Case]} & \mathcal{R}\,(\mathit{Stage}\ C\ \mathit{Rs}) = \mathcal{R}\,(C) + \sum_{x \leftarrow \mathit{Rs}} \mathcal{R}\,(x) \\
\text{[Proof]} & \mathcal{R}\ (\mathit{Cs}) = \sum_{x \leftarrow \mathit{Cs}} \mathcal{R}\,(x) \\
\text{[Result]} & \mathcal{R}\,(\mathit{End}) = 0 \\
\text{[Otherwise]} & \mathcal{R}\,(_) = 1
\end{array}
$$

and the height $\mathcal{H}$ is defined as follows:

$$
\begin{array}{lr}
\text{[Case]} & \mathcal{H}\,(\mathit{Stage}\ C\ \mathit{Rs}) = 1 + \max\big(\mathcal{H}\,(C), (\max_{x \leftarrow \mathit{Rs}}\ \mathcal{H}\,(x))\big) \\
\text{[Proof]} & \mathcal{H}\,(\mathit{Cs}) = 1 + \max_{x \leftarrow \mathit{Cs}}\ \mathcal{H}\,(x) \\
\text{[Otherwise]} & \mathcal{H}(_) = 0
\end{array}
$$

where C is a case, Cs is a sequence of cases and Rs is a sequence of results.

Looking at the tree for the induction derivation, we can see that the proof uses $l+m+n$ results and the height of the proof tree is $1+\max(l, m, n)$ (where l, m and n are the number of results for each case).

3.4.2 Comparing proofs

As an example, we prove the following assertion:

$$(\mathsf{head}\ (x : xs)) : (\mathsf{tail}\ (x : xs)) = x : xs \tag{3.10}$$

for a finite list xs.

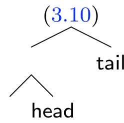
First we prove it using the standard definitions of head and tail:

$$\mathsf{head}\ (x : xs) = x$$
$$\mathsf{tail}\ (x : xs) = xs$$

The proof of (3.10) is as follows:

$$\begin{aligned}
& (\mathsf{head}\ (x : xs)) : (\mathsf{tail}\ (x : xs)) \\
= & \quad \{\text{definition of } \mathsf{tail}\} \\
& (\mathsf{head}\ (x : xs)) : xs \\
= & \quad \{\text{definition of } \mathsf{head}\} \\
& x : xs
\end{aligned}$$

and we can draw the following proof tree:



The height of this tree is 2 and uses 2 results.

Now suppose that we obfuscate head as follows:

$$\mathsf{head} = \mathsf{last.reverse}$$

where

$$\mathsf{reverse}\ [\,] = [\,]$$
$$\mathsf{reverse}\ (x : xs) = xs +\!\!+ [x]$$

and

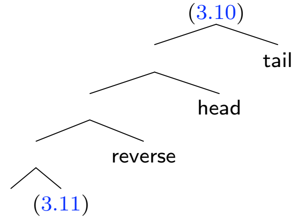
$$\mathsf{last}\ [x] = x$$
$$\mathsf{last}\ (x : xs) = \mathsf{last}\ xs$$

Note that this definition of last has overlapping cases (since the first case is a special case of the second) but since Haskell matches from top to bottom it is a valid definition. This alternate definition of head is much more expensive to compute (and we use an inefficient version of reverse) — we consider this definition only for demonstration purposes.

We prove (3.10) for this definition of head as follows:

$$\begin{aligned}
& (\mathsf{head}\ (x : xs)) : (\mathsf{tail}\ (x : xs)) \\
= & \quad \{\text{definition of } \mathsf{tail}\} \\
& (\mathsf{head}\ (x : xs)) : xs \\
= & \quad \{\text{definition of } \mathsf{head}\} \\
& (\mathsf{last}\ (\mathsf{reverse}\ (x : xs))) : xs \\
= & \quad \{\text{definition of } \mathsf{reverse}\} \\
& (\mathsf{last}\ (xs +\!\!+ [x])) : xs \\
= & \quad \{\text{property of } \mathsf{last}\ (3.11)\} \\
& x : xs
\end{aligned}$$

This derivation produces the following tree:



The height of this tree is $4 + \mathcal{H}\,(3.11)$ and the proof uses $3 + \mathcal{R}\,(3.11)$ results.

To complete the proof, we need to prove that for a finite list xs:

$$\mathsf{last}\ (xs +\!\!+ [x]) = x \tag{3.11}$$

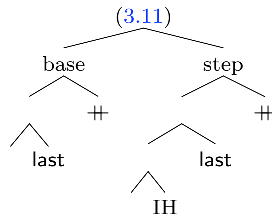
We prove this by induction on xs

Base Case Suppose that $xs = [\,]$, then

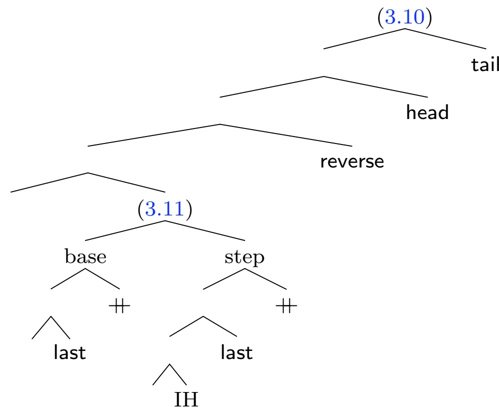
$$\begin{array}{ll} & \mathsf{last}\ ([\,] +\!\!+ [x]) \\ = & \quad \{\text{definition of } +\!\!+\} \\ & \mathsf{last}\ [x] \\ = & \quad \{\text{definition of } \mathsf{last}\} \\ & x \end{array}$$

Step Case Let $xs = y : ys$ and for the induction hypothesis, we suppose that ys satisfies (3.11). Then

$$\begin{array}{ll} & \mathsf{last}\ ((y : ys) +\!\!+ [x]) \\ = & \quad \{\text{definition of } +\!\!+\} \\ & \mathsf{last}\ (y : (ys +\!\!+ [x])) \\ = & \quad \{\text{definition of } \mathsf{last}\} \\ & \mathsf{last}\ (ys +\!\!+ [x]) \\ = & \quad \{\text{induction hypothesis}\} \\ & x \end{array}$$



This tree has a height of 4 and the proof uses 5 results. To obtain the full proof tree, we plug in this induction proof giving:



The height of this tree is 8 and the proof uses 8 results — both are 4 times more than for the standard definition of head and so we can conclude that our obfuscation of head is an assertion obfuscation.

3.4.3 Changing the measurements

In our simple head example, measuring the height and the number of results gave us a simple way of comparing the two proofs. In the second proof, we used a subsidiary result — Equation (3.11) — in the proof. To work out the height and number of results of the whole proof tree we simply plugged in the tree for (3.11) at the appropriate place. Is this a reasonable approach to take?

If we had used our subsidiary result more than once then using this approach of plugging in proof tree means that we would count the number of results that the result uses twice. Surely if we prove a result once then we do not need to prove it again! Once we prove a result then using this result again should not add to the "difficulty" of a proof. But the measure of the number of results does not reflect this observation.

We could decide that we are not allowed to use results that we have proved before but this is not what we do when proving a property. Instead if we use a proved result more than once then we have to declare it as a lemma (with an appropriate name). We make a slight change to the grammar for proofs as follows:

$$\textbf{Result} \quad = \quad \textit{Definition} \mid \textit{Law} \mid \textit{Lemma} \mid \textbf{Proof}$$

For a proof P, we define $\mathcal{L}(P)$ to be the set of lemmas for P (and so we ignore multiplicities).

Instead of computing the number of results used, we measure the *cost* of a proof. To calculate the cost, we add together the number of results used in a proof tree (where each lemma has cost 1) with the sum of the cost of each lemma used. Thus the cost $\mathcal{C}$ is defined to be:

$$\mathcal{C}\,(P) = \mathcal{R}\,(P) + \sum_{x \in \mathcal{L}(P)} \mathcal{C}\,(x)$$

Note that if for a proof P we have that $\mathcal{L}(P) = \emptyset$ then $\mathcal{C}(P) = \mathcal{R}(P)$. We leave the height measure unchanged as we already only use the height of a lemma once.

3.4.4 Comments on the definition

How useful is our definition? Does this definition give a good indication of the degree of obfuscation?

We use the definition with different main data-types — see Section 5.4 and Appendices A, B and D for examples of proof trees. Here are some comments about this definition:

- It is often hard to prove properties in a consistent way — which is essential for a fair comparison.
- To fully compare how well an operation is obfuscated, we have to prove all assertions related to that operation. We may find that an obfuscation is better for one assertion than another — how do we decide on the degree of obfuscation in such a case?
- To fully reflect on the effectiveness of an obfuscation on a data-type we have to consider all the operations. Therefore we should obfuscate according to the operations and not just the data structure.
- Since we should consider all the assertions for each data-type, is there a way of finding a minimal set of assertions?
- Every time we use a standard result in a proof, we have to prove that result as well. Declaring such a result as a lemma minimizes the amount of effort required for our proofs.
- Constructing proof trees is quite easy for our style of proof — will it be as easy for other styles?
- In Section 5.4 we consider an assertion for a particular set operation. We find that a better obfuscation is achieved if the obfuscated operation does not have a similar structure to the unobfuscated operation.
- There is a trade-off between the degree of obfuscation and the computational complexity and so we should consider the complexity as well as the degree of obfuscation.

3.5 Folds

Since we are using functional programming we could use folds in our definitions and proofs. In this section, we briefly look at some properties of folds and unfolds. Folds are commonly known in functional programming — in particular, in the Haskell prelude, foldr is defined as follows:

$$\begin{aligned}
&\mathsf{foldr} && :: (\alpha \to \beta \to \beta) \to \beta \to \mathit{List}\ \alpha \to \beta \\
&\mathsf{foldr}\ f\ e\ [\,] && = e \\
&\mathsf{foldr}\ f\ e\ (x : xs) && = f\ x\ (\mathsf{foldr}\ f\ e\ xs)
\end{aligned}$$

Unfolds are less commonly known than folds and for unfoldr we follow the definition given in [28]:

$$\begin{array}{l}
\mathsf{unfoldr} :: (\alpha \to \mathbb{B}) \to (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to \mathit{List}\ \beta \\
\mathsf{unfoldr}\ p\ f\ g\ x \\
\quad \left|\ \begin{array}{ll} p\ x & = [\,] \\ \text{otherwise} & = (f\ x) : (\mathsf{unfoldr}\ p\ f\ g\ (g\ x)) \end{array}\right.
\end{array}$$

A fold consumes a list to produce a new data-structure whilst an unfold generates a list from another data-structure. If we obfuscate a list then, in general, a conversion function can be written as a fold and the corresponding abstraction function can be written as an unfold.

The operations of foldr and unfoldr satisfy various properties which we state without proof:

Property 1 (Fold Fusion). For a function f

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

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

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

Property 2 (Unfold Fusion). For a function f,

$$(\mathsf{unfoldr}\ p\ g\ h) \cdot f = \mathsf{unfoldr}\ p'\ g'\ h'$$

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

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

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

In general, we can use these properties in derivations and proofs. We will tend not to use folds and unfolds for definitions of operations (except for matrix operations). This is because using folds often restricts the definition of an operation (*i.e.* using a fold means that an operation can only be written in one way) and thus restricts the opportunities for obfuscation. Also, we do not use folds and unfolds in proofs as we want to prove results in a consistent way and any rules (such as fusion) will have to be proved and declared as lemmas. This means that a standard derivational proof is often shorter — see [21] for a discussion of using folds to show the correctness of an operation. In [19], we derive some operations using the fusion theorem and in Section C, we prove that transpose is an involution by using fold fusion.

3.6 Example Data-Types

In the next four chapters, to show the generality of our approach we give some case studies in which we explore obfuscating some different data-types. In Chapter 4, we generalise the array split discussed in Section 1.3.6 and we give some example splits for lists. In Chapters 5 and 6 we apply splitting to sets and matrices. Finally, in Chapter 7, we develop a new obfuscation for trees.

Part III

Data-Type Case Studies

Chapter 4

Splitting Headaches

> And all dared to brave unknown terrors, to do mighty deeds, to boldly split infinitives that no man had split before.
>
> *The Hitchhiker's Guide to the Galaxy* (1979)

In this chapter, we generalise the array-split obfuscation that we discussed in Section 1.3.6 and, using this obfuscation, we show how to produce different splits for lists.

4.1 Generalising Splitting

We aim to generalise the array-split obfuscation and we want to specify splitting as a refinement (Section 3.2). The aim of a split is to break up an object t of a particular data-type T into n smaller objects (which we normally refer to as the *split components*) $t_0, t_1, \ldots, t_{n-1}$ of type T by using a so-called split sp:

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

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

4.1.1 Indexed Data-Types

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

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

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

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

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

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

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

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

We can define an indexer I_t for t as follows:

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

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

Let us look at some example IDTs:

- For arrays, $I_{array} = \mathbb{N}$ and

  $$\mathbb{A}_{array}\,(A, I) = X \Leftrightarrow A[I] = X$$

- For finite Haskell lists, the indexer I_{list} is $\mathbb{N}$ and the access function is usually written !!.

- For sequences in Z [50, Section 4.5], the indexer is $\mathbb{N}$ and the access function is written as functional application, *i.e.*

  $$\mathbb{A}_{seq}\,(s, n) = a \Leftrightarrow s\,n = a$$

- If $\mathbf{M}^{r \times c}$ is the set of matrices with r rows and c columns, then $I_{\mathbf{M}^{r\times c}} = [0..r) \times [0..c)$ and we usually write $\mathbf{M}(i, j)$ to denote the access function.

- For binary trees which have type:

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

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

4.1.2 Defining a split

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

Hence

$$
\begin{array}{rl}
t \rightsquigarrow & \langle t_0, t_1, \ldots, t_{n-1}\rangle_{sp} \\
\Leftrightarrow & \\
& ch :: I_t \to [0..n) \\
\wedge & \\
& (\forall\, f_k \in \mathcal{F}) \bullet f_k :: ch^{-1}(k) \rightarrowtail I_{t_k} \\
\wedge & \\
& \mathbb{A}_T(t_k, f_k(i)) = \mathbb{A}_T(t, i) \text{ where } ch(i) = k
\end{array}
\tag{4.1}
$$

We will call Equation (4.1) the *split relationship*.

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

$$(\forall\, a \leftarrow t) \bullet \mathsf{freq}(a, t) \leq \sum_{e \in [0..n)} \mathsf{freq}(a, t_e) \tag{4.2}$$

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

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

4.1.3 Example Splits

We now give two examples which will be used with some of our data-types. For these splits, we assume that we have an IDT $T(\alpha)$ which has an access function $\mathbb{A}_T$ and for each $t :: T(\alpha)$, the indexer $I_t \subseteq \mathbb{Z}$.

The first split we shall look at is called the *alternating split*, written as *asp*. This will split an element of an IDT into two — the first contains the elements with an even index and the second contains the rest.

The choice function is defined as

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

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

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

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

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

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

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

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

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

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

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

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

We can easily see that each f_k is injective.

We can define an access function for this split as follows:

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

4.1.4 Splits and Operations

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

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

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

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

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

$$h :: X^p \rightarrow Y$$

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

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

and for all $i \in I_x$

$$\mathbb{A}_D(y, i) = h(\mathbb{A}_D(x^1, \phi^1(i)), \ldots, \mathbb{A}_D(x^p, \phi^p(i))) \tag{4.3}$$

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

Theorem 1. Function Splitting Theorem
Suppose that the elements $x^1, \ldots, x^p$ of a data-type $D(X)$ are split for some split $sp = (ch, \mathcal{F})$. Let g be an operation

$$g :: D(X)^p \rightarrow D(Y)$$

with element y and functions h and $\{\phi^e\}_e$ as above in Equation (4.3). If there exists a family of functions $\{\theta^e\}_{e::[1..p]}$, such that for each e:

$$\theta^e \cdot ch = ch \cdot \phi^e \tag{4.4}$$

and if each ϕ^e satisfies:

$$\phi^e(f_k\,(i)) = f_{\theta^e(k)}\,(\phi^e(i)) \tag{4.5}$$

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

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

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

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

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

$$\begin{array}{cl} & \mathbb{A}_D(y_k, f_k(i)) \\ = & \quad \{\text{split relationship (4.1)}\} \\ & \mathbb{A}_D(y, i) \\ = & \quad \{\text{Equation (4.3)}\} \\ & h\,(\mathbb{A}_D(x^1, \phi^1(i)), \ldots, \mathbb{A}_D(x^p, \phi^p(i))) \\ = & \quad \{\text{split relationship (4.1) with } k_e = ch(\phi^e(i))\} \\ & h\,(\mathbb{A}_D(x^1_{k_1}, f_{k_1}(\phi^1(i))), \ldots, \mathbb{A}_D(x^p_{k_p}, f_{k_p}(\phi^p(i)))) \\ = & \quad \{\text{Property (4.4)},\ k_e = ch(\phi^e(i)) = \theta^e(ch(i)) = \theta^e(k)\} \\ & h\,(\mathbb{A}_D(x^1_{\theta^1(k)}, f_{\theta^1(k)}\,(\phi^1(i))), \ldots, \mathbb{A}_D\ (x^p_{\theta^p(k)}, f_{\theta^p(k)}\,(\phi^p(i)))) \\ = & \quad \{\text{Property (4.5)}\} \\ & h\,(\mathbb{A}_D(x^1_{\theta^1(k)}, \phi^1(f_k\ (i))), \ldots, \mathbb{A}_D(x^p_{\theta^p(k)}, \phi^p(f_k\ (i)))) \\ = & \quad \{\text{definition of } g \text{ in terms of } h \text{ and } \phi^e, \text{ Equation (4.3)}\} \\ & \mathbb{A}_D(g(x^1_{\theta^p(k)}, \ldots, x^p_{\theta^p(k)}), f_k(i)) \end{array}$$

Thus

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

□

Let us consider an operation F which has type

$$F :: (X \to Y) \to D(X) \to D(Y)$$

which is defined as follows:

$$\mathbb{A}_D(F\ f\ x, i) = f\,(\mathbb{A}_D(x, i)) \tag{4.6}$$

where $x :: D(X)$ and $i \in I_D$.

If we take a function $f :: X \to Y$ then the function $(F\ f)$ satisfies the Function Splitting Theorem where $h = f$ and $\phi^1 = id$. So, for any split $sp = (ch, \mathcal{F})$, if we let $\theta^1 = id$ then

$$\begin{array}{rcl} \theta^1 \cdot ch & = & id \cdot ch \\ & = & ch \cdot id \\ & = & ch \cdot \phi^1 \end{array}$$

Thus Equation (4.4) is satisfied. Also, if $k = ch(i)$ and $f_k \in \mathcal{F}$ then

$$\begin{aligned} f_{\theta^1(k)}(\phi^1(i)) &= f_k(i) \\ &= \phi^1(f_k(i)) \end{aligned}$$

and so Equation (4.5) is satisfied. So, we can define

$$F_{sp}\ f\ \langle x_0, \ldots, x_{n-1}\rangle_{sp} = \langle F\ f\ x_0, \ldots, F\ f\ x_{n-1}\rangle_{sp}$$

Let $x :: D(X)$ then for some $i \in I_D$

$$\begin{aligned} \mathbb{A}_D(F\ id\ x, i) &= id\,(\mathbb{A}_D(x,i)) \\ &= \mathbb{A}_D(x,i) \end{aligned}$$

and so

$$F\ id = id \tag{4.7}$$

Now let $f :: Y \to Z$ and $g :: X \to Y$. Then for any $x :: D(X)$ and for some $i \in I_D$

$$\begin{aligned} \mathbb{A}_D(F\ (f \cdot g)\ x, i) &= (f \cdot g)\,(\mathbb{A}_D(x,i)) \\ &= f\,(g\,(\mathbb{A}_D(x,i))) \\ &= f\,(\mathbb{A}_D\,(F\ g\ x, i)) \\ &= \mathbb{A}_D\,(F\ f\ (F\ g\ x), i) \\ &= \mathbb{A}_D\,((F\ f \cdot F\ g)\ x, i) \end{aligned}$$

and so

$$F\ (f \cdot g) = (F\ f) \cdot (F\ g) \tag{4.8}$$

The two equations (4.7) and (4.8) are called *functor laws* and so we can conclude that F is a functor for the data-type D.

4.2 Arrays

We need to consider whether the array splitting discussed in Section 1.3.6 matches up with our notion of data-type splitting — we defined a data-type for arrays in Section 3.1.1. Let us suppose that we want to split an array A into n smaller arrays $A_0, A_1, \ldots, A_{n-1}$. If we have

$$A \rightsquigarrow \langle A_0, A_1, \ldots, A_{n-1}\rangle_{sp}$$

then we need to define ch and $\mathcal{F}$ where $sp = (ch, \mathcal{F})$.

The relationship between A and the split components is:

$$A[i] = A_k[f_k(i)] \text{ where } k = ch(i). \tag{4.9}$$

This equation shows us how both `access` and `update` can be defined for a split. Note that with arrays, we can have the situation where

$$A \rightsquigarrow \langle A_0, A_1, \ldots, A_{n-1}\rangle_{sp}$$

but

$$|A| \leq \sum_{k=0}^{n-1} |A_k|$$

Strict inequality could occur if we had `Null` elements in some of the split components.

4.2.1 Array-Split properties

In Section 4.1.3 we defined two splits: *asp* and $b(k)$. Can these be used for arrays? Yes, in fact, we can use exactly the same definitions as before and the alternating split matches the array-split defined in Equation (1.4).

When using an array, we may have properties of the array which we might want to preserve. Two such properties are *monotonicity* and *completeness*. An array A is monotone increasing if

$$(\forall x :: \mathbf{N}; y :: \mathbf{N}) \bullet (0 \leq x < y < |A|) \Rightarrow A[x] \leq A[y]$$

We can define monotone decreasing similarly and we define strictly monotone increasing (decreasing) by replacing $\leq$ ($\geq$) in the consequent by $<$ ($>$). Monotonic arrays are sufficient for binary searches.

We say an array A is complete, if

$$(\forall i :: \mathbf{N}) \;\bullet\; (0 \leq i < |A|) \Rightarrow A[i] \neq \mathsf{Null}$$

We now state a result (without proof) about the splits *asp* and $b(k)$.

Suppose for an array A that

$$A \rightsquigarrow \langle P_0, P_1 \rangle_{asp}$$

and

$$A \rightsquigarrow \langle Q_0, Q_1 \rangle_{b(k)}$$

Then (i) if A is monotone increasing (decreasing) then P_0, P_1, Q_0, Q_1 are all monotone increasing (decreasing) and (ii) if A is complete then so are P_0, P_1, Q_0, Q_1.

Note that the converse of (i) is not always true — consider

$$\begin{array}{rcl} [3,2,5,6] & \rightsquigarrow & \langle [3,5],[2,6] \rangle_{asp} \\ \text{and } [3,5,2,6] & \rightsquigarrow & \langle [3,5],[2,6] \rangle_{b_2} \end{array}$$

The arrays $[3,5]$ and $[2,6]$ are both monotone increasing but the arrays $[3,2,5,6]$ and $[3,5,2,6]$ are not.

It is not always the case that splits preserve completeness and monotonicity. For example, let us consider the following array-split. Let us suppose that we have an array of length N and let $sp = (ch, \mathcal{F})$ with

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

and $\mathcal{F} = \{f_0, f_1\}$ where

$$\begin{array}{rl} f_0 & = (\lambda i . (N - i) \operatorname{div} 2) \\ \text{and } f_1 & = (\lambda i . (i \operatorname{div} 2) + 1) \end{array}$$

With this split, taking $N = 6$:

$$[0,1,2,3,4,5] \rightsquigarrow \langle [\mathsf{Null},4,2,0],[\mathsf{Null},1,3,5] \rangle_{sp}$$

and so this split does not preserve completeness or monotonicity.

List (α)

$List\ \alpha ::= Empty \mid Cons\ \alpha\ (List\ \alpha)$

$(:) :: \alpha \to List\ \alpha \to List\ \alpha$
$|__| :: List\ \alpha \to \mathbb{N}$
$\mathsf{null} :: List\ \alpha \to \mathbb{B}$
$\mathsf{head} :: List\ \alpha \to \alpha$
$\mathsf{tail} :: List\ \alpha \to List\ \alpha$
$\mathsf{elem} :: \alpha \to List\ \alpha \to \mathbb{B}$
$(+\!\!+) :: List\ \alpha \to List\ \alpha \to List\ \alpha$
$\mathsf{map} :: (\alpha \to \beta) \to List\ \alpha \to List\ \beta$
$\mathsf{filter} :: (\alpha \to \mathbb{B}) \to List\ \alpha \to List\ \alpha$
$(!!) :: List\ \alpha \to \mathbb{N} \to \alpha$

$$
\begin{aligned}
\mathsf{not}(\mathsf{null}\ xs) \Rightarrow (\mathsf{head}\ xs) : (\mathsf{tail}\ xs) &= xs \\
|xs +\!\!+ ys| &= |xs| +\!\!+ |ys| \\
xs\ !!\ 0 &= \mathsf{head}\ xs \\
xs\ !!\ (n+1) &= (\mathsf{tail}\ xs)\ !!\ n \\
|xs| = 0 &\Leftrightarrow \mathsf{null}\ xs \\
\mathsf{elem}\ x\ [\ ] &= False \\
\mathsf{elem}\ x\ (xs +\!\!+ ys) &= \mathsf{elem}\ x\ xs \vee \mathsf{elem}\ x\ ys \\
\mathsf{elem}\ x\ (\mathsf{filter}\ p\ xs) &= \mathsf{elem}\ x\ xs \wedge p\ x \\
\mathsf{map}\ (f \cdot g) &= \mathsf{map}\ f \cdot \mathsf{map}\ g
\end{aligned}
$$

Figure 4.1: Data-type for Lists

4.3 Lists

Our data-type for finite lists is given in Figure 4.1 and is based on the list data-type in Haskell. We will use the normal Haskell shorthand for list viz. for *Empty* we write $[\,]$ and we write *Cons* 1 (*Cons* 2 *Empty*) as $1 : (2 : [\,])$ or just $[1, 2]$ and so

$$x : xs \equiv Cons\ x\ xs$$

We will write $|xs|$ instead of $\mathsf{length}\ xs$. Haskell lists form an IDT where the access function is denoted by !! and the indexer for a list xs is $[0..|xs| - 1]$.

We use the definitions of the operations given in [7, Chapter 4]:

- length of a list

$$\begin{aligned} |[\,]| &= 0 \\ |(x : xs)| &= 1 + |xs| \end{aligned}$$

- test for empty list

$$\mathsf{null}\ xs = (xs == [\,])$$

- return first element

$$\mathsf{head}\ (x : xs) = x$$

- return all but the first element

$$\mathsf{tail}\ (x : xs) = xs$$

- test whether an element is contained within a list

$$\begin{aligned} \mathsf{elem}\ a\ [\,] &= False \\ \mathsf{elem}\ a\ (x : xs) &= (x == a) \vee (\mathsf{elem}\ a\ xs) \end{aligned}$$

- join two lists together (concatenate)

$$\begin{aligned} [\,] +\!\!+\ ys &= ys \\ (x : xs) +\!\!+\ ys &= x : (xs +\!\!+\ ys) \end{aligned}$$

- apply a function to every element in a list

$$\begin{aligned} \mathsf{map}\ f\ [\,] &= [\,] \\ \mathsf{map}\ f\ (x : xs) &= f\ x : (\mathsf{map}\ f\ xs) \end{aligned}$$

- filter a list using a Boolean function

$$\begin{aligned} \mathsf{filter}\ p\ [\,] &= [\,] \\ \mathsf{filter}\ p\ (x : xs) &= \text{if } p\ x \text{ then } x : (\mathsf{filter}\ p\ xs) \\ &\qquad \text{else } \mathsf{filter}\ p\ xs \end{aligned}$$

- list indexing

$$\begin{aligned} (x : xs)\ !!\ 0 &= x \\ (x : xs)\ !!\ (n + 1) &= xs\ !!\ n \end{aligned}$$

Note that head and tail are defined for non-empty finite lists while the rest are defined for all finite lists.

$$
\begin{array}{|l}
\hline
\textsf{SpList}\ (\alpha) \\
\quad \textit{REFINEMENT}\ [\textsf{unsplit}] : \quad \textsf{List}\ (\alpha) \\
\hline
\textit{SpList}\ \alpha ::= \langle \textit{List}\ \alpha, \textit{List}\ \alpha \rangle_{sp} \wedge \textit{dti} \\
\hline
\quad \langle\!\langle [(:), |__|, \textsf{null}, \textsf{head}, \textsf{tail}, \\
\qquad \textsf{elem}, (+\!\!+), \textsf{map}, \textsf{filter}, (!!)] \rangle\!\rangle \\
\leadsto \langle\!\langle [\textsf{cons}_{sp}, |__|_{sp}, \textsf{null}_{sp}, \textsf{head}_{sp}, \textsf{tail}_{sp}, \\
\qquad \textsf{elem}_{sp}, (+\!\!+)_{sp}, \textsf{map}_{sp}, \textsf{filter}_{sp}, (!!)_{sp}] \rangle\!\rangle \\
\hline
\\
\hline
\end{array}
$$

Figure 4.2: Data-type for Split Lists

4.3.1 List Splitting

Suppose that we have a split *sp* that splits a list into two split components. The definition for a data-type for split lists is given in Figure 4.2. Note that the invariant *dti* will be defined individually for each split.

As the list indexing operation (!!) is expensive to use, for Haskell list splitting, we will define two functions split (the *splitting function*) and unsplit (which is the inverse of split), such that

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

and

$$\textsf{unsplit}_{sp} \ \cdot \ \textsf{split}_{sp} = id \tag{4.10}$$

In Section 5.3.1 we work with sets which are represented by ordered lists and we will need to know whether a split preserves ordering — *i.e.* 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:

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

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 always sublists of xs then we know that the split has preserved the ordering.

Since lists are IDTs can we use the Function Splitting Theorem? For a list operation op, we would like to find a function h and a family of functions $\{\phi_i\}$ such that, for all n,

$$(\textsf{op}\ (xs_1, xs_2, \ldots, xs_p))\ !!\ n = h\,(xs_1\ !!\ (\phi^1(n)), \ldots, xs_p\ !!\ (\phi^p(n)))$$

If we take $h = id$ and $\phi^1 = \lambda i.i + 1$ then

$$(\mathsf{tail}\ xs)\ !!\ n = h\,(xs\ !!\ (\phi^1(n)))$$

Note that

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

and so map satisfies Equation (4.6). Thus for any split sp

$$\mathsf{map}_{sp}\ f\ \langle x_0, .., x_{n-1}\rangle_{sp} = \langle \mathsf{map}\ f\ \ x_0, \ldots, \mathsf{map}\ f\ \ x_{n-1}\rangle_{sp}$$

Now let us consider how we can split lists using the two example splits in Section 4.1.3.

4.3.2 Alternating Split

We now define the alternating split for lists. For example, we would like:

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

Using the definitions of ch and $\mathcal{F}$ from Section 4.1.3, if

$$xs \rightsquigarrow \langle l, r\rangle_{asp}$$

then

$$xs\ !!\ n = \begin{cases} l\ !!\ (n\ \mathrm{div}\ 2) & \text{if } n \bmod 2 = 0 \\ r\ !!\ (n\ \mathrm{div}\ 2) & \text{otherwise} \end{cases}$$

where $n < |xs|$.

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

$$dti \equiv |r| \leq |l| \leq |r| + 1 \tag{4.11}$$

We will strengthen this invariant in Section 5.3.1 when we work with ordered lists. As this split does not introduce any extra elements into the split components, we can strengthen Equation (4.2). So if $xs :: List\ \alpha$ and $xs \rightsquigarrow \langle l, r\rangle_{asp}$ then

$$(\forall x \leftarrow \alpha) \bullet \mathsf{freq}(x, xs) = \mathsf{freq}(x, l) + \mathsf{freq}(x, r) \tag{4.12}$$

From Equation (4.12), it is immediate that

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

We define a splitting function by stating cases for *Empty* and *Cons* as follows:

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

and here is the inverse:

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

Both of these operations take time proportional to $|l| + |r|$ $(=|xs|)$. We can easily check that Equation (4.10) is true and we also have that:

$$\mathsf{split}_{asp} \cdot \mathsf{unsplit}_{asp} = id \tag{4.13}$$

To ensure that our definition of split_{asp} matches the $(ch, \mathcal{F})$ formulation, we prove the following.

Correctness Proof 1 (Splitting function for the alternating split). If

$$\mathsf{split}_{asp}\ xs = \langle l, r\rangle_{asp}$$

then $(\forall\ n :: \mathbb{N}) \bullet\ n < |xs|$

$$xs\,!!\,n = \begin{cases} l\,!!\,(n \text{ div } 2) & \text{if } n \text{ is even} \\ r\,!!\,(n \text{ div } 2) & \text{otherwise} \end{cases} \tag{4.14}$$

where xs is a finite list.

Proof. We prove this by structural induction on xs.

Base Case 1 Suppose that $xs = [\,]$ then, by the definition of split_{asp}

$$\mathsf{split}_{asp}\ [\,] = \langle [\,], [\,]\rangle_{asp}$$

So there are no possible values of n and so Equation (4.14) is vacuously true.

Base Case 2 Suppose that $xs = [x]$ then, by the definition of split_{asp},

$$\mathsf{split}_{asp}\ [x] = \langle [x], [\,]\rangle_{asp}$$

The only possible value for n is zero and so Equation (4.14) is easily satisfied.

Step Case Suppose that $xs = x : y : ys$. If we let

$$\langle l, r\rangle_{asp} = \mathsf{split}_{asp}\ ys$$

then using the definition of split_{asp} twice,

$$\mathsf{split}_{asp}\ (x : y : ys) = \langle x : l, y : r\rangle_{asp}$$

For an induction hypothesis, suppose that ys satisfies Equation (4.14) for all natural numbers less than n.

Subcase 1 Suppose that $n = 0$. Then

$$(x : y : ys)\,!!\,0 = x = (x : l)\,!!\,0$$

and so Equation (4.14) is satisfied.

Subcase 2 Suppose that $n = 1$. Then

$$(x : y : ys)\,!!\,1 = y = (y : r)\,!!\,(1 \text{ div } 2)$$

and so Equation (4.14) is satisfied.

Subcase 3 Suppose that $n > 1$. Then

$$\begin{array}{ll}
 & (x : y : ys) \,!!\, n \\
= & \quad \{\text{property of } !!\} \\
 & (y : ys) \,!!\, (n-1) \\
= & \quad \{\text{property of } !!\} \\
 & ys \,!!\, (n-2) \\
= & \quad \{\text{induction hypothesis}\} \\
 & \begin{cases} l \,!!\, ((n-2) \text{ div } 2) & \text{if } n-2 \text{ is even} \\ r \,!!\, ((n-2) \text{ div } 2) & \text{otherwise} \end{cases} \\
= & \quad \{\text{arithmetic}\} \\
 & \begin{cases} l \,!!\, ((n \text{ div } 2) - 1) & \text{if } n \text{ is even} \\ r \,!!\, ((n \text{ div } 2) - 1) & \text{otherwise} \end{cases} \\
= & \quad \{\text{property of } !!\} \\
 & \begin{cases} (x : l) \,!!\, (n \text{ div } 2) & \text{if } n \text{ is even} \\ (y : r) \,!!\, (n \text{ div } 2) & \text{otherwise} \end{cases}
\end{array}$$

So by induction, Equation (4.14) is satisfied. □

We can consider $\mathsf{unsplit}_{asp}$ to be an abstraction function, so we can write

$$xs \leadsto \langle l, r \rangle_{asp} \Leftrightarrow xs = \mathsf{unsplit}_{asp}\ \langle l, r \rangle_{asp} \wedge (|r| \leq |l| \leq |r| + 1) \tag{4.15}$$

Following the definition of split_{asp}, we can define an operation

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

that satisfies

$$\mathsf{split}_{asp}\ (a : xs) = \mathsf{cons}_{asp}\ a\ (\mathsf{split}_{asp}\ xs) \tag{4.16}$$

From the previous section, we know that

$$\mathsf{map}_{asp}\ f\ \langle l, r \rangle_{asp} = \langle \mathsf{map}\ f\ l, \mathsf{map}\ f\ r \rangle_{asp}$$

Can we use Theorem 1 for tail with asp? For tail, $h = id$ and $\phi^1 = \lambda i . i + 1$. We can define $\theta^1 = \lambda i . 1 - i$ and so we find that

$$\theta^1 \cdot ch = ch \cdot \phi^1$$

However, if i is even then $ch(i) = 0$ and from Equation (4.5) we require that

$$\phi^1(f_0(i)) = f_{\theta^1(0)}\,(\phi^1(i))$$

The left hand side gives $1 + (i \text{ div } 2)$ and the right hand side gives $(i+1) \text{ div } 2$. Since i is even, these expressions are not equal and so Theorem 1 does not apply.

For the more straightforward operations, we can define the following:

$$\begin{array}{rcl}
|\langle l, r \rangle_{asp}|_{asp} & = & |l| + |r| \\
\mathsf{null}_{asp}\ \langle l, r \rangle_{asp} & = & \mathsf{null}\ l \\
\mathsf{head}_{asp}\ \langle l, r \rangle_{asp} & = & \mathsf{head}\ l \\
\mathsf{tail}_{asp}\ \langle l, r \rangle_{asp} & = & \langle r, \mathsf{tail}\ l \rangle_{asp} \\
\mathsf{elem}_{asp}\ a\ \langle l, r \rangle_{asp} & = & \mathsf{elem}\ a\ l\ \vee\ \mathsf{elem}\ a\ r
\end{array}$$

Let us now consider an operation cat_{asp} which corresponds to $+\!\!+$, *i.e.* $(+\!\!+) \leadsto \mathsf{cat}_{asp}$. By writing $+\!\!+$ as an uncurried operation then we can use Equation (3.9) so that cat_{asp} satisfies

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

This is equivalent to:

$$\mathsf{cat}_{asp}\ xsp\ ysp = \mathsf{split}_{asp}\ ((\mathsf{unsplit}_{asp}\ xsp)\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp)) \tag{4.17}$$

We derive a definition for cat_{asp} by structural induction on xsp.

Base Case Suppose that $xsp = \langle [\,], [\,] \rangle_{asp}$ and so $\mathsf{unsplit}_{asp}\ xsp = [\,]$.

$$\begin{aligned}
& \mathsf{split}_{asp}\ ((\mathsf{unsplit}_{asp}\ xsp)\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp)) \\
= & \quad \{\text{definitions of } xsp \text{ and } \mathsf{unsplit}_{asp}\} \\
& \mathsf{split}_{asp}\ ([\,] +\!\!+ (\mathsf{unsplit}_{asp}\ ysp)) \\
= & \quad \{\text{definition of } +\!\!+\} \\
& \mathsf{split}_{asp}\ (\mathsf{unsplit}_{asp}\ ysp) \\
= & \quad \{\text{Equation (4.13)}\} \\
& ysp
\end{aligned}$$

Step Case Suppose that $xsp = \langle x : r, l \rangle_{asp}$ and for the induction hypothesis, we assume that $\langle l, r \rangle_{asp}$ satisfies Equation (4.17).

$$\begin{aligned}
& \mathsf{split}_{asp}\ ((\mathsf{unsplit}_{asp}\ xsp)\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp)) \\
= & \quad \{\text{definition of } xsp\} \\
& \mathsf{split}_{asp}\ ((\mathsf{unsplit}_{asp}\ (\langle x : r, l \rangle_{asp}))\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp)) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
& \mathsf{split}_{asp}\ ((x : (\mathsf{unsplit}_{asp}\ \langle l, r \rangle_{asp}))\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp)) \\
= & \quad \{\text{definition of } +\!\!+\} \\
& \mathsf{split}_{asp}\ (x : ((\mathsf{unsplit}_{asp}\ \langle l, r \rangle_{asp})\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp))) \\
= & \quad \{\text{Property (4.16)}\} \\
& \mathsf{cons}_{asp}\ x\ (\mathsf{split}_{asp}\ ((\mathsf{unsplit}_{asp}\ \langle l, r \rangle_{asp})\ +\!\!+\ (\mathsf{unsplit}_{asp}\ ysp))) \\
= & \quad \{\text{induction hypothesis}\} \\
& \mathsf{cons}_{asp}\ x\ (\mathsf{cat}_{asp}\ \langle l, r \rangle_{asp}\ ysp)
\end{aligned}$$

Thus, we can define

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

As an alternative, we could define:

$$\langle l_0, r_0 \rangle_{asp} +\!\!+_{asp} \langle l_1, r_1 \rangle_{asp} = \begin{cases} \langle l_0 +\!\!+ l_1, r_0 +\!\!+ r_1 \rangle_{asp} & \text{if } |l_0| = |r_0| \\ \langle l_0 +\!\!+ r_1, r_0 +\!\!+ l_1 \rangle_{asp} & \text{otherwise} \end{cases}$$

In Appendix A, we discuss which of these two definitions produces the better obfuscation with respect to one of the assertions. We find that $+\!\!+_{asp}$ gives rise

to a more complicated assertion proof than cat_{asp}. This is because the structure of cat_{asp} is similar to $+\!\!+$ and the definition of $+\!\!+_{asp}$ uses $+\!\!+$ (so that we have to use the proof of the assertion for $+\!\!+$ in the proof of the assertion for $+\!\!+_{asp}$).

For filter_{asp}, we may expect to define

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

But, for example,

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

would give

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

which violates the Invariant (4.11). However, we can define filter_{asp} as follows:

$$\begin{array}{ll}
\mathsf{filter}_{asp}\ p\ \langle [\,],[\,]\rangle_{asp} & = \langle [\,],[\,]\rangle_{asp} \\
\mathsf{filter}_{asp}\ p\ \langle a : r, l\rangle_{asp} & = \text{if } p\ a \\
 & \quad\ \text{then } \mathsf{cons}_{asp}\ a\ (\mathsf{filter}_{asp}\ p\ \langle l, r\rangle_{asp}) \\
 & \quad\ \text{else } \mathsf{filter}_{asp}\ p\ \langle l, r\rangle_{asp}
\end{array}$$

Now computing

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

gives

$$\mathsf{cons}_{asp}\ 1\ (\mathsf{cons}_{asp}\ 5\ (\mathsf{cons}_{asp}\ 7\ \langle [\,],[\,]\rangle_{asp}))$$

which is equal to

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

We can prove that

$$\mathsf{filter}_{asp}\ p = \mathsf{split}_{asp} \cdot (\mathsf{filter}\ p) \cdot \mathsf{unsplit}_{asp}$$

The operations for the alternating split have similar efficiencies to the unsplit list operations. For $+\!\!+_{asp}$, the only extra computation is the test for $|l_0| = |r_0|$.

4.3.3 Block Split

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

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

Using the definitions of ch and $\mathcal{F}$ from Section 4.1.3 we can define an access operation, so if

$$xs \rightsquigarrow \langle l, r\rangle_{b(k)}$$

then

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

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

$$dti \equiv (|r| = 0 \wedge |l| < k) \vee (|l| = k) \tag{4.18}$$

This invariant will be strengthened in Section 5.3.2. From this we can see that if the list xs has at most k elements then,

$$xs \rightsquigarrow \langle xs, [\,]\rangle_{b(k)}$$

This split also satisfies Equation (4.12).

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

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

To ensure that our definition of $\mathsf{split}_{b(k)}$ matches up with the $(ch, \mathcal{F})$ formulation, we prove the following.

Correctness Proof 2 (Splitting function for the block split). Let xs be a non-empty list and $n :: \mathbb{N}$. If

$$\mathsf{split}_{b(k)}\ xs = \langle l, r\rangle_{b(k)}$$

then $(\forall\, n :: \mathbb{N}) \bullet n < |xs|$

$$xs \mathbin{!!} n = \begin{cases} l \mathbin{!!} n & \text{if } n < k \\ r \mathbin{!!} (n-k) & \text{otherwise} \end{cases} \tag{4.19}$$

Proof. We prove this by induction on k.

Base Case Suppose that $k = 0$. Then, by the definition of $\mathsf{split}_{b(k)}$, $l = [\,]$ and $r = xs$. We know that since $n \geq 0$ then $n < k$ is always false and so we need to compute $r \mathbin{!!} n$ which is exactly $xs \mathbin{!!} n$.

Step Case 1 Suppose that $k > 0$ and $xs = [\,]$. Then by the definition of split,

$$\mathsf{split}_{b(k)}\ [\,] = \langle [\,], [\,]\rangle_{b(k)}$$

Thus Equation (4.19) is vacuously true as there are no possible values for n.

Step Case 2 Suppose that $k > 0$ and $xs = t : ts$. Then by the definition of split,

$$\mathsf{split}_{b(k+1)}\ (t : ts) = \langle t : l, r\rangle_{b(k+1)} \quad \text{where } \langle l, r\rangle_{b(k)} = \mathsf{split}_{b(k)}\ ts$$

and for the induction hypothesis, we suppose that Equation (4.19) is true for all values at most k. If $n = 0$ then $(t : ts)\ !!\ 0 = t$ and $(t : l)\ !!\ 0 = t$.

If $n + 1 < k + 1$ then $n < k$ and consider

$$\begin{array}{ll} & (x : l)\ !!\ (n+1) \\ = & \quad \{\text{definition of } !!\} \\ & l\ !!\ n \\ = & \quad \{\text{induction hypothesis}\} \\ & ts\ !!\ n \\ = & \quad \{\text{definition of } !!\} \\ & (t : ts)\ !!\ (n+1) \end{array}$$

If $n + 1 \geq k + 1$ then $n \geq k$ and consider

$$\begin{array}{ll} & r\ !!\ ((n+1) - (k+1)) \\ = & \quad \{\text{arithmetic}\} \\ & r\ !!\ (n-k) \\ = & \quad \{\text{induction hypothesis}\} \\ & ts\ !!\ n \\ = & \quad \{\text{definition of } !!\} \\ & (t : ts)\ !!\ (n+1) \end{array}$$

Thus, by induction, Equation (4.19) holds.

□

The function $\mathsf{split}_{b(k)}$ is invertible and we can define:

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

Note that this definition is independent of k and we now show that $\mathsf{unsplit}_{b(k)}$ is a left-inverse for $\mathsf{split}_{b(k)}$.

Property 4 (Left inverse of split$_{b(k)}$). For all $k :: \mathbb{N}$ and all finite lists xs

$$\mathsf{unsplit}_{b(k)}\ (\mathsf{split}_{b(k)}\ xs) = xs \tag{4.20}$$

Proof. We prove this by considering the value of k.

Case 1 Suppose that $k = 0$. Then

$$\begin{array}{ll} & \mathsf{unsplit}_{b(0)}\ (\mathsf{split}_{b(0)}\ xs) \\ = & \quad \{\text{definition of } \mathsf{split}_{b(k)} \text{ with } k = 0\} \\ & \mathsf{unsplit}_{b(0)}\ \langle [\,], xs\rangle_{b(0)} \\ = & \quad \{\text{definition of } \mathsf{unsplit}_{b(0)}\} \\ & [\,] +\!\!+\ xs \\ = & \quad \{\text{definition of } +\!\!+\} \\ & xs \end{array}$$

Case 2 Suppose that $k > 0$. We prove Equation (4.20) by structural induction on xs.

Base Case Suppose that $xs = [\,]$. Then

$$
\begin{aligned}
& \mathsf{unsplit}_{b(k)}\ (\mathsf{split}_{b(k)}\ [\,]) \\
= & \quad \{\text{definition of } \mathsf{split}_{b(k)}\} \\
& \mathsf{unsplit}_{b(k)}\ \langle [\,], [\,] \rangle_{b(k)} \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b(k)}\} \\
& [\,] +\!\!+ [\,] \\
= & \quad \{\text{definition of } +\!\!+\} \\
& [\,]
\end{aligned}
$$

Step Case We now suppose that $xs = t : ts$ and let $ts \rightsquigarrow \langle l, r \rangle_{b(k-1)}$. For the induction hypothesis, we suppose that ts satisfies Equation (4.20), *i.e.* $ts = l +\!\!+ r$. Then

$$
\begin{aligned}
& \mathsf{unsplit}_{b(k)}\ (\mathsf{split}_{b(k)}\ (t : ts)) \\
= & \quad \{\text{definition of } \mathsf{split}_{b(k)} \text{ and refinement of } ts\} \\
& \mathsf{unsplit}_{b(k)}\ \langle t : l, r \rangle_{b(k)} \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{b(k)}\} \\
& (t : l) +\!\!+ r \\
= & \quad \{\text{definition of } +\!\!+\} \\
& t : (l +\!\!+ r) \\
= & \quad \{\text{induction hypothesis}\} \\
& t : ts
\end{aligned}
$$

□

We can consider $\mathsf{unsplit}_{b(k)}$ to be an abstraction function for this split and so

$$
\begin{aligned}
& xs \rightsquigarrow \langle l, r \rangle_{b(k)} \Leftrightarrow \\
& \quad xs = \mathsf{unsplit}_{b(k)}\ \langle l, r \rangle_{b(k)} \wedge ((|r| = 0 \wedge |l| < k) \vee (|l| = k))
\end{aligned} \tag{4.21}
$$

We would like a function $\mathsf{cons}_{b(k)}$ that satisfies:

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

and so we define:

$$
\mathsf{cons}_{b(k)}\ a\ \langle l, r \rangle_{b(k)} = \begin{cases} \langle a : l, r \rangle_{b(k)} & \text{if } |l| < k \\ \langle a : (\mathsf{init}\ l), (\mathsf{last}\ l) : r \rangle_{b(k)} & \text{otherwise} \end{cases}
$$

where the functions init and last satisfy:

$$xs = \mathsf{init}\ xs +\!\!+ [\mathsf{last}\ xs] \quad \text{where } xs \neq [\,]$$

From earlier in this chapter, by using Theorem 1, we can define

$$\mathsf{map}_{b(k)}\ f\ \langle l, r \rangle_{b(k)} = \langle \mathsf{map}\ f\ l, \mathsf{map}\ f\ r \rangle_{b(k)}$$

Can we use Theorem 1 with tail? For tail, $h = id$ and $\phi^1 = \lambda i.i+1$. If $k > 1$ then from Equation (4.4), we need a function θ^1 such that

$$(\forall i)\ \theta^1(ch(i)) = ch(i+1)$$

Taking $i = 0$ gives that $\theta^1(0) = 0$ but taking $i = k-1$ gives that $\theta^1(0) = ch(k) = 1$ and so we cannot find a function θ^1 and Theorem 1 is not satisfied for tail.

Below we define the more straightforward list operations for the block split:

$$\begin{aligned}
|\langle l, r\rangle_{b(k)}|_{b(k)} &= |l| + |r| \\
\mathsf{null}_{b(k)}\ \langle l, r\rangle_{b(k)} &= \mathsf{null}\ l \wedge \mathsf{null}\ r \\
\mathsf{head}_{b(k)}\ \langle l, r\rangle_{b(k)} &= \begin{cases} \mathsf{head}\ r & \text{if } k = 0 \\ \mathsf{head}\ l & \text{otherwise} \end{cases} \\
\mathsf{tail}_{b(k)}\ \langle l, r\rangle_{b(k)} &= \begin{cases} \langle \mathsf{tail}\ l, r\rangle_{b(k)} & \text{if } |r| = 0 \\ \langle \mathsf{tail}\ l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r\rangle_{b(k)} & \text{otherwise} \end{cases} \\
\mathsf{elem}_{b(k)}\ a\ \langle l, r\rangle_{b(k)} &= \mathsf{elem}\ a\ l \vee \mathsf{elem}\ a\ r
\end{aligned}$$

We can derive a function $+\!\!+_{b(k)}$ that satisfies

$$xsp +\!\!+_{b(k)} ysp = \mathsf{split}_{b(k)}\ ((\mathsf{unsplit}_{b(k)}\ xsp)\ +\!\!+\ (\mathsf{unsplit}_{b(k)}\ ysp)) \tag{4.22}$$

by using induction on $|xsp|$ and we obtain the following definition:

$$\begin{aligned}
\langle [\,], [\,]\rangle_{b(k)} &+\!\!+_{b(k)} ysp = ysp \\
\langle a : l, [\,]\rangle_{b(k)} &+\!\!+_{b(k)} ysp = \mathsf{cons}_{b(k)}\ a\ (\langle l, [\,]\rangle_{b(k)} +\!\!+_{b(k)} ysp) \\
\langle a : l, r\rangle_{b(k)} &+\!\!+_{b(k)} ysp = \mathsf{cons}_{b(k)}\ a\ (\langle l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r\rangle_{b(k)} +\!\!+_{b(k)} ysp)
\end{aligned}$$

In Appendix A, we find that this operation is a good obfuscation as the definition has three distinct cases and does not follow a similar pattern to the definition of $+\!\!+$. We can define a filtering function as follows:

$$\begin{aligned}
\mathsf{filter}_{b(k)}\ p\ \langle [\,], [\,]\rangle_{b(k)} &= \langle [\,], [\,]\rangle_{b(k)} \\
\mathsf{filter}_{b(k)}\ p\ \langle a : l, [\,]\rangle_{b(k)} &= \text{if } p\ a \\
&\quad\ \text{then } \mathsf{cons}_{b(k)}\ a\ (\mathsf{filter}_{b(k)}\ p\ \langle l, [\,]\rangle_{b(k)}) \\
&\quad\ \text{else } \mathsf{filter}_{b(k)}\ p\ \langle l, [\,]\rangle_{b(k)} \\
\mathsf{filter}_{b(k)}\ p\ \langle a : l, b : r\rangle_{b(k)} &= \text{if } p\ a \\
&\quad\ \text{then } \mathsf{cons}_{b(k)}\ a\ (\mathsf{filter}_{b(k)}\ p\ \langle l +\!\!+ [b], r\rangle_{b(k)}) \\
&\quad\ \text{else } \mathsf{filter}_{b(k)}\ p\ \langle l +\!\!+ [b], r\rangle_{b(k)}
\end{aligned}$$

We can show that this definition satisfies

$$\mathsf{filter}\ p = \mathsf{unsplit}_{b(k)} \cdot (\mathsf{filter}_{b(k)}\ p) \cdot \mathsf{split}_{b(k)}$$

The operations for the block split have the same complexities as the normal split operations but most operations require an extra test (usually to determine whether the second component is an empty list).

Before using the block split, we need to determine what value of k we want to use. Many of operations depend on the choice of the decision value (k) and so we need know what this value will be in advance. We can remove this requirement if we use an augmented split (Section 4.4.1).

4.4 Random List Splitting

Each of the two list splittings that we have discussed so far always split a list in the same way. Suppose that we split up a finite list randomly so that the list is split differently each time a program is executed. This means that even with the same input, different program traces are produced on different executions and this helps to confuse an attacker further. So, for a list xs, we want

$$xs \rightsquigarrow \langle l, r \rangle$$

If we want to do this randomly then an easy way to do so is to provide a random Boolean for each element of xs. We could say that if the value is *True* for an element of xs then that element should be placed into the list r and if *False* then into l. So when splitting each list, we need to provide a list of Booleans that tells us how to split the list — such a list can be a decision value for this split. For example, we want to split up the list $[1, 2, 3, 4, 5, 6]$. Then using the list $[F, F, F, T, F, T]$

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

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

To be able to use the split list, we will need to how it has been split and so we have to carry around the decision value (for the example split above, the decision value is 40). To do this we create a new type called an *augmented split list* which contains a decision value (of type β) as well as the split lists:

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

So,

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

For our example split, we take $\beta = \mathbb{N}$.

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

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

and we can define $f_i \in \mathcal{F}$ as follows:

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

Before we define some operations for this split, we derive a property of f_k:

$$f_k(s + 1, n)$$
$$= \quad \{\text{definition}\}$$
$$|\{i \mid 0 \le i < s + 1 \wedge ch(i, n) == k\}|$$
$$= \quad \{\text{range split}\}$$

$$
\begin{aligned}
& |\{i \mid i = 0 \wedge ch(i,n) == k\}| + \\
& |\{i \mid 1 \leq i < s+1 \wedge ch(i,n) == k\}| \\
= \quad & \{\text{let } j = i - 1 \text{ and definition of } ch\} \\
& |\{i \mid i = 0 \wedge n \bmod 2 == k\}| + \\
& |\{j+1 \mid 0 \leq j < s \wedge ch(j+1,n) == k\}| \\
= \quad & \{\text{arithmetic and definition of } ch\} \\
& (1 - k + n \bmod 2) \bmod 2 + \\
& |\{j+1 \mid 0 \leq j < s \wedge ch(j, n \text{ div } 2) == k\}| \\
= \quad & \{\text{since we take the size of the set we can replace } j+1 \text{ by } j\} \\
& (1 - k + n \bmod 2) \bmod 2 + |\{j \mid 0 \leq j < s \wedge ch(j, n \text{ div } 2) == k\}| \\
= \quad & \{\text{definition of } f_k\} \\
& (1 - k + n \bmod 2) \bmod 2 + f_k(s, n \text{ div } 2)
\end{aligned}
$$

Thus

$$f_k(s+1, n) = f_k(s, n \text{ div } 2) + (1 - k + n \bmod 2) \bmod 2 \tag{4.23}$$

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

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

We would like two functions split and unsplit that satisfy the following property

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

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

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

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

We give a definition for $\mathsf{unsplit}_A$ on Page 85.

Now we prove that this definition of split_A matches up with the $(ch, \mathcal{F})$ formulation.

Correctness Proof 3 (Splitting function for the augmented split). If

$$\mathsf{split}_A\ xs = \langle n, l, r\rangle_A$$

for a list xs then $(\forall\, s :: \mathbb{N}) \bullet\ s < |xs|$

$$xs\ !!\ s = \begin{cases} l\ !!\ f_0\,(s,n) & \text{if } ch(s,n) == 0 \\ r\ !!\ f_1\,(s,n) & \text{otherwise} \end{cases} \tag{4.24}$$

Proof. Suppose that $xs = [\,]$ then there are no possible values of s and so the result is vacuously true.

We now induct on s and suppose that xs is non-empty. We have two cases depending on whether n is even or odd.

Even Suppose that $n \bmod 2 = 0$. Then by the definitions of split_A and cons_A:

$$(t : ts) \rightsquigarrow \langle n, (t : l), r\rangle_A \text{ where } ts \rightsquigarrow \langle n \text{ div } 2, l, r\rangle_A$$

Base Case Let $s = 0$ then $ch(s, n) = n \bmod 2 = 0$ and so

$$(t : l) \;!!\; f_0\,(0, n) = (t : l) \;!!\; 0 = t = (t : ts) \;!!\; 0$$

Step Case Suppose that $s > 0$. For the induction hypothesis, we suppose that Equation (4.24) is true for all natural numbers at most s. If $ch(s + 1, n) = 0$ then, by the definition of ch, $ch(s, n \text{ div } 2) = 0$ and

$$\begin{array}{ll}
 & (t : l) \;!!\; f_0\,(s + 1, n) \\
= & \quad \{\text{Property (4.23) and } 1 - k + n \bmod 2 = 1\} \\
 & (t : l) \;!!\; (1 + f_0\,(s, n \text{ div } 2)) \\
= & \quad \{\text{definition of !!}\} \\
 & l \;!!\; f_0\,(s, n \text{ div } 2) \\
= & \quad \{\text{induction hypothesis}\} \\
 & ts \;!!\; s \\
= & \quad \{\text{definition of !!}\} \\
 & (t : ts) \;!!\; (s + 1)
\end{array}$$

If $ch(s + 1, n) = 1$ then $ch(s, n \text{ div } 2) = 1$ and

$$\begin{array}{ll}
 & r \;!!\; f_1\,(s + 1, n) \\
= & \quad \{\text{Property (4.23) and } 1 - k + n \bmod 2 = 0\} \\
 & r \;!!\; f_1\,(s, n \text{ div } 2) \\
= & \quad \{\text{induction hypothesis}\} \\
 & ts \;!!\; s \\
= & \quad \{\text{definition of !!}\} \\
 & (t : ts) \;!!\; (s + 1)
\end{array}$$

Odd Suppose that $n \bmod 2 = 1$. Then by the definition of split_A

$$(t : ts) \rightsquigarrow \langle n, l, (t : r)\rangle_A \text{ where } ts \rightsquigarrow \langle n \text{ div } 2, l, r\rangle_A$$

Base Case Let $s = 0$ then $ch(s, n) = n \bmod 2 = 1$ and so

$$(t : r) \;!!\; f_1\,(0, n) = (t : r) \;!!\; 0 = t = (t : ts) \;!!\; 0$$

Step Case Suppose that $s > 0$. For the induction hypothesis, we suppose that Equation (4.24) is true for all natural numbers at most s. If $ch(s+1, n) = 0$ by the definition of ch, $ch(s, n \text{ div } 2) = 0$ and

$$
\begin{array}{ll}
& l \;!!\; f_0\,(s+1, n) \\
= & \quad \{\text{Property (4.23) and } 1 - k + n \text{ mod } 2 = 0\} \\
& l \;!!\; f_0\,(s, n \text{ div } 2) \\
= & \quad \{\text{induction hypothesis}\} \\
& ts \;!!\; s \\
= & \quad \{\text{definition of } !!\} \\
& (t : ts) \;!!\; (s+1)
\end{array}
$$

If $ch(s+1, n) = 1$ then $ch(s, n \text{ div } 2) = 1$ and

$$
\begin{array}{ll}
& (t : r) \;!!\; f_1\,(s+1, n) \\
= & \quad \{\text{Property (4.23) and } 1 - k + n \text{ mod } 2 = 1\} \\
& (t : r) \;!!\; (1 + f_1\,(s, n \text{ div } 2)) \\
= & \quad \{\text{definition of } !!\} \\
& r \;!!\; f_1\,(s, n \text{ div } 2) \\
= & \quad \{\text{induction hypothesis}\} \\
& ts \;!!\; s \\
= & \quad \{\text{definition of } !!\} \\
& (t : ts) \;!!\; (s+1)
\end{array}
$$

□

We now define an inverse for split_A — $\mathsf{unsplit}_A$ — which can be written as follows:

$$
\begin{array}{ll}
\mathsf{unsplit}_A\ \langle n, [\,], rs \rangle_A & = rs \\
\mathsf{unsplit}_A\ \langle n, ls, [\,] \rangle_A & = ls \\
\mathsf{unsplit}_A\ \langle n, (l : ls), (r : rs) \rangle_A & = \text{if } m == 0 \\
& \quad \text{then } l : (\mathsf{unsplit}_A\ \langle d, ls, (r : rs) \rangle_A) \\
& \quad \text{else } \ r : (\mathsf{unsplit}_A\ \langle d, (l : ls), rs \rangle_A) \\
& \qquad \text{where } (d, m) = \mathsf{divMod}\ n\ 2
\end{array}
$$

Although

$$\mathsf{unsplit}_A \cdot (\mathsf{split}_A\ n) = id$$

it is not always the case that

$$(\mathsf{split}_A\ n) \cdot \mathsf{unsplit}_A = id$$

For example,

$$[1, 2, 3, 4] \rightsquigarrow \langle 10, [1, 3], [2, 4] \rangle_A$$

but

$$\begin{aligned}\mathsf{split}\ 11\ (\mathsf{unsplit}_A\ \langle 10, [1,3], [2,4]\rangle_A) &= \mathsf{split}_A\ 11\ [1,2,3,4] \\ &= \langle 11, [3], [1,2,4]\rangle_A \\ &\neq \langle 10, [1,3], [2,4]\rangle_A\end{aligned}$$

Instead of equality for split lists, we could set up an equivalence relation.

$$xsp \equiv ysp \Leftrightarrow \mathsf{unsplit}_A\ xsp = \mathsf{unsplit}_A\ ysp$$

or equivalently

$$xsp \equiv yss \Leftrightarrow (\exists\ xs)\ (xs \rightsquigarrow xsp \wedge xs \rightsquigarrow ysp)$$

We can define the more straightforward list operations as follows:

$$\begin{aligned}|\langle n, l, r\rangle_A|_A &= |l| + |r| \\ \mathsf{null}_A\ \langle n, l, r\rangle_A &= \mathsf{null}\ l \wedge \mathsf{null}\ r \\ \mathsf{elem}_A\ x\ \langle n, l, r\rangle_A &= \mathsf{elem}\ x\ l \vee \mathsf{elem}\ x\ r \\ \mathsf{map}_A\ f\ \langle n, l, r\rangle_A &= \langle n, \mathsf{map}\ f\ l, \mathsf{map}\ f\ r\rangle_A\end{aligned}$$

Following the definition of cons_A, we can define head_A and tail_A as follows:

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

For efficiency, we can combine these two operations in a single operation called $\mathsf{headTail}A$ which has type $ASpList\ \alpha \rightarrow (\mathbb{N}, \alpha, ASpList\ \alpha)$. We define the operation as follows:

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

This operation satisfies the following property:

$$xsp = \mathsf{cons}_A\ b\ h\ t \ \Leftrightarrow\ (b, h, t) = \mathsf{headTail}_A\ xsp \tag{4.25}$$

The proof is routine and we omit the details.

Using $\mathsf{headTail}_A$, we can define an operation analogous to $+\!\!+$:

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

which satisfies

$$xs +\!\!+ ys = \mathsf{unsplit}_A\ (\mathsf{cat}_A\ (\mathsf{split}_A\ m\ xs)\ (\mathsf{split}_A\ n\ ys))$$

Using $\mathsf{headTail}_A$, we can define a more succinct definition for $\mathsf{unsplit}$

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

and finally we can prove that $\mathsf{unsplit}_A$ is a left inverse for split_A.

Property 5 (Left inverse of split_A). For all $d :: \mathbb{N}$,

$$\mathsf{unsplit}_A\ (\mathsf{split}_A\ d\ xs) = xs \tag{4.26}$$

Proof. We prove this by induction on xs.

Base Case Suppose that $xs = [\,]$

$$\begin{aligned}
& \mathsf{unsplit}_A\ (\mathsf{split}_A\ d\ [\,]) \\
= & \quad \{\text{definition of } \mathsf{split}_A\} \\
& \mathsf{unsplit}_A\ \langle d, [\,], [\,] \rangle_A \\
= & \quad \{\text{definition of } \mathsf{unsplit}\} \\
& [\,]
\end{aligned}$$

Step Case Suppose that $xs = y : ys$ and that ys satisfies Equation (4.26). Let $n = 2d + m$ so that $(d, m) = \mathsf{divMod}\ n\ 2$ then

$$\begin{aligned}
& \mathsf{unsplit}_A\ (\mathsf{split}_A\ n\ (y : ys)) \\
= & \quad \{\text{definition of } \mathsf{split}_A\} \\
& \mathsf{unsplit}_A\ (\mathsf{cons}_A\ m\ y\ (\mathsf{split}\ d\ ys)) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_A \text{ and Equation (4.25)}\} \\
& y : (\mathsf{unsplit}_A\ (\mathsf{split}_A\ d\ ys)) \\
= & \quad \{\text{induction hypothesis}\} \\
& y : ys
\end{aligned}$$

□

Consider this definition:

$$\begin{aligned}
& \mathsf{filter}_A\ p\ \langle n, [\,], [\,] \rangle_A = \langle n, [\,], [\,] \rangle_A \\
& \mathsf{filter}_A\ p\ \ xsp \qquad\quad = \text{if } p\ h \text{ then } \mathsf{cons}_A\ b\ h\ ysp \text{ else } ysp \\
& \qquad\qquad\qquad\qquad\quad \text{where } ysp = \mathsf{filter}_A\ p\ t \\
& \qquad\qquad\qquad\qquad\qquad\quad (b, h, t) = \mathsf{headTail}_A\ xsp
\end{aligned}$$

We would like this operation to correspond to the filter operation for lists, *i.e.*

$$xs \rightsquigarrow xsp \Rightarrow \mathsf{filter}\ p\ xs \rightsquigarrow \mathsf{filter}_A\ p\ xsp$$

To show this, we can prove that

$$\mathsf{filter}\ p\ ys = \mathsf{unsplit}_A\ (\mathsf{filter}_A\ p\ (\mathsf{split}_A\ n\ ys))$$

by structural induction on ys.

The complexities of the augmented split operations are the same as for normal lists however most operations require some extra computations. Operations such as cons_A and tail_A need to perform some arithmetic and cat_A and filter_A need to compute $\mathsf{headTail}_A$.

Using augmented split lists we develop two versions of the block split that allows us to introduce some randomness in the splits.

4.4.1 Augmented Block Split

We now briefly describe (without proofs) a variation of the block split in which we store the decision value and the split components. We call this the *augmented block* splits — denoted by a subscript B. We would like

$$xs \rightsquigarrow \langle l, r \rangle_{b(k)} \Rightarrow xs \rightsquigarrow \langle k, l, r \rangle_B$$

We can define a split function as follows:

$$\begin{aligned}
&\mathsf{split}_B\ k\ [\,] &&= \langle k, [\,], [\,]\rangle_B\\
&\mathsf{split}_B\ 0\ (x : xs) &&= \langle 0, [\,], x : xs\rangle_B\\
&\mathsf{split}_B\ (k+1)\ (x : xs) &&= \langle k+1, x : l, r\rangle_B\\
& &&\quad \text{where } \langle m, l, r\rangle_B = \mathsf{split}_B\ k\ xs
\end{aligned}$$

The unsplitting function is the same as the k-block split:

$$\mathsf{unsplit}_B\ \langle n, l, r\rangle_B = l +\!\!+\, r$$

The representation $xs \rightsquigarrow \langle n, l, r\rangle_B$ satisfies the invariant

$$dti \equiv (|r| = 0 \wedge |l| < n) \vee (|l| = n) \tag{4.27}$$

By storing the decision value, we will see that the operations that we define can use this value. Hence, we can choose the decision value randomly and so on different execution runs, the same list could be split differently. Also, when performing operation on this split, we can change the decision value as well the lists themselves.

For instance, we can define a cons operation as follows:

$$\begin{aligned}
&\mathsf{cons}_B\ a\ \langle k, l, r\rangle_B\\
&\quad |\ |l| < k &&= \langle k, a : l, r\rangle_B\\
&\quad |\ \text{otherwise} &&= \langle k+1, a : l, r\rangle_B
\end{aligned}$$

The definition for the k-block split required keeping the length of the first list component the same and so the last of element of the first list was added to the front of the second list. The version of cons for the augmented block split is more efficient as we merely increase the decision value by one. However cons_B cannot be used to build up split lists starting from an empty split list as the list r will remain empty.

The straightforward list operations are defined as follows:

$$\begin{aligned}
|\langle k, l, r\rangle_B|_B &= |l| + |r|\\
\mathsf{null}_B\ \langle k, l, r\rangle_B &= \mathsf{null}\ l \wedge \mathsf{null}\ r\\
\mathsf{elem}_B\ a\ \langle k, l, r\rangle_B &= \mathsf{elem}\ a\ l \vee \mathsf{elem}\ a\ r\\
\mathsf{head}_B\ \langle k, l, r\rangle_B &= \begin{cases}\mathsf{head}\ r & \text{if } k = 0\\ \mathsf{head}\ l & \text{otherwise}\end{cases}\\
\mathsf{tail}_B\ \langle k, l, r\rangle_B &= \begin{cases}\langle k, l, \mathsf{tail}\ r\rangle_B & \text{if } k = 0\\ \langle k-1, \mathsf{tail}\ l, r\rangle_B & \text{otherwise}\end{cases}\\
\mathsf{map}_B\ f\ \langle k, l, r\rangle_B &= \langle k, \mathsf{map}\ f\ l, \mathsf{map}\ f\ r\rangle_B
\end{aligned}$$

All these definitions, with the exception of tail, match the definitions for the k-block split. The definition of tail is more efficient as we just decrease the decision value instead of joining the head of the second component to the end of the first component.

Here is a possible definition for a concatenation operation:

$$\begin{aligned}
\langle k, [\,], [\,]\rangle_B +\!\!+_B \langle k', l', r'\rangle_B &= \langle k', l', r'\rangle_B\\
\langle k, l, r\rangle_B +\!\!+_B \langle k', l', r'\rangle_B &= \langle |l +\!\!+ r|, l +\!\!+ r, l' +\!\!+ r'\rangle_B
\end{aligned}$$

and a definition for a filtering operation:

$$\begin{aligned} \mathsf{filter}_B\ p\ \langle n, l, r\rangle_B = & \ \langle |l'|, l', \mathsf{filter}\ p\ r\rangle_B \\ & \text{where } l' = \mathsf{filter}\ p\ l \end{aligned}$$

The complexities of the augmented block split operations match the complexities of the normal list operations. The operations head_A and tail_A have an extra test for $k = 0$ and $+\!\!+_B$ and filter_B need to compute some list lengths.

4.4.2 Padded Block Split

For a list $xs = ls +\!\!+ rs$, we want

$$xs \rightsquigarrow \langle |ls|, ls +\!\!+ js, rs\rangle_P$$

where js is a list of random elements — thus we can pad the split list with extra junk elements.

We decide on how many elements to put into the first component and so we pass this decision value in as a parameter. Since we store the decision value as well as the split list, we can choose the value randomly.

We can define a splitting function as follows:

$$\begin{aligned} \mathsf{split}_P\ n\ [\,] &= \langle 0, js, [\,]\rangle_P \\ \mathsf{split}_P\ 0\ (x : xs) &= \langle 0, js, x : xs\rangle_P \\ \mathsf{split}_P\ (n+1)\ (x : xs) &= \langle m+1, x : l, r\rangle_P \\ &\quad \text{where } \langle m, l, r\rangle_P = \mathsf{split}_P\ n\ xs \end{aligned}$$

where the list js is a random list. We define the unsplitting function as follows:

$$\mathsf{unsplit}_P\ \langle n, l, r\rangle_P = (\mathsf{take}\ n\ l) +\!\!+ r$$

where the operation take is specified as follows:

$$\begin{aligned} \mathsf{take}\ n\ xs = ys \text{ where } & xs = ys +\!\!+ zs \\ & |ys| = \begin{cases} |xs| & \text{if } |xs| < n \\ n & \text{otherwise} \end{cases} \end{aligned}$$

The representation $xs \rightsquigarrow \langle n, l, r\rangle_P$ satisfies the following invariant:

$$dti \equiv |l| \geq n \tag{4.28}$$

Note that for this split, Invariant (4.12) is not true. Instead, we have that for a list xs of type $List\ \alpha$

$$xs \rightsquigarrow \langle n, l, r\rangle_P \Rightarrow (\forall x :: \alpha)\ \mathsf{freq}(x, xs) = \mathsf{freq}(x, \mathsf{take}\ n\ l) + \mathsf{freq}(x, r)$$

We can define cons_P as follows:

$$\mathsf{cons}_P\ a\ \langle n, l, r\rangle_P = \langle n+1, a : l, r\rangle_P$$

This is more efficient than $\mathsf{cons}_{b(k)}$ as we just need to add the new element to the front of l and increment n.

The straightforward list operations are defined as follows:

$$
\begin{array}{rcl}
|\langle n, l, r\rangle_P|_P & = & n + |r| \\
\mathsf{null}_P\ \langle n, l, r\rangle_P & = & n == 0 \wedge \mathsf{null}\ r \\
\mathsf{elem}\ a\ \langle n, l, r\rangle_P & = & \mathsf{elem}\ a\ (\mathsf{take}\ n\ l) \vee \mathsf{elem}\ a\ r \\
\mathsf{head}_P\ \langle k, l, r\rangle_P & = & \begin{cases} \mathsf{head}\ r & \text{if } k = 0 \\ \mathsf{head}\ l & \text{otherwise} \end{cases} \\
\mathsf{tail}_P\ \langle k, l, r\rangle_P & = & \begin{cases} \langle k, l, \mathsf{tail}\ r\rangle_P & \text{if } k = 0 \\ \langle k-1, \mathsf{tail}\ l, r\rangle_P & \text{otherwise} \end{cases} \\
\mathsf{map}_P\ f\ \langle n, l, r\rangle_P & = & \langle n, \mathsf{map}\ f\ l, \mathsf{map}\ f\ r\rangle_P
\end{array}
$$

For the definition of elem_P we need to use the take function so that we discard the junk elements. The definition of tail_P is the same as for tail_B.

We can define a concatenation operation as follows:

$$
\begin{array}{l}
\langle 0, l, [\,]\rangle_P \mathbin{+\!\!+}_P \langle m, l', r'\rangle_P = \langle m, l' \mathbin{+\!\!+} l, r'\rangle_P \\
\langle n, l, r\rangle_P \mathbin{+\!\!+}_P \langle m, l', r'\rangle_P = \langle |lt| + |r|, lt \mathbin{+\!\!+} r \mathbin{+\!\!+} ld' \mathbin{+\!\!+} ld, lt' \mathbin{+\!\!+} r'\rangle_P \\
\qquad \text{where } lt = \mathsf{take}\ n\ l \\
\qquad\qquad ld = \mathsf{drop}\ n\ l \\
\qquad\qquad lt' = \mathsf{take}\ m\ l' \\
\qquad\qquad ld' = \mathsf{drop}\ m\ l'
\end{array}
$$

We actually have some degree of freedom with this operation. Instead of creating the list $lt \mathbin{+\!\!+} r \mathbin{+\!\!+} ld' \mathbin{+\!\!+} ld$, we can create $lt \mathbin{+\!\!+} r \mathbin{+\!\!+} rs$, where rs is a random list.

Finally, we define filter_P as follows:

$$
\begin{array}{l}
\mathsf{filter}_P\ f\ \langle n, l, r\rangle_P = \langle |lf|, lf \mathbin{+\!\!+} ld, \mathsf{filter}\ f\ r\rangle_P \\
\qquad \text{where } lf = \mathsf{filter}\ f\ (\mathsf{take}\ n\ l) \\
\qquad\qquad ld = \mathsf{drop}\ n\ l
\end{array}
$$

Again, we can replace ld by a random list.

Many of the operations require the use of take and drop. This means that many of the operations for the padded block split are less efficient although all have the same complexity as the corresponding unsplit list operation. For efficiency we could use the function splitAt which computes both take and drop in one pass through the list.

4.5 Conclusions

In this chapter we have seen how to generalise the array-split discussed in [10] and we have shown various ways of splitting lists. Our generalisation of a split included creating a choice function and stating an abstract data-type. We then saw that this generalisation covers the array-split discussed in [10]. Thus using the techniques from this chapter enables us to produce more array obfuscations for imperative programs. In the next chapters we extend the notion of splitting to other abstract data-types.

A consequence of our generalisation of splitting was that we could devise ways of splitting randomly. To date, random obfuscations seem to have received little attention. Randomness can help to confuse an attacker by adding an extra level of obscurity. One of the random obfuscations that we proposed involved

padding one of the components with junk elements. This obfuscation raises concerns about stealth. We must ensure that any junk elements that we create do not "stand out" so that it is not obvious which elements are "real" data. We could also make some of the operations act on the junk so that it is not noticeable that the junk is unimportant. These concerns also apply to the tree obfuscations that we discuss in Chapter 7.

In Appendix A we consider different obfuscations of $+\!\!+$. We prove the same assertion for each definition. Here is a table of results for the different proofs:

Operation	$\mathcal{C}$	$\mathcal{H}$
$+\!\!+$	11	7
$+\!\!+_{asp}$	27	13
cat_{asp}	15	10
$+\!\!+_{b(k)}$	43	16
cat_{A}	18	10
$+\!\!+_{B}$	24	13
$+\!\!+_{P}$	27	12

where the $\mathcal{C}$ is the cost of a proof tree and $\mathcal{H}$ is the height. We find that if the definition of an obfuscated operation follows a similar pattern to the definition of the original operation then the proof of the assertion is also similar. Thus an operation will be a good assertion obfuscation if its definition does not follow a similar pattern to the unobfuscated operation. In general, there is a trade-off between the degree of obfuscation and the computational complexity of an obfuscation but all of our obfuscations of $+\!\!+$ have linear complexity (although many require extra computations such as evaluating tests).

Chapter 5

Sets and the Splitting

Buffy: "This'll probably go faster if we split up."
Lily: "Can I come with you?"
Buffy: "Okay, where did I lose you on the whole splitting up thing?"

Buffy the Vampire Slayer — *Anne* (1998)

In this chapter, we consider how we can split finite sets. For the padded block split (defined in Section 4.4.2) we placed junk elements at specific places in the list. Since sets are unordered we have difficulties adding junk elements as we are not sure which elements are "real" and which are "junk". So to split sets we will represent sets using Haskell lists and then split the corresponding lists. We consider the set data-type and the representations discussed in [7, Section 8.3].

5.1 A Data-Type for Sets

The data-type for sets is defined in Figure 5.1. Note that *dti* is a data-type invariant (we specify this when we decide on a list representation).

How do these operations relate to the mathematical operations? The operation empty stands for $\emptyset$, member for $\in$, union for $\cup$, meet for $\cap$ and minus for set difference ($\backslash$). We can specify

$$\begin{aligned} \mathsf{isEmpty}\ s &\equiv s == \emptyset \\ \mathsf{insert}\ a\ s &\equiv \{a\} \cup s \\ \text{and } \mathsf{delete}\ a\ s &\equiv s \backslash \{a\} \end{aligned}$$

We will consider two list representations — unordered lists with duplicates and strictly-increasing lists. A discussion using unordered lists without duplicates is given in [20].

5.2 Unordered Lists with duplicates

We first choose to represent a set as an unordered list with duplicates — then there is no particular data-type invariant and so *dti* is *True*. We define the set

Set (α)
$\quad USING:$ List (α)

$Set\ \alpha \sim List\ \alpha \wedge dti$

$$
\begin{aligned}
\mathsf{empty} &:: Set\ \alpha \\
\mathsf{isEmpty} &:: Set\ \alpha \rightarrow \mathbb{B} \\
\mathsf{member} &:: Set\ \alpha \rightarrow \alpha \rightarrow \mathbb{B} \\
\mathsf{insert} &:: \alpha \rightarrow Set\ \alpha \rightarrow Set\ \alpha \\
\mathsf{delete} &:: \alpha \rightarrow Set\ \alpha \rightarrow Set\ \alpha \\
\mathsf{union} &:: Set\ \alpha \rightarrow Set\ \alpha \rightarrow Set\ \alpha \\
\mathsf{meet} &:: Set\ \alpha \rightarrow Set\ \alpha \rightarrow Set\ \alpha \\
\mathsf{minus} &:: Set\ \alpha \rightarrow Set\ \alpha \rightarrow Set\ \alpha
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{insert}\ a\ (\mathsf{insert}\ a\ xs) &= \mathsf{insert}\ a\ xs \\
\mathsf{insert}\ a\ (\mathsf{insert}\ b\ xs) &= \mathsf{insert}\ b\ (\mathsf{insert}\ a\ xs) \\
\mathsf{isEmpty}\ \mathsf{empty} &= True \\
\mathsf{isEmpty}\ (\mathsf{insert}\ a\ xs) &= False \\
\mathsf{member}\ \mathsf{empty}\ y &= False \\
\mathsf{member}\ (\mathsf{insert}\ a\ xs)\ y &= (a = y) \vee \mathsf{member}\ xs\ a \\
\mathsf{delete}\ a\ \mathsf{empty} &= \mathsf{empty} \\
\mathsf{delete}\ a\ (\mathsf{insert}\ b\ xs) &= \text{if } a = b \text{ then } \mathsf{delete}\ a\ xs \\
&\qquad\qquad\quad \text{else } \mathsf{insert}\ b\ (\mathsf{delete}\ a\ xs) \\
\mathsf{union}\ xs\ \mathsf{empty} &= xs \\
\mathsf{union}\ xs\ (\mathsf{insert}\ b\ ys) &= \mathsf{insert}\ b\ (\mathsf{union}\ xs\ ys) \\
\mathsf{meet}\ xs\ \mathsf{empty} &= \mathsf{empty} \\
\mathsf{meet}\ xs\ (\mathsf{insert}\ b\ ys) &= \text{if } \mathsf{member}\ xs\ b \text{ then } \mathsf{insert}\ b\ (\mathsf{meet}\ xs\ ys) \\
&\qquad\qquad\qquad\qquad \text{else } \mathsf{meet}\ xs\ ys \\
\mathsf{minus}\ xs\ \mathsf{empty} &= xs \\
\mathsf{minus}\ xs\ (\mathsf{insert}\ b\ ys) &= \mathsf{minus}\ (\mathsf{delete}\ b\ xs)\ ys
\end{aligned}
$$

Figure 5.1: Data-type for Sets

operations as follows:

$$
\begin{aligned}
\mathsf{empty} &= [\,] \\
\mathsf{isEmpty} &= (== [\,]) \\
\mathsf{member}\ xs\ a &= \text{if } \mathsf{isEmpty}\ xs \text{ then } \mathit{False} \\
&\qquad \text{else } a == (\mathsf{head}\ xs) \vee \mathsf{member}\ (\mathsf{tail}\ xs)\ a \\
\mathsf{insert}\ a\ xs &= a : xs \\
\mathsf{delete}\ a\ xs &= \mathsf{filter}\ (\neq a)\ xs \\
\mathsf{union}\ xs\ ys &= xs +\!\!+\ ys \\
\mathsf{meet}\ xs\ ys &= \mathsf{filter}\ (\mathsf{member}\ ys)\ xs \\
\mathsf{minus}\ xs\ ys &= \mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}\ ys)\ xs
\end{aligned}
$$

The advantage of this representation is that insert is a constant time operation but, in the worst case, member, delete and union have linear complexity and meet and minus have quadratic complexity.

5.2.1 Alternating Split

Let us now consider obfuscating the set operations using the alternating split. We state the operations (without proof) using the definitions from Section 4.3.2 — more detail can be found in [20].

We can define the empty set as $\langle [\,], [\,] \rangle_{asp}$ and

$$\mathsf{isEmpty}_{asp}\ \langle l, r \rangle_{asp} = \mathsf{isEmpty}\ l \wedge \mathsf{isEmpty}\ r$$

Using the definition of cons_{asp}, we define

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

As with insert this is a constant time operation. Since member is analogous to elem then we can define

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

We can show that member_{asp} satisfies:

$$\mathsf{member}_{asp}\ xsp\ a = \mathsf{member}\ (\mathsf{unsplit}_{asp}\ xsp)\ a \tag{5.1}$$

The definition of union_{asp} is:

$$\mathsf{union}_{asp}\ xs\ ys = xs +\!\!+_{asp}\ ys$$

These last two operations have linear complexity.

Using the definition of filter_{asp} we can define:

$$
\begin{aligned}
\mathsf{delete}_{asp}\ a\ xs &= \mathsf{filter}_{asp}\ (\neq a)\ xs \\
\mathsf{meet}_{asp}\ xs\ ys &= \mathsf{filter}_{asp}\ (\mathsf{member}_{asp}\ ys)\ xs \\
\mathsf{minus}_{asp}\ xs\ ys &= \mathsf{filter}_{asp}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ys)\ xs
\end{aligned}
$$

The operation delete_{asp} has linear complexity and the other two have quadratic complexity.

5.3 Strictly Ordered Lists

We now represent sets by lists which are in strictly-increasing order. If xs is a strictly-increasing ordered list then it has a data-type invariant as follows:

$$dti \equiv ((\forall m :: \mathbf{N}; n :: \mathbf{N}) \bullet 0 \leq m < n < |xs| \Rightarrow xs \;!!\; m < xs \;!!\; n) \quad (5.2)$$

The definitions of empty and isEmpty are the same as before. We can define:

$$\begin{aligned} \mathsf{member}\ xs\ a &= \text{if } \mathsf{isEmpty}\ zs \text{ then } \mathit{False} \text{ else } (a == \mathsf{head}\ zs) \\ &\quad \text{where } zs = \mathsf{dropWhile}\ (< a)\ xs \\ \mathsf{insert}\ a\ xs &= ys + (\text{if } \mathsf{isEmpty}\ zs \lor \mathsf{head}\ zs \neq a \text{ then } a : zs \text{ else } zs) \\ &\quad \text{where } (ys, zs) = \mathsf{span}\ (< a)\ xs \\ \mathsf{delete}\ a\ xs &= ys + (\text{if } \mathsf{isEmpty}\ zs \lor \mathsf{head}\ zs \neq a \text{ then } zs \text{ else } \mathsf{tail}\ zs) \\ &\quad \text{where } (ys, zs) = \mathsf{span}\ (< a)\ xs \end{aligned}$$

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

In [7, Page 267], the union operation is defined for this representation:

```
union [ ] ys      = ys
union xs [ ]      = xs
union (x : xs) (y : ys)
  | x < y         = x : union xs (y : ys)
  | x == y        = x : union xs ys
  | otherwise     = y : union (x : xs) ys
```

For the definitions of union, meet and minus we can exploit the fact that we have an ordering. Since the lists are ordered (Invariant (5.2)), we can walk through both lists in order, taking the appropriate action at each stage. Thus the definitions of minus and meet are:

```
minus [ ] ys      = [ ]
minus xs [ ]      = xs
minus (x : xs) (y : ys)
  | x < y         = x : minus xs (y : ys)
  | x == y        = minus xs ys
  | otherwise     = minus (x : xs) ys

meet [ ] ys   = [ ]
meet xs [ ]   = [ ]
meet (x : xs) (y : ys)
  | x < y         = meet xs (y : ys)
  | x == y        = x : meet xs ys
  | otherwise     = meet (x : xs) ys
```

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 — see Section 5.4 for a discussion of the insert operation.

5.3.1 Alternating Split

If we wish to use the alternating split with ordered lists then we need to strengthen Invariant (5.3). The representation $xs \rightsquigarrow \langle l, r\rangle_{asp}$ satisfies:

$$(|r| \leq |l| \leq |r| + 1) \wedge (l \trianglelefteq xs) \wedge (r \trianglelefteq xs) \tag{5.3}$$

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.

The definitions of empty_{asp}, $\mathsf{isEmpty}_{asp}$ and member_{asp} are the same as for unordered lists. The definition of delete_{asp} is:

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

Note that in the definition of delete_{asp} (and insert_{asp}), we use $\mathsf{member}\ lz\ a$ instead of $\mathsf{member}\ l\ a$ — by the definition of lz, $\mathsf{member}\ lz\ a$ reduces to checking whether $\mathsf{head}\ lz == a$. The number of steps for computing delete is proportional to $|ly| + |ry|$ — so it has linear complexity. The definition of insert_{asp} is:

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

This definition has four conditions. In the first and third conditions we know that a is a member of the split list and so we do not insert a — instead we just have to reconstruct the split list. In the other two cases, we have to insert a into the split list and so we use the definition of $+\!\!+_{asp}$ to tell us where to place the element a.

This version of insert_{asp} is still linear time — the only extra work is a check that $|ly| == |ry|$. The proof of correctness of insert_{asp} can be found in Appendix B and the correctness of delete_{asp} can be found in [20].

Using the definition of minus with ordered lists, we can easily define minus_{asp} as follows:

$$
\begin{array}{l}
\mathsf{minus}_{asp}\ \langle [\,], [\,]\rangle_{asp}\ ys\ \ = \langle [\,], [\,]\rangle_{asp} \\
\mathsf{minus}_{asp}\ xs\ \langle [\,], [\,]\rangle_{asp}\ \ = xs \\
\mathsf{minus}_{asp}\ \langle x : l_0, r_0\rangle_{asp}\ \langle y : l_1, r_1\rangle_{asp} \\
\quad \left|\ \begin{array}{ll}
x < y & = \mathsf{cons}_{asp}\ x\ (\mathsf{minus}_{asp}\ \langle r_0, l_0\rangle_{asp}\ \langle y : l_1, r_1\rangle_{asp}) \\
x == y & = (\mathsf{minus}_{asp}\ \langle r_0, l_0\rangle_{asp}\ \langle r_1, l_1\rangle_{asp}) \\
\text{otherwise} & = (\mathsf{minus}_{asp}\ \langle x : l_0, r_0\rangle_{asp}\ \langle r_1, l_1\rangle_{asp})
\end{array}\right.
\end{array}
$$

and the definitions of union_{asp} and meet_{asp} are similar. This definition of minus_{asp} follows a similar pattern to the unobfuscated version. This operation is

not very 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 components.

Since filter preserves ordering (*i.e.* $(\mathsf{filter}\ p\ xs) \trianglelefteq xs$) then we can use the definition for minus for unordered lists:

$$\mathsf{minus}\ xs\ ys = \mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}\ ys)\ xs$$

(note that this definition has quadratic complexity).

Using this, we derive an operation minus_{asp} that satisfies:

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

The right-hand of this equation becomes

$$\mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}\ (\mathsf{unsplit}_{asp}\ ysp))\ (\mathsf{unsplit}_{asp}\ xsp))$$

Using Equation (5.1), we obtain

$$\begin{aligned} &\mathsf{minus}_{asp}\ xsp\ ysp = \\ &\quad \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ xsp)) \end{aligned} \tag{5.4}$$

We derive minus_{asp} by induction on xsp.

Base Case Suppose that $xsp = \langle [\,], [\,] \rangle_{asp}$ then

$$\begin{aligned}
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ xsp)) \\
= & \quad \{\text{definition of } xsp\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ \langle [\,], [\,] \rangle_{asp})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ [\,]) \\
= & \quad \{\text{definition of } \mathsf{filter}\} \\
& \mathsf{split}_{asp}\ [\,] \\
= & \quad \{\text{definition of } \mathsf{split}_{asp}\} \\
& \langle [\,], [\,] \rangle_{asp}
\end{aligned}$$

Step Case Let $xsp = \langle x : xl, xr \rangle_{asp}$ and for the induction hypothesis, we suppose that $\langle xr, xl \rangle_{asp}$ satisfies Equation (5.4). We have two subcases.

Subcase 1 Suppose that $\mathsf{member}_{asp}\ ysp\ x = \mathit{False}$.

$$\begin{aligned}
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ xsp)) \\
= & \quad \{\text{definition of } xsp\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ \langle x : xl, xr \rangle_{asp})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ x : (\mathsf{unsplit}_{asp}\ \langle xr, xl \rangle_{asp})) \\
= & \quad \{\text{definition of } \mathsf{filter} \text{ with } \mathsf{member}_{asp}\ ysp\ x = \mathit{False}\} \\
& \mathsf{split}_{asp}\ (x : (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ \langle xr, xl \rangle_{asp})))
\end{aligned}$$

$$
\begin{aligned}
= &\quad \{\text{Equation (4.16)}\} \\
& \mathsf{cons}_{asp}\ x\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ \langle xr, xl \rangle_{asp})) \\
= &\quad \{\text{induction hypothesis}\} \\
& \mathsf{cons}_{asp}\ x\ (\mathsf{minus}\ \langle xr, xl \rangle_{asp}\ ysp)
\end{aligned}
$$

Subcase 2 Suppose that $\mathsf{member}_{asp}\ ysp\ x = True$.

$$
\begin{aligned}
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ xsp)) \\
= &\quad \{\text{definition of } xsp\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ \langle x : xl, xr \rangle_{asp})) \\
= &\quad \{\text{definition of } \mathsf{unsplit}\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ x : (\mathsf{unsplit}_{asp}\ \langle xr, xl \rangle_{asp})) \\
= &\quad \{\text{definition of } \mathsf{filter} \text{ with } \mathsf{member}_{asp}\ ysp\ x = False\} \\
& \mathsf{split}_{asp}\ (\mathsf{filter}\ (\mathsf{not} \cdot \mathsf{member}_{asp}\ ysp)\ (\mathsf{unsplit}_{asp}\ \langle xr, xl \rangle_{asp})) \\
= &\quad \{\text{induction hypothesis}\} \\
& \mathsf{minus}\ \langle xr, xl \rangle_{asp}\ ysp
\end{aligned}
$$

Putting all the cases gives the following definition:

$$
\begin{aligned}
\mathsf{minus}_{asp}\ \langle [\,], [\,] \rangle_{asp}\ ysp &= \langle [\,], [\,] \rangle_{asp} \\
\mathsf{minus}_{asp}\ \langle x : xl, xr \rangle_{asp}\ ysp &= \text{if } \mathsf{member}_{asp}\ ysp\ x \\
&\qquad \text{then } \mathsf{minus}_{asp}\ \langle xr, xl \rangle_{asp}\ ysp \\
&\qquad \text{else } \mathsf{cons}_{asp}\ x\ (\mathsf{minus}_{asp}\ \langle xr, xl \rangle_{asp}\ ysp)
\end{aligned}
$$

5.3.2 Block Split

Let us now consider how we can obfuscate the set operations using the block split. As we are now working with ordered lists, we need to strengthen Invariant (4.18). The representation $xs \rightsquigarrow \langle l, r \rangle_{b(k)}$ satisfies:

$$((|r| = 0 \wedge |l| < k) \vee (|l| = k)) \wedge (l \trianglelefteq xs) \wedge (r \trianglelefteq xs) \tag{5.5}$$

which ensures that the block split preserves ordering. Using the definition of $\mathsf{split}_{b(k)}$, we can easily check that this invariant holds and so we can use this split with ordered lists.

As with the alternating split, we state the operations for ordered lists — the proofs of correctness can be found in [20]. The $\mathsf{member}_{b(k)}$ operation is the same as usual:

$$\mathsf{member}_{b(k)}\ \langle l, r \rangle_{b(k)}\ a \quad = \quad \mathsf{member}\ l\ a \vee \mathsf{member}\ r\ a$$

For $\mathsf{insert}_{b(k)}$, we may have to break the list l into $ls +\!\!+ [l']$, where l' is the last element of l (assuming that l is not empty). Note that since $|l| \leq k$ then breaking l into ls and l' is a constant operation.

$$
\begin{aligned}
& \mathsf{insert}_{b(k)}\ a\ \langle l, r \rangle_{b(k)} \\
& \quad \left|\ \begin{aligned}
& \mathsf{member}_{b(k)}\ \langle l, r \rangle_{b(k)}\ a && = \langle l, r \rangle_{b(k)} \\
& |l| < k && = \langle \mathsf{insert}\ a\ l, r \rangle_{b(k)} \\
& l' < a && = \langle l, \mathsf{insert}\ a\ r \rangle_{b(k)} \\
& \text{otherwise} && = \langle \mathsf{insert}\ a\ ls, l' : r \rangle_{b(k)}
\end{aligned} \right. \\
& \qquad\qquad \text{where } \ ls = \mathsf{init}\ l \\
& \qquad\qquad\qquad\quad\; l' = \mathsf{last}\ l
\end{aligned}
$$

As $|l| \leq k$ then the insert operation still has linear complexity in the worst case.

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

$$
\begin{array}{l}
\mathsf{delete}_{b(k)}\ a\ \langle l, r\rangle_{b(k)} \\
\quad \left|\begin{array}{ll}
\mathsf{member}\ a\ l \wedge r == [\,] & = \langle \mathsf{delete}\ a\ l, r\rangle_{b(k)} \\
\mathsf{member}\ a\ l \wedge r \neq [\,] & = \langle (\mathsf{delete}\ a\ l) +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r\rangle_{b(k)} \\
\mathsf{otherwise} & = \langle l, \mathsf{delete}\ a\ r\rangle_{b(k)}
\end{array}\right.
\end{array}
$$

Again, in the worst case, $\mathsf{delete}_{b(k)}$ has linear complexity.

This definitions of $\mathsf{union}_{b(k)}$, $\mathsf{meet}_{b(k)}$ and $\mathsf{minus}_{b(k)}$ are similar to the definitions stated at the start of Section 5.3. As an example, here is the definition of $\mathsf{meet}_{b(k)}$:

$$
\begin{array}{l}
\mathsf{meet}_{b(k)}\ \langle [\,], [\,]\rangle_{b(k)}\ ys = \langle [\,], [\,]\rangle_{b(k)} \\
\mathsf{meet}_{b(k)}\ xs\ \langle [\,], [\,]\rangle_{b(k)} = \langle [\,], [\,]\rangle_{b(k)} \\
\mathsf{meet}_{b(k)}\ (\mathsf{cons}_{b(k)}\ x\ \langle l_0, r_0\rangle_{b(k)})\ (\mathsf{cons}_{b(k)}\ y\ \langle l_1, r_1\rangle_{b(k)}) \\
\quad \left|\begin{array}{lll}
x == y & = & \mathsf{cons}_{b(k)}\ x\ (\mathsf{meet}_{b(k)}\ \langle l_0, r_0\rangle_{b(k)}\ \langle l_1, r_1\rangle_{b(k)}) \\
x < y & = & \mathsf{meet}_{b(k)}\ \langle l_0, r_0\rangle_{b(k)}\ (\mathsf{cons}_{b(k)}\ y\ \langle l_1, r_1\rangle_{b(k)}) \\
\text{otherwise} & = & \mathsf{meet}_{b(k)}\ (\mathsf{cons}_{b(k)}\ x\ \langle l_0, r_0\rangle_{b(k)})\ \langle l_1, r_1\rangle_{b(k)}
\end{array}\right.
\end{array}
$$

5.4 Comparing different definitions

When using ordered lists we defined insert as follows:

$$
\begin{array}{l}
\mathsf{insert}\ a\ xs = ys\ +\!\!+ (\text{if } \mathsf{isEmpty}\ zs \vee \mathsf{head}\ zs \neq a \text{ then } a : zs \text{ else } zs) \\
\qquad\qquad \text{where } (ys, zs) = \mathsf{span}\ (< a)\ xs
\end{array}
$$

As an alternative, we could define:

$$
\begin{array}{l}
\mathsf{insert1}\ a\ [\,] = [a] \\
\mathsf{insert1}\ a\ (x : xs) \\
\quad \left|\begin{array}{ll}
a > x & = x : (\mathsf{insert1}\ a\ xs) \\
a == x & = x : xs \\
\mathsf{otherwise} & = a : x : xs
\end{array}\right.
\end{array}
$$

We chose the first formulation as we claimed that it had a more interesting obfuscation. If we obfuscate both of these operations by using the alternating split then we obtain:

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

and

$$
\begin{array}{l}
\mathsf{insert1}_{asp}\ a\ \langle [\,], [\,]\rangle_{asp} = \langle [a], [\,]\rangle_{asp} \\
\mathsf{insert1}_{asp}\ a\ \langle x : l, r\rangle_{asp} \\
\quad \left|\begin{array}{ll}
a > x & = \mathsf{cons}_{asp}\ x\ (\mathsf{insert1}_{asp}\ a\ \langle r, l\rangle_{asp}) \\
a == x & = \langle x : l, r\rangle_{asp} \\
\mathsf{otherwise} & = \langle a : r, x : l\rangle_{asp}
\end{array}\right.
\end{array}
$$

We saw the definition of insert_{asp} in Section 5.3.1.

These four operations all have linear complexity. The definition of insert1 is generally the most efficient as it passes through the list only once. For $\mathsf{insert1}_{asp}$ we need to swap the split components — this can be implemented in constant time. The other two operations require walking through the initial part of the list (*i.e.* the elements less than a) twice. In the definition of insert_{asp} we have extra tests and so insert_{asp} is slightly less efficient than insert.

While it is certainly true that $\mathsf{insert1}_{asp}$ is syntactically similar to insert, is it less obfuscated? We prove that each of the four operations satisfy:

$$f\ a\ (f\ a\ xs) = f\ a\ xs \tag{5.6}$$

which corresponds to the assertion:

$$\mathsf{insert}\ a\ (\mathsf{insert}\ a\ xs) = \mathsf{insert}\ a\ xs$$

(*i.e.* that $\mathsf{insert}\ a$ is idempotent).

5.4.1 First Definition

We prove Equation (5.6) for insert — so let $(ys, zs) = \mathsf{span}\ (< a)\ xs$ and we have two cases.

Case 1 Suppose that $zs = [\,] \vee \mathsf{head}\ zs \neq a$. Then

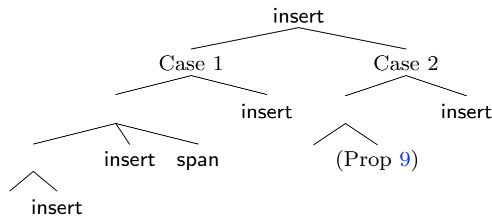
$$
\begin{aligned}
& \mathsf{insert}\ a\ (\mathsf{insert}\ a\ xs) \\
= \quad & \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ (ys +\!\!+ (a : zs)) \\
= \quad & \{\text{definition of } \mathsf{insert};\ \mathsf{span}\ (< a)\ (ys +\!\!+ (a : zs)) = (ys, a : zs)\} \\
& ys +\!\!+ (a : zs) \\
= \quad & \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ xs
\end{aligned}
$$

Case 2 Suppose that $\mathsf{head}\ zs = a$, then

$$
\begin{aligned}
& \mathsf{insert}\ a\ (\mathsf{insert}\ a\ xs) \\
= \quad & \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ (ys +\!\!+ zs) \\
= \quad & \{\text{property of } \mathsf{span} \text{ (Property 9)}\} \\
& \mathsf{insert}\ a\ xs
\end{aligned}
$$

Note that Property 9 is proved in Appendix B.1.

We can draw the following proof tree:



For this proof tree, the cost and height are as follows:

$$\begin{aligned} \mathcal{C}\,(\mathsf{insert}) &= 5 + \mathcal{C}\,(\text{Prop } 9) = 9 \\ \text{and } \mathcal{H}\,(\mathsf{insert}) &= 3 + \mathcal{H}\,(\text{Prop } 9) = 6 \end{aligned}$$

5.4.2 Second Definition

We prove Equation (5.6) for insert1 by induction on xs.

Base Case Suppose that $xs = [\,]$. Then

$$\begin{aligned} & \mathsf{insert1}\ a\ (\mathsf{insert1}\ a\ [\,]) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & \mathsf{insert1}\ a\ [a] \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & [a] \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & \mathsf{insert1}\ a\ [\,] \end{aligned}$$

Step Case Suppose that $xs = y : ys$ and for the induction hypothesis, we assume that ys satisfies Equation (5.6). We have three subcases.

Subcase 1 Suppose that $a > y$ then

$$\begin{aligned} & \mathsf{insert1}\ a\ (\mathsf{insert1}\ a\ (y : ys)) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & \mathsf{insert1}\ a\ (y : (\mathsf{insert1}\ a\ ys)) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & y : (\mathsf{insert1}\ a\ (\mathsf{insert1}\ a\ ys)) \\ = &\quad \{\text{induction hypothesis}\} \\ & y : (\mathsf{insert1}\ a\ ys) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & \mathsf{insert1}\ a\ (y : ys) \end{aligned}$$

Subcase 2 Suppose that $a = y$ then

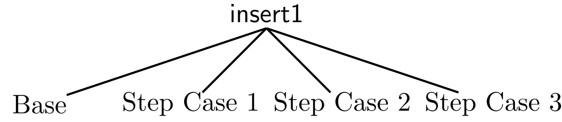
$$\begin{aligned} & \mathsf{insert1}\ a\ (\mathsf{insert1}\ a\ (y : ys)) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & \mathsf{insert1}\ a\ (y : ys) \end{aligned}$$

Subcase 3 Suppose that $a < y$ then

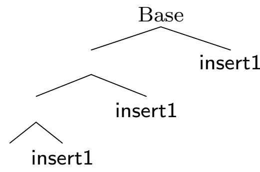
$$\begin{aligned} & \mathsf{insert1}\ a\ (\mathsf{insert1}\ a\ (y : ys)) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \\ & \mathsf{insert1}\ a\ (a : y : ys) \\ = &\quad \{\text{definition of } \mathsf{insert1}\} \end{aligned}$$

$$
\begin{aligned}
& a : y : ys \\
= & \quad \{\text{definition of } \mathsf{insert1}\} \\
& \mathsf{insert1}\ a\ (y : ys)
\end{aligned}
$$

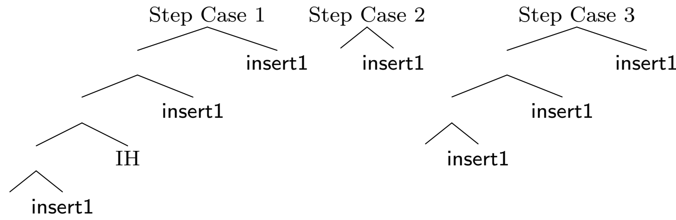
The tree for this proof is:



where



and



So,

$$
\begin{aligned}
\mathcal{C}\,(\mathsf{insert1}) &= 11 \\
\text{and } \mathcal{H}\,(\mathsf{insert1}) &= 5
\end{aligned}
$$

5.4.3 Third Definition

We prove Equation (5.6) for insert_{asp} — so we let $xs = \langle l, r\rangle_{asp}$ and

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

We have four cases.

Case 1 Suppose that $\mathsf{member}\ lz\ a$. By (B.3 $\Rightarrow$) (proved in Appendix B.1) $|ly| = |ry|$ and so,

$$
\begin{aligned}
& \mathsf{insert}_{asp}\ a\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{insert}_{asp}\ a\ (\langle ly, ry\rangle_{asp} +\!\!+_{asp} \langle lz, rz\rangle_{asp}) \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |ly| = |ry|\} \\
& \mathsf{insert}_{asp}\ a\ (\langle ly +\!\!+ lz, ry +\!\!+ rz\rangle_{asp}) \\
= & \quad \{\text{property of } \mathsf{span} \text{ (Property 9)}\} \\
& \mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}
\end{aligned}
$$

Case 2 Suppose that $\neg(\mathsf{member}\ lz\ a)$ and $|ly| = |ry|$

$$
\begin{aligned}
& \mathsf{insert}_{asp}\ a\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{insert}_{asp}\ a\ (\langle ly, ry\rangle_{asp} +\!\!+_{asp} \langle a : rz, lz\rangle_{asp}) \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |ly| = |ry|\} \\
& \mathsf{insert}_{asp}\ a\ \langle ly +\!\!+ (a : rz), ry +\!\!+ lz\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp} \text{ with } \mathsf{member}\ (a : rz)\ a\} \\
& \langle ly +\!\!+ (a : rz), ry +\!\!+ lz\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}
\end{aligned}
$$

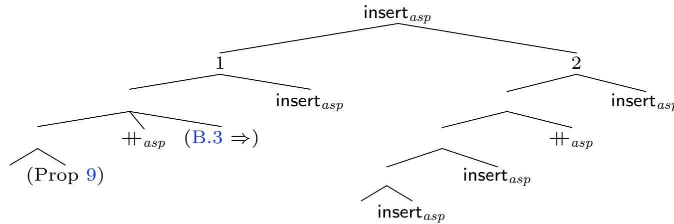
Case 3 Suppose that $\mathsf{member}\ rz\ a$ then by (B.4 $\Rightarrow$) $|ly| \neq |ry|$.

$$
\begin{aligned}
& \mathsf{insert}_{asp}\ a\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{insert}_{asp}\ a\ (\langle ly, ry\rangle_{asp} +\!\!+_{asp} \langle rz, lz\rangle_{asp}) \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |ly| \neq |ry|\} \\
& \mathsf{insert}_{asp}\ a\ (\langle ly +\!\!+ lz, ry +\!\!+ rz\rangle_{asp}) \\
= & \quad \{\text{property of } \mathsf{span} \text{ (Property 9)}\} \\
& \mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}
\end{aligned}
$$

Case 4 Suppose that $\neg(\mathsf{member}\ rz\ a)$ and $|ly| \neq |ry|$

$$
\begin{aligned}
& \mathsf{insert}_{asp}\ a\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{insert}_{asp}\ a\ (\langle ly, ry\rangle_{asp} +\!\!+_{asp} \langle a : lz, rz\rangle_{asp}) \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |ly| \neq |ry|\} \\
& \mathsf{insert}_{asp}\ a\ \langle ly +\!\!+ rz, ry +\!\!+ (a : lz)\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp} \text{ with } \mathsf{member}\ (a : lz)\ a\} \\
& \langle ly +\!\!+ rz, ry +\!\!+ (a : lz)\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}
\end{aligned}
$$

Below is the proof tree for Cases 1 and 2:



The proof tree for Cases 3 and 4 is similar (except that (B.3 $\Rightarrow$) is replaced by (B.4 $\Rightarrow$)). Both of (B.3 $\Rightarrow$) and (B.4 $\Rightarrow$) use (B.1) and so

$$\mathcal{L}\,(\mathsf{insert}_{asp}) = \{(\text{B.1}),\ (\text{Prop } 9)\}$$

Note that $\mathcal{C}\,(\text{B.3} \Rightarrow) = 6 + \mathcal{C}\,(\text{B.1})$, where $\mathcal{C}\,(\text{B.1}) = 12$, and thus

$$\begin{array}{rl} \mathcal{C}\,(\mathsf{insert}_{asp}) & = (2 \times (7+6)) + \mathcal{C}\,(\text{B.1}) + \mathcal{C}\,(\text{Prop } 9) = 42 \\ \text{and } \mathcal{H}\,(\mathsf{insert}_{asp}) & = \max(5, 3 + \mathcal{H}\,(\text{B.3} \Rightarrow), 4 + \mathcal{H}\,(\text{Prop } 9)) = 16 \end{array}$$

Thus the proof for insert_{asp} is much more complicated than for insert.

5.4.4 Last Definition

We prove Equation (5.6) for $\mathsf{insert1}_{asp}$ by induction on xs.

Base Case Suppose that $xs = \langle[\,],[\,]\rangle_{asp}$ then

$$\begin{array}{ll} & \mathsf{insert1}_{asp}\ a\ (\mathsf{insert1}_{asp}\ a\ \langle[\,],[\,]\rangle_{asp}) \\ = & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\ & \mathsf{insert1}_{asp}\ a\ \langle[a],[\,]\rangle_{asp} \\ = & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\ & \langle[a],[\,]\rangle_{asp} \\ = & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\ & \mathsf{insert1}_{asp}\ a\ \langle[\,],[\,]\rangle_{asp} \end{array}$$

Step Case Suppose that $xs = \langle y : l, r\rangle_{asp}$ and for the induction hypothesis, we assume that $\langle r, l\rangle_{asp}$ satisfies (5.6). Let $\langle p, q\rangle_{asp} = \mathsf{insert1}_{asp}\ a\ \langle r, l\rangle_{asp}$. We have three subcases

Subcase 1 Suppose that $a > y$ then

$$\begin{array}{ll} & \mathsf{insert1}_{asp}\ a\ (\mathsf{insert1}_{asp}\ a\ \langle y : l, r\rangle_{asp}) \\ = & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\ & \mathsf{insert1}_{asp}\ a\ (\mathsf{cons}_{asp}\ y\ (\mathsf{insert1}_{asp}\ a\ \langle r, l\rangle_{asp})) \\ = & \quad \{\text{definition of } \langle p, q\rangle_{asp}\} \\ & \mathsf{insert1}_{asp}\ a\ (\mathsf{cons}_{asp}\ y\ \langle p, q\rangle_{asp}) \\ = & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\ & \mathsf{insert1}_{asp}\ a\ \langle y : q, p\rangle_{asp} \\ = & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\ & \mathsf{cons}_{asp}\ y\ (\mathsf{insert1}_{asp}\ a\ \langle p, q\rangle_{asp}) \\ = & \quad \{\text{definition of } \langle p, q\rangle_{asp}\} \\ & \mathsf{cons}_{asp}\ y\ (\mathsf{insert1}_{asp}\ a\ (\mathsf{insert1}\ a\ \langle r, l\rangle_{asp})) \\ = & \quad \{\text{induction hypothesis}\} \\ & \mathsf{cons}_{asp}\ y\ (\mathsf{insert1}_{asp}\ a\ \langle r, l\rangle_{asp}) \\ = & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\ & \mathsf{insert1}_{asp}\ a\ \langle y : l, r\rangle_{asp} \end{array}$$

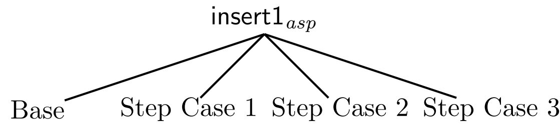
Subcase 2 Suppose that $a = y$ then

$$
\begin{aligned}
& \mathsf{insert1}_{asp}\ a\ (\mathsf{insert1}_{asp}\ a\ \langle y : l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\
& \mathsf{insert1}_{asp}\ a\ \langle y : l, r\rangle_{asp}
\end{aligned}
$$

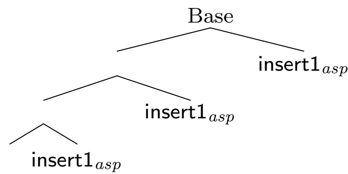
Subcase 3 Suppose that $a < y$ then

$$
\begin{aligned}
& \mathsf{insert1}_{asp}\ a\ (\mathsf{insert1}\ a\ \langle y : l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\
& \mathsf{insert1}_{asp}\ a\ \langle a : r, x : l\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\
& \langle a : r, x : l\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{insert1}_{asp}\} \\
& \mathsf{insert1}\ a\ \langle y : l, r\rangle_{asp}
\end{aligned}
$$

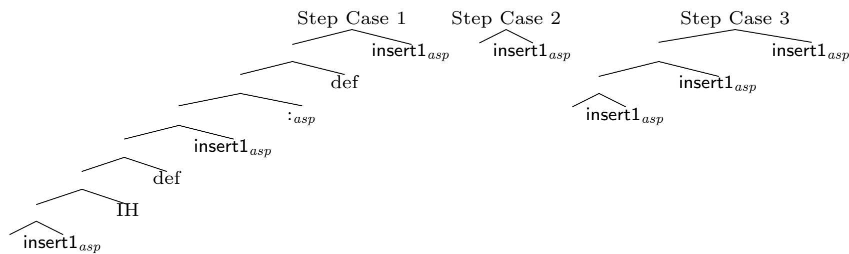
Here is a tree for this proof:



where



and



So,

$$
\begin{aligned}
\mathcal{C}\,(\mathsf{insert1}_{asp}) &= 14 \\
\text{and } \mathcal{H}\,(\mathsf{insert1}_{asp}) &= 8
\end{aligned}
$$

5.5 Conclusions

The following table summarises the results from the proof trees in the last section:

Version	$\mathcal{C}$	$\mathcal{H}$
insert	9	6
insert1	11	5
insert_{asp}	42	16
$\mathsf{insert1}_{asp}$	14	8

In Section 5.3 we claimed that the definition of insert which uses span produces a more interesting obfuscation. We can see that this claim is supported by the results in the table. The definition of $\mathsf{insert1}_{asp}$ has a similar structure to insert1 while the definition of insert_{asp} is more complicated than insert. Although the two unobfuscated definitions are similar in the difficulty of the proof, there is a much greater difference between the proofs for the obfuscated operations. As with the $+\!\!+$ operation discussed in Appendix A we can see that an obfuscated operation seems to be a good assertion obfuscation if the definition has a different structure to the unobfuscated operation.

Chapter 6

The Matrix Obfuscated

The Doctor: "I deny this reality.
This reality is a computation matrix."

Doctor Who — *The Deadly Assassin* (1976)

The next data-type we consider is *matrices* and we briefly develop some splits for this data-type — more details can be found in [19]. A matrix $\mathbf{M}$ which has r rows and c columns with elements of type α will be denoted by $\mathbf{M}^{r \times c}(\alpha)$. We write $\mathbf{M}(i, j)$ to denote the access function which denotes the element located at the ith row and the jth column. An indexer for a matrix $\mathbf{M}^{r \times c}$ is $[0..r) \times [0..c)$. Thus since matrices are examples of IDTs, we can perform splits.

6.1 Example

Matrices are used for a variety of applications in Computer Science. One such application is computer graphics in which coordinates have type Float^3. We use matrices, taking $\alpha = \mathit{Float}$, to represent transformations, such as rotations and scaling and column vectors to represent coordinates. So that we can define a translation matrix, we use *homogeneous coordinates* [25, Chapter 5]; we write the point (x, y, z) as (x, y, z, w) where $w \neq 0$. Two homogeneous coordinates are equal if one is a multiple of the other. When using 3D homogeneous coordinates we need to have 4×4 transformation matrices.

Let us consider a scaling matrix which has the form:

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To scale the point (x, y, z, w), we compute

$$S(s_x, s_y, s_z) \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x \times s_x \\ y \times s_y \\ z \times s_z \\ w \end{pmatrix}$$

How could we obfuscate this operation? Consider the matrix

$$\mathbf{A} = \begin{pmatrix} a_{(0,0)} & \cdots & a_{(0,3)} \\ \vdots & \ddots & \vdots \\ a_{(3,0)} & \cdots & a_{(3,3)} \end{pmatrix}$$

In each of the first three rows, we choose which of the first three columns to place each scaling factor and the last number in each row indicates the choice of column. The rest of the matrix can be filled with junk. So we can define:

$$S(s_x, s_y, s_z) \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x \times a_{(0, a_{(0,3)} \bmod 3)} \\ y \times a_{(1, a_{(1,3)} \bmod 3)} \\ z \times a_{(2, a_{(2,3)} \bmod 3)} \\ w \end{pmatrix}$$

6.2 Matrix data-type

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

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

We represent $\mathbf{M}^{r\times c}$ by a list of lists *mss* where $|mss| = r$ and each list in *mss* has length c.

Our data-type for matrices is shown in Figure 6.1. Note that we saw the operation cross in Section 3.2.3:

$$\mathsf{cross}\ (f_1, f_2)\ (x_1, x_2) = (f_1\ x_1, f_2\ x_2)$$

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

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

for matrices $\mathbf{M}^{r\times c}$, $\mathbf{N}^{r\times c}$ and $\mathbf{P}^{c\times d}$ with $i :: [0..r)$, $j :: [0..c)$ and $k :: [0..d)$.

For simplicity, we assume the our element type is $\mathbb{Z}$ and so we simply write $\mathbf{M}$ instead of $\mathbf{M}(\mathbb{Z})$.

We assume that basic arithmetic operations take constant time and so the computational complexities of add $\mathbf{M}$ $\mathbf{N}$, scale s $\mathbf{M}$ and transpose $\mathbf{M}$ are all $r\times c$ and the complexity of mult $\mathbf{M}$ $\mathbf{P}$ is $r\times c\times d$. In fact, to reduce the number of multiplications in the calculation of mult, we could use *Strassen's algorithm* [13, Section 28.2] which performs matrix multiplication by establishing simultaneous equations. The algorithm requires starting with a $2n\times 2n$ matrix and splitting

Matrix (α)

$Matrix\ \alpha ::= List\ (List\ \alpha) \wedge (\forall\ mss :: Matrix\ \alpha \bullet \mathsf{valid}\ mss)$

$$\begin{aligned}
\mathsf{scale} &:: \alpha \to Matrix\ \alpha \to Matrix\ \alpha \\
\mathsf{add} &:: (Matrix\ \alpha, Matrix\ \alpha) \to Matrix\ \alpha \\
\mathsf{transpose} &:: Matrix\ \alpha \to Matrix\ \alpha \\
\mathsf{mult} &:: (Matrix\ \alpha, Matrix\ \alpha) \to Matrix\ \alpha \\
(!!!) &:: Matrix\ \alpha \to \mathbb{N} \times \mathbb{N} \to \alpha
\end{aligned}$$

$$\begin{aligned}
\mathsf{transpose} \cdot \mathsf{transpose} &= id \\
\mathsf{transpose} \cdot \mathsf{add} &= \mathsf{add} \cdot \mathsf{cross}\,(\mathsf{transpose}, \mathsf{transpose}) \\
\mathsf{transpose} \cdot (\mathsf{scale}\ s) &= (\mathsf{scale}\ s) \cdot \mathsf{transpose} \\
(\mathsf{add}\ (\mathbf{M}, \mathbf{N}))\ !!!\ (r, c) &= (\mathbf{M}\ !!!\ (r, c)) + (\mathbf{N}\ !!!\ (r, c)) \\
\mathsf{transpose}\ (\mathsf{mult}\ (\mathbf{M}, \mathbf{N})) &= \mathsf{mult}\ (\mathsf{transpose}\ \mathbf{N}, \mathsf{transpose}\ \mathbf{M}) \\
\mathsf{add}\ (\mathbf{M}, \mathbf{N}) &= \mathsf{add}\ (\mathbf{N}, \mathbf{M})
\end{aligned}$$

Figure 6.1: Data-type for Matrices

it into four $n \times n$ matrices but by padding a matrix with zeros, this method can be adapted for more general matrices.

We must ensure that when we obfuscate these operations we do not change the complexity.

6.2.1 Definitions of the operations

We now define operations to match the data-type given in Figure 6.1. Two of our definitions will use the function zipWith — the definition below contains a catch-all 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 infix operation $(\otimes)$ to correspond to zipWith $(+\!\!+)$.

We define the matrix operations functionally as follows:

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

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

6.3 Splitting Matrices

Since matrices are an IDT we can use the Function Splitting Theorem (Theorem 1 from Section 4.1.4). Can we express our matrix operations using functions h and ϕ^e? For scale $(\times s)$, we can take $h = (\times s)$ and $\phi^1 = id$; for transpose, $h = id$ 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 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 (4.5) is satisfied for any split. If, for some split sp,

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

then

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

and

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

Note that it is not surprising that scale satisfies the Function Splitting Theorem as it is defined in terms of map.

6.3.1 Splitting in squares

A simple matrix split is one which splits a square matrix into four matrices — two of which are square. Using this split we can give definitions of our operations for split matrices. Suppose that we have a square matrix $\mathbf{M}^{r\times r}$ and choose a positive integer k such that $k < n$. The choice function $ch(i,j)$ is defined as

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

which can be written as a single formula

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

where sgn is the signum function. The family of functions $\mathcal{F}$ is $\{f_0, f_1, f_2, f_3\}$ where

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

Again, we can write this in a single formula:

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

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

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

So if

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

then we can write

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

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

In Section 6.3.4, we will see that for this split

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

or, pictorially,

$$\left(\begin{array}{cc} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{array}\right)^T = \left(\begin{array}{cc} {\mathbf{M}_0}^T & {\mathbf{M}_2}^T \\ {\mathbf{M}_1}^T & {\mathbf{M}_3}^T \end{array}\right)$$

This operation has complexity $n \times n$.

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

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

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

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

By considering the product

$$\left(\begin{array}{cc} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{array}\right) \times \left(\begin{array}{cc} \mathbf{N}_0 & \mathbf{N}_1 \\ \mathbf{N}_2 & \mathbf{N}_3 \end{array}\right)$$

we obtain the following result:

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

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

6.3.2 Modelling the split in Haskell

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

$$SpMat\ \alpha \quad = \quad \langle Matrix\ \alpha,\ Matrix\ \alpha,\ Matrix\ \alpha,\ Matrix\ \alpha\,\rangle$$

to describe split matrices. Let us suppose that we have a square matrix mss with $\mathsf{dim}\ mss = (n, n)$. Then the representation

$$mss \leadsto \langle as, bs, cd, ds\rangle_{s(k)}$$

satisfies the invariant

$$\begin{array}{r} \mathsf{dim}\ as = (k,k) \wedge \mathsf{dim}\ bs = (k, n-k) \quad \wedge \\ \mathsf{dim}\ cs = (n-k, k) \wedge \mathsf{dim}\ ds = (n-k, n-k) \end{array} \qquad (6.1)$$

for some $k :: (0..n)$. The operation dim returns the dimensions of a matrix and 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

$$|\langle as, bs, cs, ds\rangle_{s(k)}|_{s(k)} = |as| + |cs|$$

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

$$\mathsf{split}_{s(k)} :: Matrix\ \alpha \rightarrow SpMat\ \alpha$$

that operates on square matrices and returns the corresponding split matrix. Thus $\mathsf{split}_{s(k)}$ must satisfy:

$$mss \leadsto \langle as, bs, cs, ds\rangle_{s(k)} \Leftrightarrow \mathsf{split}_{s(k)}\ mss = \langle as, bs, cs, ds\rangle_{s(k)}$$

For this split, we define

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

Since we can undo $\mathsf{splitAt}$ by using $+\!\!+$, we define the (left and right) inverse of $\mathsf{split}_{s(k)}$ 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 take $\mathsf{unsplit}_{s(k)}$ to be our abstraction function for this split. Thus

$$mss \leadsto \langle as, bs, cs, ds\rangle_{s(k)} \Leftrightarrow$$

$$mss = \mathsf{unsplit}_{s(k)}\ \langle as, bs, cs, ds\rangle_{s(k)} \ \wedge \left(\begin{array}{l} \mathsf{dim}\ as = (k,k) \wedge \\ \mathsf{dim}\ bs = (k, n-k) \wedge \\ \mathsf{dim}\ cs = (n-k, k) \wedge \\ \mathsf{dim}\ ds = (n-k, n-k) \end{array}\right)$$

6.3.3 Properties

To make derivations easier we will use some properties about map, ++ and zipWith. The proofs of some of these properties can in found in [19].

First, two properties of ++:

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

Next, we will need a way of concatenating two zipWiths together.

Property 6. Let $(\bowtie)$ be a function $\bowtie :: List\ \alpha \to List\ \beta \to List\ \gamma$ and suppose that we have lists as and cs of type $List\ \alpha$ and bs and ds of type $List\ \beta$ such that $|as| = |bs|$. Then:

$$\begin{aligned} &(\mathsf{zipWith}\ (\bowtie)\ as\ bs) ++ (\mathsf{zipWith}\ (\bowtie)\ cs\ ds) \\ &\quad = \mathsf{zipWith}\ (\bowtie)\ (as ++ cs)\ (bs ++ ds) \end{aligned}$$

Using this property, we can write an alternative definition for unsplit:

$$\mathsf{unsplit}\ \langle as, bs, cs, ds\rangle_{s(k)} = (as ++ cs)\ \otimes\ (bs ++ ds)$$

The next two properties link transpose and $\otimes$.

Property 7. For matrices mss and nss

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

We can see this property pictorially:

$$\left(\frac{mss}{nss}\right)^T = \left(\ mss^T \;\middle|\; nss^T\ \right)$$

Property 8. For matrices mss and nss where $|mss| = |nss|$

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

6.3.4 Deriving transposition

Using Equation (3.6) we can produce an equation that enables us to derive operations for split matrices:

$$\mathsf{op}_{s(k)} = \mathsf{split}_{s(k)} \cdot \mathsf{op} \cdot \mathsf{unsplit}_{s(k)} \tag{6.2}$$

Using this equation, we derive $\mathsf{transpose}_{s(k)}$ using the properties from Section 6.3.3.

$$\begin{aligned} &\quad \mathsf{transpose}_{s(k)}\ \langle a, b, c, d\rangle_{s(k)} \\ = &\quad \{\text{using Equation (6.2)}\} \\ &\quad \mathsf{split}_{s(k)}(\mathsf{transpose}\ (\mathsf{unsplit}_{s(k)}\ \langle a, b, c, d\rangle_{s(k)})) \\ = &\quad \{\text{definition of } \mathsf{unsplit}_{s(k)}\} \\ &\quad \mathsf{split}_{s(k)}\ (\mathsf{transpose}\ ((a\ \otimes\ b) ++ (c\ \otimes\ d))) \\ = &\quad \{\text{Property 7}\} \end{aligned}$$

$$
\begin{aligned}
& \mathsf{split}_{s(k)}\ ((\mathsf{transpose}\ (a \otimes b)) \otimes (\mathsf{transpose}\ (c \otimes d))) \\
= & \quad \{\text{Property } 8\} \\
& \mathsf{split}_{s(k)}\ ((\mathsf{transpose}\ a +\!\!+\ \mathsf{transpose}\ b) \otimes (\mathsf{transpose}\ c +\!\!+\ \mathsf{transpose}\ d)) \\
= & \quad \{\text{Property } 6,\ (\otimes) = \mathsf{zipWith}(+\!\!+)\} \\
& \mathsf{split}_{s(k)}\ ((\mathsf{transpose}\ a \otimes \mathsf{transpose}\ c) +\!\!+ \\
& \qquad\qquad (\mathsf{transpose}\ b \otimes \mathsf{transpose}\ d)) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{s(k)}\} \\
& \mathsf{split}_{s(k)}\ (\mathsf{unsplit}\ \langle \mathsf{transpose}\ a,\ \mathsf{transpose}\ c, \\
& \qquad\qquad\qquad \mathsf{transpose}\ b,\ \mathsf{transpose}\ d\rangle_{s(k)}) \\
= & \quad \{\mathsf{split}_{s(k)} \cdot \mathsf{unsplit}_{s(k)} = id\} \\
& \langle \mathsf{transpose}\ a,\ \mathsf{transpose}\ c,\ \mathsf{transpose}\ b,\ \mathsf{transpose}\ d\,\rangle_{s(k)}
\end{aligned}
$$

Hence,

$$
(\langle a, b, c, d\rangle_{s(k)})^T = \langle a^T, c^T, b^T, d^T\rangle_{s(k)} \tag{6.3}
$$

The derivation for $\mathsf{transpose}_{s(k)}$ is quite short and matches the form given in Section 6.3.1. Derivations for the other operations can be found in [19].

We could use the Function Splitting Theorem (Theorem 1), by taking $g = \mathsf{transpose}$, $h = id$ and $\phi^1(i,j) = (j,i)$ to define transposition for our split. If we let

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

then we can verify that Equations (4.4) and (4.5) are satisfied to give a definition that matches Equation (6.3).

We shall show that this operation satisfies the first assertion in Section 6.2, namely that $\mathsf{transpose}_{s(k)}$ is self-inverse:

$$
\mathsf{transpose}_{s(k)} \cdot \mathsf{transpose}_{s(k)} = id \tag{6.4}
$$

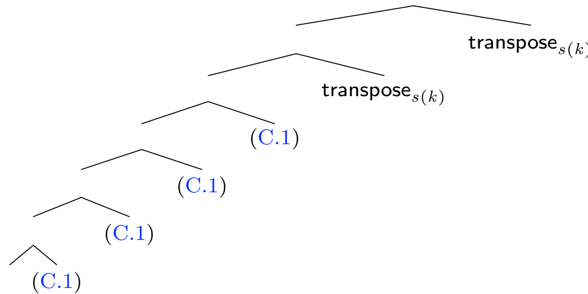
Proof. To prove this, we will need that $\mathsf{transpose}$ is self-inverse (Equation (C.1)) — the proof of (C.1) is in Appendix C.

$$
\begin{aligned}
& \mathsf{transpose}_{s(k)}\ (\mathsf{transpose}_{s(k)}\ \langle a, b, c, d\rangle_{s(k)}) \\
= & \quad \{\text{definition of } \mathsf{transpose}_{s(k)}\} \\
& \mathsf{transpose}_{s(k)}\ \langle \mathsf{transpose}\ a,\ \mathsf{transpose}\ c, \\
& \qquad\qquad\qquad \mathsf{transpose}\ b,\ \mathsf{transpose}\ d\rangle_{s(k)} \\
= & \quad \{\text{definition of } \mathsf{transpose}_{s(k)}\} \\
& \langle \mathsf{transpose}\ (\mathsf{transpose}\ a),\ \mathsf{transpose}\ (\mathsf{transpose}\ b), \\
& \ \ \mathsf{transpose}\ (\mathsf{transpose}\ c),\ \mathsf{transpose}\ (\mathsf{transpose}\ d)\rangle_{s(k)} \\
= & \quad \{\text{Equation (C.1) four times}\} \\
& \langle a, b, c, d\rangle_{s(k)}
\end{aligned}
$$

□

This proof that $\mathsf{transpose}_{s(k)}$ is self-inverse requires the proof that $\mathsf{transpose}$ is also self-inverse. This means that this obfuscation is an assertion obfuscation.

The tree for the proof of this assertion is:



6.3.5 Problems with arrays

One of the reasons 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:

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

Then, using $k = 3$,

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

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. So to obfuscate the transposition operation for matrices we could represent matrices by an array and the matrix width.

6.4 Other splits and operations

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

6.4.1 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

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

The representation

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

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

$$\begin{array}{r} \mathsf{dim}\ as = (k, l) \wedge \mathsf{dim}\ bs = (k, c - l) \wedge \\ \mathsf{dim}\ cs = (r - k, l) \wedge \mathsf{dim}\ ds = (r - k, c - l) \end{array} \tag{6.5}$$

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

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

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

The Function Splitting Theorem (Theorem 1 from Section 4.1.4) immediately provides the definitions of addition and scalar multiplication 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 (4.4)

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

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

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

Thus

$$\theta^1(ch(k{-}1, l{-}1)) = ch(\phi^1(k{-}1, l{-}1)) \Rightarrow \theta^1(0) \neq 0$$

So, we cannot find a function θ^1 that satisfies Equation (4.4). Also, we have difficulties defining a multiplication operation for this split; we cannot use our definition for the (k, k)-square split because the matrices that we would try to multiply could be non-conformable.

6.4.2 A more general square splitting

Suppose that we have a matrix $\mathbf{M}^{k\times k}$ which we want to split into n^2 blocks with the condition that the blocks down the main diagonal are square. We will call this the *n-square matrix split*, denoted by $sq(n)$. For this, we will need a set of numbers $S_0, S_1, \ldots, S_m$ such that

$$0 = S_0 < S_1 < S_2 < \ldots < S_{n-1} < S_n = k - 1$$

We require strict inequality so that we have exactly n^2 blocks with both dimensions of each block at least 1.

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

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

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

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

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

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

As with the (k,k)-square split 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:

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

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

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

and thus Equation (4.4) is satisfied. Also,

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

and thus Equation (4.5) is satisfied. Hence the Function Splitting Theorem applies and so if

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

then

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

Without the Function Splitting Theorem, the derivation of this obfuscation is much harder.

6.4.3 Extending the data-type of matrices

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, 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.

6.5 Conclusions

We have found that splitting provides a way of obfuscating another data-type. We are able to obfuscate standard matrix operations producing obfuscations that we know are correct and whose complexities match the unobfuscated operations. The Function Splitting Theorem shows us how to produce obfuscations for scale and add for *all* splits. We have also used the theorem to help us construct obfuscations for transpose for some obfuscations. Since our data-type splitting is a generalisation of array splitting we can use the obfuscations developed here to help us obfuscate matrix operations that are defined imperatively using arrays.

We showed that our definition for $\mathsf{transpose}_{s(k)}$ is an assertion obfuscation with respect to one of the assertions. Further work is needed to establish if our obfuscated operations are good assertion obfuscations. Another area for research to see how we can use randomness to create other matrix obfuscations.

Chapter 7

Cultivating Tree Obfuscations

> Many were increasingly of the opinion that they'd all made a big mistake in coming down from the trees in the first place. And some said that even the trees had been a bad move, and that no one should ever have left the ocean.
>
> *The Hitchhiker's Guide to the Galaxy* (1979)

For our final data-type, we consider binary trees. We have seen that we can split arrays, lists and matrices — is splitting a suitable obfuscation for trees?

7.1 Binary Trees

In this section, we state our binary tree data-type — many of the definitions of the tree operations are taken from [7, Chapter 6]. The kinds of binary tree that we consider are finite trees which contain a well-defined value of type α at each node (so we do not allow $\perp$). Our data-type for binary trees is shown in Figure 7.1.

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

$$\begin{aligned}
&\mathsf{flatten}\ \mathit{Null} &&= [\,]\\
&\mathsf{flatten}\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) &&= (\mathsf{flatten}\ \mathit{lt}) \mathbin{+\!\!+} [v] \mathbin{+\!\!+} (\mathsf{flatten}\ \mathit{rt})
\end{aligned}$$

The height operation measures how far away the furthest leaf is from the root:

$$\begin{aligned}
&\mathsf{height}\ \mathit{Null} &&= 0\\
&\mathsf{height}\ (\mathit{Fork}\ \mathit{lt}\ v\ \mathit{rt}) &&= 1 + (\mathsf{max}\ (\mathsf{height}\ \mathit{lt})\ (\mathsf{height}\ \mathit{rt}))
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{mktree} :: \mathit{List}\ \alpha \to \mathit{Tree}\ \alpha\ \bullet\ \mathit{xt} = \mathsf{mktree}\ \mathit{ls}\ \Rightarrow\\
&(\mathsf{flatten}\ \mathit{xt} = \mathit{ls}\ \wedge (\forall\ \mathit{yt})\ (\mathsf{flatten}\ \mathit{yt}\ = \mathit{ls} \Rightarrow \mathsf{height}\ \mathit{yt} \geq \mathsf{height}\ \mathit{xt}))
\end{aligned}$$

Tree (α)
$\quad USING:$ List (α)

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

$\mathsf{mktree} :: List\ \alpha \rightarrow Tree\ \alpha$
$\mathsf{flatten} :: Tree\ \alpha \rightarrow List\ \alpha$
$\mathsf{member} :: \alpha \rightarrow Tree\ \alpha \rightarrow \mathbb{B}$
$\mathsf{height} :: Tree\ \alpha \rightarrow \mathbb{N}$
$\mathsf{modify} :: \alpha \rightarrow \alpha \rightarrow Tree\ \alpha \rightarrow Tree\ \alpha$

$$\mathsf{flatten} \cdot \mathsf{mktree} = id$$
$$\mathsf{member}\ p\ (\mathsf{mktree}\ xs) = \mathsf{elem}\ p\ xs$$
$$\mathsf{member}\ p\ t \Rightarrow \mathsf{member}\ q\ (\mathsf{modify}\ p\ q\ t)$$
$$\mathsf{height}\ t \leq |\mathsf{flatten}\ t|$$

Figure 7.1: Data-Type for Binary Trees

Bird [7, Page 183] proposes a definition for mktree that builds a binary tree of minimum height:

$$\begin{aligned}
&\mathsf{mktree}\ [\,] = Null\\
&\mathsf{mktree}\ xs = Fork\ (\mathsf{mktree}\ ys)\ z\ (\mathsf{mktree}\ zs)\\
&\quad \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ |xs|\ 2)\ xs
\end{aligned}$$

We can routinely verify that the mktree requirement is satisfied. This definition of mktree not only builds minimal height trees but also builds balanced trees. Note that we do not require that the trees in our data-types are balanced.

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

$$\begin{aligned}
&\mathsf{member}\ p\ Null = False\\
&\mathsf{member}\ p\ (Fork\ lt\ v\ rt) = (v == p) \vee \mathsf{member}\ p\ lt \vee \mathsf{member}\ p\ rt
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{modify}\ p\ q\ Null = Null\\
&\mathsf{modify}\ p\ q\ (Fork\ lt\ v\ rt)\\
&\quad |\ p == v \quad = Fork\ (\mathsf{modify}\ p\ q\ lt)\ q\ (\mathsf{modify}\ p\ q\ rt)\\
&\quad |\ \mathsf{otherwise} = Fork\ (\mathsf{modify}\ p\ q\ lt)\ v\ (\mathsf{modify}\ p\ q\ rt)
\end{aligned}$$

7.1.1 Binary Search Trees

We now consider a special kind of binary tree — *binary search trees*. Binary search trees have the property that the value, of type α, of a node is greater

$$\begin{array}{l}
\mathsf{BST}\ (\alpha) \\
\quad REFINEMENT\ [id]: \quad \mathsf{Tree}\ (\alpha)
\end{array}$$

$$\begin{array}{l}
(Ord\ \alpha) \Rightarrow BST\ \alpha ::= Null \mid Fork\ (BST\ \alpha)\ \alpha\ (BST\ \alpha) \\
\qquad\qquad\qquad \wedge\ \mathsf{inc}\ (\mathsf{flatten}\ t)
\end{array}$$

$$\begin{array}{l}
\quad \langle\!|\mathsf{mktree}, \mathsf{flatten}, \mathsf{member}, \mathsf{height}, \mathsf{modify}\,|\!\rangle \\
\rightsquigarrow \langle\!|\mathsf{mkBST}, \mathsf{flatten}, \mathsf{memBST}, \mathsf{height}, \mathsf{modBST}\,|\!\rangle \\
\\
\mathsf{insert} :: \alpha \rightarrow BST\,\alpha \rightarrow BST\,\alpha \\
\mathsf{delete} :: \alpha \rightarrow BST\,\alpha \rightarrow BST\,\alpha
\end{array}$$

$$\begin{array}{l}
\mathsf{memBST}\ p\ (\mathsf{insert}\ p\ t) = True \\
\mathsf{memBST}\ p\ (\mathsf{delete}\ p\ t) = False
\end{array}$$

Figure 7.2: Data-Type for Binary Search Trees

than the values in the left subtree but less than the values in the right subtree (this assumes we have an total ordering on α). This property can be stated as:

$$\mathsf{inc}\ (\mathsf{flatten}\ t) \tag{7.1}$$

where inc is a Boolean function that tests whether a list is in strictly increasing order. Our data-type for binary search trees is shown in Figure 7.2. Since BSTs are binary trees then we can take id as the abstraction function for the refinement.

The definitions of flatten and height are the same as before (and so we keep the names the same). For BSTs, memBST is implemented efficiently as follows:

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

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

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

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

$$\mathsf{mkbst} = \mathsf{mktree.strict_sort}$$

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

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

If the list is strictly-increasing then so is each sublist and thus Equation (7.1) is maintained.

We can define insert quite easily as follows:

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

However, the definition of delete is more complicated:

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

where the function join satisfies the equation:

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

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

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

The functions headTree and tailTree satisfy

$$\mathsf{headTree} = \mathsf{head} \cdot \mathsf{flatten} \tag{7.2}$$

$$\mathsf{flatten} \cdot \mathsf{tailTree} = \mathsf{tail} \cdot \mathsf{flatten} \tag{7.3}$$

and we define

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

and

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

We can define modBST in terms of insert and delete:

$$
\begin{array}{lcl}
\mathsf{modBST}\ p\ q\ t & = & \text{if } \mathsf{memBST}\ p\ t \\
 & & \text{then } \mathsf{insert}\ q\ (\mathsf{delete}\ p\ t) \\
 & & \text{else } t
\end{array}
$$

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

7.2 Obfuscating Trees

Now that we have defined our binary tree data-type, we need to consider how we can obfuscate it. In the previous chapters, we have used splitting to obfuscate lists, sets and matrices. Can we obfuscate trees in the same way?

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

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

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

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

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

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

Using of split trees produces reasonable obfuscations (the proof trees for split tree operations generally have a greater height and cost) but the definitions of the obfuscated operations are similar to the unobfuscated versions. To show that our data-type approach is applicable to more than just splitting, we will consider a different obfuscation.

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

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

7.2.1 Ternary Trees

We now consider converting a binary tree to a ternary tree by adding an extra subtree at every node of the binary tree. If a binary tree t_2 is represented by a ternary tree t_3, then by Equation (3.1) we need an abstraction function af and predicate dti such that

$$t_2 = af(t_3) \wedge dti(t_3)$$

$\mathsf{Tree}_3\ (\alpha)$ $\quad$ *REFINEMENT* [to2] : $\quad \mathsf{Tree}\ (\alpha)$
$Tree_3\ \alpha ::= Null_3 \mid Fork_3\ \alpha\ (Tree_3\ \alpha)\ (Tree_3\ \alpha)\ (Tree_3\ \alpha)$
$\quad \langle\!\langle \mathsf{mktree}, \mathsf{flatten}, \mathsf{member}, \mathsf{height}, \mathsf{modify} \rangle\!\rangle$ $\leadsto \langle\!\langle \mathsf{mktree}_3, \mathsf{flatten}_3, \mathsf{member}_3, \mathsf{height}_3, \mathsf{modify}_3 \rangle\!\rangle$

Figure 7.3: Data-Type for Ternary Trees

In Section 7.2.2, we define a function

$$\mathsf{to2} :: Tree_3\ \alpha \to Tree\ \alpha$$

which serves as an abstraction function. The data-type invariant for binary trees is *True* and for search trees is $\mathsf{inc}\ (\mathsf{flatten}\ (\mathsf{to2}\ t_3))$.

7.2.2 Particular Representation

In this section, we give a particular ternary tree representation of binary trees with our data-type for ternary trees stated in Figure 7.3. We can represent search trees by a similar declaration. For demonstration purposes, we assume that α is the type of integers.

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

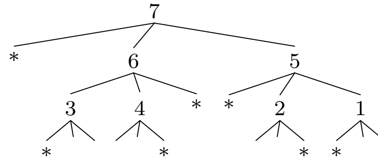
The conversion that we consider is as follows. The tree $Fork\ lt\ v\ rt$ is converted to

- $Fork_3\ v\ lt_3\ rt_3\ junk_1$ if v is even
- $Fork_3\ v\ junk_2\ rt_3\ lt_3$ if v is odd

and the tree *Null* is converted to $Null_3$. So, the binary tree

```
et =      7
        /   \
       5     6
      / \   / \
     1   2 3   4
```

is converted to a ternary tree of the form



where "*" stand for different arbitrary ternary trees.

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

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

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

$$
\mathsf{to2} \cdot \mathsf{to3} = \mathit{id} \tag{7.4}
$$

and so

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

In Section 3.2, we called such a function a *conversion* function (and this is a conversion with respect to a particular refinement).

To convert a binary tree into a ternary tree, we have many choices of how to create the junk to be inserted into the tree at the appropriate places. Thus we can construct many functions that satisfy Equation (7.4). The complexity of the binary tree operations defined in Section 7.1 depends on the number of nodes or on the height. Thus if we want our ternary tree operation to have a similar complexity then we should make sure that the height is similar and we do not introduce too many extra elements. As an example, we define:

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

where

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

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

(7.5)

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

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

$$\mathsf{op}_3 = \mathsf{to3} \cdot \mathsf{op} \cdot \mathsf{to2} \tag{7.6}$$

However, the operations that we would obtain would be specialised to this particular conversion from binary to ternary since the junk information is built up in a specific way. We will see that we have a certain degree of freedom in how we deal with the junk information. Yet, we could use Equation (7.6) to give us a guide in how an obfuscation might look and so aid us in giving the definition of a ternary tree operation.

7.2.3 Ternary Operations

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

$$\begin{array}{l}
\mathsf{height}_3\ Null_3\ = 0 \\
\mathsf{height}_3\ (Fork_3\ v\ lt\ ct\ rt) \\
\quad \mid \mathsf{even}\ v \quad\ = 1 + (\mathsf{max}\ (\mathsf{height}_3\ lt)\ (\mathsf{height}_3\ ct)) \\
\quad \mid \mathsf{otherwise} = 1 + (\mathsf{max}\ (\mathsf{height}_3\ rt)\ (\mathsf{height}_3\ ct))
\end{array}$$

$$\begin{array}{l}
\mathsf{flatten}_3\ Null_3\ = [\,] \\
\mathsf{flatten}_3\ (Fork_3\ v\ lt\ ct\ rt) \\
\quad \mid \mathsf{even}\ v \quad\ = (\mathsf{flatten}_3\ lt) + [v] + (\mathsf{flatten}_3\ ct) \\
\quad \mid \mathsf{otherwise} = (\mathsf{flatten}_3\ rt) + [v] + (\mathsf{flatten}_3\ ct)
\end{array}$$

and we can easily show that these definitions satisfy

$$\mathsf{op}_3 = \mathsf{op} \cdot \mathsf{to2} \tag{7.7}$$

We can see that these operations have an extra test which adds an extra conditional and so gives an extra case when proving assertions.

For member_3, we can use Equation (7.7) to derive the operation using structural induction which gives the following definition:

$$\begin{array}{l}
\mathsf{member}_3\ p\ Null_3 = False \\
\mathsf{member}_3\ p\ (Fork_3\ v\ lt\ ct\ rt) = \\
\qquad v == p\ \vee\ (\mathsf{member}_3\ p\ ct)\ \vee\ (\text{if even } v\ \text{ then } (\mathsf{member}_3\ p\ lt) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } (\mathsf{member}_3\ p\ rt))
\end{array}$$

The details for this proof are in [21]. We can define modify_3 as follows:

$$
\begin{array}{l}
\mathsf{modify}_3\ p\ q\ Null_3 = Null_3 \\
\mathsf{modify}_3\ p\ q\ (Fork_3\ v\ lt\ ct\ rt) \\
\quad \left|\ \begin{array}{ll}
v == p \land \mathsf{mod}\ (p-q)\ 2\ ==\ 0 & = Fork_3\ q\ lt'\ ct'\ rt' \\
v == p \land \mathsf{mod}\ (p-q)\ 2\ ==\ 1 & = Fork_3\ q\ rt'\ ct'\ lt' \\
\text{otherwise} & = Fork_3\ v\ lt'\ ct'\ rt'
\end{array}\right. \\
\qquad \text{where } lt' = \mathsf{modify}_3\ p\ q\ lt \\
\qquad\qquad\quad\ rt' = \mathsf{modify}_3\ p\ q\ rt \\
\qquad\qquad\quad\ ct' = \mathsf{modify}_3\ p\ q\ ct
\end{array}
$$

In this definition we have collapsed some cases; for example, we can combine the predicates $\mathsf{even}\ p \land \mathsf{even}\ q$ and $\mathsf{odd}\ p \land \mathsf{odd}\ q$ into one predicate: $\mathsf{mod}\ (p-q) == 0$. A fuller discussion of modify_3 can be found in [21].

7.2.4 Making Ternary Trees

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

$$\mathsf{mktree} = \mathsf{to2} \cdot \mathsf{mktree}_3 \tag{7.8}$$

However as with the definition of a conversion function, we have many choices of mktree_3 that satisfy the above equation. The general form of a function that satisfies Equation (7.8) is:

$$
\begin{array}{l}
\mathsf{mktree}_3\ [\,] = Null_3 \\
\mathsf{mktree}_3\ xs \\
\quad \left|\ \begin{array}{ll}
\mathsf{even}\ z & = Fork_3\ z\ (\mathsf{mktree}_3\ ys)\ (\mathsf{mktree}_3\ zs)\ jt \\
\text{otherwise} & = Fork_3\ z\ kt\ (\mathsf{mktree}_3\ zs)\ (\mathsf{mktree}_3\ ys)
\end{array}\right. \\
\qquad \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ |xs|\ 2)\ xs
\end{array}
$$

where the expressions jt and kt represent arbitrary ternary trees.

As an example, we define mktree_3 so that

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

where $\mathsf{to3}$ as defined in Section 7.2.2. This equation gives the following definition:

$$
\begin{array}{l}
\mathsf{mktree}_3\ [\,] = Null_3 \\
\mathsf{mktree}_3\ xs \\
\quad \left|\ \begin{array}{ll}
\mathsf{even}\ z & = Fork_3\ z\ (\mathsf{mktree}_3\ ys)\ (\mathsf{mktree}_3\ zs)\ (\mathsf{mktree'}_3\ ys) \\
\text{otherwise} & = Fork_3\ z\ (\mathsf{mktree'}_3\ zs)\ (\mathsf{mktree}_3\ zs)\ (\mathsf{mktree}_3\ ys)
\end{array}\right. \\
\qquad \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ (|xs|\ 2)\ xs
\end{array}
$$

$$
\begin{array}{l}
\mathsf{mktree'}_3\ [\,] = Null_3 \\
\mathsf{mktree'}_3\ xs = Fork_3\ (3 * z + 2)\ (\mathsf{mktree}_3\ ys)\ (\mathsf{mktree'}_3\ ys)\ (\mathsf{mktree'}_3\ zs) \\
\qquad \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ (|xs|\ 2)\ xs
\end{array}
$$

We can construct this function by using Equation (7.6) and so this function is specialised to the choice of conversion of binary to ternary. Computing

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

produces the ternary tree (7.5).

Since the input for this operation is a list, we can also perform an obfuscation on the input list. In Appendix D we use a split list as well as ternary trees to obfuscate the definition of mktree. We prove an assertion for different versions of mktree and we find that using ternary trees increases the cost but not the height of proof trees, but using a split list increases both.

7.2.5 Operations for Binary Search Trees

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

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

We define operations corresponding to memBST and insert routinely:

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

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

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

7.2.6 Deletion

To define delete for our ternary tree, we first need to define headTree and tailTree for ternary trees which satisfy analogues of Equations (7.2) and (7.3):

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

Using these equations, we obtain:

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

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

and we can easily verify that:

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

Now we need to define a function join_3 that satisfies the equation

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

We propose the following definition:

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

In this definition we have collapsed two cases into one. Suppose that we let $v = \mathsf{headTree}_3\ yt$. When v is even we want to have a tree of the form:

$$\mathit{Fork}_3\ v\ xt\ (\mathsf{tailTree}_3\ yt)\ jt$$

and when v is odd we want:

$$\mathit{Fork}_3\ v\ kt\ (\mathsf{tailTree}_3\ yt)\ xt$$

where jt and kt are junk ternary trees. We can combine these cases by making both jt and kt equal xt.

This definition can be shown to satisfy:

$$\mathsf{join}\ (\mathsf{to2}\ xt)\ (\mathsf{to2}\ yt) = \mathsf{to2}\ (\mathsf{join}_3\ xt\ yt) \tag{7.9}$$

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

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

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

The values jt and kt represent ternary trees — we could in fact choose to write, for example, $\mathsf{delete}_3\ x\ rt$ or even $\mathsf{insert}_3\ x\ rt$ in place of these which would aid in making the function more obfuscated. To show that this operation is correct, we need to prove that for a ternary tree t and a value x, delete satisfies Equation (3.5), *i.e.*:

$$\mathsf{delete}\ x\ (\mathsf{to2}\ t) = \mathsf{to2}\ (\mathsf{delete}_3\ x\ t) \tag{7.10}$$

This is routine and proved in [21].

7.3 Other Methods

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

7.3.1 Abstraction Functions

Consider how we can design a more general abstraction function than to2 given in Section 7.2.2. We need a function af that satisfies

$$t_2 \leadsto t_3 \Leftrightarrow t_2 = af\ t_3$$

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

We first define a *partitioning function* with type

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

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

$$\begin{array}{l} af\ (Fork_3\ v\ lt\ ct\ rt) \\ \quad \left|\begin{array}{ll} p(v) == 0 & = Fork\ (af\ (xt_0))\ v\ (af\ (yt_0)) \\ p(v) == 1 & = Fork\ (af\ (xt_1))\ v\ (af\ (yt_1)) \\ \ldots & \\ \text{otherwise} & = Fork\ (af\ (xt_{n-1}))\ v\ (af\ (yt_{n-1})) \end{array}\right. \end{array}$$

We should ensure that p partitions α uniformly — thus a good choice for p would be a hash function. If we define $p = \lambda v.v \bmod 2$ and let $xt_0 = lt$, $yt_0 = ct$, $xt_1 = rt$ and $yt_1 = ct$ then we can write to2 in this form.

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

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

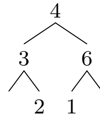
$$\begin{array}{l} \mathsf{toT}\ Null = Null_3 \\ \mathsf{toT}\ (Fork\ lt\ v\ rt) \\ \quad \left|\begin{array}{ll} \mathsf{even}\ v & = Fork_3\ (v+2)\ (\mathsf{toT}\ rt)\ jt\ (\mathsf{toT}\ lt) \\ \text{otherwise} & = Fork_3\ (v \times 3)\ kt\ (\mathsf{toT}\ lt)\ (\mathsf{toT}\ rt) \end{array}\right. \end{array}$$

where jt and kt are arbitrary ternary trees.

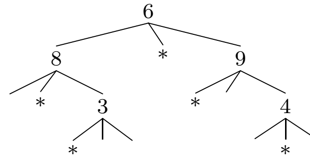
To create an abstraction function for this conversion, we need to invert the functions $\lambda\, i.(i+2)$ and $\lambda\, i.(3i)$. Thus we can define the abstraction function as follows:

$$\begin{array}{l} \mathsf{toB}\ Null_3 = Null \\ \mathsf{toB}\ (Fork_3\ w\ lt\ ct\ rt) \\ \quad \left|\begin{array}{ll} \mathsf{even}\ w & = Fork\ (\mathsf{toB}\ rt)\ (w-2)\ (\mathsf{toB}\ lt) \\ \text{otherwise} & = Fork\ (\mathsf{toB}\ ct)\ (\mathsf{div}\ w\ 3)\ (\mathsf{toB}\ rt) \end{array}\right. \end{array}$$

Using this abstraction function, the binary tree



can be represented by trees of the following form:



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

$$\begin{aligned} \mathsf{toB}\ (\mathsf{toT}\ (\mathit{Fork}\ \mathit{Null}\ 2\ \mathit{Null})) &= \mathsf{toB}\ (\mathit{Fork}_3\ 3\ \mathit{Null}\ \mathit{jt}\ \mathit{Null}) \\ &= \mathit{Fork}\ (\mathsf{toB}\ \mathit{jt})\ 1\ \mathit{Null} \\ &\neq \mathit{Fork}\ \mathit{Null}\ 2\ \mathit{Null} \end{aligned}$$

and would not be the case that $\mathsf{toB} \cdot \mathsf{toT} = \mathit{id}$.

7.3.2 Heuristics

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

- When creating a partitioning function for the definition of an abstraction function, we should ensure that each value in the range is equally likely. In particular, we should not construct a case in the definition of the abstraction function which is never considered or else it could be eliminated. We could use a hashing function for the partitioning function. In fact, for obfuscation purposes, we could have a set of hashing functions and randomly choose which function we use (*cf.* Universal Hashing [8]).

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

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

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

- We must ensure that the junk we generate does not "stand out" — this is so that it is not obvious what is junk and what is "real" data. Thus we should the keep the junk values within the range of the values of our binary tree and we could possibly repeat values. This was also a concern for the padded block split discussed in Section 4.4.2.

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

7.4 Conclusions

In this chapter we have discussed a refinement that is suitable for creating tree obfuscations. The examples given have been made simple in order to demonstrate the techniques involved. We have also given some heuristics (in Section 7.3.2) that need to be considered when producing obfuscations. We can adapt the refinement to deal with more complicated obfuscations and also for obfuscating other kinds of tree such as red-black trees [13, Chapter 13].

We have demonstrated how to obfuscate binary trees by representing them as ternary trees. This representation has allowed us to add bogus elements and so hide the "real" information. We must ensure that we do not adversely affect the efficiency by adding too much junk which would increase the size of the trees. This extra information gives us further scope for obfuscation. For example, when trying to insert an element in a tree we could try to delete it from the junk part of the tree. We can actually specify a set of operations that represent the same function and we could choose one operation randomly. In fact, our approach gives plenty of scope for randomization. On each execution, we can have a different abstraction function (by choosing a suitable partition function), different junk values in a ternary tree representation and a different definition for each operation. Each of these choices can be made randomly and so we create different program traces which compute the same value. This randomness provides additional confusion for an adversary and helps to keep the unobfuscated operations secret.

For our tree obfuscation, we have a trade-off between how much junk we place in a ternary tree and the complexity of the obfuscated operations. Putting in too much junk makes building the ternary trees expensive and operating on the junk can have a severe impact on the efficiency. However, we should ensure that our operations act on the junk in some way (so that it is not obvious that the junk is not significant). Thus we should aim to keep the height and size of the ternary tree roughly the same as that of the binary tree it represents.

Part IV

Conclusions

Chapter 8

Conclusions and Further Work

The Doctor: “It’s the end but the moment has been prepared for”.

Doctor Who — *Logopolis* (1981)

8.1 Discussion of our approach to Obfuscation

The current view of obfuscation concentrates on obfuscating object-oriented languages (or the underlying intermediate representations). The obfuscations given in [10] focus on concrete data-types such as variables and arrays. Proofs of correctness for imperative obfuscations are hard to construct and require difficult methods. We would like to have a workable definition of obfuscation which is more rigorous than the metric-based definition of Collberg *et al.* [10] and overcomes the impossibility result of Barak *et al.* [6] for their strong cryptographic definition. In this thesis, we have aimed to give an alternative approach to obfuscation. On Page 7, we have stated that we wanted our approach to have the following objectives:

- to yield proofs of correctness (or even yield derivations) of all our obfuscations
- to use simple, established refinement techniques, leaving the ingenuity for obfuscation
- to generalise obfuscations to make obfuscations more applicable.

For our approach, we have obfuscated abstract data-types and used techniques from data refinement and functional programming. In the rest of this section, we discuss whether our data-type approach has met the above objectives and we also provide some benefits of our approach.

8.1.1 Meeting the Objectives

We have proposed a new approach to obfuscation by studying abstract data-types and considering obfuscation as functional refinement. To demonstrate the

flexibility of our approach, we have presented various case studies for different data-types and discussed obfuscations for each data-type. We have developed obfuscations for simple lists and for more complicated types such as matrices and trees which have a 2-dimensional structure. The example operations and obfuscations that we have given have been made simple in order to demonstrate our obfuscation techniques. It seems clear that more intricate, realistic obfuscations can be developed similarly for other abstract data-types.

It is important to check that an obfuscation is correct — *i.e.* it does not change the functionality of an operation. In Section 2.4.3, we have seen that proving correctness is a challenging task. Considering obfuscation as refinement has allowed us to set up equations to prove the correctness of *all* our obfuscations of data-type operations. Additionally if the abstraction function is injective then it has a unique inverse with which we can derive obfuscations. Using simple derivational techniques and modelling our operations in Haskell has allowed us to establish correctness easily — this is a real strength of our approach. Specifying operations in Haskell gives us the benefits of the elegance of the functional style and the consequent abstraction of side-effects. Thus our functional approach will provide support for purely imperative obfuscations.

We have given a generalisation of the array split [10] by considering a more general class of data-types. Using this generalisation, we have proved a theorem in Section 4.1.4 that shows us how to to define obfuscations for certain splits and operations. In Section 6.4.2 this theorem was used to give a quick derivation of an obfuscation of transpose for split matrices. In Sections 3.2.2 and 3.2.3 we have given a correctness equation for general operations. For instance, if we have a non-homogeneous operation and abstraction functions for obfuscations of the domain and the range then we can construct an obfuscated operation that has two different obfuscations. For example, in Appendix D, we show how to obfuscate mktree by splitting the input list and turning the output into a ternary tree.

To establish a functional refinement, we have to state an abstraction function. From an obfuscation point of view, the abstraction function acts as a deobfuscation function. With knowledge of the abstraction function, we could reconstruct unobfuscated operations and so it is vital that we hide this function from an attacker.

8.1.2 Complexity

One of the requirements for an obfuscated operation is that the running time should be at most polynomially more than the original operation. When splitting data-types we have been fortunate in that, in general, our obfuscations do not change the complexity of the operations. For lists and sets, the obfuscated operations require some extra computations such as evaluating tests. For trees, we have added junk elements to hide the "real" information. However if we put too much junk into the trees then the efficiencies of the tree operations will be compromised. So there is a trade-off between how much junk we put into a ternary tree and the complexity of our operations.

Further work is needed to see how the complexity changes with more complicated obfuscations and for other data-types. Also we need to consider how the complexity changes when implementing the operations and obfuscations imperatively.

8.1.3 Definition

In Section 1.2 we have stated two definitions of obfuscation and discussed some problems with the definitions. For data-types we have proposed a new definition — an assertion obfuscation.

Does our definition actually reflect what it means to be obfuscated? When trying to find out what a program does we often want to know what properties the program has (*i.e.* what the program does). We would expect that if a program is obfuscated then it should be harder to find out what properties this program has. Our definition reflects this observation by taking "properties of a program" as "assertions" and considering how difficult it is to prove these assertions. Thus we have interpreted "harder to understand" as "making the proofs of assertions more complicated". So, our definition tries to measure the degree of obfuscation by considering the complexity of *proofs* rather than the complexity of operations.

For our definition we are required to give a list of assertions when we declare a data-type. Also we have had to qualify what it means for a proof to be "more complicated". We have decided to consider drawing trees for the proofs and then taking "more complicated" (informally) to mean increasing the height and number of nodes in the proof tree. For a fair comparison of proofs, we have to ensure that we prove assertions in a consistent manner and so we have discussed some of the ways that would help us to be consistent. Some other problems with the definition were highlighted in Section 3.4.4. One major problem is that to reflect completely on the effectiveness of an obfuscated operation we have to construct proofs for all assertions involving the operation. To show how our definition performs we have proved assertions for each of the data-types in our case studies and we found that the assertion proofs were quite straightforward since we chose to model the operations in Haskell. For our case studies, we have found that a good assertion obfuscation is produced when the structures of the operation and its obfuscation are quite different.

Our definition does not consider syntactic properties such as variable names nor whether we use nested conditionals or guarded equations and it also does not consider how easy it is to undo our obfuscations. How does the assertion definition match up with the traditional view of obfuscation? In particular, if a transformation is an assertion obfuscation then is it also an obfuscation under the definitions of Collberg [10] and Barak [6]?

8.1.4 Randomness

One benefit of considering abstract data-types is that we have been able to introduce randomness in our obfuscations. In particular, we have shown some examples of random splits in Chapter 4 and in Chapter 7 we discussed how to put random junk into trees. Random obfuscations can help confuse an attacker further by creating different program traces on different executions with the same input.

For the tree obfuscation, we have plenty of scope for randomization:

- The obfuscation inserts junk into the trees which we can choose randomly. Though we must ensure that the junk does not "stand out" from the "real" information.

- When defining some operations (such as insert and delete) we are free to choose how the operation acts on the junk. This means that we can have a set of definitions that act in different ways on the junk. Hence we can choose randomly which definition we want to execute.
- In Section 7.3.1, we have shown more general abstraction functions in which we can supply a partitioning function p which has type: $\alpha \rightarrow [0..n)$. If we fix the value of n then we can define our tree operations to contain occurrences of the test for the value of p. Thus we can supply the actual definition for p separately and so choose a partitioning function at random.

Thus on each execution, we can have different junk, different definitions and different abstraction functions — all of which can be chosen randomly.

In Chapter 4 we have seen how to create random splits for lists. Further work is needed to see how to extend these random splits (and other random obfuscations) to other data-types.

8.1.5 Contributions

We have seen that our approach has met the objectives stated on Page 7 — these objectives can be hard to achieve in an object-oriented setting. In addition, we have proposed a new definition of obfuscation, we have some promising results on the complexity of our obfuscations and we have the ability to create random obfuscations. Thus considering data-types has made a valuable contribution to the study of obfuscation.

Can our techniques be applied to help to obfuscate imperative programs? In Section 3.1.2 we imposed some restrictions on how we used Haskell as a modelling language. In particular, we specified finite, well-defined data-types and we did not exploit laziness. We demanded these restrictions so that we can implement our obfuscations imperatively. In [21] and on Page 8, we have given examples of how to apply our tree obfuscation from Chapter 7 to obfuscate some imperative methods. These obfuscations were achieved by manually adapting the original imperative methods — further work is needed to automate this conversion. Using imperative programs provides different opportunities for obfuscation. For instance, trees can be implemented using pointers and the difficulty in performing pointer analysis could be exploited to construct obfuscations. Binary trees could be changed into ternary trees and in the junk, bogus pointers that point back up the tree could be added. As well as implementing lists using arrays, linked lists could be used. So in addition to splitting the lists, bogus pointers into the list can be added.

Less satisfactory aspects of our approach are that there are some problems with our definition of obfuscation (discussed in Section 3.4.4) and our abstraction functions act as deobfuscation functions.

8.2 Further Work

We briefly consider some possible areas for further study.

8.2.1 Deobfuscation

We have not mentioned how easy it is to undo our obfuscations. Since we have considered obfuscation to be a functional refinement, we have an abstraction function which maps the obfuscated data-type back to the original data-type. So, the abstraction function acts as a "deobfuscation" and therefore it is important to keep this function secret from an attacker. In particular, where possible, traces of the abstraction function from the definition of our obfuscations should be removed. This is a particular problem with operations that convert an obfuscated data-type to another data-type. For instance, the definition of $\mathsf{flatten}_3$ in Section 7.2.3 gives away some knowledge of the abstraction function (to2). To prevent this, further obfuscations (such as using split lists) should be added.

The resilience of our obfuscation to reverse engineering should be explored. One simple technique for reverse engineering is to execute the program and then study the program traces. If random obfuscations (see Section 8.1.4) are used then different program traces can be created. Another technique is *refactoring* [27] which is the process of improving the design of existing code by performing behaviour-preserving transformations. One particular area to explore is the refactoring of functional programs — [35] gives a discussion of a tool for refactoring Haskell called *HaRe*. What happens to our obfuscations if they are refactored? Could HaRe be used to specify obfuscations?

8.2.2 Other techniques

In Chapter 2, we have discussed a language with which we can specify obfuscations for IL and so can generate obfuscations automatically. Is it possible to develop such a system for our obfuscations? One possible approach could be to write operations and abstraction functions as folds or unfolds. As stated in Section 3.5, an abstraction function can usually be written as an unfold and a conversion function as a fold. If folds are used then derivations and proofs can be calculated using fusion and so an automated system, such as the one described in [15], could be used. A concern of some of the matrix operations in [19] that were derived using fusion was that traces of the splitting functions can be seen in the definition.

Functional programming and data refinement have been used to specify obfuscations for abstract data-types. In our objectives (on Page 7), we have stated that we wanted to use simple, established techniques and to be able to generalise obfuscations. While it is certainly true that our approach has used simple techniques, is it general enough? For more generality *categories* could be obfuscated and concepts such as hylomorphisms [38] could be used. Barak *et al.* [6] provide a formal cryptographic treatment of obfuscation by obfuscating Turing machines. How does our approach and, in particular, our definition compare with their approach? In Section 3.2, we restricted ourselves to functional refinements. More general notions of refinement [16] could be studied instead and in doing so partial and non-deterministic operations can be considered. What would be gained by using these more general methods? To what extent will these methods support obfuscation of imperative programs?

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Anakrino. .NET decompiler. Avaliable from URL: http://www.saurik.com/net/exemplar/.

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

[4] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[5] Brenda S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, 1977.

[6] 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.

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

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

[9] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, January 1999.

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

[11] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages*, page 28. IEEE Computer Society, 1998.

[12] Cristian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *ACM SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1998.

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

[14] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher Order Symbolic Computation*, 16(1-2):15–35, 2003.

[15] 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.

[16] 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.

[17] Kris de Volder. *Type-oriented logic meta-programming*. PhD dissertation, Vrije Universiteit Brussels, 1998.

[18] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag, Texts and Monographs in Computer Science, 1990.

[19] Stephen Drape. The Matrix Obfuscated. Technical Report PRG-RR-04-12, Programming Research Group, Oxford University Computing Laboratory, June 2004.

[20] Stephen Drape. Obfuscating Set Representations. Technical Report PRG-RR-04-09, Programming Research Group, Oxford University Computing Laboratory, May 2004.

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

[22] 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.

[23] ECMA. Standard ECMA-335: Common Language Infrastructure. URL: http://www.ecma-international.org/publications.

[24] Ana M. Erosa and Laurie J. Hendren. Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In *ICCL*, 1994.

[25] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (Second Edition in C)*. Addison-Wesley, 1996.

[26] Force5 Software. JCloak. Available from URL: http://www.force5.com/JCloak/ProductJCloak.html.

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

[28] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 273–279. ACM Press, 1998.

[29] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63–74, 1981.

[30] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Trans. Comput. Syst.*, 5(4):371–393, 1987.

[31] David Lacey. *Specifying Compiler Optimisations in Temporal Logic*. DPhil thesis, Oxford University Computing Laboratory, 2003.

[32] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *LNCS*, pages 52–68. Springer Verlag, 2001.

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

[34] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231. ACM Press, 2003.

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

[36] Johannes J. Martin. *Data types and data structures*. Prentice Hall, 1986.

[37] Thomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[38] 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.

[39] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[40] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *Journal of Systems Software*, 20(3):217–225, 1993.

[41] R. Paige. Viewing a program transformation system at work. In Manuel Hermenegildo and Jaan Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, pages 5–24. Springer Verlag, 1994.

[42] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th Annual Computer Security Applications Conference, ACSAC '00*, pages 308–316. IEEE, 2000.

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

[44] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM (JACM)*, 35(4):893–920, 1988.

[45] RemoteSoft. Salamander .NET decompiler. Available from URL: http://www.remotesoft.com/salamander/index.html.

[46] RemoteSoft. Salamander .NET obfuscator. Available from URL: http://www.remotesoft.com/salamander/obfuscator.html.

[47] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–38. ACM Press, 2004.

[48] Smokescreen. Java obfuscator. Available from URL: http://www.leesw.com/smokescreen/index.html.

[49] PreEmptive Solutions. Dotfuscator. Available from URL: http://www.preemptive.com.

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

[51] J. M. Spivey. *Logic Programming for Programmers*. Prentice Hall, 1996.

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

[53] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, pages 18–34, 2000.

[54] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[55] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, number 2051 in Lecture Notes in Computer Science, pages 357–362. Springer-Verlag, May 2001.

[56] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.

[57] Zelix. Klassmaster Java Obfuscator. Available from URL: http://www.zelix.com/klassmaster/index.html.

Part V

Appendices

Appendix A

List assertion

We want to prove that, for all finite lists xs and ys,

$$|xs +\!\!+ ys| = |xs| + |ys| \tag{A.1}$$

which is one of our assertions for the list data-type (Figure 4.1). We prove this assertion for the different concatenation operations defined in Chapter 4.

A.1 Unobfuscated version

For standard lists, we define $+\!\!+$ as follows:

$$\begin{aligned}
[\,] +\!\!+ ys &= ys \\
(x : xs) +\!\!+ ys &= x : (xs +\!\!+ ys)
\end{aligned}$$

We prove Equation (A.1) by structural induction on xs.

Base Case Suppose that $xs = [\,]$. Then

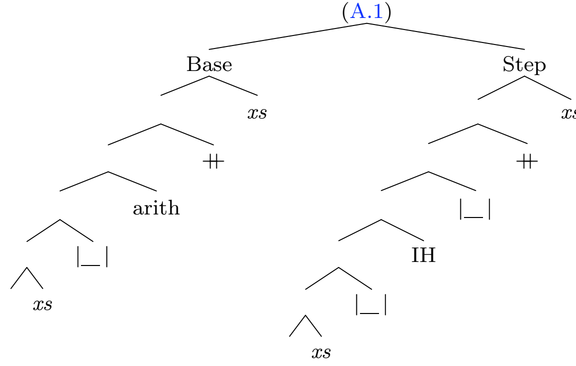
$$\begin{aligned}
& |xs +\!\!+ ys| \\
= & \quad \{\text{definition of } xs\} \\
& |[\,] +\!\!+ ys| \\
= & \quad \{\text{definition of } +\!\!+\} \\
& |ys| \\
= & \quad \{\text{arithmetic}\} \\
& 0 + |ys| \\
= & \quad \{\text{definition of } |__|\} \\
& |[\,]| + |ys| \\
= & \quad \{\text{definition of } xs\} \\
& |xs| + |ys|
\end{aligned}$$

Step Case Suppose that $xs = t : ts$ and that ts satisfies Equation (A.1). Then

$$|xs +\!\!+ ys|$$

$$
\begin{aligned}
&= \quad \{\text{definition of } xs\} \\
&\quad |(t : ts) +\!\!+ ys| \\
&= \quad \{\text{definition of } +\!\!+\} \\
&\quad |t : (ts +\!\!+ ys)| \\
&= \quad \{\text{definition of } |_|\} \\
&\quad 1 + |ts +\!\!+ ys| \\
&= \quad \{\text{induction hypothesis}\} \\
&\quad 1 + |ts| + |ys| \\
&= \quad \{\text{definition of } |_|\} \\
&\quad |t : ts| + |ys| \\
&= \quad \{\text{definition of } xs\} \\
&\quad |xs| + |ys|
\end{aligned}
$$

The tree is this proof is:



For this, $\mathcal{C}$ (A.1) = 11 and $\mathcal{H}$ (A.1) = 7.

A.2 Alternating Split for Lists

We want to prove that

$$|xsp \ +\!\!+_{asp} \ ysp|_{asp} = |xsp|_{asp} + |ysp|_{asp} \tag{A.2}$$

For the alternating split, we have two operations equivalent to $+\!\!+$.

A.2.1 Version 1

We first define:

$$\langle l_0, r_0 \rangle_{asp} +\!\!+_{asp} \langle l_1, r_1 \rangle_{asp} = \begin{cases} \langle l_0 +\!\!+ l_1, r_0 +\!\!+ r_1 \rangle_{asp} & \text{if } |l_0| = |r_0| \\ \langle l_0 +\!\!+ r_1, r_0 +\!\!+ l_1 \rangle_{asp} & \text{otherwise} \end{cases}$$

To prove (A.2), let $xsp = \langle l_0, r_0 \rangle_{asp}$ and $ysp = \langle l_1, r_1 \rangle_{asp}$ and we have two cases:

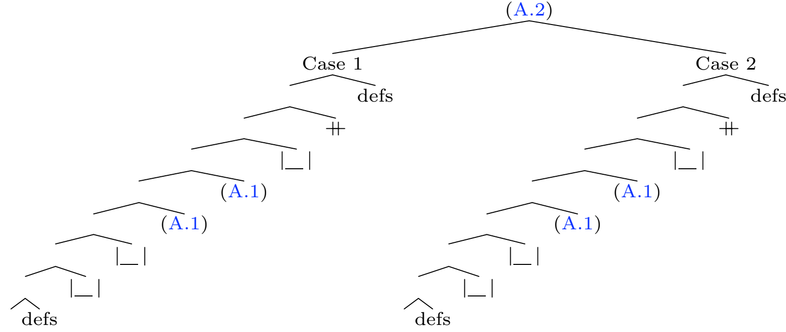
Case 1 Suppose that $|l_0| = |r_0|$. Then

$$
\begin{aligned}
& |xsp +\!\!+_{asp}\ ysp|_{asp} \\
= & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\
& |\langle l_0, r_0\rangle_{asp} +\!\!+_{asp} \langle l_1, r_1\rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } +\!\!+_{asp}\} \\
& |\langle l_0 +\!\!+ l_1, r_0 +\!\!+ r_1\rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& |l_0 +\!\!+ l_1| + |r_0 +\!\!+ r_1| \\
= & \quad \{\text{Equation (A.1)}\} \\
& |l_0| + |l_1| + |r_0 +\!\!+ r_1| \\
= & \quad \{\text{Equation (A.1)}\} \\
& |l_0| + |l_1| + |r_0| + |r_1| \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& |\langle l_0, r_0\rangle_{asp}|_{asp} + |l_1| + |r_1| \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& |\langle l_0, r_0\rangle_{asp}|_{asp} + |\langle l_1, r_1\rangle_{asp}|_{asp} \\
= & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\
& |xsp|_{asp} + |ysp|_{asp}
\end{aligned}
$$

Case 2 Suppose that $|l_0| \neq |r_0|$. Then

$$
\begin{aligned}
& |xsp +\!\!+_{asp}\ ysp|_{asp} \\
= & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\
& |\langle l_0, r_0\rangle_{asp} +\!\!+_{asp} \langle l_1, r_1\rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } +\!\!+_{asp}\} \\
& |\langle l_0 +\!\!+ r_1, r_0 +\!\!+ l_1\rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& |l_0 +\!\!+ r_1| + |r_0 +\!\!+ l_1| \\
= & \quad \{\text{Equation (A.1)}\} \\
& |l_0| + |r_1| + |r_0 +\!\!+ l_1| \\
= & \quad \{\text{Equation (A.1)}\} \\
& |l_0| + |r_1| + |r_0| + |l_1| \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& |\langle l_0, r_0\rangle_{asp}|_{asp} + |l_1| + |r_1| \\
= & \quad \{\text{definition for } |_|_{asp}\} \\
& |\langle l_0, r_0\rangle_{asp}|_{asp} + |\langle l_1, r_1\rangle_{asp}|_{asp} \\
= & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\
& |xsp|_{asp} + |ysp|_{asp}
\end{aligned}
$$

For this proof, we declare (A.1) as a lemma and so $\mathcal{L}$ (A.2) = {(A.1)}. The tree for this proof is:



For this proof,

$\mathcal{C}$ (A.2) $= 16 + \mathcal{C}$ (A.1) $= 27$
and $\mathcal{H}$ (A.2) $= 6 + \max(3, \mathcal{H}$ (A.1)$) = 13$

A.2.2 Version 2

As an alternative, we can define

$$\mathsf{cat}_{asp} \ \langle [\,], [\,] \rangle_{asp} \quad ysp = ysp$$
$$\mathsf{cat}_{asp} \ \langle x : r_0, l_0 \rangle_{asp} \quad ysp = \mathsf{cons}_{asp} \ x \ (\mathsf{cat}_{asp} \ \langle l_0, r_0 \rangle_{asp} \ ysp)$$

For this operation, we want to prove

$$|\mathsf{cat}_{asp} \ xsp \ ysp|_{asp} = |xsp|_{asp} + |ysp|_{asp} \tag{A.3}$$

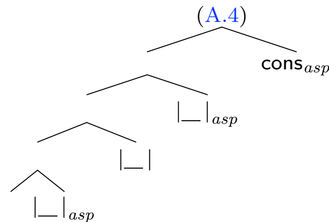
Before we prove (A.3), first we prove

$$|\mathsf{cons}_{asp} \ a \langle l, r \rangle_{asp}|_{asp} = 1 + |\langle l, r \rangle_{asp}|_{asp} \tag{A.4}$$

So,

$$
\begin{aligned}
& |\mathsf{cons}_{asp} \ a \ \langle l, r \rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\
& |\langle a : r, l \rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& |a : r| + |l| \\
= & \quad \{\text{definition of } |_|\} \\
& 1 + |r| + |l| \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
& 1 + |\langle l, r \rangle_{asp}|_{asp}
\end{aligned}
$$

The tree for this proof is:



For this proof, $\mathcal{C}\,(\text{A.4}) = 4$ and $\mathcal{H}\,(\text{A.4}) = 4$. We are now ready to prove Equation (A.3) by induction on xsp.

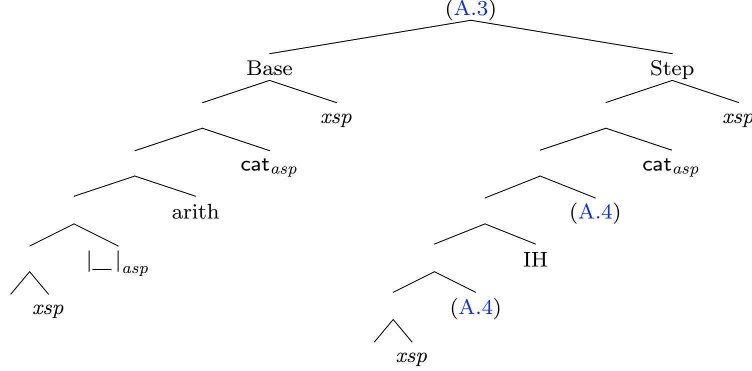
Base Case Suppose that $xsp = \langle [\,], [\,] \rangle_{asp}$ and so $|xsp|_{asp} = 0$. Then

$$
\begin{array}{ll}
 & |\mathsf{cat}_{asp}\ xsp\ ysp|_{asp} \\
= & \quad \{\text{definition of } xsp\} \\
 & |\mathsf{cat}_{asp}\ \langle [\,], [\,] \rangle_{asp}\ ysp|_{asp} \\
= & \quad \{\text{definition of } \mathsf{cat}_{asp}\} \\
 & |ysp|_{asp} \\
= & \quad \{\text{arithmetic}\} \\
 & 0 + |ysp|_{asp} \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
 & |xsp|_{asp} + |ysp|_{asp} \\
= & \quad \{\text{definition of } xsp\} \\
 & |xsp|_{asp} + |ysp|_{asp}
\end{array}
$$

Step Case Suppose that $xsp = \mathsf{cons}_{asp}\ x\ \langle l_0, r_0 \rangle_{asp}$, (and then $xsp = \langle x : r_0, l_0 \rangle_{asp}$) and $\langle l_0, r_0 \rangle_{asp}$ satisfies (A.2). Then

$$
\begin{array}{ll}
 & |\mathsf{cat}_{asp}\ xsp\ ysp\,|_{asp} \\
= & \quad \{\text{definition of } xsp\} \\
 & |\mathsf{cat}_{asp}\ \langle x : r_0, l_0 \rangle_{asp}\ ysp|_{asp} \\
= & \quad \{\text{definition of } \mathsf{cat}_{asp}\} \\
 & |\mathsf{cons}_{asp}\ x\ (\mathsf{cat}_{asp}\ \langle l_0, r_0 \rangle_{asp}\ ysp)|_{asp} \\
= & \quad \{\text{Equation (A.4)}\} \\
 & 1 + |\mathsf{cat}_{asp}\ \langle l_0, r_0 \rangle_{asp}\ ysp\,|_{asp} \\
= & \quad \{\text{induction hypothesis}\} \\
 & 1 + |\langle l_0, r_0 \rangle_{asp}|_{asp} + |ysp|_{asp} \\
= & \quad \{\text{Equation (A.4)}\} \\
 & |\mathsf{cons}\ x\ \langle l_0, r_0 \rangle_{asp}|_{asp} + |ysp|_{asp} \\
= & \quad \{\text{definition of } xsp\} \\
 & |xsp|_{asp} + |ysp|_{asp}
\end{array}
$$

The tree for this proof is:



We declare (A.4) as a lemma and so $\mathcal{L}(A.3) = \{(A.4)\}$. So, for this proof:

$$\begin{aligned} &\mathcal{C}\,(A.3) = 11 + \mathcal{C}\,(A.4) = 15 \\ \text{and } &\mathcal{H}\,(A.3) = 6 + \max(1, \mathcal{H}\,(A.4)) = 10 \end{aligned}$$

A.3 Block Split for Lists

We want to prove that

$$|xsp \ +\!\!+_{b(k)} \ ysp|_{b(k)} = |xsp|_{b(k)} + |ysp|_{b(k)} \tag{A.5}$$

The definition of $+\!\!+_{b(k)}$ is

$$\begin{aligned} \langle [\,], [\,] \rangle_{b(k)} \quad &+\!\!+_{b(k)} \ ysp = ysp \\ \langle a : l, [\,] \rangle_{b(k)} \ &+\!\!+_{b(k)} \ ysp = \mathsf{cons}_{b(k)} \ a \ (\langle l, [\,] \rangle_{b(k)} \ +\!\!+_{b(k)} \ ysp) \\ \langle a : l, r \rangle_{b(k)} \quad &+\!\!+_{b(k)} \ ysp = \mathsf{cons}_{b(k)} \ a \ (\langle l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r \rangle_{b(k)} \ +\!\!+_{b(k)} \ ysp) \end{aligned}$$

We first need to prove that

$$|\mathsf{cons}_{b(k)} \ a \langle l, r \rangle_{b(k)}|_{b(k)} = 1 + |\langle l, r \rangle_{b(k)}|_{b(k)} \tag{A.6}$$

We have two cases:

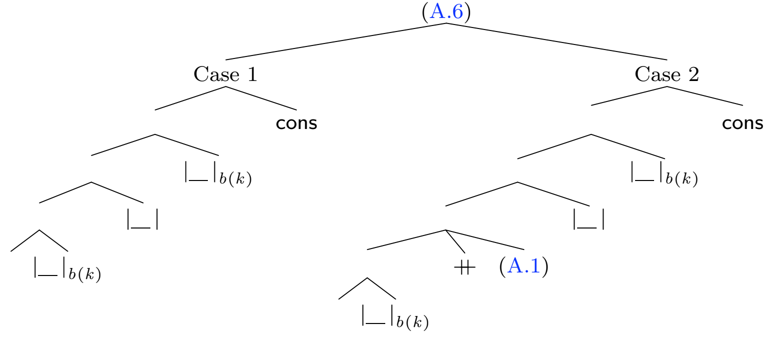
Case 1 If $|l| < k$ then

$$\begin{aligned} & |\mathsf{cons}_{b(k)} \ a \ \langle l, r \rangle_{b(k)}|_{b(k)} \\ = \quad & \{\text{definition of } \mathsf{cons}_{b(k)} \text{ with } |l| < k\} \\ & |\langle a : l, r \rangle_{bk}|_{b(k)} \\ = \quad & \{\text{definition of } |_|_{b(k)}\} \\ & |a : l| + |r| \\ = \quad & \{\text{definition of } |_|\} \\ & 1 + |l| + |r| \\ = \quad & \{\text{definition of } |_|_{b(k)}\} \\ & 1 + |\langle l, r \rangle_{b(k)}|_{b(k)} \end{aligned}$$

Case 2 If $|l| \geq k$ then

$$
\begin{aligned}
& |\mathsf{cons}_{b(k)}\ a\ \langle l, r\rangle_{b(k)}|_{b(k)} \\
= & \quad \{\text{definition of } \mathsf{cons}_{b(k)} \text{ with } |l| \geq k\} \\
& |\langle a : (\mathsf{init}\ l), (\mathsf{last}\ l) : r\rangle_{b(k)}|_{b(k)} \\
= & \quad \{\text{definition of } |_|_{b(k)}\} \\
& |a : (\mathsf{init}\ l)| + |(\mathsf{last}\ l) : r| \\
= & \quad \{\text{definition of } |_|\} \\
& 1 + |\mathsf{init}\ l| + |\mathsf{last}\ l| + |r| \\
= & \quad \{\mathsf{init}\ l +\!\!+ [\mathsf{last}\ l] = l \text{ and (A.1)}\} \\
& 1 + |l| + |r| \\
= & \quad \{\text{definition of } |_|_{b(k)}\} \\
& 1 + |\langle l, r\rangle_{b(k)}|_{b(k)}
\end{aligned}
$$

The tree for this proof is:



Since (A.1) is only used once, we do not declare it as a lemma. So, for this proof,

$$
\begin{aligned}
& \mathcal{C}\,(\text{A.6}) = 9 + \mathcal{C}\,(\text{A.1}) = 20 \\
\text{and } & \mathcal{H}\,(\text{A.6}) = 5 + \max(1, \mathcal{H}\,(\text{A.1})) = 12
\end{aligned}
$$

Now, we prove Equation (A.5) by induction on xsp.

Base Case Suppose that $xsp = \langle [\,], [\,]\rangle_{b(k)}$ and so $|xsp|_{b(k)} = 0$. Then

$$
\begin{aligned}
& |xsp +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{definition of } xsp\} \\
& |\langle [\,], [\,]\rangle_{b(k)} +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{definition of } +\!\!+_{b(k)}\} \\
& |ysp|_{b(k)} \\
= & \quad \{\text{arithmetic}\} \\
& 0 + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } xsp\} \\
& |xsp|_{b(k)} + |ysp|_{b(k)}
\end{aligned}
$$

Step Case Suppose that $xsp = \langle x_0, x_1 \rangle_{b(k)}$ (and so $|xsp|_{b(k)} = |x_0| + |x_1|$). Then we have two subcases depending on whether x_1 is empty.

Subcase 1 Suppose that $x_0 = a : l$, $x_1 = [\,]$ and $\langle l, [\,] \rangle_{b(k)}$ satisfies Equation (A.5). Then

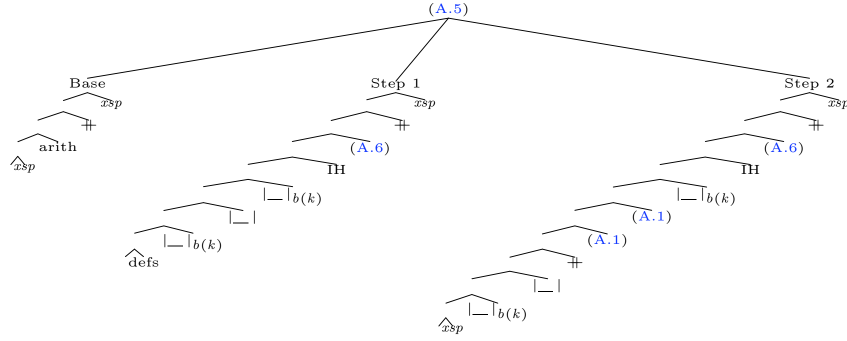
$$
\begin{array}{cl}
 & |xsp +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{definition of } xsp\} \\
 & |\langle a : l, [\,] \rangle_{b(k)} +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{definition of } +\!\!+_{b(k)}\} \\
 & |\mathsf{cons}_{b(k)}\ a\ (\langle l, [\,] \rangle_{b(k)} +\!\!+_{b(k)}\ ysp)|_{b(k)} \\
= & \quad \{\text{Equation (A.6)}\} \\
 & 1 + |\langle l, [\,] \rangle_{b(k)} +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{induction hypothesis}\} \\
 & 1 + |\langle l, [\,] \rangle_{b(k)}|_{b(k)} + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } |_|_{b(k)}\} \\
 & 1 + |l| + |[\,]| + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } |_|\} \\
 & |a : l| + |[\,]| + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } |_|_{b(k)}\} \\
 & |\langle a : l, [\,] \rangle_{b(k)}|_{b(k)} + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } xsp\} \\
 & |xsp|_{b(k)} + |ysp|_{b(k)}
\end{array}
$$

Subcase 2 Now suppose that $x_0 = a : l$, $x_1 = r$ (where $r \neq [\,]$) and $\langle l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r \rangle_{b(k)}$ satisfies (A.5). Then

$$
\begin{array}{cl}
 & |xsp +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{definition of } xsp\} \\
 & |\langle a : l, r \rangle_{b(k)} +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{definition of } +\!\!+_{b(k)}\} \\
 & |\mathsf{cons}_{b(k)}\ a\ (\langle l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r \rangle_{b(k)} +\!\!+_{b(k)}\ ysp)|_{b(k)} \\
= & \quad \{\text{Equation (A.6)}\} \\
 & 1 + |\langle l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r \rangle_{b(k)} +\!\!+_{b(k)}\ ysp|_{b(k)} \\
= & \quad \{\text{induction hypothesis}\} \\
 & 1 + |\langle l +\!\!+ [\mathsf{head}\ r], \mathsf{tail}\ r \rangle_{b(k)}|_{b(k)} + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } |_|_{b(k)}\} \\
 & 1 + |l +\!\!+ [\mathsf{head}\ r]| + |\mathsf{tail}\ r| + |ysp|_{b(k)} \\
= & \quad \{\text{Equation (A.1)}\}
\end{array}
$$

$$
\begin{aligned}
& 1 + |l| + |[\mathsf{head}\ r]| + |\mathsf{tail}\ r| + |ysp|_{b(k)} \\
= & \quad \{\text{Equation (A.1)}\} \\
& 1 + |l| + |[\mathsf{head}\ r] + \!\!+ \ \mathsf{tail}\ r| + |ysp|_{b(k)} \\
= & \quad \{[\mathsf{head}\ r] + \!\!+ (\mathsf{tail}\ r) = r\} \\
& 1 + |l| + |r| + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } |_|\} \\
& |a : l| + |r| + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } |_|_{b(k)}\} \\
& |\langle a : l, r\rangle_{b(k)}|_{b(k)} + |ysp|_{b(k)} \\
= & \quad \{\text{definition of } xsp\} \\
& |xsp|_{b(k)} + |ysp|_{b(k)}
\end{aligned}
$$

The tree for the proof of (A.5) is:



We declare (A.1) and (A.6) as lemmas and so $\mathcal{L}$ (A.5) = {(A.1), (A.6)}. Note that $\mathcal{C}$ (A.6) = 9 + $\mathcal{C}$ (A.1) and so

$$\mathcal{C}\,(\text{A.5}) = 23 + 9 + \mathcal{C}\,(\text{A.1}) = 43$$
and $\mathcal{H}\,(\text{A.5}) = 4 + \max(7, \mathcal{H}\,(\text{A.6}), 4 + \mathcal{H}\,(\text{A.1})) = 16$

A.4 Augmented Split

We want to prove that

$$|\mathsf{cat}_A\ xsp\ ysp|_A = |xsp|_A + |ysp|_A \tag{A.7}$$

The definition of cat_A is

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

We first need to prove

$$|\mathsf{cons}_A\ m\ x\ \langle d, l, r\rangle_A|_A = 1 + |\langle d, l, r\rangle_A|_A \tag{A.8}$$

We have two cases depending on whether $m = 0$.

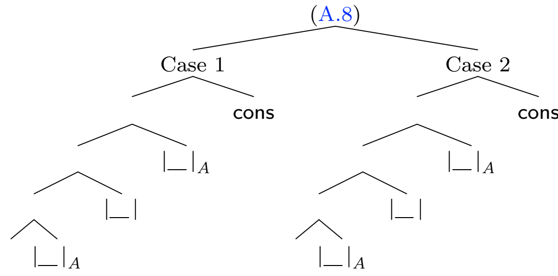
Case 1 Suppose that $m = 0$.

$$
\begin{array}{ll}
& |\mathsf{cons}_A \ m \ x \ \langle d, l, r \rangle_A|_A \\
= & \quad \{\text{definition of } \mathsf{cons}_A\} \\
& |\langle (2d + m), (x : l), r \rangle_A|_A \\
= & \quad \{\text{definition of } |_|_A\} \\
& |x : l| + |r| \\
= & \quad \{\text{definition of } |_|\} \\
& 1 + |l| + |r| \\
= & \quad \{\text{definition of } |_|_A\} \\
& 1 + |\langle d, l, r \rangle_A|_A
\end{array}
$$

Case 2 Suppose that $m \neq 0$.

$$
\begin{array}{ll}
& |\mathsf{cons}_A \ m \ x \ \langle d, l, r \rangle_A|_A \\
= & \quad \{\text{definition of } \mathsf{cons}_A\} \\
& |\langle (2d + m), l, (x : r) \rangle_A|_A \\
= & \quad \{\text{definition of } |_|_A\} \\
& |l| + |x : r| \\
= & \quad \{\text{definition of } |_|\} \\
& 1 + |l| + |r| \\
= & \quad \{\text{definition of } |_|_A\} \\
& 1 + |\langle d, l, r \rangle_A|_A
\end{array}
$$

The tree for this proof is:



For this proof, $\mathcal{C}$ (A.8) = 8 and $\mathcal{H}$ (A.8) = 5. We now prove (A.7) by induction on xsp.

Base Case Suppose that $xsp = \langle n, [\,], [\,] \rangle_A$. Then $|xsp|_A = 0$ and so

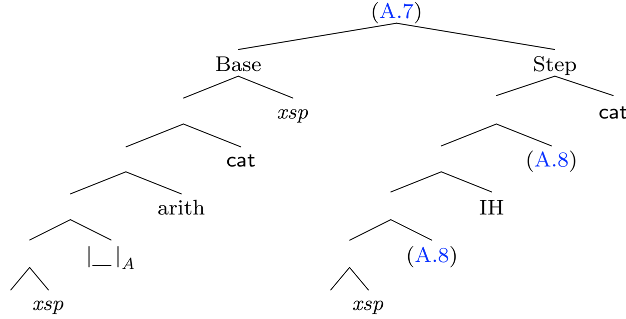
$$
\begin{array}{ll}
& |\mathsf{cat}_A \ xsp \ ysp|_A \\
= & \quad \{\text{definition of } xsp\} \\
& |\mathsf{cat}_A \ \langle n, [\,], [\,] \rangle_A \ ysp|_A \\
= & \quad \{\text{definition of } \mathsf{cat}_A\} \\
& |ysp|_A \\
= & \quad \{\text{arithmetic}\}
\end{array}
$$

$$
\begin{array}{ll}
 & 0 + |ysp|_A \\
= & \quad \{\text{definition of } |_|_A\} \\
 & |\langle n, [\,], [\,]\rangle_A|_A + |ysp|_A \\
= & \quad \{\text{definition of } xsp\} \\
 & |xsp|_A + |ysp|_A
\end{array}
$$

Step Case Suppose that $xsp = \mathsf{cons}_A\ b\ h\ tsp$ and then by Equation (4.25) $(b, h, tsp) = \mathsf{headTail}_A\ xsp$. Also suppose that tsp satisfies Equation (A.7). So

$$
\begin{array}{ll}
 & |\mathsf{cat}_A\ xsp\ ysp|_A \\
= & \quad \{\text{definition of } \mathsf{cat}_A\} \\
 & |\mathsf{cons}_A\ b\ h\ (\mathsf{cat}_A\ tsp\ ysp)|_A \\
= & \quad \{\text{Equation (A.8)}\} \\
 & 1 + |\mathsf{cat}_A\ tsp\ ysp|_A \\
= & \quad \{\text{induction hypothesis}\} \\
 & 1 + |tsp|_A + |ysp|_A \\
= & \quad \{\text{Equation (A.8)}\} \\
 & |\mathsf{cons}_A\ b\ h\ tsp|_A + |ysp|_A \\
= & \quad \{\text{definition of } xsp\} \\
 & |xsp|_A + |ysp|_A
\end{array}
$$

The tree for the proof of (A.7) is:



We declare (A.8) as a lemma (then $\mathcal{L}(\text{A.7}) = \{(\text{A.8})\}$) and so:

$$\mathcal{C}\,(\text{A.7}) = 10 + \mathcal{C}\,(\text{A.8}) = 18$$
and $\mathcal{H}\,(\text{A.7}) = 5 + \max(1, \mathcal{H}\,(\text{A.8})) = 10$

A.5 Augmented Block Split

We want to prove that

$$|xsp +\!\!+_B\ ysp|_B = |xsp|_B + |ysp|_B \tag{A.9}$$

The definition of $+\!\!+_B$ is

$$
\begin{array}{rl}
\langle k, [\,], [\,]\rangle_B +\!\!+_B \langle k', l', r'\rangle_B & = \langle k', l', r'\rangle_B \\
\langle k, l, r\rangle_B +\!\!+_B \langle k', l', r'\rangle_B & = \langle |l +\!\!+ r|, l +\!\!+ r, l' +\!\!+ r'\rangle_B
\end{array}
$$

We have two cases to prove.

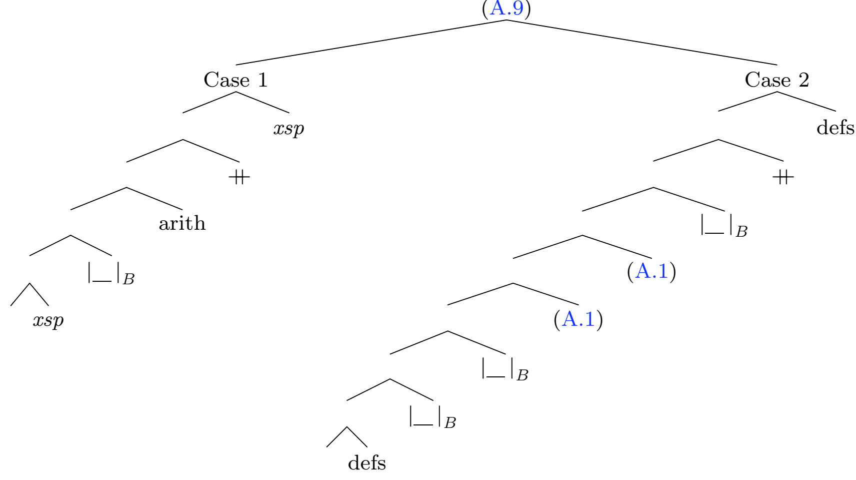
Case 1 Suppose that $xsp = \langle k, [\,], [\,] \rangle_B$. Then $|xsp|_B = 0$ and

$$
\begin{array}{ll}
 & |xsp +\!\!+_B\ ysp|_B \\
= & \quad \{\text{definition of } xsp\} \\
 & |\langle k, [\,], [\,] \rangle_B +\!\!+_B\ ysp|_B \\
= & \quad \{\text{definition of } +\!\!+_B\} \\
 & |ysp|_B \\
= & \quad \{\text{arithmetic}\} \\
 & 0 + |ysp|_B \\
= & \quad \{\text{definition of } |_|_B\} \\
 & |\langle k, [\,], [\,] \rangle_B|_B + |ysp|_B \\
= & \quad \{\text{definition of } xsp\} \\
 & |xsp|_B + |ysp|_B
\end{array}
$$

Case 2 Suppose that $xsp = \langle k, l, r \rangle_B$ and $ysp = \langle k', l', r' \rangle_B$. Then $|xsp|_B = |l| + |r|$ and $|ysp|_B = |l'| + |r'|$ and so

$$
\begin{array}{ll}
 & |xsp +\!\!+_B\ ysp|_B \\
= & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\
 & |\langle k, l, r \rangle_B +\!\!+_B \langle k', l', r' \rangle_B|_B \\
= & \quad \{\text{definition of } +\!\!+_B\} \\
 & |\langle |l +\!\!+ r|, l +\!\!+ r, l' +\!\!+ r' \rangle_B|_B \\
= & \quad \{\text{definition of } |_|_B\} \\
 & |l +\!\!+ r| + |l' +\!\!+ r'| \\
= & \quad \{\text{Equation (A.1)}\} \\
 & |l| + |r| + |l' +\!\!+ r'| \\
= & \quad \{\text{Equation (A.1)}\} \\
 & |l| + |r| + |l'| + |r'| \\
= & \quad \{\text{definition of } |_|_B\} \\
 & |\langle k, l, r \rangle_B|_B + |l'| + |r'| \\
= & \quad \{\text{definition of } |_|_B\} \\
 & |\langle k, l, r \rangle_B|_B + |\langle k', l', r' \rangle_B|_B \\
= & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\
 & |xsp|_B + |ysp|_B
\end{array}
$$

The tree for the proof of (A.9) is:



We declare (A.1) as a lemma (then $\mathcal{L}$ (A.9) $= \{$(A.1)$\}$) and so for this proof,

$$\begin{array}{l} \mathcal{C}\,(\text{A.9}) = 13 + \mathcal{C}\,(\text{A.1}) = 24 \\ \text{and } \mathcal{H}\,(\text{A.9}) = 6 + \max(3, \mathcal{H}\,(\text{A.1})) = 13 \end{array}$$

A.6 Padded Block Split

We want to prove that

$$|xsp \mathbin{+\!\!+}_P ysp|_P = |xsp|_P + |ysp|_P \tag{A.10}$$

The definition of $\mathbin{+\!\!+}_P$ is

$$\begin{array}{rl} \langle 0, l, [\,]\rangle_P \mathbin{+\!\!+}_P \langle m, l', r'\rangle_P &= \langle m, l' \mathbin{+\!\!+} l, r'\rangle_P \\ \langle n, l, r\rangle_P \mathbin{+\!\!+}_P \langle m, l', r'\rangle_P &= \langle |lt| + |r|, lt \mathbin{+\!\!+} r \mathbin{+\!\!+} ld' \mathbin{+\!\!+} ld, lt' \mathbin{+\!\!+} r'\rangle_P \\ & \quad \text{where } lt = \mathsf{take}\ n\ l \\ & \quad\quad\quad\; ld = \mathsf{drop}\ n\ l \\ & \quad\quad\quad\; lt' = \mathsf{take}\ m\ l' \\ & \quad\quad\quad\; ld' = \mathsf{drop}\ m\ l' \end{array}$$

By the definition of take then

$$|\mathsf{take}\ n\ l| = \mathsf{max}\,(|l|, n)$$

Let $ysp = \langle m, l', r'\rangle_P$ with $|l'| \geq m$ (and so $|ysp|_P = m + |r'|$) and we prove Equation (A.10) by considering two cases.

Case 1 Let $xsp = \langle 0, l, [\,]\rangle_P$ (and so $|xsp|_P = 0$). Then

$$\begin{array}{cl} & |xsp \mathbin{+\!\!+}_P ysp|_P \\ = & \quad \{\text{definitions of } xsp \text{ and } ysp\} \\ & |\langle 0, l, [\,]\rangle_P \mathbin{+\!\!+}_P \langle m, l', r'\rangle_P|_P \\ = & \quad \{\text{definition of } \mathbin{+\!\!+}_P\} \\ & |\langle m, l' \mathbin{+\!\!+} l, r'\rangle_P|_P \end{array}$$

$$
\begin{aligned}
= & \quad \{\text{definition of } |_|_P\} \\
& m + |r'| \\
= & \quad \{\text{arithmetic}\} \\
& 0 + m + |r'| \\
= & \quad \{\text{definition of } |_|_P\} \\
& |\langle 0, l, [\,]\rangle_P|_P + m + |r'| \\
= & \quad \{\text{definition of } |_|_P\} \\
& |\langle 0, l, [\,]\rangle_P|_P + |\langle m, l', r'\rangle_P|_P \\
= & \quad \{\text{definitions of } \mathit{xsp} \text{ and } \mathit{ysp}\} \\
& |\mathit{xsp}|_P + |\mathit{ysp}|_P
\end{aligned}
$$

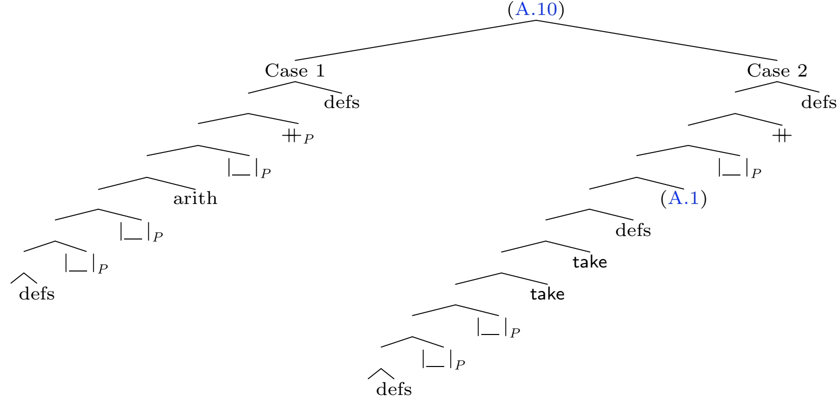
Case 2 Suppose that $\mathit{xsp} = \langle n, l, r\rangle_P$ with $|l| \geq n$ (so $|\mathit{xsp}|_P = n + |r|$) and let

$$
\begin{aligned}
& \mathit{lt} = \mathsf{take}\ n\ l \\
& \mathit{ld} = \mathsf{drop}\ n\ l \\
& \mathit{lt}' = \mathsf{take}\ m\ l' \\
& \mathit{ld}' = \mathsf{drop}\ m\ l'
\end{aligned}
$$

Then

$$
\begin{aligned}
& |\mathit{xsp} +\!\!+_P\ \mathit{ysp}|_P \\
= & \quad \{\text{definitions of } \mathit{xsp} \text{ and } \mathit{ysp}\} \\
& |\langle n, l, r\rangle_B +\!\!+_P \langle m, l', r'\rangle_P|_P \\
= & \quad \{\text{definition of } +\!\!+_P\} \\
& |\langle |\mathit{lt}| + |r|, \mathit{lt} +\!\!+ r +\!\!+ \mathit{ld}' +\!\!+ \mathit{ld}, \mathit{lt}' +\!\!+ r'\rangle_P|_P \\
= & \quad \{\text{definition of } |_|_P\} \\
& |\mathit{lt}| + |r| + |\mathit{lt}' +\!\!+ r'| \\
= & \quad \{\text{Equation (A.1)}\} \\
& |\mathit{lt}| + |r| + |\mathit{lt}'| + |r'| \\
= & \quad \{\text{definitions of } \mathit{lt} \text{ and } \mathit{lt}'\} \\
& |\mathsf{take}\ n\ l| + |r| + |\mathsf{take}\ m\ l'| + |r'| \\
= & \quad \{\text{property of } \mathsf{take}\} \\
& |n| + |r| + |\mathsf{take}\ m\ l'| + |r'| \\
= & \quad \{\text{property of } \mathsf{take}\} \\
& |n| + |r| + |m| + |r'| \\
= & \quad \{\text{definition of } |_|_P\} \\
& |\langle n, l, r\rangle_P|_P + |m| + |r'| \\
= & \quad \{\text{definition of } |_|_P\} \\
& |\langle n, l, r\rangle_P|_P + |\langle m, l', r'\rangle_P|_P \\
= & \quad \{\text{definitions of } \mathit{xsp} \text{ and } \mathit{ysp}\} \\
& |\mathit{xsp}|_P + |\mathit{ysp}|_P
\end{aligned}
$$

The tree for the proof of (A.10) is:



We do not declare (A.1) as a lemma and so

$$\begin{aligned} &\mathcal{C}\,(\text{A.10}) = 16 + \mathcal{C}\,(\text{A.1}) = 27 \\ \text{and } &\mathcal{H}\,(\text{A.10}) = 5 + \max(6, \mathcal{H}\,(\text{A.1})) = 12 \end{aligned}$$

A.7 Comparing the proofs

Below is a table which summarises the results for each of the proof trees:

Operation	$\mathcal{C}$	$\mathcal{H}$
$+\!\!+$	11	7
$+\!\!+_{asp}$	27	13
cat_{asp}	15	10
$+\!\!+_{b(k)}$	43	16
cat_{A}	18	10
$+\!\!+_{B}$	24	13
$+\!\!+_{P}$	27	12

We can see that for this assertion $+\!\!+_{b(k)}$ produces the best obfuscation and cat_{asp} produces the worst. Considering the two operations for the alternating split we find that for this assertion, the operation $+\!\!+_{asp}$ produces the best obfuscation. There are two reasons for this:

- The definition of cat_{asp} follows a similar pattern to the definition of $+\!\!+$ and so the proofs for $+\!\!+$ and cat_{asp} will have a similar structure.

- The definition of $+\!\!+_{asp}$ uses the operation $+\!\!+$ so the proof of the assertion for $+\!\!+_{asp}$ uses the assertion for $+\!\!+$.

Likewise, as the structure of cat_{A} is similar to that of $+\!\!+$, the proof is only slightly more complicated and since $+\!\!+_{B}$ uses $+\!\!+$ in its definition then it has a longer proof. The definition of $+\!\!+_{b(k)}$ is the only operation to have three distinct cases and so produces the longest proof.

Appendix B

Set operations

In this appendix, we prove that the operation insert_{asp} from Section 5.3.1 is correct.

B.1 Span Properties

Before we show the proof of correctness, we will need results about span, where span is defined to be:

$$\begin{aligned} &\mathsf{span}\ p\ [\,] &&= [\,] \\ &\mathsf{span}\ p\ (x : xs) &&= \text{if } p\ x \text{ then } (x : ys, zs) \text{ else } ([\,], xs) \\ & &&\quad \text{where } (ys, zs) = \mathsf{span}\ p\ xs \end{aligned}$$

Property 9. If $(ys, zs) = \mathsf{span}\ p\ xs$ then $xs = ys +\!\!+ zs$.

Proof. We prove this by induction on xs.

Base Case Suppose that $xs = [\,]$ then by the definition of span, $([\,], [\,]) = \mathsf{span}\ p\ [\,]$ and the result is immediately true.

Step Case Suppose that $xs = v : vs$ and for the induction hypothesis, we suppose that if $(ys, zs) = \mathsf{span}\ p\ vs$ then $vs = ys +\!\!+ zs$. Now let us consider $\mathsf{span}\ p\ (v : vs)$

If $\neg p\ v$ then

$$\mathsf{span}\ p\ (v : vs) = ([\,], v : vs)$$

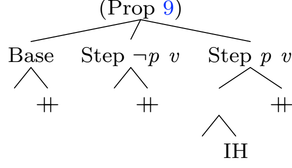
and so the result is immediate.

If $p\ v$ then by the definition of span then

$$(v : ys, zs) = \mathsf{span}\ p\ (v : vs)$$

Now $(v : ys) +\!\!+ zs = v : (ys +\!\!+ zs) = v : vs$ □

Now let us draw the proof tree for this property (this proof tree is needed in Section 5.4).



For this proof, $\mathcal{C}$ (Prop 9) = 4 and $\mathcal{H}$ (Prop 9) = 3.

Property 10. Suppose that $xs \leadsto \langle l, r\rangle_{asp}$, where xs is a strictly-increasing list. So

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

for some a. Then

(i) if $|ly| = |ry|$ then (B.1)

$$\mathsf{span}\ (< a)\ xs = (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}, \mathsf{unsplit}_{asp}\ \langle lz, rz\rangle_{asp})$$

(ii) if $|ly| \neq |ry|$ then (B.2)

$$\mathsf{span}\ (< a)\ xs = (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}, \mathsf{unsplit}_{asp}\ \langle rz, lz\rangle_{asp})$$

Proof. First let us prove (i) by induction on xs and we suppose that $|ly| = |ry|$

Base Case If $xs = [\,]$, $l = [\,]$ and $r = [\,]$. Since $\mathsf{span}\ (< a)\ [\,] = ([\,],[\,])$ then the result holds immediately.

Step Case We have two subcases.

Subcase 1 Suppose that $ly = [\,]$, $ry = [\,]$ and so $lz = l$ and $rz = r$. Since xs is an increasing list and by the definition of span either $l = [\,]$ or $\mathsf{head}\ l \geq a$. Since $\mathsf{head}\ xs = \mathsf{head}\ l$ then,

$$\begin{array}{cl} & \mathsf{span}\ (< a)\ xs \\ = & \quad \{\text{definition of } \mathsf{span}\} \\ & ([\,], xs) \\ = & \quad \{\text{definitions}\} \\ & (\mathsf{unsplit}_{asp}\ \langle [\,],[\,]\rangle_{asp}, \mathsf{unsplit}_{asp}(\langle l, r\rangle_{asp})) \\ = & \quad \{\text{definitions}\} \\ & (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}, \mathsf{unsplit}_{asp}\ \langle lz, rz\rangle_{asp}) \end{array}$$

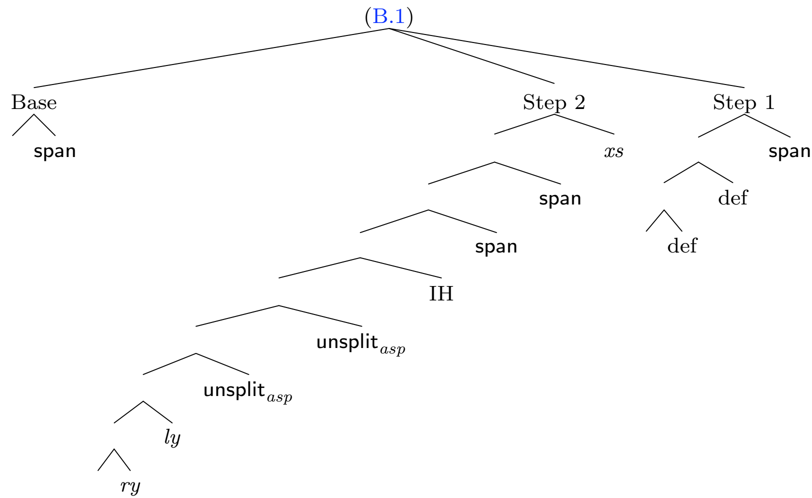
Subcase 2 Now suppose that $ly \neq [\,]$. Since $ry \neq [\,]$ there must be $s < t < a$ such that $xs = s : t : xs'$. Then $xs' \leadsto \langle l', r'\rangle_{asp}$ where $l = s : l'$ and $r = t : r'$. Suppose that $(ly', lz) = \mathsf{span}\ (< a)\ l'$ and $(ry', rz) = \mathsf{span}\ (< a)\ r'$. For the

induction hypothesis, we suppose that xs' (and ly' and ry') satisfy Equation (B.1).

$$
\begin{aligned}
& \mathsf{span}\ (<a)\ xs \\
= & \quad \{\text{definition of } xs\} \\
& \mathsf{span}\ (<a)\ (s : t : xs') \\
= & \quad \{\text{definition of } \mathsf{span} \text{ with } s < t < a\} \\
& (s : ys'', zs'') \text{ where } (ys'', zs'') = \mathsf{span}\ (<a)\ (t : xs') \\
= & \quad \{\text{definition of } \mathsf{span} \text{ with } t < a\} \\
& (s : t : ys', zs') \text{ where } (ys', zs') = \mathsf{span}\ (<a)\ xs' \\
= & \quad \{\text{induction hypothesis}\} \\
& (s : t : (\mathsf{unsplit}_{asp}\ \langle ly', ry' \rangle_{asp}), \mathsf{unsplit}_{asp}\ (\langle lz, rz \rangle_{asp})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
& (s : (\mathsf{unsplit}_{asp}\ \langle t : ry', ly' \rangle_{asp}), \mathsf{unsplit}_{asp}\ (\langle lz, rz \rangle_{asp})) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\langle s : ly', t : ry' \rangle_{asp}), \mathsf{unsplit}_{asp}\ (\langle lz, rz \rangle_{asp}) \\
= & \quad \{(s : ly', lz) = \mathsf{span}\ (<a)\ (s : l') \text{ and } l = s : l' \text{ so } ly = s : ly'\} \\
& \mathsf{unsplit}_{asp}\ (\langle ly, t : ry' \rangle_{asp}), \mathsf{unsplit}_{asp}\ (\langle lz, rz \rangle_{asp}) \\
= & \quad \{\text{similarly } ry = t : ry'\} \\
& \mathsf{unsplit}_{asp}\ (\langle ly, ry \rangle_{asp}), \mathsf{unsplit}_{asp}\ (\langle lz, rz \rangle_{asp})
\end{aligned}
$$

□

We can draw the proof tree as follows:



Thus for this proof, $\mathcal{C}$ (B.1) = 12 and $\mathcal{H}$ (B.1) = 9.

Proof of (B.2) We can use the previous result to prove (B.2). This gives us a shorter (text) proof but the tree for this proof contains the tree for the proof of (B.1). So if we want to use or compare this proof then we should prove the result directly.

Proof. Suppose that $|ly| \neq |ry|$ and since we have strictly-increasing lists, $|ly| = 1 + |ry|$. Thus we can find some $s < a$ and lists l' and ly' such that $ly = s : ly'$ and $l = s : l'$ and then $|ly'| = |ry|$. By the definition of span (and $s < a$), $\mathsf{span}\ (< a)\ l' = (ly', lz)$. Consider xs' where $xs = s : xs'$. Then by the definition of split_{asp}, $xs' \rightsquigarrow \langle r, l' \rangle_{asp}$ and since $|ry| = |ly'|$, xs' satisfies (B.1). So

$$
\begin{array}{ll}
 & \mathsf{span}\ (< a)\ xs \\
= & \quad \{\text{definition of } xs'\} \\
 & \mathsf{span}\ (< a)\ (s : xs') \\
= & \quad \{\text{definition of } \mathsf{span} \text{ and } s < a\} \\
 & (s : ys', zs') \ \ \text{where } (ys', zs') = \mathsf{span}\ (< a)\ xs' \\
= & \quad \{\text{Equation (B.1)}\} \\
 & (s : (\mathsf{unsplit}_{asp}\ \langle ry, ly' \rangle_{asp}), \mathsf{unsplit}\ \langle rz, lz \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
 & (\mathsf{unsplit}_{asp}\ \langle s : ly', lz \rangle_{asp}, \mathsf{unsplit}\ \langle rz, lz \rangle_{asp}) \\
= & \quad \{ly = s : ly'\} \\
 & (\mathsf{unsplit}_{asp}\ \langle ly, lz \rangle_{asp}, \mathsf{unsplit}\ \langle rz, lz \rangle_{asp})
\end{array}
$$

□

Property 11. With $xs \rightsquigarrow \langle l, r \rangle_{asp}$ and ly and ry as above. If $\mathsf{member}\ xs\ a$ then

$$(i)\ \mathsf{member}\ l\ a \ \Leftrightarrow\ |ly| = |ry| \tag{B.3}$$
$$(ii)\ \mathsf{member}\ r\ a \ \Leftrightarrow\ |ly| \neq |ry| \tag{B.4}$$

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

Proof of (i) ($\Rightarrow$) Since $\mathsf{member}\ l\ a$ then by the definition of split_{asp}, $(\exists k)$ such that $xs\ !!\ 2k = a$. Since xs is an increasing list then the number of elements of xs that are less than a is $2k$. So,

$$
\begin{array}{ll}
 & \mathsf{member}\ l\ a \\
\Rightarrow & \quad \{\text{definition of } asp,\ xs \text{ is increasing}\} \\
 & |\mathsf{takeWhile}\ (< a)\ xs| = 2k \\
\Rightarrow & \quad \{\text{definition of } \mathsf{span}\} \\
 & |ys| = 2k \\
\Rightarrow & \quad \{\text{Property (B.1)}\} \\
 & |\mathsf{unsplit}\ \langle ly, ry \rangle_{asp}| = 2k \\
\Rightarrow & \quad \{\text{definition of } |_|_{asp}\} \\
 & |ly| + |ry| = 2k \\
\Rightarrow & \quad \{\text{invariant (5.3), arithmetic}\}
\end{array}
$$

$|ly| = |ry|$

Proof of (i) $(\Leftarrow)$ If $\mathsf{member}\ xs\ a$ then $\mathsf{head}\ zs = a$ and

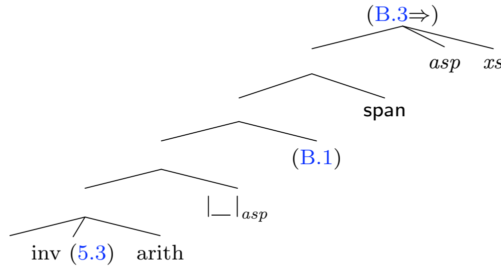
$$
\begin{array}{ll}
 & |ly| = |ry| \\
\Rightarrow & \quad \{\text{Property (B.1)}\} \\
 & \mathsf{unsplit}\ \langle lz, rz\rangle_{asp} = zs \\
\Rightarrow & \quad \{\text{applying } \mathsf{head} \text{ to both sides (which are finite non-empty lists)}\} \\
 & \mathsf{head}\ (\mathsf{unsplit}\ \langle lz, rz\rangle_{asp}) = \mathsf{head}\ zs \\
\Rightarrow & \quad \{\mathsf{member}\ xs\ a\} \\
 & \mathsf{head}\ (\mathsf{unsplit}\ \langle lz, rz\rangle_{asp}) = a \\
\Rightarrow & \quad \{\text{property of } \mathsf{head}_{asp}\} \\
 & \mathsf{head}_{asp}\ \langle lz, rz\rangle_{asp} = a \\
\Rightarrow & \quad \{\text{definition of } \mathsf{head}_{asp}\} \\
 & \mathsf{head}\ lz = a \\
\Rightarrow & \quad \{\text{property of } \mathsf{member}\} \\
 & \mathsf{member}\ l\ a
\end{array}
$$

Proof of (ii) We can prove this result in the same way as we proved (i) or we can note that:

$$\neg(\mathsf{member}\ l\ a) \Leftrightarrow \neg(|ly| = |ry|)$$

Since $\mathsf{member}\ xs\ a$ then $\neg(\mathsf{member}\ l\ a) \equiv \mathsf{member}\ r\ a$ and so (ii) holds. □

The tree for (i) $(\Rightarrow)$ is



Since (B.1) is only used once for this proof we do not declare it as a lemma (note that if we wanted the tree for the entire proof then (B.1) would be declared as a lemma). Thus

$$
\begin{array}{lll}
\mathcal{C}\,(\text{B.3} \Rightarrow) & = & 6 + \mathcal{C}\,(\text{B.1}) = 18 \\
\mathcal{H}\,(\text{B.3} \Rightarrow) & = & 3 + \max(2, \mathcal{H}\,(\text{B.1})) = 12
\end{array}
$$

B.2 Insertion for the alternating split

To prove the correctness of insert_{asp}, we need to show that Equation (3.6) holds for insert_{asp}. Therefore, we are required to prove:

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

We will need the properties from Sections 4.3.2 and B.1 and use the definition of $+\!\!+_{asp}$. The definition of insert_{asp} is

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

Suppose that $xs \rightsquigarrow \langle l, r\rangle_{asp}$ and consider $\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}$. Let $(ly, lz) = \mathsf{span}\ (< a)\ l$, $(ry, rz) = \mathsf{span}\ (< a)\ r$ and $(ys, zs) = \mathsf{span}\ (< a)\ xs$. We have four cases corresponding to the conditions in the definition of insert_{asp}.

Case (i) Suppose that $\mathsf{member}\ a\ lz$ and $\mathsf{head}\ zs = a$. Since $\mathsf{member}\ a\ l$ then by Equation (B.3), $|ly| = |ry|$. So,

$$\begin{array}{ll}
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ (\mathsf{split}_{asp}\ xs)) \\
= & \quad \{xs \rightsquigarrow \langle l, r\rangle_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\langle ly, ry\rangle_{asp}\ +\!\!+_{asp}\ \langle lz, rz\rangle_{asp}) \\
= & \quad \{\text{property of } +\!\!+_{asp}\} \\
& (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp})\ +\!\!+\ (\mathsf{unsplit}_{asp}\ \langle lz, rz\rangle_{asp}) \\
= & \quad \{\text{Property (B.1)}\} \\
& ys +\!\!+ zs \\
= & \quad \{\text{definition of } \mathsf{insert} \text{ with } \mathsf{head}\ zs = a\} \\
& \mathsf{insert}\ a\ xs
\end{array}$$

Case (ii) Now suppose that $\neg(\mathsf{member}\ a\ lz)$ and $|ly| = |ry|$. By Equation (B.3), $\neg(\mathsf{member}\ a\ xs)$. Then,

$$\begin{array}{ll}
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ (\mathsf{split}_{asp}\ xs)) \\
= & \quad \{xs \rightsquigarrow \langle l, r\rangle_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\langle ly, ry\rangle_{asp}\ +\!\!+_{asp}\ \langle a : rz, lz\rangle_{asp}) \\
= & \quad \{\text{property of } +\!\!+_{asp}\} \\
& (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp})\ +\!\!+\ (\mathsf{unsplit}_{asp}\ \langle a : rz, lz\rangle_{asp})
\end{array}$$

$$
\begin{aligned}
= &\quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
& (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}) +\!\!+ (a : \mathsf{unsplit}_{asp}\ \langle lz, rz\rangle_{asp}) \\
= &\quad \{\text{Property (B.1)}\} \\
& ys +\!\!+ (a : zs) \\
= &\quad \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ xs
\end{aligned}
$$

Case (iii) Now suppose that $\mathsf{member}\ rz\ a$ and then $\mathsf{head}\ zs = a$. Since $\mathsf{member}\ r\ a$ then by Equation (B.4), $|ly| \neq |ry|$. So,

$$
\begin{aligned}
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ (\mathsf{split}_{asp}\ xs)) \\
= &\quad \{xs \rightsquigarrow \langle l, r\rangle_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= &\quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\langle ly, ry\rangle_{asp} +\!\!+_{asp} \langle rz, lz\rangle_{asp}) \\
= &\quad \{\text{property of } +\!\!+_{asp}\} \\
& (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}) +\!\!+ (\mathsf{unsplit}_{asp}\ \langle rz, lz\rangle_{asp}) \\
= &\quad \{\text{Property (B.2)}\} \\
& ys +\!\!+ zs \\
= &\quad \{\text{definition of } \mathsf{insert} \text{ with } \mathsf{head}\ zs = a\} \\
& \mathsf{insert}\ a\ xs
\end{aligned}
$$

Case (iv) Now suppose that $\neg(\mathsf{member}\ xs\ a)$ and $|ly| \neq |ry|$. Then

$$
\begin{aligned}
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ (\mathsf{split}_{asp}\ xs)) \\
= &\quad \{xs \rightsquigarrow \langle l, r\rangle_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\mathsf{insert}_{asp}\ a\ \langle l, r\rangle_{asp}) \\
= &\quad \{\text{definition of } \mathsf{insert}_{asp}\} \\
& \mathsf{unsplit}_{asp}\ (\langle ly, ry\rangle_{asp} +\!\!+_{asp} \langle a : lz, rz\rangle_{asp}) \\
= &\quad \{\text{property of } +\!\!+_{asp}\} \\
& (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}) +\!\!+ (\mathsf{unsplit}_{asp}\ \langle a : lz, rz\rangle_{asp}) \\
= &\quad \{\text{definition of } \mathsf{unsplit}_{asp}\} \\
& (\mathsf{unsplit}_{asp}\ \langle ly, ry\rangle_{asp}) +\!\!+ (a : \mathsf{unsplit}_{asp}\ \langle rz, lz\rangle_{asp}) \\
= &\quad \{\text{Property (B.2)}\} \\
& ys +\!\!+ (a : zs) \\
= &\quad \{\text{definition of } \mathsf{insert}\} \\
& \mathsf{insert}\ a\ xs
\end{aligned}
$$

Appendix C

Matrices

We prove the assertion that tranpose is an involution, *i.e.*

$$\mathsf{transpose} \cdot \mathsf{transpose} = id \tag{C.1}$$

Note that this equation is only valid for lists of lists that represent matrices.

C.1 Rewriting the definition

To prove Equation (C.1), we give alternative formulations of the definition of transpose. First, we can write the definition of transpose using foldr instead of foldr1:

$$\mathsf{transpose}\ mss = \mathsf{foldr}\ (\mathsf{zipWith}\ (+\!\!+))\ blanks\ (\mathsf{map}\ (\mathsf{map\ wrap})\ mss)$$

where we define $blanks$ to be repeat $[\,]$ (*i.e.* $blanks = [\,[\,], [\,], , \ldots])$. By the Fold Map Fusion (Property 3) for lists, we get:

$$\mathsf{transpose}\ = \mathsf{foldr}\ ((\mathsf{zipWith}\ (+\!\!+)) \cdot (\mathsf{map\ wrap}))\ blanks$$

Can we simplify this definition? Let

$$f_1 = (\mathsf{zipWith}\ (+\!\!+)) \cdot (\mathsf{map\ wrap})$$

and so f_1 has type $List\ \alpha \rightarrow List\ (List\ \alpha) \rightarrow List\ (List\ \alpha)$. Then consider $f_1\ xs\ yss$ where $xs :: List\ \alpha$ and $ys :: List\ (List\ \alpha)$. If $xs = [\,]$ then $f_1\ xs\ yss = [\,]$. However if $yss = [[\,]]$ and $xs = v : vs$ then $f_1\ xs\ yss = [[v]]$.

Now suppose that $xs = v : vs$ and $yss = ws : wss$. Then

$$
\begin{array}{ll}
 & f_1\ (v : vs)\ (ws : wss) \\
= & \quad \{\text{definitions}\} \\
 & ((\mathsf{zipWith}\ (+\!\!+)) . (\mathsf{map\ wrap}))\ (v : vs)\ (ws : wss) \\
= & \quad \{\text{definition of } \mathsf{map}\} \\
 & \mathsf{zipWith}\ (+\!\!+)((\mathsf{wrap}\ v) : (\mathsf{map\ wrap}\ vs))\ (ws : wss) \\
= & \quad \{\text{definition of } \mathsf{wrap}\} \\
 & \mathsf{zipWith}\ (+\!\!+)([v] : (\mathsf{map\ wrap}\ vs))\ (ws : wss) \\
= & \quad \{\text{definition of } \mathsf{zipWith}\}
\end{array}
$$

$$
\begin{aligned}
& ([v] + ws) : (((\mathsf{zipWith}\ (+\!\!+)) . (\mathsf{map\ wrap}))\ vs\ wss) \\
= & \quad \{\text{property of : and } +\!\!+\} \\
& (v : ws) : (((\mathsf{zipWith}\ (+\!\!+)) . (\mathsf{map\ wrap}))\ vs\ wss) \\
= & \quad \{\text{definition of } f\} \\
& (v : ws) : (f_1\ vs\ wss)
\end{aligned}
$$

Now if we let $f_2 = \mathsf{zipWith}\ (:)$ then

$$f_2\ (v : vs)\ (ws : wss) = (v : ws) : (f_2\ vs\ wss)$$

which is the same pattern as above. Note that $f_2\ [\,]\ yss = [\,]$ and $f_2\ (v : vs)\ [[\,]] = [[v]]$. Thus

$$(\mathsf{zipWith}\ (+\!\!+)) \cdot (\mathsf{map\ wrap}) = \mathsf{zipWith}\ (:)$$

and so we can write

$$\mathsf{transpose} = \mathsf{foldr}\ (\mathsf{zipWith}\ (:))\ blanks$$

C.2 Using fold fusion

We need to prove Equation (C.1) and, by the results above, we need to prove that

$$\mathsf{transpose}\ (\mathsf{foldr}\ (\mathsf{zipWith}\ (:))\ blanks\ mss) = mss$$

Note that this result only holds if $\mathsf{valid}\ mss$ is true, *i.e.* when mss represents a matrix. We prove this result by using the Fold Fusion Theorem (Property 1) taking $f = \mathsf{transpose}$, $g = \mathsf{zipWith}\ (:)$ and $a = blanks$. So, f is strict and $f\ a = [\,]$ and so we take $b = [\,]$. Now we need a function h that $f\ (g\ x\ y) = h\ x\ (f\ y)$. Since (C.1) is only true for matrices we need to ensure that $|x| = |y|$. Before we prove that transpose is self-inverse we need a property involving f and g.

Property 12. With the definitions of f and g above: $f\ (x : xs) = g\ x\ (f\ xs)$.

Proof.

$$
\begin{aligned}
& f\ (x : xs) \\
= & \quad \{f = \mathsf{transpose}\} \\
& \mathsf{transpose}\ (x : xs) \\
= & \quad \{\mathsf{foldr}\ \text{definition of}\ \mathsf{transpose}\} \\
& \mathsf{foldr}\ (\mathsf{zipWith}\ (:))\ blanks\ (x : xs) \\
= & \quad \{\text{definition of}\ \mathsf{foldr}\} \\
& (\mathsf{zipWith}\ (:))\ x\ (\mathsf{foldr}\ (\mathsf{zipWith}\ (:))\ blanks\ xs) \\
= & \quad \{\text{definitions of } f \text{ and } g\} \\
& g\ x\ (f\ xs)
\end{aligned}
$$

□

For the fold fusion, we propose that $h = (:)$ (note that $\mathsf{foldr}\ (:)\ [\,] = id$) and we prove this by induction, *i.e.* that if $|xs| = |yss|$ then

$$f\ (g\ xs\ yss) = xs : (f\ yss) \tag{C.2}$$

Proof. We prove Equation (C.2) by induction on $|xs|$.

Base Case Suppose that $xs = [x]$ and let $yss = [ys]$ so that $|xs| = 1 = |yss|$

$$
\begin{aligned}
& f\ (g\ xs\ yss) \\
= & \quad \{\text{definitions of } xs,\ yss \text{ and } g\} \\
& f\ [x : ys] \\
= & \quad \{\mathsf{foldr1} \text{ definition of } f\} \\
& \mathsf{foldr1}\ (\mathsf{zipWith}\ (+\!\!+))\ (\mathsf{map}\ (\mathsf{map}\ \mathsf{wrap})\ [x : ys]) \\
= & \quad \{\text{definitions of } \mathsf{map} \text{ and } \mathsf{wrap}\} \\
& \mathsf{foldr1}\ (\mathsf{zipWith}\ (+\!\!+))\ [[x] : (\mathsf{map}\ \mathsf{wrap}\ ys)] \\
= & \quad \{\text{definition of } \mathsf{foldr1}\} \\
& [x] : (\mathsf{map}\ \mathsf{wrap}\ ys) \\
= & \quad \{[\text{definition of } \mathsf{foldr1}\} \\
& [x] : (\mathsf{foldr1}\ (\mathsf{zipWith}\ (+\!\!+))\ [\mathsf{map}\ \mathsf{wrap}\ ys]) \\
= & \quad \{\text{definition of } \mathsf{map}\} \\
& [x] : (\mathsf{foldr1}\ (\mathsf{zipWith}\ (+\!\!+))\ (\mathsf{map}\ (\mathsf{map}\ \mathsf{wrap})\ [ys])) \\
= & \quad \{\mathsf{foldr1} \text{ definition of } f\} \\
& [x] : (f\ [ys]) \\
= & \quad \{\text{definitions}\} \\
& xs : (f\ yss)
\end{aligned}
$$

Step Case Suppose that $xs = v : vs$ and $yss = ws : wss$ where $|vs| = |wss|$. For the induction hypothesis, we suppose that vs and wss satisfy Equation (C.2).

$$
\begin{aligned}
& f\ (g\ xs\ yss) \\
= & \quad \{\text{definitions}\} \\
& f\ (g\ (v : vs)\ (ws : wss)) \\
= & \quad \{\text{definition of } g\ (g \text{ is an instance of } \mathsf{zipWith})\} \\
& f\ ((v : ws) : (g\ vs\ wss)) \\
= & \quad \{\text{Property } 12\} \\
& g\ (v : ws)\ (f\ (g\ vs\ wss)) \\
= & \quad \{\text{induction hypothesis}\} \\
& g\ (v : ws)\ (vs : (f\ wss)) \\
= & \quad \{g \text{ is an instance of } \mathsf{zipWith}\} \\
& (v : vs) : (g\ ws\ (f\ wss))) \\
= & \quad \{\text{Property } 12\} \\
& (v : vs) : (f\ (ws : wss)) \\
= & \quad \{\text{definitions}\} \\
& xs : (f\ yss)
\end{aligned}
$$

□

Appendix D

Proving a tree assertion

The operation mktree takes a list and produces a binary tree and so we can obfuscate the input (a list) and the output (a tree). Suppose that mktree^O is an obfuscation of mktree with $\mathsf{unsplit}$ the abstraction function which converts split lists back to lists and $\mathsf{to2}$ the abstraction function for the conversion from ternary to binary trees. Then mktree^O will satisfy Equation (3.7), *i.e.*

$$\mathsf{mktree} \cdot \mathsf{unsplit} = \mathsf{to2} \cdot \mathsf{mktree}^O$$

This equation allows us to prove the correctness of mktree^O.

We will consider the original operation mktree and three obfuscations of this operation. We will prove that each of the operations satisfies the following assertion:

$$\mathsf{member}\ p\ (\mathsf{mktree}\ xs) = \mathsf{elem}\ p\ xs \tag{D.1}$$

and we will discuss how obfuscated each operation is.

D.1 Producing a binary tree from a list

First, we consider the unobfuscated operation and so our operation for making trees has the following type

$$\mathsf{mktree} :: [\alpha] \to \mathit{Tree}\ \alpha$$

and is defined to be:

$$\begin{array}{l} \mathsf{mktree}\ [\,] = \mathit{Null} \\ \mathsf{mktree}\ xs = \mathit{Fork}\ (\mathsf{mktree}\ ys)\ z\ (\mathsf{mktree}\ zs) \\ \qquad \text{where } (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ |xs|\ 2)\ xs \end{array}$$

D.1.1 Operations

We now state some operations that we will need. First, we define a membership operation for trees as follows:

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

and a membership operation for lists:

$$\begin{aligned}\mathsf{elem}\ p\ [\,] &= \mathit{False}\\ \mathsf{elem}\ p\ (x : xs) &= p == x \lor \mathsf{elem}\ p\ xs\end{aligned}$$

Some list functions that we will need:

$$\begin{aligned}\mathsf{splitAt}\ 0\ xs &= ([\,], xs)\\ \mathsf{splitAt}\ _\ [\,] &= ([\,],[\,])\\ \mathsf{splitAt}\ n\ (x : xs) &= (x : ys, zs)\\ &\quad \mathsf{where}\ (ys, zs) = \mathsf{splitAt}\ (n-1)\ xs\end{aligned}$$

$$\begin{aligned}[\,] +\!\!+\, ys &= ys\\ (x : xs) +\!\!+\, ys &= x : (xs +\!\!+\, ys)\end{aligned}$$

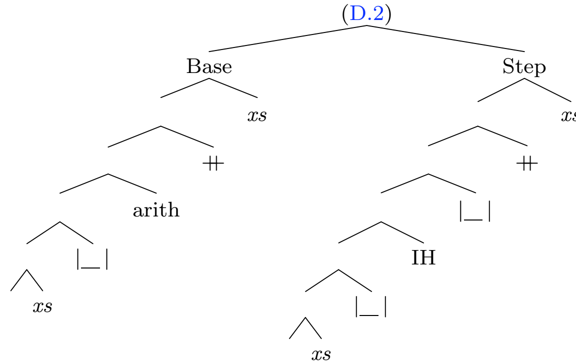
$$\begin{aligned}|[\,]| &= 0\\ |x : xs| &= 1 + |xs|\end{aligned}$$

D.1.2 Proving Properties

First, we prove some properties of the list operations.

- $$|ts +\!\!+\, ys| = |ts| + |ys| \tag{D.2}$$

  The proof for this is in Section A.1 which produces the following tree:



  For this tree, $\mathcal{C}\,(\text{D.2}) = 11$ and $\mathcal{H}\,(\text{D.2}) = 7$.

- $$ts = ys +\!\!+\, zs \text{ where } (ys, zs) = \mathsf{splitAt}\ n\ ts \tag{D.3}$$

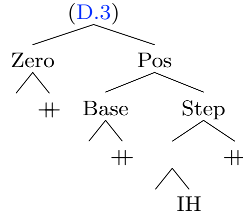
Case for zero If $n = 0$ then $\mathsf{splitAt}\ 0\ ts = ([\,], ts)$. By the definition of $+\!\!+$ then $ts = [\,] +\!\!+\, ts$.

Positive case Let $n > 0$ and prove by structural induction on ts

Base Case Suppose that $ts = [\,]$, then $\mathsf{splitAt}\ n\ [\,] = ([\,],[\,])$ and so $[\,] = [\,] +\!\!+\, [\,]$

Step Case Suppose that $ts = x : xs$ and, for the induction hypothesis, that xs satisfies (D.3). Let $(x : ys, zs) = \mathsf{splitAt}\ n\ (x : xs)$ where $(ys, zs) = \mathsf{splitAt}\ (n-1)\ xs$.

$$\begin{array}{ll} & (x : ys) +\!\!+\ zs \\ = & \quad \{\text{definition of } +\!\!+\} \\ & x : (ys +\!\!+\ zs) \\ = & \quad \{\text{inductive hypothesis}\} \\ & x : xs \end{array}$$



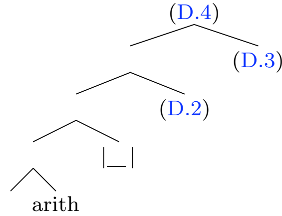
For this tree, $\mathcal{C}\,(\text{D.3}) = 4$ and $\mathcal{H}\,(\text{D.3}) = 4$.

- Suppose that $(ys, z : zs) = \mathsf{splitAt}\ n\ xs$ where $0 \leq n < |xs|$. Then

$$\begin{array}{ll} & |xs| \\ = & \quad \{\text{Equation (D.3)}\} \\ & |ys +\!\!+\ (z : zs)| \\ = & \quad \{\text{Equation (D.2)}\} \\ & |ys| + |z : zs| \\ = & \quad \{\text{definition of } |_|\} \\ & |ys| + 1 + |zs| \end{array}$$

and so (by arithmetic)

$$(ys, z : zs) = \mathsf{splitAt}\ n\ xs \Rightarrow |ys| < |xs| \land |zs| < |xs| \tag{D.4}$$



For this tree,

$$\begin{array}{lcl} \mathcal{C}\,(\text{D.4}) & = & 2 + \mathcal{C}\,(\text{D.3}) + \mathcal{C}\,(\text{D.2}) = 17 \\ \mathcal{H}\,(\text{D.4}) & = & 1 + \max(\mathcal{H}\,(\text{D.3}), 1 + \mathcal{H}\,(\text{D.2}), 3) = 9 \end{array}$$

- $$\mathsf{elem}\ p\ (ts +\!\!+\ ys) = \mathsf{elem}\ p\ ts \lor \mathsf{elem}\ p\ ys \tag{D.5}$$

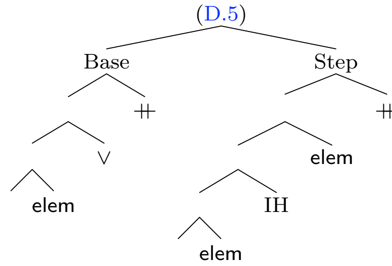
We prove this by structural induction on ts.

Base Case Suppose that $ts = [\,]$.

$$
\begin{aligned}
& \mathsf{elem}\ p\ ([\,] \mathbin{+\!\!+} ys) \\
= & \quad \{\text{definition of } \mathbin{+\!\!+}\} \\
& \mathsf{elem}\ p\ ys \\
= & \quad \{\text{property of } \vee\} \\
& \mathit{False} \vee \mathsf{elem}\ p\ ys \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& \mathsf{elem}\ p\ [\,] \vee \mathsf{elem}\ p\ ys
\end{aligned}
$$

Step Case Suppose that $ts = x : xs$ and, for the induction hypothesis, that xs satisfies (D.5).

$$
\begin{aligned}
& \mathsf{elem}\ p\ ((x : xs) \mathbin{+\!\!+} ys) \\
= & \quad \{\text{definition of } \mathbin{+\!\!+}\} \\
& \mathsf{elem}\ p\ (x : (xs \mathbin{+\!\!+} ys)) \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& p == x \vee \mathsf{elem}\ p\ (xs \mathbin{+\!\!+} ys) \\
= & \quad \{\text{induction hypothesis}\} \\
& p == x \vee \mathsf{elem}\ p\ xs \vee \mathsf{elem}\ p\ ys \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& \mathsf{elem}\ p\ (x : xs) \vee \mathsf{elem}\ p\ ys
\end{aligned}
$$



For this tree, $\mathcal{C}$ (D.5) $= 7$ and $\mathcal{H}$ (D.5) $= 5$.

D.1.3 Proving the assertion

Now, we are able to prove Equation (D.1) by induction on $|xs|$.

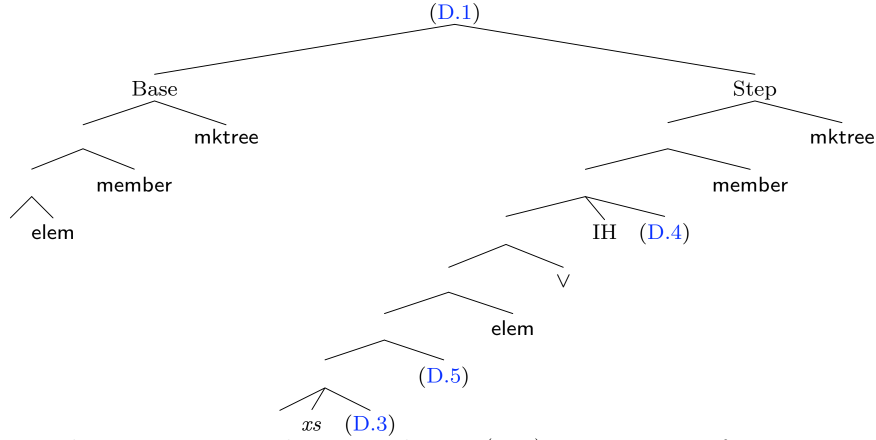
Base Case Suppose that $|xs| = 0$, by the definition of $|_|$ this means that $xs = [\,]$

$$
\begin{aligned}
& \mathsf{member}\ p\ (\mathsf{mktree}\ [\,]) \\
= & \quad \{\text{definition of } \mathsf{mktree}\} \\
& \mathsf{member}\ p\ \mathit{Null} \\
= & \quad \{\text{definition of } \mathsf{member}\}
\end{aligned}
$$

$$
\begin{array}{ll}
& \mathit{False} \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& \mathsf{elem}\ p\ [\,]
\end{array}
$$

Step Case Now, consider a list xs. For the induction hypothesis, we suppose that (D.1) is true $(\forall ts) \bullet |ts| < |xs|$. Let $(ys, (z : zs)) = \mathsf{splitAt}\ n\ xs$

$$
\begin{array}{ll}
& \mathsf{member}\ p\ (\mathsf{mktree}\ xs) \\
= & \quad \{\text{definition of } \mathsf{mktree}\} \\
& \mathsf{member}\ p\ (\mathit{Fork}\ (\mathsf{mktree}\ ys)\ z\ (\mathsf{mktree}\ zs)) \\
= & \quad \{\text{definition of } \mathsf{member}\} \\
& z == p \vee \mathsf{member}\ p\ (\mathsf{mktree}\ ys) \vee \mathsf{member}\ p\ (\mathsf{mktree}\ zs) \\
= & \quad \{\text{induction hypothesis, using (D.4)}\} \\
& z == p \vee \mathsf{elem}\ p\ ys \vee \mathsf{elem}\ p\ zs \\
= & \quad \{\text{commutativity and associativity of } \vee\} \\
& \mathsf{elem}\ p\ ys \vee (z == p \vee \mathsf{elem}\ p\ zs) \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& \mathsf{elem}\ p\ ys \vee \mathsf{elem}\ p\ (z : zs) \\
= & \quad \{\text{Equation (D.5)}\} \\
& \mathsf{elem}\ p\ (ys +\!\!+ (z : zs)) \\
= & \quad \{\text{definition of } xs\text{, Equation (D.3)}\} \\
& \mathsf{elem}\ p\ xs
\end{array}
$$



For this tree it seems that we only use (D.3) once. But in fact, we use it twice: once in the tree above and once in the proof of (D.4). This means that we must declare (D.3) as a lemma and so we obtain

$$
\begin{array}{lcl}
\mathcal{C}\,(\text{D.1}) & = & 10 + \mathcal{C}\,(\text{D.4}) + \mathcal{C}\,(\text{D.5}) = 34 \\
\mathcal{H}\,(\text{D.1}) & = & 4 + \max(\mathcal{H}\,(\text{D.4}), 3 + \mathcal{H}\,(\text{D.5}), 4 + \mathcal{H}\,(\text{D.3})) = 13
\end{array}
$$

D.2 Producing a binary tree from a split list

Now we use the alternating split to obfuscate mktree. Thus our operation for making trees has the following type

$$\mathsf{mktree}_{asp} :: \langle[\alpha],[\alpha]\rangle_{asp} \to \mathit{Tree}\ \alpha$$

and is defined to be

$$\begin{aligned}
&\mathsf{mktree}_{asp}\ \langle[\,], _\rangle_{asp} = \mathit{Null}\\
&\mathsf{mktree}_{asp}\ \langle l, r\rangle_{asp} = \mathit{Fork}\ (\mathsf{mktree}_{asp}\ ysp)\ z\ (\mathsf{mktree}_{asp}\ \langle zr, zl\rangle_{asp})\\
&\quad \text{where } (ysp, \langle z : zl, zr\rangle_{asp}) = \mathsf{breakAt}\ |r|\ \langle l, r\rangle_{asp}
\end{aligned}$$

D.2.1 Operations

We use the tree membership operation from Section D.1. Here is a membership operation for split lists:

$$\begin{aligned}
&\mathsf{elem}_{asp}\ p\ \langle[\,],[\,]\rangle_{asp} &&= \mathit{False}\\
&\mathsf{elem}_{asp}\ p\ \langle x : l, r\rangle_{asp} &&= p == x \vee \mathsf{elem}_{asp}\ p\ \langle r, l\rangle_{asp}
\end{aligned}$$

and a split list operation corresponding to splitAt:

$$\begin{aligned}
&\mathsf{breakAt}\ 0\ xsp &&= (\langle[\,],[\,]\rangle_{asp}, xsp)\\
&\mathsf{breakAt}\ _\ \langle[\,],[\,]\rangle_{asp} &&= (\langle[\,],[\,]\rangle_{asp}, \langle[\,],[\,]\rangle_{asp})\\
&\mathsf{breakAt}\ n\ \langle x : l, r\rangle_{asp} &&= (\mathsf{cons}_{asp}\ x\ ys, zs)\\
& &&\quad \text{where } (ys, zs) = \mathsf{breakAt}\ (n-1)\ \langle r, l\rangle_{asp}
\end{aligned}$$

Other split list operations that we need:

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

$$\langle l_0, r_0\rangle_{asp} +\!\!+_{asp} \langle l_1, r_1\rangle_{asp} = \begin{cases} \langle l_0 +\!\!+ l_1, r_0 +\!\!+ r_1\rangle_{asp} & \text{if } |l_0| = |r_0|\\ \langle l_0 +\!\!+ r_1, r_0 +\!\!+ l_1\rangle_{asp} & \text{otherwise}\end{cases}$$

$$|\langle l, r\rangle_{asp}|_{asp} = |l| + |r|$$

Thus the assertion that we want to prove is:

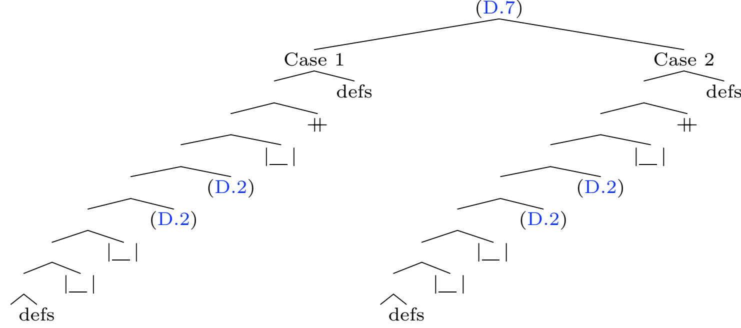
$$\mathsf{member}\ p\ (\mathsf{mktree}_{asp}\ xs) = \mathsf{elem}_{asp}\ p\ xs \tag{D.6}$$

D.2.2 Preliminaries

First, we prove some properties of the split list operations. Note that for structural induction on split lists, we have two cases — $\langle[\,],[\,]\rangle_{asp}$ and lists of the form $\langle x : l, r\rangle_{asp}$.

- $$|xsp +\!\!+_{asp} ysp|_{asp} = |xsp|_{asp} + |ysp|_{asp} \tag{D.7}$$

This is proved in Section A.2.1 and the tree for this proof is:



For this proof, we declare (D.2) as a lemma and so

$$\mathcal{C}\,(\text{D.7}) = 16 + \mathcal{C}\,(\text{D.2}) = 27$$
$$\text{and } \mathcal{H}\,(\text{D.7}) = 6 + \max(3, \mathcal{H}\,(\text{D.2})) = 13$$

- $$(ysp, zsp) = \mathsf{breakAt}\ n\ xsp \Rightarrow xsp = ysp +\!\!+_{asp}\ zsp \tag{D.8}$$

Zero case Let $n = 0$. From the definition,

$$\mathsf{breakAt}\ 0\ xsp = (\langle [\,], [\,] \rangle_{asp}, xsp)$$

and so

$$\begin{aligned} & (\langle [\,], [\,] \rangle_{asp}) +\!\!+_{asp}\ xsp \\ = & \quad \{\text{definition of } +\!\!+_{asp}\} \\ & xsp \end{aligned}$$

Positive Case Suppose that $n > 0$, we prove Equation (D.8) by structural induction on xsp.

Base Case Suppose that $xsp = \langle [\,], [\,] \rangle_{asp}$. From the definition

$$\mathsf{breakAt}\ n\ \langle [\,], [\,] \rangle_{asp} = (\langle [\,], [\,] \rangle_{asp}, \langle [\,], [\,] \rangle_{asp})$$

and so

$$\begin{aligned} & (\langle [\,], [\,] \rangle_{asp}) +\!\!+_{asp}\ (\langle [\,], [\,] \rangle_{asp}) \\ = & \quad \{\text{definition of } +\!\!+_{asp}\} \\ & \langle [\,], [\,] \rangle_{asp} \end{aligned}$$

Step Case Suppose $xsp = \mathsf{cons}_{asp}\ x\ \langle l, r \rangle_{asp} = \langle x : l, r \rangle_{asp}$. Let

$$\begin{aligned} & (\mathsf{cons}_{asp}\ x\ \langle yl, yr \rangle_{asp}, \langle zl, zr \rangle_{asp}) = \mathsf{breakAt}\ n\ \langle x : l, r \rangle_{asp} \\ & \quad \text{where } (\langle yl, yr \rangle_{asp}, \langle zl, zr \rangle_{asp}) = \mathsf{breakAt}\ (n-1)\ \langle r, l \rangle_{asp} \end{aligned}$$

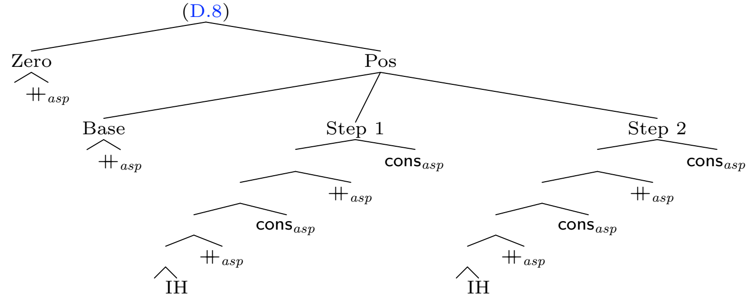
For the induction hypothesis, we suppose that $\langle l, r \rangle_{asp}$ satisfies Equation (D.8). We have two cases depending on the lengths of yl and yr.

Subcase 1 Suppose that $|yl| = |yr|$. Then

$$
\begin{aligned}
& (\mathsf{cons}_{asp}\ x\ \langle yl, yr\rangle_{asp}) +\!\!+_{asp} \langle zl, zr\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\
& \langle x : yr, yl\rangle_{asp} +\!\!+_{asp} \langle zl, zr\rangle_{asp} \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |x : yr| \neq |yl|\} \\
& \langle x : yr +\!\!+ zr, yl +\!\!+ zl\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\
& \mathsf{cons}_{asp}\ x\ \langle yl +\!\!+ zl, yr +\!\!+ zr\rangle_{asp} \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |yl| = |yr|\} \\
& \mathsf{cons}_{asp}\ x\ (\langle yl, yr\rangle_{asp} +\!\!+_{asp} \langle zl, zr\rangle_{asp}) \\
= & \quad \{\text{induction hypothesis}\} \\
& \mathsf{cons}_{asp}\ x\ \langle l, r\rangle_{asp}
\end{aligned}
$$

Subcase 2 Suppose that $|yl| \neq |yr|$. Then by invariant (4.11), $|yl| = |yr| + 1$ and so

$$
\begin{aligned}
& (\mathsf{cons}_{asp}\ x\ \langle yl, yr\rangle_{asp}) +\!\!+_{asp} \langle zl, zr\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\
& \langle x : yr, yl\rangle_{asp} +\!\!+_{asp} \langle zl, zr\rangle_{asp} \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |x : yr| = |yl|\} \\
& \langle x : yr +\!\!+ zl, yl +\!\!+ zr\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\
& \mathsf{cons}_{asp}\ x\ \langle yl +\!\!+ zr, yr +\!\!+ zl\rangle_{asp} \\
= & \quad \{\text{definition of } +\!\!+_{asp} \text{ with } |yl| \neq |yr|\} \\
& \mathsf{cons}_{asp}\ x\ (\langle yl, yr\rangle_{asp} +\!\!+_{asp} \langle zl, zr\rangle_{asp}) \\
= & \quad \{\text{induction hypothesis}\} \\
& \mathsf{cons}_{asp}\ x\ \langle l, r\rangle_{asp}
\end{aligned}
$$



For this tree, $\mathcal{C}$ (D.8) = 12 and $\mathcal{H}$ (D.8) = 7.

- $$|ysp|_{asp} + |zsp|_{asp} = |xsp|_{asp} \quad \text{where } (ysp, zsp) = \mathsf{breakAt}\ n\ xsp \tag{D.9}$$

Then

$$
\begin{aligned}
& |xsp|_{asp} \\
= & \quad \{\text{Equation (D.8)}\} \\
& |ysp \mathbin{+\!\!+}_{asp} zsp|_{asp} \\
= & \quad \{\text{Equation (D.7)}\} \\
& |ysp|_{asp} + |zsp|_{asp}
\end{aligned}
$$

(D.9)
— (D.7)
— (D.8)

For this tree, (we do not declare the two results in the tree as lemmas):

$$
\begin{aligned}
\mathcal{C}\,(\text{D.9}) &= \mathcal{C}\,(\text{D.7}) + \mathcal{C}\,(\text{D.8}) = 39 \\
\mathcal{H}\,(\text{D.9}) &= 1 + \max(1 + \mathcal{H}\,(\text{D.7}), \mathcal{H}\,(\text{D.8})) = 15
\end{aligned}
$$

- $$
\begin{aligned}
& \mathsf{elem}_{asp}\ p\ ysp \lor \mathsf{elem}_{asp}\ p\ zsp = \mathsf{elem}_{asp}\ p\ xsp \\
& \quad \text{where}\ \ (ysp, zsp) = \mathsf{breakAt}\ n\ xsp
\end{aligned} \tag{D.10}
$$

We prove this by structural induction on xsp.

Base Case Suppose $xsp = \langle [\,], [\,] \rangle_{asp}$. From the definition

$$\mathsf{breakAt}\ n\ \langle [\,], [\,] \rangle_{asp} = (\langle [\,], [\,] \rangle_{asp}, \langle [\,], [\,] \rangle_{asp})$$

and then

$$
\begin{aligned}
& \mathsf{elem}_{asp}\ p\ \langle [\,], [\,] \rangle_{asp} \lor \mathsf{elem}_{asp}\ p\ \langle [\,], [\,] \rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
& \mathit{False} \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
& \mathsf{elem}_{asp}\ p\ \langle [\,], [\,] \rangle_{asp}
\end{aligned}
$$

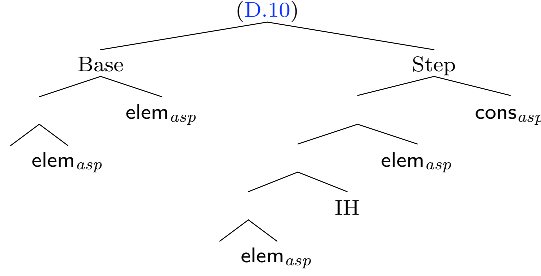
Step Case Suppose $xsp = \langle x : l, r \rangle_{asp}$ and

$$
\begin{aligned}
& (\mathsf{cons}_{asp}\ x\ \langle yl, yr \rangle_{asp}, zsp) = \mathsf{breakAt}\ n\ \langle x : l, r \rangle_{asp} \\
& \quad \text{where}\ (\langle yl, yr \rangle_{asp}, zsp) = \mathsf{breakAt}\ (n-1)\ \langle r, l \rangle_{asp}
\end{aligned}
$$

For the induction hypothesis, we suppose that $\langle r, l \rangle_{asp}$ satisfies (D.10) and so

$$
\begin{aligned}
& \mathsf{elem}_{asp}\ p\ (\mathsf{cons}_{asp}\ x\ \langle yl, yr \rangle_{asp}) \lor \mathsf{elem}_{asp}\ p\ zsp \\
= & \quad \{\text{definition of } \mathsf{cons}_{asp}\} \\
& \mathsf{elem}_{asp}\ p\ \langle x : yr, yl \rangle_{asp} \lor \mathsf{elem}_{asp}\ p\ zsp \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\}
\end{aligned}
$$

$$
\begin{array}{ll}
 & p == x \vee \mathsf{elem}_{asp}\ p\ \langle yl, yr\rangle_{asp} \vee \mathsf{elem}_{asp}\ p\ zsp \\
= & \quad \{\text{induction hypothesis}\} \\
 & p == x \vee \mathsf{elem}_{asp}\ p\ \langle r, l\rangle_{asp} \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
 & \mathsf{elem}_{asp}\ p\ \langle x : l, r\rangle_{asp}
\end{array}
$$



For this tree, $\mathcal{C}\,(\text{D.10}) = 6$ and $\mathcal{H}\,(\text{D.10}) = 5$.

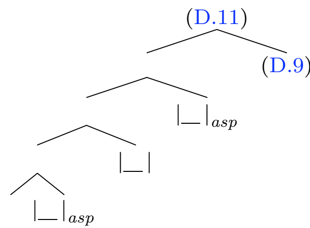
- For a non-empty split list xsp and $0 \leq n < |xsp|_{asp}$

$$
\begin{array}{l}
\text{If } (ysp, \langle z : zl, zr\rangle_{asp}) = \mathsf{breakAt}\ n\ xsp \\
\text{then } |ysp|_{asp} < |xsp|_{asp}\ \wedge\ |\langle zr, zl\rangle_{asp}|_{asp} < |xsp|_{asp}
\end{array} \tag{D.11}
$$

The proof is as follows:

$$
\begin{array}{ll}
 & |xsp|_{asp} \\
= & \quad \{\text{Equation (D.9)}\} \\
 & |ysp|_{asp} + |\langle z : zl, zr\rangle_{asp}|_{asp} \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
 & |ysp|_{asp} + |z : zl| + |zr| \\
= & \quad \{\text{definition of } |_|\} \\
 & 1 + |ysp|_{asp} + |zl| + |zr| \\
= & \quad \{\text{definition of } |_|_{asp}\} \\
 & 1 + |ysp|_{asp} + |\langle zr, zl\rangle_{asp}|_{asp}
\end{array}
$$

and thus we can conclude $|ysp|_{asp} < |xsp|_{asp}$ and $|\langle zr, zl\rangle_{asp}|_{asp} < |xsp|_{asp}$.



For this tree,

$$
\begin{array}{rcl}
\mathcal{C}\,(\text{D.11}) & = & 3 + \mathcal{C}\,(\text{D.9}) = 42 \\
\mathcal{H}\,(\text{D.11}) & = & 1 + \max(\mathcal{H}\,(\text{D.9}), 3) = 16
\end{array}
$$

D.2.3 Proof of the assertion

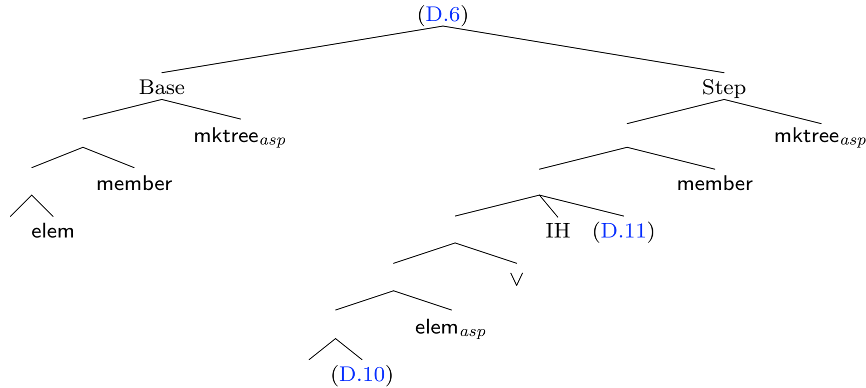
Now, we prove Equation (D.6) by induction on $|xsp|_{asp}$.

Base Case Suppose that $|xsp|_{asp} = 0$. So $xsp = \langle [\,], [\,] \rangle_{asp}$.

$$
\begin{array}{ll}
 & \mathsf{member}\ p\ (\mathsf{mktree}_{asp}\ \langle [\,], [\,] \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{mktree}_{asp}\} \\
 & \mathsf{member}\ p\ \mathit{Null} \\
= & \quad \{\text{definition of } \mathsf{member}\} \\
 & \mathit{False} \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
 & \mathsf{elem}_{asp}\ p\ \langle [\,], [\,] \rangle_{asp}
\end{array}
$$

Step Case Let $xsp = \langle l, r \rangle_{asp}$ and we suppose that Equation (D.6) is true for all split lists which have length less than $|xsp|_{asp}$. Let $(ysp, \langle z : zl, zr \rangle_{asp}) = \mathsf{breakAt}\ |r|\ \langle l, r \rangle_{asp}$.

$$
\begin{array}{ll}
 & \mathsf{member}\ p\ (\mathsf{mktree}_{asp}\ \langle l, r \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{mktree}_{asp}\} \\
 & \mathsf{member}\ p\ (\mathit{Fork}\ (\mathsf{mktree}_{asp}\ ysp)\ z\ (\mathsf{mktree}_{asp}\ \langle zr, zl \rangle_{asp})) \\
= & \quad \{\text{definition of } \mathsf{member}\} \\
 & z == p \vee \mathsf{member}\ p\ (\mathsf{mktree}_{asp}\ ysp) \\
 & \qquad\quad \vee\ \mathsf{member}\ p\ (\mathsf{mktree}_{asp}\ \langle zr, zl \rangle_{asp}) \\
= & \quad \{\text{induction hypothesis using Equation (D.11)}\} \\
 & z == p \vee \mathsf{elem}_{asp}\ p\ ysp \vee \mathsf{elem}_{asp}\ p\ \langle zr, zl \rangle_{asp} \\
= & \quad \{\text{associativity and commutativity of } \vee\} \\
 & \mathsf{elem}_{asp}\ p\ ysp \vee (z == p \vee \mathsf{elem}_{asp}\ p\ \langle zr, zl \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
 & \mathsf{elem}_{asp}\ p\ ysp \vee \mathsf{elem}_{asp}\ p\ \langle x : zl, zr \rangle_{asp} \\
= & \quad \{\text{Equation (D.10)}\} \\
 & \mathsf{elem}_{asp}\ p\ \langle l, r \rangle_{asp}
\end{array}
$$



For this tree,

$$\begin{aligned}\mathcal{C}\,(\text{D.6}) &= 8 + \mathcal{C}\,(\text{D.10}) + \mathcal{C}\,(\text{D.11}) = 56\\ \mathcal{H}\,(\text{D.6}) &= 4 + \max(\mathcal{H}\,(\text{D.11}), 3 + \mathcal{H}\,(\text{D.10})) = 20\end{aligned}$$

D.3 Producing a ternary tree from a list

In this section, our operation for making trees has the following type:

$$\mathsf{mktree} :: [\alpha] \rightarrow \mathit{Tree}_3\ \alpha$$

and is defined to be:

$$\begin{array}{l}\mathsf{mktree3}\ [\,] = \mathit{Null}_3\\ \mathsf{mktree3}\ xs\\ \quad \mid \mathsf{even}\ z \quad\ = \mathit{Fork}_3\ z\ (\mathsf{mktree3}\ ys)\ (\mathsf{mktree3}\ zs)\ jt\\ \quad \mid \text{otherwise} = \mathit{Fork}_3\ z\ kt\ (\mathsf{mktree3}\ zs)\ (\mathsf{mktree3}\ ys)\\ \quad \text{where}\ (ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ |xs|\ 2)\ xs\end{array}$$

D.3.1 Operations

Membership operation for ternary trees is defined as follows:

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

and we use the list operations from Section D.1. Thus the assertion that we want to prove is

$$\mathsf{member}_3\ p\ (\mathsf{mktree3}\ xs) = \mathsf{elem}\ p\ xs \tag{D.12}$$

D.3.2 Proof

We prove the assertion by induction on length of xs.

Base Case Suppose that $|xs| = 0$ ($i.e.$ $xs = [\,]$).

$$\begin{array}{ll} & \mathsf{member}_3\ p\ (\mathsf{mktree3}\ [\,])\\ = & \quad \{\text{definition of}\ \mathsf{mktree3}\}\\ & \mathsf{member}_3\ p\ \mathit{Null}_3\\ = & \quad \{\text{definition of}\ \mathsf{member}_3\}\\ & \mathit{False}\\ = & \quad \{\text{definition of}\ \mathsf{elem}\}\\ & \mathsf{elem}\ p\ [\,]\end{array}$$

Step Case Consider a list xs and for the induction hypothesis, we suppose that Equation (D.12) is true $(\forall ts) \bullet |ts| < |xs|$. Let $(ys, (z : zs)) = \mathsf{splitAt}\ (\mathsf{div}\ |xs|\ 2)\ xs$.

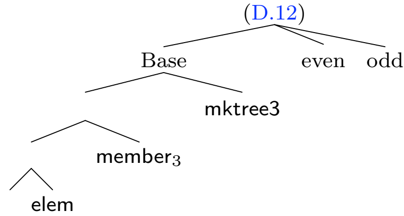
Subcase 1 Suppose that z is even:

$$
\begin{aligned}
& \mathsf{member}_3\ p\ (\mathsf{mktree3}\ (x : xs)) \\
= & \quad \{\text{definition of } \mathsf{mktree3}\} \\
& \mathsf{member}_3\ p\ (Fork_3\ z\ (\mathsf{mktree3}\ ys)\ (\mathsf{mktree3}\ zs)\ jt) \\
= & \quad \{\text{definition of } \mathsf{member}_3\text{, with } z \text{ even}\} \\
& z == p \lor \mathsf{member}_3\ p\ (\mathsf{mktree3}\ zs) \lor \mathsf{member}_3\ p\ (\mathsf{mktree3}\ ys) \\
= & \quad \{\text{induction hypothesis, using (D.4)}\} \\
& z == p \lor \mathsf{elem}\ p\ ys \lor \mathsf{elem}\ p\ zs \\
= & \quad \{\text{commutativity and associativity of } \lor\} \\
& \mathsf{elem}\ p\ ys \lor (z == p \lor \mathsf{elem}\ p\ zs) \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& \mathsf{elem}\ p\ ys \lor \mathsf{elem}\ p\ (z : zs) \\
= & \quad \{\text{Equation (D.5)}\} \\
& \mathsf{elem}\ p\ (ys +\!\!+ (z : zs)) \\
= & \quad \{\text{definition of } xs\text{, Equation (D.3)}\} \\
& \mathsf{elem}\ p\ xs
\end{aligned}
$$

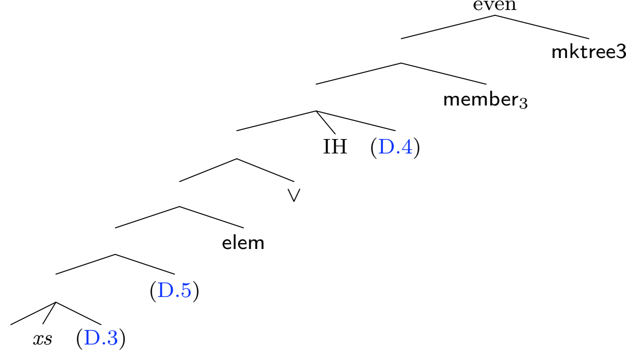
Subcase 2 Suppose that z is odd:

$$
\begin{aligned}
& \mathsf{member}_3\ p\ (\mathsf{mktree3}\ (x : xs)) \\
= & \quad \{\text{definition of } \mathsf{mktree3}\} \\
& \mathsf{member}_3\ p\ (Fork_3\ z\ kt\ (\mathsf{mktree3}\ zs)\ (\mathsf{mktree3}\ ys)) \\
= & \quad \{\text{definition of } \mathsf{member}_3\text{, with } z \text{ odd}\} \\
& z == p \lor \mathsf{member}_3\ p\ (\mathsf{mktree3}\ zs) \lor \mathsf{member}_3\ p\ (\mathsf{mktree3}\ ys) \\
= & \quad \{\text{induction hypothesis, using (D.4)}\} \\
& z == p \lor \mathsf{elem}\ p\ ys \lor \mathsf{elem}\ p\ zs \\
= & \quad \{\text{commutativity and associativity of } \lor\} \\
& \mathsf{elem}\ p\ ys \lor (z == p \lor \mathsf{elem}\ p\ zs) \\
= & \quad \{\text{definition of } \mathsf{elem}\} \\
& \mathsf{elem}\ p\ ys \lor \mathsf{elem}\ p\ (z : zs) \\
= & \quad \{\text{Equation (D.5)}\} \\
& \mathsf{elem}\ p\ (ys +\!\!+ (z : zs)) \\
= & \quad \{\text{definition of } xs\text{, Equation (D.3)}\} \\
& \mathsf{elem}\ p\ xs
\end{aligned}
$$

The basic structure of the proof tree is:



The tree for the case when z even is:



The case for z odd produces an identical proof tree. We declare (D.3), (D.4) and (D.5) as lemmas. So (remembering that (D.4) uses (D.3)):

$$
\begin{aligned}
\mathcal{C}\,(\text{D.12}) &= 3 + (2 \times 9) + \mathcal{C}\,(\text{D.4}) + \mathcal{C}\,(\text{D.5}) = 45 \\
\mathcal{H}\,(\text{D.12}) &= 4 + \max(\mathcal{H}\,(\text{D.4}), 3 + \mathcal{H}\,(\text{D.5}), 4 + \mathcal{H}\,(\text{D.3})) = 13
\end{aligned}
$$

D.4 Producing a ternary tree from a split list

In this section, our operation for making trees has the following type:

$$\mathsf{mktree} :: \langle [\alpha], [\alpha] \rangle_{asp} \rightarrow \mathit{Tree}_3\ \alpha$$

and is defined to be:

$$
\begin{aligned}
&\mathsf{mktree3}_{asp}\ \langle [\,], _\ \rangle_{asp} = \mathit{Null}_3 \\
&\mathsf{mktree3}\ _{asp}\ \langle l, r \rangle_{asp} \\
&\quad \mid \mathsf{even}\ z \quad = \mathit{Fork}_3\ z\ (\mathsf{mktree3}_{asp}\ \mathit{ysp}) \\
&\qquad\qquad\qquad\qquad (\mathsf{mktree3}_{asp}\ \langle zr, zl \rangle_{asp})\ \mathit{jt} \\
&\quad \mid \text{otherwise} = \mathit{Fork}_3\ z\ \mathit{kt} \\
&\qquad\qquad\qquad\qquad (\mathsf{mktree3}\ \langle zr, zl \rangle_{asp})\ (\mathsf{mktree3}\ \mathit{ysp}) \\
&\quad\ \text{where}\ (\mathit{ysp}, \langle z : zl, zr \rangle_{asp}) \ = \ \mathsf{breakAt}\ |r|\ \langle l, r \rangle_{asp}
\end{aligned}
$$

We use the definition of member_3 from Section D.3 and the split list operations from Section D.2. Thus the assertion that we want to prove is

$$\mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ \mathit{xsp}) = \mathsf{elem}_{asp}\ p\ \mathit{xsp} \tag{D.13}$$

D.4.1 Proof

We prove the assertion by induction on $|\mathit{xsp}|_{asp}$.

Base case Suppose that $|\mathit{xsp}|_{asp} = 0$ (and so $\mathit{xsp} = \langle [\,], [\,] \rangle_{asp}$).

$$
\begin{aligned}
&\quad \mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ \langle [\,], [\,] \rangle_{asp}) \\
=&\quad \{\text{definition of } \mathsf{mktree}_{asp}\} \\
&\quad \mathsf{member}_3\ p\ \mathit{Null}_3 \\
=&\quad \{\text{definition of } \mathsf{member}_3\}
\end{aligned}
$$

$$
\begin{aligned}
& \quad \mathit{False} \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
& \quad \mathsf{elem}_{asp}\ p\ \langle [\,], [\,] \rangle_{asp}
\end{aligned}
$$

Step Case Let $xsp = \langle l, r \rangle_{asp}$ and we suppose that Equation (D.13) is true for all split lists which have length less than $|xsp|_{asp}$. Let $(ysp, \langle z : zl, zr \rangle_{asp}) = \mathsf{breakAt}\ |r|\ \langle l, r \rangle_{asp}$.

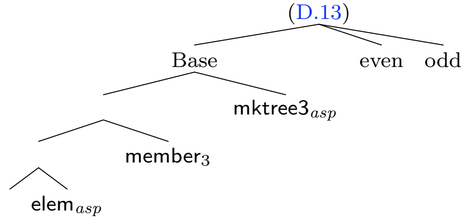
Subcase 1 Suppose that z is even:

$$
\begin{aligned}
& \quad \mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ \langle l, r \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{mktree3}_{asp}\} \\
& \quad \mathsf{member}_3\ p\ (\mathit{Fork}_3\ z\ \ (\mathsf{mktree3}_{asp}\ ysp) \\
& \qquad\qquad\qquad\qquad (\mathsf{mktree3}_{asp}\ \langle zr, zl \rangle_{asp})\ jt) \\
= & \quad \{\text{definition of } \mathsf{member}_3 \text{ with } z \text{ even}\} \\
& \quad z == p \lor (\mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ ysp)) \\
& \qquad\quad \lor\ (\mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ \langle zr, zl \rangle_{asp})) \\
= & \quad \{\text{induction hypothesis using (D.11)}\} \\
& \quad z == p \lor \mathsf{elem}_{asp}\ p\ ysp \lor \mathsf{elem}_{asp}\ p\ \langle zr, zl \rangle_{asp} \\
= & \quad \{\text{associativity and commutativity of } \lor\} \\
& \quad \mathsf{elem}_{asp}\ p\ ysp \lor (z == p \lor \mathsf{elem}_{asp}\ p\ \langle zr, zl \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
& \quad \mathsf{elem}_{asp}\ p\ ysp \lor \mathsf{elem}_{asp}\ p\ \langle x : zl, zr \rangle_{asp} \\
= & \quad \{\text{Equation (D.10)}\} \\
& \quad \mathsf{elem}_{asp}\ p\ \langle l, r \rangle_{asp}
\end{aligned}
$$

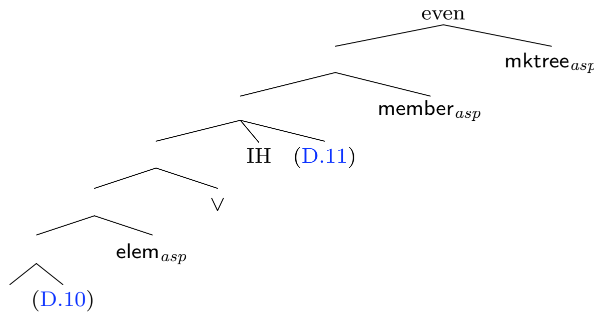
Subcase 2 Suppose that z is odd:

$$
\begin{aligned}
& \quad \mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ \langle l, r \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{mktree3}_{asp}\} \\
& \quad \mathsf{member}_3\ p\ (\mathit{Fork}_3\ z\ kt\ (\mathsf{mktree3}_{asp}\ \langle zr, zl \rangle_{asp})) \\
& \qquad\qquad\qquad\qquad\quad (\mathsf{mktree3}_{asp}\ ysp) \\
= & \quad \{\text{definition of } \mathsf{member}_3 \text{ with } z \text{ odd}\} \\
& \quad z == p \lor (\mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ ysp)) \\
& \qquad\quad \lor (\mathsf{member}_3\ p\ (\mathsf{mktree3}_{asp}\ \langle zr, zl \rangle_{asp})) \\
= & \quad \{\text{induction hypothesis using (D.11)}\} \\
& \quad z == p \lor \mathsf{elem}_{asp}\ p\ ysp \lor \mathsf{elem}_{asp}\ p\ \langle zr, zl \rangle_{asp} \\
= & \quad \{\text{associativity and commutativity of } \lor\} \\
& \quad \mathsf{elem}_{asp}\ p\ ysp \lor (z == p \lor \mathsf{elem}_{asp}\ p\ \langle zr, zl \rangle_{asp}) \\
= & \quad \{\text{definition of } \mathsf{elem}_{asp}\} \\
& \quad \mathsf{elem}_{asp}\ p\ ysp \lor \mathsf{elem}_{asp}\ p\ \langle x : zl, zr \rangle_{asp} \\
= & \quad \{\text{Equation (D.10)}\} \\
& \quad \mathsf{elem}_{asp}\ p\ \langle l, r \rangle_{asp}
\end{aligned}
$$

The structure of the proof tree is:



The tree for z even is:



and the proof for z odd produces the same tree. We declare (D.10) and (D.11) as lemmas and so

$$\begin{aligned}\mathcal{C}\,(\text{D.13}) &= 3 + (2 \times 7) + \mathcal{C}\,(\text{D.10}) + \mathcal{C}\,(\text{D.11}) = 65 \\ \mathcal{H}\,(\text{D.13}) &= 4 + \max(\mathcal{H}\,(\text{D.11}), 3 + \mathcal{H}\,(\text{D.10})) = 20\end{aligned}$$

D.5 Comparing the proofs

Below is a table which summarises the results for each of the proof trees:

Obfuscation	$\mathcal{C}$	$\mathcal{H}$
None	34	13
Split List	56	20
Ternary Tree	45	13
Split List and Ternary Tree	65	20

From this table we can see that introducing a split list increases both the cost and the height of the proof. Making a ternary tree increases the cost but the height remains the same. As the definition of mktree3 has two cases with each case similar is to the definition of mktree, the cost increases but the height does not.

Appendix E

Obfuscation Example

On Page 8, we gave an example obfuscation and we asked what the method *start* does to the elements of an (integer) array. In fact, *start* sorts the array elements into ascending order by creating a BST and then flattening the tree. The original program is shown in Figure E.1.

The method *start* takes an integer valued array and then places all elements of this array into a tree by using the method *insert*. The tree that is constructed by *insert* is a BST (so it satisfies Equation (7.1)). Next, the method *sort* is used to put all elements of the tree into an array in such a way that the elements are in ascending order.

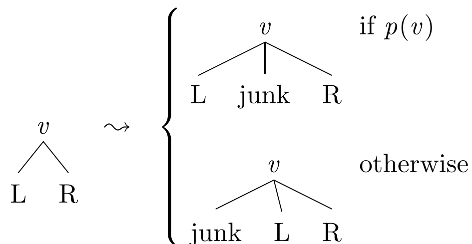
The program was obfuscated by changing binary trees to ternary trees, performing a variable transformation and renaming variables.

For the variable transformation, we take

$$f = \lambda n.3n + 1$$
$$g = \lambda n.(n-1) \text{ div } 3$$

We store $f(v)$ in the tree instead of v and when retrieving the values, we use the function g to recover the correct value. Note that $(\forall v :: \mathbf{Z}) \bullet\ g(f(v)) = v$.

The next step is to perform the tree transformation. Each node in the binary tree is transformed as follows:



We can choose the predicate p at runtime.

For further confusion, we add in the extra predicate h which is used on the junk values in the tree. Also we change the names of the variables and methods but we cannot change the names of b and *start* as they are public methods.

```
public class class1
{   tree t1 = new tree();
    public int [] b;
    int r;

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

    public int [] start(int [] b)
    {   t1 = null;
        r = 0;
        while (r < b.Length) { insert(ref t1, b[r]);
                               r++;
                             }
        r = 0;
        sort(t1);
        return b;
    }

    void sort(tree t2)
    {   if (t2 == null) return;
        sort(t2.left);
        b[r] = t2.val;
        r++;
        sort(t2.right);
    }

    void insert(ref tree t2, int key)
    {   if (t2 == null){ t2 = new tree();
                         t2.val = key;
                       }
        else { if (key < t2.val) { insert(ref t2.left, key); }
               else { insert(ref t2.right, key); }
             }
    }
}
```

Figure E.1: Original Program