A Calculational Approach to Program Inversion

Shin-Cheng Mu
Linacre College

23rd January 2003

Oxford University Computing Laboratory
Programming Research Group

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Oxford

To my grandmother, Mrs. Mao-Qian Chen.

謹獻給外婆陳毛茜女士

Abstract

Many problems in computation can be specified in terms of computing the inverse of an easily constructed function. However, studies on how to derive an algorithm from a problem specification involving inverse functions are relatively rare. The aim of this thesis is to demonstrate, in an example-driven style, a number of techniques to do the job. The techniques are based on the framework of relational, algebraic program derivation.

Simple program inversion can be performed by just taking the converse of the program, sometimes known as to "run a program backwards". The approach, however, does not match the pattern of some more advanced algorithms. Previous results, due to Bird and de Moor, gave conditions under which the inverse of a total function can be written as a fold. In this thesis, a generalised theorem stating the conditions for the inverse of a partial function to be a hylomorphism is presented and proved. The theorem is applied to many examples, including the classical problem of rebuilding a binary tree from its preorder and inorder traversals.

This thesis also investigates into the interplay between the above theorem and previous results on optimisation problems. A greedy linear-time algorithm is derived for one of its instances — to build a tree of minimum height. The necessary monotonicity condition, though looking intuitive, is difficult to establish. For general optimal bracketing problems, however, the thinning strategy gives an exponential-time algorithm. The reason and possible improvements are discussed in a comparison with the traditional dynamic programming approach. The greedy theorem is also generalised to a generic form allowing mutually defined algebras. The generalised theorem is applied to the optimal marking problem defined on non-polynomial based datatypes. This approach delivers polynomial-time algorithms without the need to convert the inputs to polynomial based datatypes, which is sometimes not convenient to do.

The many techniques are applied to solve the Countdown problem, a problem derived from the popular television program of the same name. Different strategies toward deriving an efficient algorithm are experimented and compared.

Finally, it is shown how to derive from its specification the inverse of the Burrows-Wheeler transform, a string-to-string transform useful in compression. Not only do we identify the key property why the inverse algorithm works, but, as a bonus, we also outline how two generalisations of the transform may be derived.

Most of the main part of this thesis has been published or is under preparation to be published. Chapter 4 and Chapter 7 have been published as [68] and [19] respectively. Journal versions of these articles have been submitted. The first two sections of Chapter 5 has been published in [18]. The full proof in Section 5.1.2, however, is only seen in this thesis. We are currently preparing an article covering Chapter 6.

Acknowledgements

The past three years in Oxford were among the happiest time in my life. I would like to express my deep gratitude to Richard Bird, who has been a great supervisor in every aspect. He has been a constant source of inspiration and ideas. His experience and insight always helped whenever I got stuck. I benefited from his supervision not only through his technical advice but also though his ever-open door for students, his frank comments on my work, and his insistence that behind every problem there ought to be a beautiful solution that is worth striving for. I will certainly miss every moment of our meetings in his office, his college, or a local pub, including the way he shouted "out!" in the end when he needed to deal with other urgent business.

I would also like to thank Oege de Moor, who always understood my work better than myself and told me how important it is, for his constant interest and encouragement. My thanks also go to Jeremy Gibbons, who often managed to quickly understand my work however badly I presented it in our meetings, and would then explain to other puzzled members. I enjoyed every meeting of the Algebra of Programming group, whose regular members include Geraint Jones, Clare Martin, and Barney Stratford. With them I learnt how enjoyable it can be to work with a group of excellent people sharing the same interests.

All this could not have happened if it were not for the encouragement from Tyng-Ruey Chuang, who is not only a good teacher at work but also one I can consult with for many important decisions. I would also like to thank Shi-Jean Tai, for teaching me many important things in life and encouraging me to take the offer from Oxford.

Many people gave me useful and constructive comments on contents of this thesis. I would like to thank Roland Backhouse, Ralf Hinze, and Graham Hutton. Johan Jeuring, Andres Leoh, and Isao Sasano painstakingly read through much of this thesis and pointed out many errors. To them I must express my sincere gratitude.

Most of my time in Oxford was spent in the attic in Wolfson Building, staying up late with Malcolm Low and Sunil Nakarani. Many things we did together, such as our successful interior design and our movie nights, will be my sweet memories. Thank-you to my colleagues Stephen Drape, Will Greenland, Yorck Hunke, David Lacy, and Tom Newcomb for the calendar, the dart board, and many happy nights in pubs. I would also like to thank Silvija Seres for her good cooperation and useful advice, many of which triggered important opportunities for me. A thank-you to Christian Greiffenhagen, for his friendly chats and for taking a frightened foreign student who just arrived on the other side of the world to his first movie in Oxford.

Being in a foreign country, I am lucky to have the firm friendship of many people around me. Shih-Hsin Kan spoiled me with his cooking in our first year. Duen-Wei Hsu then took over by dragging me to pubs. I would like to thank Bert Chen, Hsiao-Hui Chen, Jwu-Ching Shu, Hsiao-Ting Lin, Hsin-Yu Lin, Hsiu-Hsu Lin, Po-Hsien Liu, I-Chun Shih, Pi-Ho Wu for their company. I would also like to thank Yu-Chan Lu for reminding me what the most important section in a thesis is. My sincere thank to Yuling Chang for sharing a memorable time with me.

Special thanks to Akiko Nakata, who always tries hard to make me happy, whose love and care always warmed my heart.

My parents Ren-Ho Mu and Pu-Mei Chen generously supported my study both financially and emotionally. I would also like to express my gratitude to my uncle Yen-Ru Chen, who has been my role-model since childhood, and will remain to be so.

I was raised by my grandmother Mrs. Mao-Qian Chen. With her unconditional love, she gave me everything I have and made me who I am. I would always feel safe wherever I am, knowing that she would be home waiting for me with her kind smile. She passed away on 2nd August, 2002. Words cannot describe my sorrow. It is a pity that I did not finish this thesis earlier and make her proud.

Contents

Chapter 1

Introduction

This thesis is about relational program derivation. It will be shown in an example-driven style how various theories and techniques can be applied to derive algorithms from a relational specification. Most examples in this thesis involve inverting functions as a common theme.

It has long been known that program construction by trial and error is doomed to failure and a more systematic approach is required. One methodology toward constructing correct and efficient programs is through program transformation [22, 27]. One starts from a specification which is obviously correct but either inexecutable or inefficient. The specification is then manipulated via successive transformations until an executable program that is efficient, yet still a valid refinement of the original specification, is constructed. We will call the progress from the specification to the resulting program a program *derivation*. Among the many approaches to program derivation, ours is a descendant of the Bird-Meertens Formalism [61, 62, 12, 33]. The characteristics of this style includes a uniform and concise notation for both specification and programming constructs, and a linear, equational reasoning style of program refinement.

There are at least two reasons why program derivation involving inverse functions deserves a thesis of its own. Firstly, inverse functions are useful in specification. Many problems in computation can be specified in terms of computing the inverse of an easily constructed function. Among many obvious examples, parsing is the inverse of printing, while compression and decompression are inverses of each other. As we shall soon see, inverse functions sometimes even arise in unexpected situations. Surprisingly, relatively little research has been done about program derivation when inverse functions are present.

The second reason is that we already have the tools to talk about inverse functions properly. During the last decade, there was a trend in the programming derivation community to move from functions to a relational framework. Relations serve as such a convenient tool for specification that one might begin to believe that it is the right model to develop a theory of programming on. Indeed, it has been proposed that non-determinism should be taken as primitive in a programming and deterministic programming a special case [28, 3]. Considering applications, the best explored area for relational derivation is that of optimisation problems [26, 17]. As the inverse of a function is most conveniently

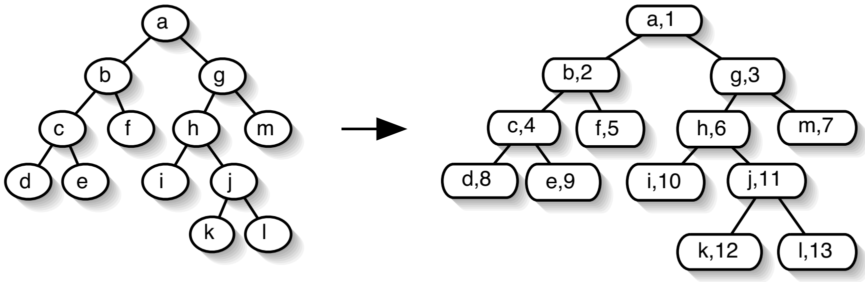

Figure 1.1: Breadth-first labelling a tree on the left with [1..].

described as a relation, it might be another area for which the relational model can be of use. This thesis explores such a possibility, as well as recording some new results on optimisation problems.

1.1 A Teaser

To convince the reader that inverse functions are useful for specification, and to give a feel what this thesis is about, let us consider, as a teaser, the problem of breadth-first labelling.

To breadth-first label a tree with respect to a given list is to augment the nodes of the tree with values in the list in breadth-first order. Figure 1.1 shows the result of breadth-first labelling a tree with 13 nodes with the infinite list [1..]. While everybody knows how to do breadth-first traversal, the closely related problem of efficient breadth-first labelling is not so widely understood.

How would one specify this problem, and what does it have to do with inverse functions? Let us call the type of binary trees $\mathit{Tree}\ A$ and assume that we have at hand the function $\mathit{bft} :: \mathit{Tree}\ A \to \mathit{List}\ A$, for breadth-first traversal, and $\mathit{zipTree} :: \mathit{Tree}\ A \to \mathit{Tree}\ B \to \mathit{Tree}\ (A \times B)$, a *partial* function zipping together two trees of the same shape. To perform breadth-first labelling given a tree t and a list x, we want to zip t with another tree u. What, then, must this tree u satisfy? Firstly, it must be of the right shape, a condition that can be enforced by $\mathit{zipTree}$. Secondly, its breadth-first traversal must be a prefix of the given list x. We thus come up with the following specification:

$$\begin{array}{lcl} \mathit{bfl}\ t\ x & = & \mathit{zipTree}\ t\ u \\ \quad \textbf{where} & & \begin{array}{lcl} \mathit{bft}\ u & = & y \\ y +\!\!+\, z & = & x \end{array} \end{array}$$

Now look at the flow of information in the above specification. The functions bft and $+\!\!+$ appear on the left-hand side, meaning that we wish the data to go *backwards* through them. Let us denote the inverse of a function f by f°, pronounced "the converse of f" or more briefly "f wok". The formal definition of f° will be delayed to Chapter 2. For now, let us say that $f^\circ\ y$ non-deterministically yields some x such that $f\ x = y$. It follows that we can alter the directions of functions and make the specification a pipeline from the right to the left, resulting in the following equivalent point-free specification:

$$bfl\ t \quad = \quad zipTree\ t \cdot bft^{\circ} \cdot fst \cdot cat^{\circ}$$

where $cat = uncurry\,(+\!\!+)$. Here cat° non-deterministically splits the input list in two, therefore $fst \cdot cat^{\circ}$ takes an arbitrary prefix of the input list. The inverse of bft gives us a tree whose breadth-first traversal matches the prefix. The tree is then zipped with the input t.

This is an example where inverses arise unexpectedly in specification. Concise as it is, how does one derive an algorithm from it? The answer, among many other examples, is to be presented in this thesis.

1.2 Background

The use of relations to model programming can be traced to the 80's. Earlier work (for example, [63, 10]) started with modelling common imperative programming constructs as input/output relations. The relational approach was later extended to model datatypes as well as operations on them. Some focused on relations [6, 7, 31, 32], while some took a category theoretical approach [64, 17]. Both approaches gave a formal treatment of important building blocks of functional programs, such as fold and unfold.

The idea of program inversion can be traced at least back to Dijkstra [29]. However, given the importance of inversion as a tool for specification, surprisingly few papers have been devoted to the topic. Among those that do, most deal with imperative program inversion in the context of refinement calculus. A program is inverted by running it "backwards", and the challenging part is when one encounters a branch or a loop [75]. The classic example was to construct a binary tree given its inorder and preorder traversal [38, 39, 24, 81, 78].

Inversion of functional programs has received even less attention. Most published results (e.g. [55, 40]) are based on a "compositional" approach, which is essentially the same as its imperative counterpart: the inverses of the sequentially composed components are recursively constructed, before being combined "in reverse". The recursive process continues until we reach primitives whose inverses are pre-defined.

A matter of concern is: what does it mean to "invert" a function or a program? We know that a function f has a left inverse f^{-1} if for all a, $f^{-1}\,(f\ a) = a$. To invert a function can be thought of as constructing f^{-1} given f. However, f^{-1} exists as a function only if f is injective. To generalise the notion of inversion to arbitrary functions, one possible choice is to switch to set-valued functions (for example, in [41]). To invert a function $f :: A \to B$ is to construct a function $f^{-1} :: B \to Set\ A$, where $f^{-1}\ b$ yields the set of all values a such that $f\ a = b$. In [40], both the domain and range of such a function were lifted to powerdomains. The approach in this thesis, on the other hand, is to work on relations rather than functions and take relational converse to be the "inverse" of a function.

Development on the side of refinement calculus seems to be more advanced. In [24], a program T was defined to be an inversion of program S under precondition P if

$$\{P \wedge Q\}S; T\{Q\}$$

for all predicates Q – that is, when the side condition P holds, T is supposed to put the computer back to *the* initial condition which S started with. This notion of inversion was further discussed in [84, 4], where a program S has an inverse S^- if

$$S^-; S \leq \mathit{skip} \quad \wedge \quad \{\mathit{wp}(S, \mathit{true})\} \leq S; S^-$$

where $\leq$ is the refinement ordering and $\{P\}$ is the predicate transformer for assertions, defined by $\mathit{wp}(\{P\}, Q) = P \wedge Q$. It was also shown in [4] that this view is equivalent to taking the relational converse of a program.

The above techniques, as well as those of this thesis, aim at producing an optimised algorithm by hand. As a consequence, the derivation usually works in a case-by-case basis and human inspiration is an essential part of the derivation. In contrast, efforts have also been made on automatically performing inversion for programs in general, such as in [74, 49, 2]. In [2], two different approaches toward inversion were distinguished: the aim of *inverse computation* is to determine what inputs would deliver in a certain output, while *inverse compilation* or *program inversion* aims at producing a program performing the inverse task of a given program. In this thesis, on the other hand, we will use the term function inversion or program inversion interchangeably to refer to algorithm derivation.

1.3 Outline

Earlier, in the teaser, we made use of inverses without a rigorous definition. This will be remedied in Chapter 2, where the minimal theory necessary for the rest of the thesis will be introduced. We will then show in Chapter 3 some examples of simple program inversion using what we will call the *compositional* approach, culminating in rephrasing the problem of rebuilding a binary tree from its prefix and infix traversals in a functional setting.

However, many algorithms involving inversion do not follow from the simple compositional approach. In Chapter 4, we will present the *converse-of-a-function theorem*, which states the conditions under which the converse of a function is a fold. The power of the theorem will be demonstrated by a number of examples, including the derivation of another algorithm that solves the classical problem of rebuilding a binary tree from its traversals. In this chapter, we will also solve the breadth-first labelling problem described in the teaser. A more general theorem, which allows to write the converse of a partial function as a hylomorphism, is proved and its applications are discussed.

Studies on optimisation problems have been a fruitful area of application for relational program derivation. Chapter 5 starts with an exploration of the interplay between the converse-of-a-function theorem and existing *greedy* and *thinning* theorems for optimisation problems. A greedy linear-time algorithm is derived for a special case of the optimal bracketing problem — to build a tree of minimum height. We will then apply the thinning strategy to solve optimal bracketing problems in general. The result, however, is an exponential-time algorithm. The reason and possible improvements are discussed in a comparison with the traditional dynamic programming approach. The greedy theorem is also generalised to a generic form allowing mutually defined algebras. The generalised theorem is applied to the optimal marking problem defined on non-polynomial based datatypes. This approach delivers polynomial-time algorithms without the need to convert the inputs to polynomial based datatypes, which is sometimes not flexible to do.

In Chapter 6 we present a larger example. The Countdown problem is derived from the popular television program of the same name. The many techniques developed earlier in the thesis, as well as some problem specific optimisations, are applied to derive an efficient algorithm to tackle the problem. Different strategies are subjected to experiment and compared. Some strategies have poor performance, while some deliver up to a three-fold improvement in efficiency.

In Chapter 7 we turn to another example, which is mostly independent from the other chapters. It is shown how to derive from its specification the inverse of the Burrows-Wheeler transform, a string-to-string transform useful in compression. Not only do we identify the key property of why the inverse algorithm works but, as a bonus, we also outline how to derive the inverses of two generalisations of the transform.

Finally we conclude in Chapter 8, give a brief summary of related work and discuss some interesting future directions.

Chapter 2

Preliminaries

This chapter introduces some basic concepts and notations that we will use throughout the thesis.

This chapter aims at tackling two tasks. The first one is to present, using category theory, a uniform treatment of the family of *fold* functions. The family of functions $\mathit{foldr}, \mathit{foldr}1\ldots$ etc. is ubiquitous in functional programming. For example, we can define the following generalised variant of the Haskell Prelude function $\mathit{fold}1$, a fold defined on non-empty lists[1]:

$$\begin{array}{lcl} \mathit{foldrn} & :: & ((A \times B) \to B) \to (A \to B) \to \mathit{List}\ A \to B \\ \mathit{foldrn}\ f\ g\ [x] & = & g\ x \\ \mathit{foldrn}\ f\ g\ (x : \mathit{xs}) & = & f(x, \mathit{foldrn}\ f\ g\ \mathit{xs}) \end{array}$$

We can also define a fold for the datatype *Tree* below, representing tip-valued binary trees:

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Tree}\ A & = & \mathit{tip}\ A \\ & \mid & \mathit{bin}\ (\mathit{Tree}\ A \times \mathit{Tree}\ A) \end{array}$$

$$\begin{array}{lcl} \mathit{foldTree} & :: & ((A \times A) \to A) \to (B \to A) \to \mathit{Tree}\ B \to A \\ \mathit{foldTree}\ f\ g\ (\mathit{tip}\ a) & = & g\ a \\ \mathit{foldTree}\ f\ g\ (\mathit{bin}\ (x, y)) & = & f(\mathit{foldTree}\ f\ g\ x, \mathit{foldTree}\ f\ g\ y) \end{array}$$

In general, every regular datatype gives rise to a corresponding fold function.

We want to be able to talk about properties of these folds in general, as well as to present and to prove some common properties they all share. However, folds for different datatypes may take different numbers of functional arguments, each of different arities.

Category theory offers a concise notation for theorems and proofs, and enables us to talk about properties of many different datatypes as a whole. In the first few sections of this chapter, we will quickly review some fundamental concepts of category theory

[1]The notation here deviates a little from Haskell and is closer to that in [17]. Types begin with capital letters while value constructors, perceived as an injective function, begin with lower-case letters. Single-letter functors are written in sans serif font while multiple-letter functors are written in normal italic font.

just to the extent that is sufficient for our purposes. Then we will show how these basic building blocks help us to model the concept of datatypes. For a more complete account of category theory, the reader is directed to [17, 8].

The second task is to generalise from functions to relations. The inverse of a function is not necessarily a function. One of the ways to formally talk about inverses is to generalise to relations, of which functions are a special case. While the semantics of Haskell is built upon functions between CPOs, for program derivation, we find relations between sets much easier to deal with. This is the approach we will take in this thesis.

2.1 Categories and Functors

A *category* consists of a collection of objects and arrows, together with four operations:

- Two total operations *source* and *target* both assign an object to an arrow. We write $f :: A \to B$ when an arrow f has source A and target B.

- A total operation id takes an object A to an arrow $id_A :: A \to A$. The subscript is sometimes omitted when it is clear from the context.

- A partial operation *composition* takes two arrows $f :: B \to C$ and $g :: A \to B$ to another arrow $f \cdot g :: A \to C$. It is required to be associative and takes id as unit.

While [17] takes an axiomatic approach to category theory, for the purpose of this thesis we can just focus on two special cases: the category **Fun**, whose objects are sets and arrows are total functions between sets, and the category **Rel**, whose objects are sets and arrows are relations. In **Fun**, the arrow id is interpreted as the identity function $id\ a = a$ and composition is just functional composition $(f \cdot g)\ a = f\ (g\ a)$.

A *functor* is a mapping between categories. It consists of two total operations: one maps objects to objects and another maps arrows to arrows, but we usually denote the two mappings by the same name. A functor F satisfies the following properties:

- It respects identity: $\mathsf{F}\, id_A = id_{\mathsf{F} A}$

- It respects composition: $\mathsf{F}(f \cdot g) = \mathsf{F}f \cdot \mathsf{F}g$

These properties, when being used in later chapters, will be referred to as "functor". The notion of functors can be generalised to take more than one argument. A *bifunctor* takes two arguments and satisfies the extended laws:

- $\mathsf{F}(id_A, id_B) = id_{\mathsf{F}(A,B)}$

- $\mathsf{F}(f \cdot h, g \cdot k) = \mathsf{F}(f, g) \cdot \mathsf{F}(h, k)$

Finally, an object is called *initial* if there exists a unique arrow from the initial object to every object in the category.

2.2 Products and Coproducts

The cartesian product $\times$ can be thought of as a bifunctor in **Fun**. The operation of $\times$ on objects is defined by

$$A \times B \quad = \quad \{(a, b) \mid a \in A \wedge b \in B\}$$

Familiar functions $\mathit{fst} :: (A \times B) \rightarrow A$ and $\mathit{snd} :: (A \times B) \rightarrow B$ extract the left and right components of a pair respectively. For every pair of functions $f :: A \rightarrow B$ and $g :: A \rightarrow C$, the function $\langle f, g\rangle :: A \rightarrow (B \times C)$ (pronounced "f fork g") for is defined by:

$$\langle f, g\rangle\, a \quad = \quad (f\ a, g\ a)$$

It satisfies the universal property:

$$h = \langle f, g\rangle \quad \equiv \quad \mathit{fst} \cdot h = f \wedge \mathit{snd} \cdot h = g$$

With fork, the operation of $\times$ on arrows can be defined by

$$f \times g \quad = \quad \langle f \cdot \mathit{fst}, g \cdot \mathit{snd}\rangle$$

The following laws useful for calculation can be derived from the universal property of product:

- cancellation : $\mathit{fst} \cdot \langle f, g\rangle = f$ and $\mathit{snd} \cdot \langle f, g\rangle = g$
- fusion : $\langle f, g\rangle \cdot h = \langle f \cdot h, g \cdot h\rangle$
- absorption : $(h \times k) \cdot \langle f, g\rangle = \langle h \cdot f, k \cdot g\rangle$

If we reverse the directions of all the arrows of a product, we get a *coproduct*. In Fun, coproduct can be defined by

$$A + B \quad = \quad \{\mathit{inl}\ a \mid a \in A\} \cup \{\mathit{inr}\ b \mid b \in B\}$$

A coproduct gives a disjoint union, and the arrows $\mathit{inl} :: A \rightarrow (A + B)$ and $\mathit{inr} :: B \rightarrow (A + B)$ become injections. Just as with fork, we can define for each pair of functions $f :: A \rightarrow C$ and $g :: B \rightarrow C$ an arrow $[f, g] :: (A + B) \rightarrow C$ (pronounced "f join g"):

$$\begin{array}{rcl} [f, g]\,(\mathit{inl}\ a) & = & f\ a \\ [f, g]\,(\mathit{inr}\ b) & = & g\ b \end{array}$$

It satisfies the following universal property:

$$h = [f, g] \quad \equiv \quad f = h \cdot \mathit{inl} \wedge g = h \cdot \mathit{inr}$$

Like product, the coproduct can also be defined to be a bifunctor. The operation of $+$ on arrows can be defined by:

$$f + g \quad = \quad [\mathit{inl} \cdot f, \mathit{inr} \cdot g]$$

As a dual of product, we also have a set of laws

- cancellation : $[f, g] \cdot inl = f$ and $[f, g] \cdot inr = g$,
- fusion : $h \cdot [f, g] = [h \cdot f, h \cdot g]$,
- absorption : $[f, g] \cdot (h + k) = [f \cdot h, g \cdot k]$

It is easy to check that the above definitions for $\times$ and $+$ do satisfy the conditions for being bifunctors.

2.3 Algebras and folds

An arrow of type $\mathsf{F}A \to A$ for some A is called a F-*algebra*, with A being its *carrier*. A F-*homomorphism* from F-algebra $\alpha :: \mathsf{F}A \to A$ to F-algebra $\beta :: \mathsf{F}B \to B$ is an arrow $\gamma :: A \to B$ such that

$$\gamma \cdot \alpha \quad = \quad \beta \cdot \mathsf{F}\gamma$$

The reason for introducing F-algebras is to capture the structures of many different datatypes in a unified form. Look at the following datatype definition for non-empty lists of integers[2]:

$$\textbf{data}\, \mathit{ListInt}_1 \quad = \quad \mathit{wrap}\, \mathcal{Z} \mid \mathit{cons}\, (\mathcal{Z} \times \mathit{ListInt}_1)$$

For brevity we use $\mathcal{Z}$ to denote the set of integers. Note that data constructor *wrap* has type $\mathcal{Z} \to \mathit{ListInt}_1$ and *cons* type $(\mathcal{Z} \times \mathit{ListInt}_1) \to \mathit{ListInt}_1$. If we define F to be a functor whose operations on objects and arrows are respectively

$$\begin{array}{lcl} \mathsf{F}X & = & \mathcal{Z} + (\mathcal{Z} \times X) \\ \mathsf{F}f & = & id_{\mathcal{Z}} + (id_{\mathcal{Z}} \times f) \end{array}$$

then the coproduct $[\mathit{wrap}, \mathit{cons}]$ yields type $\mathsf{F}\mathit{ListInt}_1 \to \mathit{ListInt}_1$. It is thus a F-algebra with carrier $\mathit{ListInt}_1$.

Similarly, consider the type *TreeInt* of leaf-valued binary trees defined below:

$$\textbf{data}\, \mathit{TreeInt} \quad = \quad \mathit{tip}\, \mathcal{Z} \mid \mathit{bin}\, (\mathit{TreeInt} \times \mathit{TreeInt})$$

The join of its constructors $[\mathit{tip}, \mathit{bin}]$ is a G-algebra with *TreeInt* being the carrier, where G is

$$\begin{array}{lcl} \mathsf{G}X & = & \mathcal{Z} + (X \times X) \\ \mathsf{G}f & = & id_{\mathcal{Z}} + (f \times f) \end{array}$$

The two instances above are not the only F-algebra and G-algebra. For any $g :: \mathcal{Z} \to A$ and $f :: (\mathcal{Z} \times A) \to A$, the coproduct $[g, f]$ forms a F-algebra of type $\mathsf{F}A \to A$. Similarly for G. For example, $[id, plus]$, where *plus* is the uncurried addition function on integers, can be seen both as an F-algebra and a G-algebra with carrier $\mathcal{Z}$. Coproduct $[\mathit{wrap}, \mathit{cat}]$,

[2]We use a notation similar to Haskell for datatype declarations. Whereas Haskell prefers curried data constructors, we find uncurried ones more suitable for our purpose.

on the other hand, is a G-algebra with carrier $\mathit{ListInt}_1$, where *cat* is the uncurried variant of $+\!\!+$, concatenating two lists represented by datatype $\mathit{ListInt}_1$.

An important result shown in [58] is that for any F belonging to a certain class of functors (which fortunately includes the examples we are currently interested in), all the F-algebras themselves form a category whose objects are F-algebras and arrows are homomorphisms between F-algebras. Furthermore, initial objects in such categories exist.

When we see the definition of non-empty lists above, we think of it as defining $[\mathit{wrap}, \mathit{cons}]$ to be an initial object in the category of F-algebras, with $\mathit{ListInt}_1$ being its carrier. That $[\mathit{wrap}, \mathit{cons}]$ is initial means that, for any F-algebra h, there exists a unique homomorphism, which we will denote, adopting the concise banana bracket notation, by $(\!|h|\!)_{\mathsf{F}}$. The condition for $(\!|h|\!)_{\mathsf{F}}$ to be a homomorphism reads:

$$(\!|h|\!)_{\mathsf{F}} \cdot [\mathit{wrap}, \mathit{cons}] = h \cdot (\mathit{id}_{\mathcal{Z}} + (\mathit{id}_{\mathcal{Z}} \times (\!|h|\!)_{\mathsf{F}}))$$

Since a coproduct-forming arrow can be represented as a coproduct of arrows [35], we can assume h has form $[g, f]$ without loss of generality. If we split it to pointwise style and write $\mathit{foldrn}\, f\, g$ for $(\!|g, f|\!)_{\mathsf{F}}$ (for this particular F), we obtain the characterisation of fold on non-empty lists, which should look familiar:

$$\begin{array}{lcl} \mathit{foldrn}\, f\, g\, (\mathit{wrap}\ a) & = & g\ a \\ \mathit{foldrn}\, f\, g\, (\mathit{cons}\ (a, x)) & = & f\, (a, \mathit{foldrn}\, f\, g\ x) \end{array}$$

Similarly, for any $f :: (A \times A) \to A$ and $g :: \mathcal{Z} \to A$, the arrow $(\!|g, f|\!)_{\mathsf{G}}$ is the unique homomorphism from initial algebra $[\mathit{tip}, \mathit{bin}] :: \mathsf{G}\,\mathit{TreeInt} \to \mathit{TreeInt}$ to the algebra $[g, f] :: \mathsf{G}A \to A$. Expanding the homomorphic condition and writing $\mathit{foldTree}\, f\, g$ for $(\!|g, f|\!)_{\mathsf{G}}$, we obtain:

$$\begin{array}{lcl} \mathit{foldTree}\, f\, g\, (\mathit{tip}\ a) & = & g\ a \\ \mathit{foldTree}\, f\, g\, (\mathit{bin}\ (x, y)) & = & f\, (\mathit{foldTree}\, f\, g\ x, \mathit{foldTree}\, f\, g\ y) \end{array}$$

In particular, $\mathit{flatten} = \mathit{foldTree}\ \mathit{cat}\ \mathit{wrap}$ is the function flattening a tree to a list by concatenating the elements from the left to the right.

In general, for any functor F and F-algebra $h :: \mathsf{F}A \to A$, a fold $(\!|h|\!)_{\mathsf{F}}$ has type $\mathsf{T} \to A$, where T is the carrier of F. We will call F the *base functor* defining T. The condition for $(\!|h|\!)_{\mathsf{F}}$ to be a homomorphism is

$$(\!|h|\!)_{\mathsf{F}} \cdot \alpha_{\mathsf{F}} \quad = \quad h \cdot \mathsf{F}(\!|h|\!)_{\mathsf{F}}$$

where α_{F} is the initial algebra, or the data constructor, whose type is $\mathsf{FT} \to \mathsf{T}$. It serves both as a definition of $(\!|h|\!)_{\mathsf{F}}$ and an important law for program calculation.

Initiality means not only such a homomorphism $(\!|h|\!)_{\mathsf{F}}$ exists, but it is unique. The uniqueness of the homomorphism implies the following *fold fusion theorem.*

$$h \cdot (\!|f|\!)_{\mathsf{F}} = (\!|g|\!)_{\mathsf{F}} \quad \Leftarrow \quad h \cdot f = g \cdot \mathsf{F}h$$

The fold fusion theorem is considered a very important law for the algebra of programming.

Finally we come to polymorphic datatypes. Datatypes are often parameterised. In that case α_{F} has type $\mathsf{F}_A(\mathsf{T}A) \to \mathsf{T}A$. For example, the base functor for cons-lists over an arbitrary type can be defined by $\mathsf{F}_A\, X = 1 + (A \times X)$. Sometimes we will write $\mathsf{F}(A, X)$ instead of $\mathsf{F}_A\, X$, thinking of F as a bifunctor. As other examples, the base functor for non-empty lists is $\mathsf{F}(A, X) = A + (A \times X)$, and that for leaf-valued binary trees is $\mathsf{F}(A, X) = A + (X \times X)$. The initial algebra α now has type $\mathsf{F}(A, \mathsf{T}A) \to \mathsf{T}A$. When describing the action of F on types, we will always write F as a bifunctor. For its action on functions, we will write $\mathsf{F}f$ in place of $\mathsf{F}(id, f)$ for brevity. We will also omit the type subscripts when it is clear from the context which base functor we are referring to.

Functional programmers are familiar with folds but, curiously, get confused when they see banana brackets. Therefore, we will use *foldr*, *foldrn* or *foldTree*, etc, when we talk about specific folds and use the banana bracket notation only when proving general properties of folds. For example, *foldr* and *foldrn* are defined by:

$$\begin{array}{lcll} \mathit{foldr}\; f\; e & = & (\!|\mathit{const}\; e, f|\!)_{\mathsf{F}} & \textbf{where}\; \mathsf{F}(A, X) = 1 + (A \times X) \\ \mathit{foldrn}\; f\; g & = & (\!|g, f|\!)_{\mathsf{F}} & \textbf{where}\; \mathsf{F}(A, X) = A + (A \times X) \end{array}$$

similarly for other folds. We follow the Haskellish convention of putting the "more complicated" argument in the front, while taking constants rather than constant functions. Unlike in Haskell, however, we use uncurried versions of the arguments.

2.4 Relations

Now it is time to generalise from functions to relations. Set-theoretically speaking, a relation $R :: A \to B$ is a set of pairs (a, b) where a has type A and b type B. For $R :: B \to C$ and $S :: A \to B$, the composition $R \cdot S :: A \to C$ is defined by

$$(a, c) \in R \cdot S \quad \equiv \quad (\exists b : b \in B : (b, c) \in R \wedge (a, b) \in S)$$

Since we use forward arrows for types, we take the left component of the pair as the "input". This notational decision differs from the one used in [7].

Since a relation is just a set, relations of the same type are ordered by set inclusion. Usual set-theoretic operations such as union, intersection and subtraction apply to relations as well. We will not make use of negations, however.

The *converse* of a relation is defined by flipping the pairs, that is,

$$(b, a) \in R^{\circ} \quad \equiv \quad (a, b) \in R$$

It is the notion of "inversion" we are going to adopt in this thesis. When we say to "invert a function", we actually mean to construct its relational converse. Converse distributes into union and intersection, that is, $(R \cup S)^{\circ} = R^{\circ} \cup S^{\circ}$ and $(R \cap S)^{\circ} = R^{\circ} \cap S^{\circ}$. Furthermore, converse is contravariant with respect to composition, i.e., $(R \cdot S)^{\circ} = S^{\circ} \cdot R^{\circ}$.

For each type A, a relation id_A is defined by $id_A = \{(a, a) \mid a \in A\}$. We will omit the subscript when it is clear from the context. A relation $R :: A \to B$ is called *simple* if

$R \cdot R^\circ \subseteq id_B$. That is, every value in A is mapped to at most one value in B. In other words, R is a partial function. A relation R is called *entire* if $id_A \subseteq R^\circ \cdot R$, that is, every value in A is mapped to at least one value in B. A relation is a (total) function if it is both simple and entire.

We follow the convention in [17] that single lower-case letters always denote functions, so we do not have to state so explicitly. Single capital letters or longer identifiers in lower-case letters denote relations in general.

A relation is called a *coreflexive* if it is a subset of id. Coreflexives are useful for modelling predicates. The ? operator converts a boolean-valued (partial) function to a coreflexive:

$$(a, a) \in p? \quad \equiv \quad p\ a$$

For convenience, when p is partial, we let $(a, a) \notin p?$ both when $p\ a$ yields *False* and when a is not in the domain of p. If we perform two consecutive tests, one of them being stronger than the other, the stronger one can absorb the weaker one:

$$(p\ a \Rightarrow q\ a) \quad \Rightarrow \quad p? \cdot q? = p? \tag{2.1}$$

Given a relation $R :: A \to B$, the coreflexive $dom\ R$ determines the domain of R and is defined by

$$(a, a) \in dom\ R \quad \equiv \quad (\exists b : b \in B : (a, b) \in R)$$

Alternatively, $dom\ R = R^\circ \cdot R \cap id$, where $\cap$ denotes set intersection. It follows that

$$dom\ R \subseteq R^\circ \cdot R \tag{2.2}$$

The coreflexive $ran\ R$ determines the range of a relation and is defined by $ran\ R = dom\ R^\circ$.

When writing in the pointwise style, relations can be introduced by the choice operator $\Box$. The expression $x \Box y$ non-deterministically yields either x or y. For example, the following relation *prefix* maps a list to one of its prefixes:

$$\begin{array}{lll}
prefix & :: & List\ A \to List\ A \\
prefix & = & foldr\ step\ [\,] \\
 & & \quad \mathbf{where} \quad \begin{array}{lll} step & :: & A \to List\ A \to List\ A \\ step\ a\ x & = & (a : x)\ \Box\ [\,] \end{array}
\end{array}$$

In each step of the fold we can choose either to cons the current item to some prefix of the sublist, or just return the empty sequence $[\,]$, which is a prefix of every list. When lambda binding and variable substitution are involved, giving a formal semantics for pointwise relational programming is a more involved task than it seems. The semantics of an expression is no longer simply a relation. The reader is referred to [67] for more details. In this thesis we will avoid using those constructs that complicate the semantics, therefore we can just think of the use of $\Box$ operator as syntax sugar to save us from writing in point-free style when the latter is more complicated.

2.5 Power Transpose

We use relations to model non-deterministic behaviour of programs. An alternative approach is to appeal to set-valued functions. The two views can be converted via the power transpose operator Λ. It converts a relation $R :: A \to B$ to a function $\Lambda R :: A \to \mathit{Set}\, B$, defined by

$$(\Lambda R)\, a \quad = \quad \{b \mid (a, b) \in R\}$$

The function ΛR is also called the *breadth* of R. The reverse operation, membership relation $(\in) :: \mathit{Set}\, A \to A$, maps a set x to any of its members. It is defined by $\{(x, b) \mid x :: \mathit{Set}\, A, b \in x\}$. Together they satisfy the universal property:

$$f = \Lambda R \quad \equiv \quad \in \cdot f = R$$

Instantiating f to $\Lambda\, R$, we get the cancellation law:

$$\in \cdot \Lambda R \quad = \quad R$$

As a consequence of the cancellation law we obtain the fusion law:

$$\Lambda(R \cdot f) \quad = \quad \Lambda R \cdot f$$

Furthermore, instantiating f and R to id and $\in$ results in the reflection law $\Lambda \in = id$.

The existential image functor E converts a relation $R :: A \to B$ to a function $\mathsf{E}R :: \mathit{Set}\, A \to \mathit{Set}\, B$. When given a set, $\mathsf{E}R$ applies R to every element of the set and then collects the result.

$$(\mathsf{E}R)x \quad = \quad \{b \mid \exists a : a \in x : (a, b) \in R\}$$

Or, equivalently, $\mathsf{E}R = \Lambda(R \cdot \in)$. The following absorption law is immediate from its definition:

$$\mathsf{E}R \cdot \Lambda S \quad = \quad \Lambda(R \cdot S) \tag{2.3}$$

With the absorption property it is not difficult to see that E is indeed a functor.

The restriction of the existential image functor to functions is written P. There is a further extension of P to take relational arguments, which will be discussed in Section 5.2.2. For now, we only need to know that E and P coincide on functions.

The familiar function $union :: \mathit{Set}\,(\mathit{Set}\, A) \to \mathit{Set}\, A$, which takes the union of a set of sets, can be defined by $union = \mathsf{E} \in$. The following law relates $union$, the existential image functor and the power transpose:

$$\mathsf{E}R \quad = \quad union \cdot \mathsf{P}(\Lambda R) \tag{2.4}$$

2.6 Relators

The notion of functors can be generalised to **Rel** as well. Furthermore, a monotonic functor in **Rel** is called a *relator*. That is, a functor satisfying

$$\mathsf{F}R \subseteq \mathsf{F}S \quad \Leftarrow \quad R \subseteq S$$

for all R and S. The definition is equivalent to saying that F preserves functions and converses, that is, $\mathsf{F}f$ is a function and that:

$$(\mathsf{F}R)^\circ \quad = \quad \mathsf{F}(R^\circ)$$

We can thus omit the surrounding brackets.

The same definition of coproducts still suffices to be a relator in **Rel**. Laws for cancellation, absorption, and fusion still hold. As for products, however, we need to define the fork to be:

$$\langle R, S \rangle \quad = \quad (\mathit{fst}^\circ \cdot R) \cap (\mathit{snd}^\circ \cdot S)$$

The definition for the product remains the same: $(R \times S) = \langle R \cdot \mathit{fst}, S \cdot \mathit{snd} \rangle$. It is still a relator and the absorption law still holds. The fork, however, does not satisfy the same universal property. Instead it is weakened to:

$$\mathit{fst} \cdot \langle R, S \rangle \quad = \quad R \cdot \mathit{dom}\, S \tag{2.5}$$

$$\mathit{snd} \cdot \langle R, S \rangle \quad = \quad S \cdot \mathit{dom}\, R \tag{2.6}$$

2.7 Relational Folds

With the ingredients prepared in the previous two sections, finally we are able to generalise folds to relations. A fold taking a relational argument is defined in terms of its functional counterpart, as in:

$$(\!|R|\!) \quad = \quad \in \cdot (\!|\Lambda(R \cdot \mathsf{F} \in)|\!)$$

where F is the base functor of the fold. It can be proved that, under the above definition, $(\!|R|\!)$ still satisfies the universal property

$$X = (\!|R|\!) \quad \equiv \quad X \cdot \alpha = R \cdot \mathsf{F}X$$

Since $(\!|R|\!)$ is unique, it is both the least fixed-point of the inequation $R \cdot \mathsf{F}X \cdot \alpha^\circ \subseteq X$ and the greatest fixed-point of $X \subseteq R \cdot \mathsf{F}X \cdot \alpha^\circ$. The fusion theorem thus has two variants when it comes to relational folds.

$$R \cdot (\!|S|\!) \subseteq (\!|T|\!) \quad \Leftarrow \quad R \cdot S \subseteq T \cdot \mathsf{F}R$$

$$(\!|T|\!) \subseteq R \cdot (\!|S|\!) \quad \Leftarrow \quad T \cdot \mathsf{F}R \subseteq R \cdot S$$

The definition of relational folds also tells us how to distribute Λ into a fold:

$$\begin{array}{cl}
& \Lambda(\![R]\!) \\
= & \quad \{\text{by definition}\} \\
& \Lambda(\in \cdot (\![\Lambda(R \cdot \mathsf{F}\in)]\!)) \\
= & \quad \{(\![\Lambda(R \cdot \mathsf{F}\in)]\!) \text{ a function, } \Lambda \text{ fusion}\} \\
& \Lambda\in \cdot (\![\Lambda(R \cdot \mathsf{F}\in)]\!) \\
= & \quad \{\text{reflection law: } \Lambda\in = id\} \\
& (\![\Lambda(R \cdot \mathsf{F}\in)]\!)
\end{array}$$

This is often referred to as the *Eilenberg-Wright Lemma*.

For a fuller account of relator theory and relational catamorphisms, the reader is referred to [6, 7].

2.8 Hylomorphisms and Fixed-points

The converse of a fold is called an *unfold*. A fold after an unfold is called a *hylomorphism*. The unfolding phase generates an intermediate data structure, while the folding phase consumes it. More precisely, consider a hylomorphism $(\![R]\!) \cdot (\![S]\!)^\circ$, where R has type $\mathsf{F}A \to A$, S has type $\mathsf{F}B \to B$, and F is the base functor of T. The unfolding phase $(\![S]\!)^\circ :: B \to \mathsf{T}$ produces a value of type T, which is then consumed by $(\![R]\!) :: \mathsf{T} \to A$. All primitive recursive functions can be written as hylomorphisms.

A hylomorphisms can be characterised as a least fixed point:

$$(\![R]\!) \cdot (\![S]\!)^\circ \quad = \quad \mu(X \mapsto R \cdot \mathsf{F}X \cdot S) \tag{2.7}$$

Here we denote anonymous functions by the $\mapsto$ notation and fixed-point operator by μ. Since $(\![\alpha]\!) = id$, both folds and unfolds are special cases of (2.7) by substituting α for R or S respectively. Furthermore, folds and unfolds are unique fixed-points characterised by (2.7). The question of when this fixed-point is unique has been answered by [31, 32, 30]. We will talk more about that in Section 4.6.

A calculus of fixed-points becomes handy when the, usually simpler but less general, laws on folds do not apply. A summary of fixed-point calculus can be found in [5]. Among the many rules, we will only cite the *fixed-point fusion* theorem below and leave the others to be introduced when they are needed. The theorem says that, provided that h is a lower adjoint in a Galois connection, we have:

$$h(\mu f) = \mu g \quad \Leftarrow \quad h \cdot f = g \cdot h$$

The Galois connection is an important, re-occurring concept in many fields of mathematics and computation, although it is beyond the scope of this thesis to go into a fuller discussion. What is immediately relevant to us now is that the converse operator and the power-transpose operator are both lower adjoints. The reader can then verify that $(\mu\, f)^\circ = \mu(X \mapsto (f\, X^\circ)^\circ)$. Or, more concisely:

$$(\mu\, f)^\circ \quad = \quad \mu((^\circ) \cdot f \cdot (^\circ)) \tag{2.8}$$

Chapter 3

The Compositional Approach

Many program inversions are performed via what we will call a *compositional* approach: given the definition of a function of interest in terms of simpler components, we construct the converse of each component and thereby construct the converse of the given function. Laws of use here are various distributivity laws of the converse operator, such as $(R \cdot S)^\circ = S^\circ \cdot R^\circ$, $(R \cup S)^\circ = R^\circ \cup S^\circ$, etc.

In this chapter we will look at three such examples. The first one is to split a list into two in all possible ways by inverting *cat*. It is basically rephrasing the same example discussed in [40] in a relational setting. As a second example, we consider the similar problem of inverting *concat*, in order to set the stage for when we consider the same problem again in later chapters. Finally, we review the famous problem of constructing a tree from its inorder and preorder traversal, but in a non-imperative setting.

3.1 Splitting a List into Two

The function $cat :: (List\ A \times List\ A) \rightarrow List\ A$ is the uncurried variant of the Haskell Prelude function $+\!\!+$:

$$
\begin{array}{lcl}
cat\,([\,], y) & = & y \\
cat\,(a : x, y) & = & a : cat\,(x, y)
\end{array}
$$

For convenience in the next section, however, we will instead consider the variant $cat_1 :: (List_1\ A \times List_1\ A) \rightarrow List_1\ A$, whose domain and range are restricted to non-empty lists only:

$$
\begin{array}{lcl}
cat_1\,([a], b : y) & = & a : b : y \\
cat_1\,(a : x, y) & = & a : cat_1\,(x, y)
\end{array}
$$

This variation is a partial function concatenating two non-empty lists into one. Conversely, the relation cat_1°, also partial, splits a given list into two *non-empty* lists in an arbitrary way. In this section we will show how to derive Λcat_1°. Since it is our first example of relational program derivation, we will go through it in finer details, even though this seems to be a rather hairy approach to a simple problem. We will come back to this problem again in Section 4.7.1, where another approach to the problem will be mentioned.

We will rewrite cat_1 as the least fixed-point of cat_F, defined by:

$$\begin{aligned} cat_F\, X &= (cons \cdot (wrap^\circ \times (not \cdot null)?)) \cup \\ &\quad\ (cons \cdot (id \times X) \cdot assocl \cdot (cons^\circ \times id)) \end{aligned}$$

where $cons$ is the uncurried version of the list constructor (:), $null$ is the predicate testing whether the given argument is an empty list, and the plumping function $assocl :: ((A \times B) \times C) \rightarrow (A \times (B \times C))$ is defined by $assocl\ ((a, b), c) = (a, (b, c))$.

Our aim is to derive $\Lambda(\mu cat_F)^\circ$. We will do so in two steps: first by promoting the converse into the fixed-point using (2.8), recited below:

$$(\mu f)^\circ \quad = \quad \mu((^\circ) \cdot f \cdot (^\circ))$$

and second by promoting Λ into the resulting fixed-point.

To promote the converse operator into μcat_F, we just need to construct $(^\circ) \cdot cat_F \cdot (^\circ)$. We reason:

$$\begin{aligned} & (cat_F\, X^\circ)^\circ \\ = & \quad \{\text{by definition}\} \\ & (cons \cdot (wrap^\circ \times (not \cdot null)?) \cup cons \cdot (id \times X^\circ) \cdot assocl \cdot (cons^\circ \times id))^\circ \\ = & \quad \{\text{since composition distributes into union}\} \\ & (cons \cdot ((wrap^\circ \times (not \cdot null)?) \cup (id \times X^\circ) \cdot assocl \cdot (cons^\circ \times id)))^\circ \\ = & \quad \{\text{since } (R \cdot S)^\circ = S^\circ \cdot R^\circ\} \\ & ((wrap^\circ \times (not \cdot null)?) \cup (id \times X^\circ) \cdot assocl \cdot (cons^\circ \times id))^\circ \cdot cons^\circ \end{aligned}$$

Since $(R \cup S)^\circ = R^\circ \cup S^\circ$, we will consider the two sides of the union separately. For the left-hand side, the converse operator distributes into product, resulting in $(wrap \times (not \cdot null)?)$. For the other side, we reason:

$$\begin{aligned} & ((id \times X^\circ) \cdot assocl \cdot (cons^\circ \times id))^\circ \\ = & \quad \{\text{since } (R \cdot S)^\circ = S^\circ \cdot R^\circ\} \\ & (cons^\circ \times id)^\circ \cdot assocl^\circ \cdot (id \times X^\circ)^\circ \\ = & \quad \{\text{since relators preserve converse}\} \\ & (cons \times id) \cdot assocl^\circ \cdot (id \times X) \end{aligned}$$

Defining $assocr\ (a, (b, c)) = ((a, b), c)$, we have

$$assocl^\circ \quad = \quad assocr \tag{3.1}$$

For the curious reader, a proof of (3.1) will be presented in Appendix A.

The above reasoning shows that $(^\circ) \cdot cat_F \cdot (^\circ) = split_F$, where $split_F$ is defined by

$$\begin{aligned} split_F\, X &= ((wrap \times (not \cdot null)?) \cup \\ &\quad\ (cons \times id) \cdot assocr \cdot (id \times X)) \cdot cons^\circ \end{aligned}$$

Therefore, we have $(\mu cat_F)^\circ = \mu split_F$. Although the derivation involves manipulating long expressions, it is essentially just mechanically pushing the converse operator as deeply inside the expression as possible.

The second step is to calculate $\Lambda(\mu split_F)$. Expanding $\mu split_F$ and using (2.3), we get

$$\begin{array}{lcl}\Lambda(\mu split_F) & = & \mathsf{E}((wrap \times (not \cdot null)?) \cup \\ & & \quad (cons \times id) \cdot assocr \cdot (id \times \mu split_F)) \cdot \Lambda cons^\circ\end{array}$$

Let cup denote uncurried set union, defined by $cup = \Lambda(\in \cdot (fst \cup snd))$. The following property, also proved in Appendix A, enables us to distribute power transpose into union:

$$\Lambda(R \cup S) \quad = \quad cup \cdot \langle \Lambda R, \Lambda S \rangle \tag{3.2}$$

Since $\mathsf{E}R = \Lambda(R \cdot \in)$, as a corollary we have

$$\mathsf{E}(R \cup S) = cup \cdot \langle \mathsf{E}R, \mathsf{E}S \rangle$$

Therefore, we can rewrite $\Lambda(\mu split_F)$ as:

$$\begin{array}{lcl}\Lambda(\mu split_F) & = & cup \cdot \langle \mathsf{E}(wrap \times (not \cdot null)?), \\ & & \quad \mathsf{E}((cons \times id) \cdot assocr \cdot (id \times \mu split_F))\rangle \cdot \Lambda cons^\circ\end{array}$$

The left component of the fork, $\mathsf{E}(wrap \times (not \cdot null)?)$, simply constructs a singleton set containing a pair of lists if the predicate holds. We thus continue with simplifying the right component. Define the function cpr by:

$$cpr\, f\, (a, b) \quad = \quad \{(a, c) \mid c \in f\, b\}$$

such that, given a set-valued function $f :: B \to Set\, C$, the function $cpr\, f :: (A \times B) \to Set\,(A \times C)$ implements $\Lambda\,(id \times \in \cdot f)$. We reason:

$$\begin{array}{ll} & \mathsf{E}((cons \times id) \cdot assocr \cdot (id \times \mu split_F)) \\ = & \quad \{\text{functor}\} \\ & \mathsf{E}((cons \times id) \cdot assocr) \cdot \mathsf{E}(id \times \mu split_F) \\ = & \quad \{(2.4)\} \\ & \mathsf{E}((cons \times id) \cdot assocr) \cdot union \cdot \mathsf{P}\Lambda(id \times \mu split_F) \\ = & \quad \{\text{introducing } cpr\text{, where } f = \Lambda(\mu split_F)\} \\ & \mathsf{E}((cons \times id) \cdot assocr) \cdot union \cdot \mathsf{P}(cpr\, \Lambda(\mu split_F)) \\ = & \quad \{\text{since } \mathsf{E} \text{ and } \mathsf{P} \text{ coincide on functions}\} \\ & \mathsf{P}((cons \times id) \cdot assocr) \cdot union \cdot \mathsf{P}(cpr\, \Lambda(\mu split_F))\end{array}$$

Therefore, we conclude that:

$$\begin{array}{lcl}\Lambda\mu split_F & = & cup \cdot \langle \mathsf{E}(wrap \times (not \cdot null)?), \\ & & \quad \mathsf{P}((cons \times id) \cdot assocr) \cdot union \cdot \mathsf{P}(cpr\, \Lambda\mu split_F)\rangle \cdot \Lambda cons^\circ)\end{array}$$

Equivalently, $\Lambda\mu split_F$ is a fixed-point of the relation-valued function:

$$\begin{array}{lcl}X & \mapsto & cup \cdot \langle \mathsf{E}(wrap \times (not \cdot null)?), \\ & & \quad \mathsf{P}((cons \times id) \cdot assocr) \cdot union \cdot \mathsf{P}(cpr\, X)\rangle \cdot \Lambda cons^\circ)\end{array}$$

```
splits1 [] = []
splits1 (a:x) = [([a],x) | not (null x)] ++
                [(a:y,z) | (y,z) <- splits1 x]
```

Figure 3.1: Haskell code implementing $splits_1$.

For reasons to be discussed in Section 4.6.1, $\Lambda\mu split_F$ is actually the unique fixed-point of the above relation-valued function. We can therefore take the recursive equation as the definition of $\Lambda\mu split_F$.

In the implementation we will represent sets by lists. The power functor P can be implemented by *map*, and *cup* and *union* by list concatenation. If we switch back to pointwise definition and list comprehension, we get the code shown in Figure 3.1. where $\Lambda cons^\circ$ is implemented by pattern matching. The case for $splits_1\,[\,]$ yields an empty set (represented as an empty list) because $(\Lambda cons^\circ)\,[\,]$ yields an empty set.

As a side remark, had we started from the point-free definition of *cat*:

$$\begin{array}{rcl} cat & = & \mu(X \mapsto (snd \cdot (null? \times id) \cup \\ & & \qquad (cons \cdot (id \times X) \cdot assocl \cdot (cons^\circ \times id))) \end{array}$$

we would have recovered the standard Haskell definition of *splits*:

$$\begin{array}{rcl} splits\,[\,] & = & [([\,],[\,])] \\ splits\,(a : x) & = & [([\,], a : x)] \mathbin{+\!\!+} [(a : y, z) \mid (y, z) \leftarrow splits\,x] \end{array}$$

3.2 Partitioning a List

As the second example, let us consider the problem of computing all partitions of a list. For example, given $[1, 2, 3]$, we want the set:

$$\{[[1], [2], [3]], [[1], [2, 3]], [[1, 2], [3]], [[1, 2, 3]]\}$$

Since we are not interested in an infinite sequence of empty lists, we restrict our attention to non-empty lists. Let $concat :: List\,(List_1\,A) \rightarrow List\,A$ be the function concatenating a list of non-empty lists. It turns out that its definition is best given in two steps, one step dealing with the empty case separately. Let *concat* filter out the case when the input list is empty:

$$\begin{array}{rcl} concat\,[\,] & = & [\,] \\ concat\,xs & = & concat_1\,xs \end{array}$$

Here, $concat_1 :: List_1\,(List_1\,A) \rightarrow List_1\,A$ can be defined simply as a fold on non-empty lists:

$$concat_1 \;=\; foldrn\;cat_1\;id$$

The aim is to calculate $partitions = \Lambda concat^\circ$. Since $concat_1$ returns only non-empty lists, we can deal with the empty case separately, that is, $partitions\,[\,] = \{[\,]\}$. Now we will focus on constructing $partitions_1 = \Lambda concat_1^\circ$.

```
partitions [] = [[]]
partitions x = [[x]] ++ [y:zs | (y,z) <- splits1 x, zs <- partitions z]
```

Figure 3.2: Haskell code implementing *partitions*.

Similar to the last section, we start with looking for a recursive characterisation for $concat_1^\circ$. This step, however, is a bit easier than in the previous section because, since $concat_1$ is a fold, its converse is an unfold. We can therefore directly appeal to (2.7) and conclude:

$$concat_1^\circ \quad = \quad \mu(X \to [wrap, cons] \cdot (id + (id \times X)) \cdot [id, cat_1]^\circ)$$

By laws of coproduct, the above is equivalent to:

$$concat_1^\circ \quad = \quad \mu(X \to wrap \cup (cons \cdot (id \times X) \cdot cat_1^\circ))$$

The next step, to work out $\Lambda concat_1^\circ$, is similar to that in the previous section. We unfold the fixed-point definition of $concat_1^\circ$, and try to promote the Λ operator to the leaf of the expression. Calculation on the more complicated branch goes:

$$\begin{array}{ll}
& \Lambda(wrap \cup (cons \cdot (id \times concat_1^\circ) \cdot cat_1^\circ)) \\
= & \quad \{\text{by } (3.2)\} \\
& cup \cdot \langle \Lambda wrap, \Lambda(cons \cdot (id \times concat_1^\circ) \cdot cat_1^\circ) \rangle \\
= & \quad \{\Lambda \text{ absorption}\} \\
& cup \cdot \langle \Lambda wrap, \mathsf{E} cons \cdot union \cdot \mathsf{P}\Lambda(id \times concat_1^\circ) \cdot \Lambda cat_1^\circ \rangle \\
= & \quad \{\text{introduce } cpr \text{, letting } f = \Lambda concat_1^\circ\} \\
& cup \cdot \langle \Lambda wrap, \mathsf{P} cons \cdot union \cdot \mathsf{P}(cpr\, \Lambda concat_1^\circ) \cdot \Lambda cat_1^\circ \rangle
\end{array}$$

It is also the unique fixed-point. We therefore conclude that:

$$\Lambda concat_1^\circ \quad = \quad \mu(X \to cup \cdot \langle \Lambda wrap, \mathsf{P} cons \cdot union \cdot \mathsf{P}(cpr\, X) \cdot \Lambda cat_1^\circ \rangle)$$

Refining sets to lists and putting the empty case back, we get the pointwise definition familiar to Haskell programmers, shown in Figure 3.2.

This implementation, however, is very inefficient because of overlapping recursive calls. For example, both $([1], [2, 3, 4, 5])$ and $([1, 2], [3, 4, 5])$ are possible splits of $[1, 2, 3, 4, 5]$. To compute its partitions, one will need to recurse on $[2, 3, 4, 5]$ and $[3, 4, 5]$, among others. To compute the partitions of $[2, 3, 4, 5]$, however, another call to *partitions* $[3, 4, 5]$ will be made. One therefore might wish to switch to a bottom-up algorithm, reusing the computed results. In Section 4.2, we will demonstrate another approach to derive an alternative implementation for $\Lambda concat^\circ$. More discussions on top-down v.s. bottom-up algorithms will be given in Chapter 6.

3.3 Rebuilding a Tree from its Traversals

It is well known that, given the inorder and preorder traversal of a binary tree whose labels are all distinct, one can reconstruct the tree uniquely. The problem has been recorded in [53, Section 2.3.1, Exercise 7] as an exercise, where Knuth briefly described why it can be done and commented that it "would be an interesting exercise" to write a program for the task. Indeed, it has become a classic problem to tackle for those who study program inversion, for example, see [24, 81]. As van de Snepscheut noted in [81], one class of solution attempts to invert an iterative algorithm while the other class delivers a recursive algorithm. In this section we will look at the second alternative. The derivation here is a rephrasing of that in [81] in a non-imperative style. The class of iterative solutions, on the other hand, will be discussed in Section 4.5.

To formalise the problem, consider internally labelled binary trees defined by the following datatype:

$$\textbf{data}\ \mathit{Tree}\ A \quad = \quad \mathit{null} \mid \mathit{node}\,(A \times (\mathit{Tree}\ A \times \mathit{Tree}\ A))$$

The fold function for *Tree* is defined by:

$$\begin{array}{lcl}
\mathit{foldTree} :: (A \times (B \times B)) \to B \to \mathit{Tree}\ A \to B & & \\
\mathit{foldTree}\ f\ e\ \mathit{null} & = & e \\
\mathit{foldTree}\ f\ e\,(\mathit{node}\,(a,(t,u))) & = & f\,(a,(\mathit{foldTree}\ f\ e\ t, \mathit{foldTree}\ f\ e\ u))
\end{array}$$

Inorder and preorder traversal on the trees can then be defined in terms of *foldTree*:

$$\begin{array}{lcl}
\mathit{preorder} & = & \mathit{foldTree}\ \mathit{pre}\ [\,] \\
\mathit{pre}\,(a,(x,y)) & = & [a] +\!\!+\, x +\!\!+\, y \\
& & \\
\mathit{inorder} & = & \mathit{foldTree}\ \mathit{inf}\ [\,] \\
\mathit{inf}\,(a,(x,y)) & = & x +\!\!+\, [a] +\!\!+\, y
\end{array}$$

Define $\mathit{pinorder} = \langle \mathit{preorder}, \mathit{inorder} \rangle$, of type $\mathit{Tree}\ A \to (\mathit{List}\ A \times \mathit{List}\ A)$, and let *distinct* be a predicate on trees yielding true for trees whose labels are all distinct. The task is to derive $\mathit{distinct}? \cdot \mathit{pinorder}^\circ$. It should also be shown that $\mathit{distinct}? \cdot \mathit{pinorder}^\circ$ is a simple relation, so that the tree is indeed uniquely determined by the two traversals. Finally, we wish that the derived algorithm has a linear-time complexity.

3.3.1 An Attempt via Direct Inversion

A standard tupling transform (see, for example [44] or [17, Chapter 3]) yields the following definition of *inpreorder* as a fold:

$$\begin{array}{lcl}
\mathit{pinorder} & :: & \mathit{Tree}\ A \to (\mathit{List}\ A \times \mathit{List}\ A) \\
\mathit{pinorder} & = & \mathit{foldTree}\ \mathit{pi}\ ([\,],[\,]) \\
\quad \textbf{where}\ \mathit{pi} & = & \langle \mathit{pre} \cdot \mathsf{F}_1 \mathit{fst}, \mathit{inf} \cdot \mathsf{F}_1 \mathit{snd} \rangle
\end{array}$$

where $\mathsf{F}_1 f = (id \times (f \times f))$. We can invert *pinorder* as an unfold if we can invert *pi*. Furthermore, since we want the inverse to be a simple relation, pi° had better be simple too. However, if we expand the definition of *pi* in pointwise style:

$$pi\,(a, ((x_1, y_1), (x_2, y_2))) \quad = \quad ([a] \mathbin{+\!\!+} x_1 \mathbin{+\!\!+} x_2, y_1 \mathbin{+\!\!+} [a] \mathbin{+\!\!+} y_2)$$

it is clear that *pi* has as an inverse a simple relation only if a is not present in either y_1 or y_2 – otherwise there would be more than one possible decomposition to split the second list. Even then, we are still left with resolving the non-determinism in inverting $[a] \mathbin{+\!\!+} x_1 \mathbin{+\!\!+} x_2$. That is why we need the labels in the tree to be all distinct. It is not difficult to show that

$$pinorder \cdot distinct? \quad = \quad eqSet? \cdot (nodup? \times nodup?) \cdot pinorder$$

where *nodup* is a predicate on lists yielding true for lists that contain no duplicated elements, and $eqSet :: (List\ A \times List\ A) \rightarrow Bool$ is defined by $eqSet\,(x, y) \equiv setify\,x = setify\,y$, ensuring that the two lists contain the same elements. The invariant can be fused into *pinorder* via fold fusion, thus in inverting *pi*, we can split the first list in a unique way and split the second list according to how the first was split.

This is how Knuth explained in [54] that there is indeed such a unique construction of the tree. We will not go into the details, however, since a naive implementation following this line would result in a cubic time algorithm.

3.3.2 Adding Redundancy

In the next two sections we will construct a linear-time algorithm to rebuild a tree from its preorder and inorder traversals. However, to reduce the amount of detail, we will make the fusion of the invariant $(nodup? \times nodup?)$ into *pinorder* implicit and simply assume in this section that x and y are both lists containing no duplicated elements. The derivation in the next two sections will be presented in pointwise style. To reduce the number of brackets, we will also make use of triples, which can be defined easily in terms of binary tuples.

What we will do now is to come up with a function more general than *pinorder*, but whose inversion is trivial. At least two factors contribute to the cubic behaviour of the previous algorithm. One is due to the data structure we use – searching and cutting a list in the middle is a linear-time operation. This problem will be solved by introducing an accumulating parameter.

The second problem is more fundamental. We have to look back and forth on the input pair of lists to decide where to cut them, and that is because we simply do not have enough information to do it more quickly[1]. As a similar example, consider the function

[1] Oege de Moore kindly pointed out that the problem can also be resolved by defining:

$$\begin{array}{lll} rebuild\,a\,(x, y) & = & (pinorder^\circ\,(x_1, y_1), x_2, y_2) \\ \quad \mathbf{where}\ (x_1, x_2) & = & splitAt\,(length\,y_1)\,x \\ \quad\quad y_1 \mathbin{+\!\!+} [a] \mathbin{+\!\!+} y_2 & = & y \end{array}$$

and deriving *rebuild*. This route is much simpler than the one in this section. The author is planning to integrate it to a future paper. For now, however, we will stick with the approach resembling that in [81].

x	y	xor
0	0	0
0	1	1
1	0	1
1	1	0

x	y	xor	y
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

Figure 3.3: Truth tables of the functions xor and $\langle xor, snd\rangle$.

xor whose truth table is shown in Figure 3.3. The function xor is not invertible (to a function) because it is not injective. However, $\langle xor, snd\rangle$ is. The extra output records some information about the history of the computation, enabling us to put the machine back to its original state. Such logic gates with extra "garbage lines" are essential in [83] to the construction of *logically reversible* devices, which are of interest to physicists and researchers in quantum computing. Similar ideas were also lifted to a higher level in the design of programming languages for reversible programs [85]. In this section we will also need to make the program produce some "redundant" outputs before being able to invert it.

Now let us start with the first point. To introduce accumulating parameters, let us define:

$$\begin{array}{lcl} prin & :: & (Tree\ A \times (List\ A \times List\ A)) \to (List\ A \times List\ A) \\ prin\,(u,(x,y)) & = & (z +\!\!+\, x, w +\!\!+\, y) \\ \quad \mathbf{where} & & (z,w) = pinorder\ u \end{array}$$

Equivalently,

$$prin\,(u,(x,y)) \;=\; (preorder\ u +\!\!+\, x, inorder\ u +\!\!+\, y)$$

By standard techniques we can easily derive a recursive definition of $prin$:

$$\begin{array}{l} prin\,(null,(x,y)) \;=\; (x,y) \\ prin\,(node\,(a,(u,v)),(x,y)) \;=\; \\ \quad \mathbf{let}\ \ (x',y') \;=\; prin\,(v,(x,y)) \\ \qquad\ \ (x'',y'') \;=\; prin\,(u,(x',[a] +\!\!+\, y')) \\ \quad \mathbf{in}\ \ ([a] +\!\!+\, x'', y'') \end{array}$$

Can we invert $prin$ to a partial function? One problem would be that given an output (x,y), we are not sure where it came from: did it come from the first pattern, in which case we should return $(null,(x,y))$, or did it come from the second pattern, in which case we should return a $node$? The problem comes from the fact that the first case returns (x,y) unaltered, so we cannot detect whether it has been invoked. It would be better to move some of the work to the first case such that both cases leave some footprints. Therefore we come up with this new definition for $prin$:

$$\begin{array}{l} prin :: (Tree\ A \times (A \times List\ A \times List\ A)) \to (List\ A \times List\ A) \\ prin\,(u,(b,x,y)) \;=\; (preorder\ u +\!\!+\, x, inorder\ u +\!\!+\, [b] +\!\!+\, y) \end{array}$$

Beside the pair of lists (x, y), the new definition takes an extra element b and appends it in front of y. As a result, when u is a null tree, it should return $(x, [b] +\!\!+\, y)$, while in the case when u is not null we can make use of this extra b and move some work to the first case.

Now we come to the second problem mentioned in the beginning of this section. It will turn out that we also need to quote b as part of the output in order to invert *prin*. Our final choice for a proper definition of *prin* will thus be:

$$\begin{array}{l} \mathit{prin} :: (\mathit{Tree}\ A \times (A \times \mathit{List}\ A \times \mathit{List}\ A)) \rightarrow (A \times \mathit{List}\ A \times \mathit{List}\ A) \\ \mathit{prin}\,(u, (b, x, y)) \quad = \quad (b, \mathit{preorder}\ u +\!\!+\, x, \mathit{inorder}\ u +\!\!+\, [b] +\!\!+\, y) \end{array}$$

By simple reasoning we can derive the recursive definition below for *prin*:

$$\begin{array}{lll} \mathit{prin}\,(\mathit{null}, (b, x, y)) & = & (b, x, b : y) \qquad \text{(a)} \\ \mathit{prin}\,(\mathit{node}\,(a, (u, v)), (b, x, y)) & = & \\ \quad \textbf{let} \quad (\_b, x', y') = \mathit{prin}\,(v, (b, x, y)) & & \qquad \text{(b)} \\ \quad \phantom{\textbf{let}} \quad (\_a, x'', y'') = \mathit{prin}\,(u, (a, x', y')) & & \qquad \text{(c)} \\ \quad \textbf{in} \quad (b, a : x'', y'') & & \end{array}$$

Notice the non-standard use of patterns like $\_a$ and $\_b$. The pattern $\_a$ is just a syntax sugar for a coreflexive $(a ==)?$. It is like a "don't-care" pattern in that the matched results are thrown away, except for that it only matches certain values.

Finally, check that *prin* maintains this invariant: provided that the values in all the initial arguments to *prin* are all distinct, the values in x, y, a and b remains all distinct at each recursive call. This invariant is important in making the inversion possible.

3.3.3 The Inversion

All the hassle we have been through was just to put *prin* in a form easy to invert, and our effort indeed pays. Denote by *rebuild* the converse of *prin*. Look at the recursive definition of *prin* in the last section and consider a triple returned by *prin*. If it looks like $(b, x, b : y)$, that is, the head of the last list equals b, oweing to the invariant, the triple must have come from the case marked (a). We shall thus just perform the converse action of returning $(\mathit{null}, (b, x, y))$, resulting in case (a') below. Otherwise it must have come from the second case, which is also inverted by switching the roles of input and output. For example, the let binding $(\_b, x', y') = \mathit{prin}\,(v, (b, x, y))$ is inverted to $(v, (\_b, x, y)) = \mathit{prin}^{\circ}\,(b, x', y')$. The resulting program is shown below:

$$\begin{array}{lll} \mathit{rebuild} :: (A \times \mathit{List}\ A \times \mathit{List}\ A) \rightarrow (\mathit{Tree}\ A \times (A \times \mathit{List}\ A \times \mathit{List}\ A)) & & \\ \mathit{rebuild}\,(b, x, b : y) & = & (\mathit{null}, (b, x, y)) \qquad \text{(a')} \\ \mathit{rebuild}\,(b, a : x'', y'') & = & \\ \quad \textbf{let} \quad (u, (\_a, x', y')) = \mathit{rebuild}\,(a, x'', y'') & & \qquad \text{(c')} \\ \quad \phantom{\textbf{let}} \quad (v, (\_b, x, y)) = \mathit{rebuild}\,(b, x', y') & & \qquad \text{(b')} \\ \quad \textbf{in} \quad (\mathit{node}\,(a, (u, v)), (b, x, y)) & & \end{array}$$

Notice how *prin* and *rebuild* are symmetrical: (a) is inverted to (a'), and the two *let*-binding, (b) and (c) in *prin* are inverted respectively to (b') and (c') and performed in reverse order. Since we have made the plumping implicit by working in pointwise style, the inversion has the feel of running the program backwards. Also note that the pattern $(b, x, b : y)$ is PROLOGish in that we require the two occurrences of b to have the same value.

Having inverted *prin*, we have yet to relate the latest definition of *prin* to the original problem. How is *prin* related to *pinorder*? The function *prin* takes not only two lists, but also a label b. To assign to b a value, we assume the existence of a value ϕ distinct from all values in the tree. The value will then be taken away after the traversal. The relationship between *prin* and *pinorder* can be expressed as the following equality:

$$\begin{array}{lll} \mathit{pinorder} & = & \mathit{cut} \cdot \mathit{prin} \cdot \mathit{init} \\ & \mathbf{where} & \mathit{init}\, u \quad = \quad (u, (\phi, [\,], [\,])) \\ & & \mathit{cut}\, (\phi, x, y +\!\!+ [\phi]) \quad = \quad (x, y) \end{array}$$

Having inverted *prin* as *rebuild*, the inverse of *pinorder* is simply

$$\begin{array}{lll} \mathit{pinorder}^{\circ} & = & \mathit{proj} \cdot \mathit{rebuild} \cdot \mathit{tag} \\ & \mathbf{where} & \mathit{tag}\, (x, y) \quad = \quad (\phi, x, y +\!\!+ [\phi]) \\ & & \mathit{proj}\, (u, (\phi, [\,], [\,])) \quad = \quad u \end{array}$$

where apparently $\mathit{init}^{\circ} = \mathit{proj}$, $\mathit{cut}^{\circ} = \mathit{tag}$, and $\mathit{prin}^{\circ} = \mathit{rebuild}$. The Haskell implementation is shown in Figure 3.4, where we declare a type class *Lifted* to denote the types that have a distinct ϕ.

3.4 Discussion

Most attempts to program inversion were based on the compositional approach, be them procedural [29, 39, 24, 84, 81, 78] or functional [40]. The basic strategy is to promote the converse operator inside with the help of various distributivity laws, such as $(R \cdot S)^{\circ} = S^{\circ} \cdot R^{\circ}$, $(R \cup S)^{\circ} = R^{\circ} \cup S^{\circ}$, and that $(\mu f)^{\circ} = \mu((^{\circ}) \cdot f \cdot (^{\circ}))$, until we reach some primitives whose inverses are either predefined or trivial. In procedural programming where sequencing is ubiquitous, or when the use of plumping functions is implicit like in Section 3.3, this approach gives one the feeling of "running a program backwards", since inverses of sub-components are composed in reverse. The challenging part is when we encounter branches, in such cases we had better somehow decide which branch the result used to come from.

This rather control-oriented view is complemented by a more data-oriented view in [47, 48]. In their paper, Jansson and Jeuring generalised functions to arrows. They then considered polytypic operations on datatypes and ensured that an operation and its inverse carrying things out in reverse order (such as "map from the left" and "map from the right", "traverse from the left branch", and "traverse from the right branch") are always constructed in pairs.

The tricky bits of Section 3.3 is to transform the original program to a form easier to invert. In [81], the same transform was presented in a procedural style. In a procedural

```
data Tree a = Null | Node a (Tree a) (Tree a)
       deriving (Show,Eq)

class Lifted a where phi :: a
instance Lifted Int where phi = -1

unpinorder :: (Eq a, Lifted a) => ([a],[a]) -> Tree a
unpinorder = proj . rebuild . init
 where init (x,y) = (phi, x, y ++ [phi])
       proj (u,(phi, [],[])) = u

rebuild :: Eq a => (a,[a],[a]) -> (Tree a,(a,[a],[a]))
rebuild (b, x, b':y)
  | b == b' = (Null, (b, (x,y)))
rebuild (b, a:x'', y'') =
  let (u, (_, x',y'))  = rebuild (a, x'', y'')
      (v, (_, x,y))    = rebuild (b, x', y')
  in (Node a u v, (b, x,y))
```

Figure 3.4: Haskell code implementing *unpinorder*.

language, the distinction between input and output is not explicit. In our formulation here, however, we have to explicitly copy some inputs to the output to achieve invertability. This is related to the construction of logically reversible circuits [83, 85] where the same action sometimes needs to be done.

The formalisation in [40] is based on functions lifted to sets, while we use relations here. One of the advantages of using relations is that the separation between inverting a function and taking its breadth reveals more structure of the program. In Section 3.2, for instance, *concat* is a fold and its inverse is naturally an unfold. Bringing in breadths too early obscures the symmetry.

In the next chapter, however, we will see a quite different approach to function inversion, where the inverse of a function, even defined as a fold, might be constructed as a fold as well.

Chapter 4

The Converse-of-a-Function Theorem

In the previous chapter, the function $concat_1$, defined as a fold, is inverted as an unfold. Functional programmers are aware that flattening a structure is usually performed by a fold operation. Consequently, building a structure is usually performed by the converse operation, unfold. However, there is no reason why the converse operation should necessarily involve an unfold. The converse-of-a-function theorem, to which this chapter is devoted, gives us conditions under which the inverse of a function can be written as a fold. When the theorem applies, the important thing is not how the function to be inverted was defined, but the properties it satisfies.

In the following sections we will show how this theorem can be applied to derive solutions to several problems, including the breadth-first labelling problem we promised to solve in Chapter 1. A proof of the theorem will then be given. In fact, we will prove a generalised theorem which gives conditions under which a simple relation can be inverted as a hylomorphism. Finally we will demonstrate some applications of the generalised theorem.

4.1 Inverting a Function as a Fold

The converse-of-a-function theorem, introduced in [17, 67], tells us how to write the inverse of a function as a fold. It reads:

Theorem 4.1 (Converse of a function) Given a function $f :: B \to \mathsf{T}A$. If $R :: \mathsf{F}(A, B) \to B$ is surjective and $f \cdot R \subseteq \alpha_{\mathsf{F}} \cdot \mathsf{F}f$, where F is the base functor for T, then $f^{\circ} = (\![R]\!)_{\mathsf{F}}$.

The specialisation of this theorem to functions over lists reads as follows: let $f :: B \to List\ A$ be given. If $base :: B$ and $step :: (A \times B) \to B$ are jointly surjective (meaning that $\{(base, base)\} \cup ran\ step = id_B$) and satisfy

$$f\ base \;=\; [\,]$$

$$f\,(step\,(a,x)) \;=\; a : f\,x$$

then $f^{\circ} = foldr\ step\ base$.

Similarly, to invert a total function f on non-empty lists, Theorem 4.1 states that if $base :: A \to B$ and $step :: (A \times B) \to B$ are jointly surjective (that is, $ran\ base \cup ran\ step = id_B$) and satisfy

$$\begin{aligned} f\,(base\ a) &= [a] \\ f\,(step\,(a,x)) &= a : f\,x \end{aligned}$$

then $f^{\circ} = foldrn\ step\ base$.

We will postpone the proof of Theorem 4.1 to Section 4.6, where in fact a more general result is proved. For now, let us see some of its applications.

4.2 Partitioning a List Revisited

First of all, let us revisit the problem dealt with in Section 3.2: given the function $concat :: List\,(List_1\ A) \to List\ A$, to construct $concat^{\circ}$. Theorem 4.1 says that if we can find a pair of relations $base$ and $step$ such that

$$\begin{aligned} concat\ base &= [\,] \\ concat\,(step\,(a,xs)) &= a : concat\ xs \end{aligned} \tag{4.1}$$

then we have $concat^{\circ} = foldr\ step\ base$.

Notice that the type of $base$ ought to be $List\,(List_1\ A)$. Therefore the only choice of $base$ we have is the empty list, as $[[\,]]$ can only be given the type $List\,(List\ A)$. What about $step$? We start the reasoning from the right-hand side of (4.1):

$$\begin{aligned} & a : concat\ xs \\ = & \quad \{\text{lists}\} \\ & [a] +\!\!+ concat\ xs \\ = & \quad \{\text{since } [a] \text{ and } concat\ xs \text{ are non-empty}\} \\ & cat_1\,([a], concat\ xs) \\ = & \quad \{\text{definition of } concat\} \\ & concat\,([a] : xs) \end{aligned}$$

Therefore, choosing $step\,(a,xs) = [a] : xs$ satisfies (4.1). However, $base$ and $step$ are not jointly surjective, because there is no way for $step$ to return a list of lists whose head is not a singleton list. We therefore consider the following case when the argument to $concat$ in left-hand side of (4.1) is not empty:

$$\begin{aligned} & a : concat\ xs \\ = & \quad \{\text{assumption: } xs \text{ non-empty}\} \\ & a : concat\,(head\ xs : tail\ xs) \end{aligned}$$

```
partitions = foldr step base
   where base = [[]]
         step a xss = [[a] : xs | xs <- xss] ++
                      [(a:x):xs | (x:xs) <- xss]
```

Figure 4.1: Implementing *partitions* by a fold.

$$
\begin{array}{ll}
= & \{\text{definition of } \mathit{concat}\} \\
 & a : (\mathit{head}\ \mathit{xs} \mathbin{+\!\!+} \mathit{concat}\ (\mathit{tail}\ \mathit{xs})) \\
= & \{\text{since } \mathbin{+\!\!+} \text{ is commutive}\} \\
 & (a : \mathit{head}\ \mathit{xs}) \mathbin{+\!\!+} \mathit{concat}\ (\mathit{tail}\ \mathit{xs}) \\
= & \{\text{definition of } \mathit{concat}\} \\
 & \mathit{concat}\ ((a : \mathit{head}\ \mathit{xs}) : \mathit{tail}\ \mathit{xs})
\end{array}
$$

It has just been shown that returning $(a : x) : xs$ is another action *step* may safely perform: taking $\mathit{step}\,(a, x) = [a] : xs \;\Box\; (a : \mathit{head}\ xs) : \mathit{tail}\ xs$ still satisfies (4.1). Furthermore, *base* and *step* are now jointly surjective. We therefore come up with the following definition for concat°:

$$
\begin{array}{lll}
\mathit{concat}^\circ & = & \mathit{foldr}\ \mathit{step}\ \mathit{base} \\
\quad\textbf{where} & \mathit{base} & = [\,] \\
 & \mathit{step}\,(a, xs) & = [a] : xs \;\Box\; (a : \mathit{head}\ xs) : \mathit{tail}\ xs
\end{array}
$$

Define $\mathit{partitions} = \Lambda \mathit{concat}^\circ$ and promote Λ inside, we get the Haskell program in Figure 4.1.

This approach of partitioning a list via a fold is well-known as the engine of many optimisation algorithms, such as that for paragraph formatting [66].

4.3 Building a Tree from Its Depths

Consider the following datatype *Tree A* of tip-valued binary trees:

$$\textbf{data}\ \mathit{Tree}\ A \;=\; \mathit{tip}\ A \mid \mathit{bin}\,(\mathit{Tree}\ A \times \mathit{Tree}\ A)$$

Suppose we are given a list representing the depths of the tips of a tree in left-to-right order. How can we reconstruct (the shape of) the tree from the list? This particular problem arises, for instance, in the final phase of the Hu-Tucker algorithm [43]. For simplicity, we will identify tip values with their depths, as in Figure 4.2. Of course, not every list corresponds to a tree.

We will start with a formal specification of the problem. First of all, the familiar function *flatten*, which takes a tree and returns its tips in left-to-right order, can be written as a fold:

$$
\begin{array}{lll}
\mathit{flatten} & :: & \mathit{Tree}\ A \to \mathit{List}_1\ A \\
\mathit{flatten} & = & \mathit{foldTree}\ \mathit{cat}_1\ \mathit{wrap}
\end{array}
$$

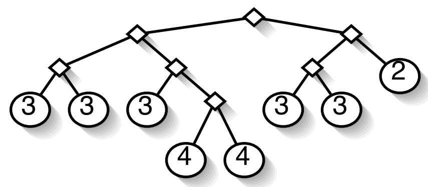

Figure 4.2: A tree whose tips have depths $[3, 3, 3, 4, 4, 3, 3, 2]$

A tree of integers is *well-formed* if one can assign to it a *level*, where the level of a tip is the number at the tip, and the level of a non-tip is defined only if its two subtrees have the same level, in which case it is one less than the levels. The partial function *level* can be defined by:

$$\begin{array}{lcl} \mathit{level} & :: & \mathit{Tree}\,\mathcal{Z} \rightarrow \mathcal{Z} \\ \mathit{level} & = & \mathit{foldtree}\ \mathit{up}\ \mathit{id} \\ & & \quad \mathbf{where}\ \mathit{up}\,(a, b) = \mathbf{if}\ \ a == b\ \mathbf{then}\ a - 1 \end{array}$$

Note that the **if** clause in the definition of *up* does not have an **else** branch. Therefore, *level* is a partial function which only returns a value for a tree when its left and right subtrees are assigned the same level. A tree is *well-formed* if it is in the domain of *level*.

Our problem can thus be specified by

$$\mathit{build} \quad = \quad ((0\ ==) \cdot \mathit{level})? \cdot \mathit{flatten}^{\circ}$$

Now we have got the problem specification. In the following sections we will transform it to a program in two major steps. The first is to use the converse-of-a-function theorem to construct $\mathit{flatten}^{\circ}$ as a relational fold. The second step is then to exploit $((0\ ==)\cdot \mathit{level})?$ to eliminate the non-determinism in the fold.

4.3.1 Building a Tree with a Fold

Our aim is to apply the converse-of-a-function theorem to invert *flatten*. We need a pair of relations $\mathit{one} :: A \rightarrow \mathit{Tree}\ A$ and $\mathit{add} :: (A \times \mathit{Tree}\ A) \rightarrow \mathit{Tree}\ A$ that are jointly surjective and satisfy

$$\begin{array}{rcl} \mathit{flatten}\,(\mathit{one}\ a) & = & [a] \\ \mathit{flatten}\,(\mathit{add}\,(a, u)) & = & a : \mathit{flatten}\ u \end{array}$$

Look at the second equation. It says that if we have a tree u which flattens to some list x, the relation *add* must be able to create a new tree v out of a and u such that v flattens to $a : x$. One way to do that is illustrated in Fig. 4.3. We divide the left spine of u in two parts, move down the lower part for one level, and attach a to the end.

To facilitate this operation, we introduce an alternative *spine representation*. A tree is represented by the list of subtrees along its left spine, plus the left-most tip. The function

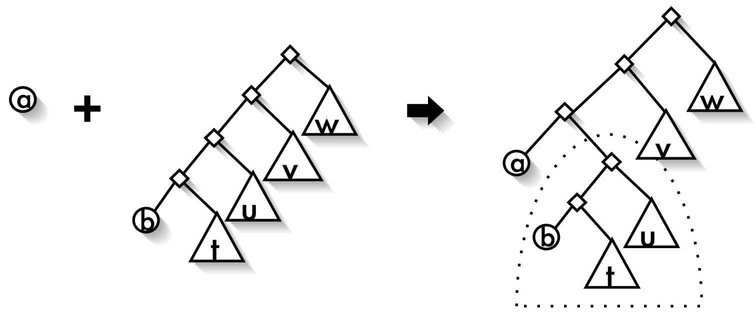

Figure 4.3: Adding a new node to a tree

roll converts a spine back into a single tree, and is in fact an isomorphism between *Spine A* and *Tree A*.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Spine}\ A & = & A \times \mathit{List}\,(\mathit{Tree}\ A) \\
\\
\mathit{roll} & :: & \mathit{Spine}\ A \to \mathit{Tree}\ A \\
\mathit{roll}(a, x) & = & \mathit{foldl}\ \mathit{bin}\,(\mathit{tip}\ a)\ x
\end{array}
$$

The advantage of this representation is that we can trace the spine upward from the left-most leaf, rather than downwards from the root. As we will see in the end of the next section, this is necessary for an efficient implementation.

The function $\mathit{flatten} \cdot \mathit{roll}$ flattens a spine tree. Our task now is to invert it as a fold. We need a pair of relations $\mathit{one} :: A \to \mathit{Spine}\ A$ and $\mathit{add} :: (A \times \mathit{Spine}\ A) \to \mathit{Spine}\ A$ satisfying

$$\mathit{flatten}\,(\mathit{roll}\,(\mathit{one}\ a)) \ = \ [a] \tag{4.2}$$

$$\mathit{flatten}\,(\mathit{roll}\,(\mathit{add}\ a\,(b, \mathit{us}))) \ = \ a : \mathit{flatten}\,(\mathit{roll}\,(b, \mathit{us})) \tag{4.3}$$

We claim that the following definition for *one* and *add* does the job:

$$
\begin{array}{lcl}
\mathit{one}\ a & = & (a, [\,]) \\
\mathit{add}\,(a, (b, \mathit{us})) & = & (a, \mathit{roll}\,(b, \mathit{vs}) : \mathit{ws}) \\
\quad \textbf{where}\ \ \mathit{vs} + \!\!\! + \ \mathit{ws} & = & \mathit{us}
\end{array}
$$

The non-deterministic pattern in the definition of *add*, dividing the list *xs* into two parts, indicates that *add* is not a function. The relations *one* and *add* are jointly surjective because *roll*, being an isomorphism, is surjective; thus, given any spine tree (a, ws), either *ws* is empty, in which case it is covered by *one a*, or there always exists a spine tree (b, vs) such that it rolls into the head of *ws*, in which case (a, ws) would be one of the results of $\mathit{add}\ a\,(b, \mathit{vs} + \!\!\! + \ \mathit{tail}\ \mathit{ws})$.

It is clear that the function *one* satisfies (4.2). To show that *add* satisfies (4.3), we will need the following fact, whose proof is left to Appendix A:

$$\mathit{flatten}(\mathit{roll}\,(a, \mathit{us})) \ = \ a : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ \mathit{us}) \tag{4.4}$$

Now we will show that *add* satisfies (4.3):

$$
\begin{array}{ll}
& a : \mathit{flatten}(\mathit{roll}\ (b, vs \mathbin{+\!\!+} ws)) \\
= & \quad \{(4.4)\} \\
& a : b : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ (vs \mathbin{+\!\!+} ws)) \\
= & \quad \{\mathit{concat} \text{ and } \mathit{map} \text{ distributes over } \mathbin{+\!\!+}\ \} \\
& a : b : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ vs) \mathbin{+\!\!+} \mathit{concat}(\mathit{map}\ \mathit{flatten}\ ws) \\
= & \quad \{(4.4)\} \\
& a : \mathit{flatten}(\mathit{roll}\ (b, vs)) \mathbin{+\!\!+} \mathit{concat}(\mathit{map}\ \mathit{flatten}\ ws) \\
= & \quad \{\text{definition of } \mathit{concat} \text{ and } \mathit{map}\} \\
& a : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ (\mathit{roll}(b, vs) : ws)) \\
= & \quad \{(4.4)\} \\
& \mathit{flatten}\ (\mathit{roll}\ (a, \mathit{roll}(b, vs) : ws))
\end{array}
$$

Thus $(\mathit{flatten} \cdot \mathit{roll})^{\circ} = \mathit{foldrn}\ \mathit{add}\ \mathit{one}$ by Theorem 4.1.

4.3.2 The Derivation

Having inverted $\mathit{flatten} \cdot \mathit{roll}$, we can start the derivation:

$$
\begin{array}{ll}
& \mathit{build} \\
= & \quad \{\text{definition}\} \\
& ((0 ==) \cdot \mathit{level})? \cdot \mathit{flatten}^{\circ} \\
= & \quad \{\mathit{roll} \text{ is an isomorphism}\} \\
& ((0 ==) \cdot \mathit{level})? \cdot (\mathit{flatten} \cdot \mathit{roll} \cdot \mathit{roll}^{\circ})^{\circ} \\
= & \quad \{\text{converse is contravariant}\} \\
& ((0 ==) \cdot \mathit{level})? \cdot \mathit{roll} \cdot (\mathit{flatten} \cdot \mathit{roll})^{\circ} \\
= & \quad \{\text{inverting } \mathit{flatten} \cdot \mathit{roll} \text{ as in the last section}\} \\
& ((0 ==) \cdot \mathit{level})? \cdot \mathit{roll} \cdot \mathit{foldrn}\ \mathit{add}\ \mathit{one} \\
= & \quad \{\text{since } p? \cdot f = f \cdot (p \cdot f)?, \text{ let } \mathit{wellform} = (0 ==) \cdot \mathit{level} \cdot \mathit{roll}\} \\
& \mathit{roll} \cdot \mathit{wellform}? \cdot \mathit{foldrn}\ \mathit{add}\ \mathit{one}
\end{array}
$$

Whereas $(0 ==) \cdot \mathit{level}$ checks whether a tree is well-formed, *wellform* is its counterpart defined on spine trees. Intuitively, a spine tree (b, us) is well-formed if either us is empty and $b = 0$, or all the trees in us has a level number, the leftmost one being b, the next one being $b - 1$,... and the rightmost one being 1.

As $\mathit{roll} \cdot \mathit{wellform}?$ is a partial function, it can be easily implemented in Haskell. However, *add* is still a relation. If we can fuse $\mathit{wellform}?$ into the fold and thereby refine *add* to a partial function, the whole expression will be implementable.

However, $\mathit{wellform}?$ is a rather strong condition to enforce. It is not possible to maintain this invariant within the fold before and after each applications of *add*. It is time to take the second inventive step: to invent a weaker condition. The predicate *decform* holds for a spine tree (b, us) if the level number of the first tree in *us* is at most b and the trees in *us* have strictly decreasing level numbers:

$$\begin{array}{lcl} \mathit{decform}\,(b, \mathit{us}) & = & \mathit{leading}\,(b, \mathit{us}) \wedge \mathit{decreasing}\,(\mathit{map\ level\ us}) \\ \mathit{leading}\,(b, \mathit{us}) & = & \mathit{null\ us} \vee \mathit{level}(\mathit{head\ us}) \leq b \end{array}$$

Note that the application of *level* to all the trees in *xs* implicitly states the requirement that all the trees have level numbers.

The predicate *decform* is weaker than *wellform*. We can thus derive:

$$\begin{array}{cl} & \mathit{roll} \cdot \mathit{wellform}? \cdot \mathit{foldrn\ add\ one} \\ = & \quad \{(2.1)\} \\ & \mathit{roll} \cdot \mathit{wellform}? \cdot \mathit{decform}? \cdot \mathit{foldrn\ add\ one} \\ = & \quad \{\text{fold fusion, see below}\} \\ & \mathit{roll} \cdot \mathit{wellform}? \cdot \mathit{foldrn\ add'\ one} \end{array}$$

The equality established by fold fusion in the last step ensures that no result is lost from the refinement. Fortunately, it can be shown that the following fusion condition is valid:

$$\mathit{decform}? \cdot \mathit{add} \quad = \quad \mathit{add'} \cdot (\mathit{id} \times \mathit{decform}?)$$

where add' is defined by rolling the given spine tree up to the point when the two left-most trees do not have the same level number:

$$\begin{array}{lcll} \mathit{add'}\,(a, (b, \mathit{us})) & = & \mathit{leading}?\,(a, \mathit{decRoll}\,(\mathit{tip}\ b)\ \mathit{us}) & \\ \mathit{decRoll}\ u\,[\,] & = & [u] & \\ \mathit{decRoll}\ u\,(v : \mathit{ws}) & | & (\mathit{level}\ u == \mathit{level}\ v) & = \mathit{decRoll}\,(\mathit{bin}\ u\ v)\ \mathit{ws} \\ & | & \mathit{otherwise} & = u : v : \mathit{ws} \end{array}$$

The code is shown in Fig. 4.4. We refine the data structure to avoid recomputing *level* by defining type *SpineI* and maintain the invariant that $\mathit{level}\, x = n$ for all pairs (x, n) along the spine. Constructors *tip* and *bin* are lifted accordingly. The function *rollwf* implements $\mathit{roll} \cdot \mathit{wellform}?$. The partial function *bin* performs a check each time two trees are joined. This algorithm is linear in the number of nodes in the tree, as each call to *join* either stops or builds a new node.

Some reader might recognise we are actually performed a specific kind of parsing. Indeed, one of the early application of the converse-of-a-function was to derive the Floyd's algorithm for precedence parsing [67]. We will discuss more about that in Section 8.3.

4.4 Breadth-First Labelling

We are now equipped with sufficient tools to solve the problem mentioned as a teaser in Section 1.1: to breadth-first label a tree with respect to a given list!

To remind the reader of the problem, we recall the example diagram in Figure 4.5, where a tree with 13 nodes is labelled with the infinite list [1..]. While everybody knows

```
data Tree a = Tip a | Bin (Tree a) (Tree a) deriving Show
type SpineI = (Int, [(Tree Int, Int)])

build :: [Int] -> Tree Int
build = rollwf . foldrn add' one

one a = (a,[])

add' a (b,us) | leading (a,ws) = (a,ws)
  where ws = decRoll (tip b) us
decRoll u [] = [u]
decRoll u (v:ws) | level u == level v = decRoll (bin u v) ws
                 | otherwise          = u:v:ws
leading (a,us) = level (head us) <= a

tip a = (Tip a, a)
bin (u,m) (v,n) | m == n = (Bin u v, m-1)
level = snd

rollwf :: SpineI -> Tree Int
rollwf (b,us) = pick (foldl bin (tip b) us)
  where pick (u,0) = u

foldrn f g [x] = g x
foldrn f g (a:x) = f a (foldrn f g x)
```

Figure 4.4: Code for rebuilding a tree from the depths of its tips

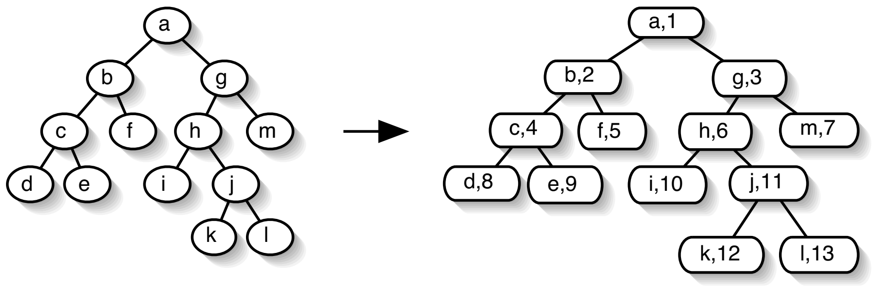

Figure 4.5: Breadth-first labelling a tree on the left with [1..].

how to do breadth-first traversal, efficient breadth-first labelling is not so widely understood. Jones and Gibbons [37] proposed a neat solution to this problem, based on a clever use of cyclic data structures. The problem was revisited by Okasaki [71] in his talk in International Conference on Functional Programming 2000, where he challenged all the audience to come up with a good algorithm that does not exploit laziness. We are going to show how Okasaki's algorithm can be derived using the converse-of-a-function theorem.

Let us go through again the specification in finer detail. Consider the data structure of internally and externally labelled binary trees:

$$\mathbf{data}\; \mathit{Tree}\; A \;=\; \mathit{tip}\; A \mid \mathit{bin}\,(A \times \mathit{Tree}\; A \times \mathit{Tree}\; A)$$

The queue-based algorithm for breadth-first traversal is well-known:

$$\begin{array}{lcl} \mathit{bft} & :: & \mathit{Tree}\; A \rightarrow \mathit{List}\; A \\ \mathit{bft}\; u & = & \mathit{bftF}\,[u] \end{array}$$

$$\mathbf{type}\; \mathit{Forest}\; A \;=\; \mathit{List}\,(\mathit{Tree}\; A)$$

$$\begin{array}{lcl} \mathit{bftF} & :: & \mathit{Forest}\; A \rightarrow \mathit{List}\; A \\ \mathit{bftF}\,[\,] & = & [\,] \\ \mathit{bftF}\,(\mathit{tip}\; a : \mathit{us}) & = & a : \mathit{bftF}\; \mathit{us} \\ \mathit{bftF}\,(\mathit{bin}\,(a, u, v) : \mathit{us}) & = & a : \mathit{bftF}\,(\mathit{us} \mathbin{+\!\!+} [u, v]) \end{array}$$

To perform the labelling, we use the following partial function *zipTree*:

$$\begin{array}{lcl} \mathit{zipTree} & :: & \mathit{Tree}\; A \rightarrow \mathit{Tree}\; B \rightarrow \mathit{Tree}\,(A \times B) \\ \mathit{zipTree}\,(\mathit{tip}\; a)\,(\mathit{tip}\; b) & = & \mathit{tip}\,(a, b) \\ \mathit{zipTree}\,(\mathit{bin}\,(a, x, y))\,(\mathit{bin}\,(b, u, v)) & = & \mathit{bin}\,((a, b), \mathit{zipTree}\; x\; u, \mathit{zipTree}\; y\; v) \end{array}$$

Breadth-first labelling of a tree u can then be seen as zipping u with another tree v, in which the breadth-first traversal of v is a prefix of the given list x:

$$\begin{array}{lcl} \mathit{bfl} & :: & \mathit{List}\; A \rightarrow \mathit{Tree}\; B \rightarrow \mathit{Tree}\,(A \times B) \\ \mathit{bfl}\; x\; u & = & \mathit{zipTree}\; v\; u \\ & & \quad \mathbf{where}\;\; (\mathit{bft}\; v) \mathbin{+\!\!+} y \;=\; x \end{array}$$

Equivalently,

$$\begin{aligned} \mathit{bfl}\ x\ u &= \mathit{zipTree}\ ((\mathit{bft}^\circ \cdot \mathit{prefix})\ x)\ u \\ &= (\mathit{zipTree} \cdot \mathit{bft}^\circ \cdot \mathit{prefix})\ x\ u \end{aligned}$$

This completes the specification. The relation *prefix* non-deterministically maps a list to one of its finite prefixes. The prefix is then passed to bft°, yet again being non-deterministically mapped to a tree whose breadth-first traversal equals the chosen prefix. It is important that *zipTree* is a partial function which yields a value only when the given two trees are of exactly the same shape. Therefore, the tree composed by $\mathit{bft}^\circ \cdot \mathit{prefix}$ can be zipped with the input tree only if it is of the correct size and shape. The partial function *zipTree* plays the role of a filter.

Since breadth-first traversal is an algorithm more naturally defined in terms of queues of trees (or forests) rather than of a single tree, it is reasonable to try to invert *bftF* rather than *bft*. The problem can be rephrased in terms of *bftF*:

$$\mathit{bfl}\ x\ u \;=\; \mathit{wrap}^\circ\ ((\mathit{zipForest} \cdot \mathit{bftF}^\circ \cdot \mathit{prefix})\ x\ [u])$$

Here $\mathit{zipForest} :: \mathit{Forest}\ A \to \mathit{Forest}\ B \to \mathit{Forest}\ (A \times B)$ is a simple extension of *zipTree* to forests, which, like *zipTree*, is a partial function:

$$\begin{aligned} \mathit{zipForest}\ [\,]\ [\,] &= [\,] \\ \mathit{zipForest}\ (u : us)\ (v : vs) &= \mathit{zipTree}\ u\ v : \mathit{zipForest}\ us\ vs \end{aligned}$$

Once the decision to focus on *bftF* is made, the rest is mechanical. To invert *bftF*, we are to find *base* and *step* such that

$$\begin{aligned} \mathit{bftF}\ \mathit{base} &= [\,] \\ \mathit{bftF}\ (\mathit{step}\ a\ us) &= a : \mathit{bftF}\ us \end{aligned}$$

The value of *base* can only be $[\,]$. The derivation for *step* is not too difficult either. We start with the general case which does not assume any structure in *us*:

$$\begin{aligned} & a : \mathit{bftF}\ us \\ = & \quad \{\text{definition of } \mathit{bftF}\} \\ & \mathit{bftF}\ (\mathit{tip}\ a : us) \end{aligned}$$

Therefore *step a us* might contain $(\mathit{tip}\ a : us)$ as one of the possible values. But this choice alone does not make *step* jointly surjective with $[\,]$, since it cannot generate a forest with a non-tip tree as its head. We therefore consider the case when *us* contains more than two trees:

$$\begin{aligned} & a : \mathit{bftF}(us +\!\!+ [u, v]) \\ = & \quad \{\text{definition of } \mathit{bftF}\} \\ & \mathit{bftF}\ (\mathit{bin}\ (a, u, v) : us) \end{aligned}$$

Therefore we define *step* to be:

$$\begin{aligned} &\mathit{step} &&:: (A \times \mathit{Forest}\ A) \to \mathit{Forest}\ A \\ &\mathit{step}\ (a, us) &&= (\mathit{tip}\ a : us)\ \Box\ (\mathit{bin}\ (a, u, v) : ws) \\ &\quad \textbf{where} && ws +\!\!+ [u, v] = us \end{aligned}$$

Since a forest either begins with a tip tree, begins with a non-tip tree, or is empty, *step* is jointly surjective with $[\,]$. The converse of bftF is thus constructed as $\mathit{bftF}^\circ = \mathit{foldr}\ \mathit{step}\ [\,]$.

Now that $\mathit{bftF}^\circ :: \mathit{List}\ A \rightarrow \mathit{Forest}\ A$ is a fold, we can fuse *zipForest* and bftF° by fold fusion:

$$
\begin{array}{lcl}
\mathit{zipForest} \cdot \mathit{bftF}^\circ & :: & \mathit{List}\ A \rightarrow \mathit{Forest}\ B \rightarrow \mathit{Forest}\,(A \times B) \\
\mathit{zipForest} \cdot \mathit{bftF}^\circ & = & \mathit{foldr}\ \mathit{rzip}\ \mathit{stop}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stop} & :: & \mathit{Forest}\ B \rightarrow \mathit{Forest}\,(A \times B) \\
\mathit{stop}\,[\,] & = & [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{rzip}\ ::\ (A \times (\mathit{Forest}\ B \rightarrow \mathit{Forest}\,(A \times B))) \rightarrow \mathit{Forest}\ B \rightarrow \mathit{Forest}\,(A \times B) \\
\begin{array}{lcl}
\mathit{rzip}\,(a, f)\,(\mathit{tip}\ b : \mathit{us}) & = & \mathit{tip}\,(a, b) : f\ \mathit{us} \\
\mathit{rzip}\,(a, f)\,(\mathit{bin}\,(b, u, v) : \mathit{us}) & = & \mathit{bin}\,((a, b), x, y) : \mathit{xs}
\end{array} \\
\qquad \textbf{where}\quad \mathit{xs} \mathbin{+\!\!+} [x, y] \;=\; f\,(\mathit{us} \mathbin{+\!\!+} [u, v])
\end{array}
$$

Consider $(\mathit{zipForest} \cdot \mathit{bftF}^\circ)\,x$ where x is a list of labels. Constructors building x are replaced by *rzip* and *stop*, yielding a relation mapping an unlabelled forest to a labelled forest. A pattern matching error will be invoked by *stop* if x is too short, and by *rzip* if x is too long. Applying fold fusion again to fuse $\mathit{zipForest} \cdot \mathit{bftF}^\circ$ with *prefix* in effect adds another case for *rzip*, that is, $\mathit{rzip}\,(a, f)\,[\,] = [\,]$, which cuts the list of labels when the forest is consumed earlier than the list. Still, the list of labels cannot be too short.

The resulting code is shown in Fig. 4.6. It can be made linear if we use an implementation of deques supporting constant-time addition and deletion [25, 70] for both the input and output of *rzip*. For clarity, we will just leave it as it is. It is nothing more than an adaption of Okasaki's algorithm in [71] to lists. In his paper, Okasaki raised the question why most people did not come up with this algorithm but instead appealed to more complicated approaches. Our answer is because they did not know the converse-of-a-function theorem.

4.5 Rebuilding a Tree from its Traversals Revisited

As the third example, we come back to the problem of rebuilding a tree from its inorder and preorder traversals. Let us recall the datatype definition for internally labelled binary trees:

$$
\textbf{data}\ \mathit{Tree}\ A \;=\; \mathit{null} \mid \mathit{node}\,(A \times (\mathit{Tree}\ A \times \mathit{Tree}\ A))
$$

Inorder and preorder traversal on the trees are defined by

$$
\begin{array}{l}
\mathit{inorder} \;=\; \mathit{foldTree}\ \mathit{inf}\ [\,] \\
\qquad \textbf{where}\quad \mathit{inf}\,(a, (x, y)) \;=\; x \mathbin{+\!\!+} [a] \mathbin{+\!\!+} y \\
\mathit{preorder} \;=\; \mathit{foldTree}\ \mathit{pre}\ [\,] \\
\qquad \textbf{where}\quad \mathit{pre}\,(a, (x, y)) \;=\; [a] \mathbin{+\!\!+} x \mathbin{+\!\!+} y
\end{array}
$$

Finally, the predicate *distinct* yields true for a tree whose node values are all distinct. The aim is to construct $\mathit{distinct}? \cdot \langle \mathit{preorder}, \mathit{inorder} \rangle^\circ$.

```
data Tree a = Tip a | Bin a (Tree a) (Tree a) deriving Show

bfl :: [a] -> Tree b -> Tree (a,b)
bfl xs = unwrap . foldr rzip stop xs . wrap
 where stop [] = []
       rzip a f [] = []
       rzip a f (Tip b:us) = Tip (a,b) : f us
       rzip a f (Bin b u v :us) = Bin (a,b) x y : ys'
          where ys = f (us ++ [u,v])
                (ys',x,y) = (init (init ys), last (init ys), last ys)

wrap a = [a]
unwrap [a] = a
```

Figure 4.6: Code for breadth-first labelling

We will try to follow the same strategy that worked in the previous sections: to construct the converse of a function as a relational fold, and then impose some constraints on the fold. However, due to its type, $\langle \mathit{preorder}, \mathit{inorder} \rangle^\circ$ apparently cannot be a fold on a recursive datatype. The first step, then, is to transform $\langle \mathit{preorder}, \mathit{inorder} \rangle^\circ$ to a filter after the converse of some function.

To do so, notice that currying can be specified as:

$$\begin{array}{lcl} \mathit{curry} & :: & ((A \times B) \rightarrow C) \rightarrow A \rightarrow B \rightarrow C \\ \mathit{curry}\; S\; a & = & S \cdot \langle \mathit{const}\; a, \mathit{id} \rangle \end{array}$$

where $S :: (A \times B) \rightarrow C$ and $a :: A$. In words, *curry* remembers the constant a and pair it with whatever input before feeding the pair to S. Furthermore, when S is the converse of a fork, the following lemma allows us to convert it to a filter after a converse of a function, which is the form we want:

Lemma 4.2 $\langle R, f \rangle^\circ \cdot \langle \mathit{const}\; a, \mathit{id} \rangle \;\; = \;\; ((a \mathrel{==}) \cdot R)? \cdot f^\circ$

Writing down the types helps us to get an intuition of the lemma. Assume that a has type A. Let $R :: C \rightarrow A$ and $f :: C \rightarrow B$. The term $\langle R, f \rangle^\circ$ thus has type $(A \times B) \rightarrow C$. The left-hand side takes a value b of type B, constructs the pair (a, b), and reduces it to a value $c :: C$ such that that $(c, a) \in R$ and $(c, b) \in f$. The right-hand side does the same by mapping b to an arbitrary c through f, and taking only those c satisfying $(c, a) \in R$. A proof of the lemma is given in Appendix A.

Now let us substitute *preorder* for R and *inorder* for f, we get:

$$\mathit{curry}\; \langle \mathit{preorder}, \mathit{inorder} \rangle^\circ\; x \;\; = \;\; ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot \mathit{inorder}^\circ$$

Putting it the other way round, if we define *rebuild* to be

$$\mathit{rebuild}\; x \;\; = \;\; ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot \mathit{inorder}^\circ$$

we have $\langle preorder, inorder\rangle^\circ = uncurry\, rebuild$. Recall that our aim is to construct $distinct? \cdot \langle preorder, inorder\rangle^\circ$. The aim now is thus to derive *rebuild*.

The relation $inorder^\circ$ constructs all trees whose inorder traversal meet a given list. The coreflexive $((x =\!=) \cdot preorder)?$ then picks the one whose preorder traversal is the list x. The derivation therefore again proceeds in two parts: to invert *inorder*, and to impose a constrain on the constructed fold such that it only generates the tree we want. Furthermore, the predicate *distinct* implies that x must not contain duplicated elements; it is thus safe to assume so in the derivation of *rebuild*.

4.5.1 Unflattening an Internally Labelled Binary Tree

In this section we aim to construct *inorder* as a fold. To do so, it is also helpful to switch to a spine representation. The following type *Spine A* represents the spine of an internally labelled binary tree:

$$\mathbf{type}\; Spine\; A \;=\; List\,(A \times Tree\; A)$$

The conversion from a spine tree to the ordinary representation can be performed by the function *roll* defined below:

$$\begin{array}{lll} roll & :: & Spine\; A \rightarrow Tree\; A \\ roll & = & foldl\; join\; null \\ & & \mathbf{where}\quad join\,(u,(a,v)) \;=\; node\,(a,(u,v)) \end{array}$$

The converse-of-a-function theorem says that $inorder^\circ = foldr\; add\; zero$ if we can find *add* and *zero* satisfying:

$$\begin{aligned} inorder\,(roll\; zero) &= [\,] \\ inorder\,(roll\,(add\,(a, us))) &= a : inorder\,(roll\; us) \end{aligned} \tag{4.5}$$

An easy choice for *zero* would be $[\,]$. As for *add*, we claim that the following definition satisfies (4.5):

$$\begin{array}{lll} add & :: & (A \times Spine\; A) \rightarrow Spine\; A \\ add\,(a, us) & = & (a, roll\; vs) : ws \\ & & \mathbf{where}\; vs \mathbin{+\!\!+} ws \;=\; us \end{array}$$

Figure 4.7 illustrates the idea. The tree on the left-hand side is represented by the list

$$[(b,t),(c,u),(d,v),(e,w)]$$

One of the possible ways to extended the tree with a new node a is to cut the list in the middle, yielding:

$$[(a, roll\,[(b,t),(c,u)]),(d,v),(e,w)]$$

It is shown in the figure on the right-hand side. The proof that the definition of *add* above does satisfy (4.5) is similar to that in Section 4.3.1. We will also need a property distributing *inorder* into the subtrees on the spine:

$$inorder \cdot roll \;=\; concat \cdot map\,(cons \cdot (id \times inorder)) \tag{4.6}$$

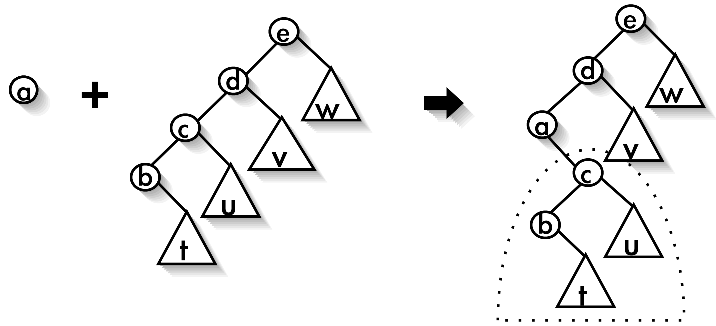

Figure 4.7: Spine representation for internally labelled trees.

The proof goes:

$$
\begin{array}{ll}
 & a : \mathit{inorder}\,(\mathit{roll}\,(\mathit{vs} \mathbin{+\!\!+} \mathit{ws})) \\
= & \quad \{(4.6)\} \\
 & a : \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,(\mathit{vs} \mathbin{+\!\!+} \mathit{ws})) \\
= & \quad \{\text{since } \mathit{concat} \text{ and } \mathit{map} \text{ distributes over } \mathbin{+\!\!+}\} \\
 & a : \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,\mathit{vs}) \mathbin{+\!\!+} \\
 & \quad \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,\mathit{ws}) \\
= & \quad \{(4.6)\} \\
 & a : \mathit{inorder}\,(\mathit{roll}\,\mathit{vs}) \mathbin{+\!\!+} \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,\mathit{ws}) \\
= & \quad \{\text{definition of } \mathit{concat} \text{ and } \mathit{map}\} \\
 & \mathit{concat}\,(\mathit{map}\,(\mathit{cons} \cdot (\mathit{id} \times \mathit{inorder}))\,((a, \mathit{roll}\,\mathit{vs}) : \mathit{ws})) \\
= & \quad \{(4.6)\} \\
 & \mathit{inorder}\,(\mathit{roll}\,((a, \mathit{roll}\,\mathit{vs}) : \mathit{ws}))
\end{array}
$$

It is also not difficult to see that $[\,]$ and *add* are jointly surjective, since if a spine tree is not $[\,]$, it must be a result of adding its left-most element on the spine to some tree. We therefore conclude that $\mathit{inorder}^{\circ} = \mathit{foldr}\ \mathit{add}\ \mathit{null}$.

4.5.2 Enforcing a Preorder

Now recall our the specification of *rebuild*

$$\mathit{rebuild}\ x \quad = \quad ((x \mathbin{==}) \cdot \mathit{preorder})? \cdot \mathit{inorder}^{\circ}$$

In the last section we have inverted *inorder* as a relational fold and switched to a spine representation, yielding:

$$\mathit{rebuild}\ x \quad = \quad \mathit{roll} \cdot (\mathit{hasPreorder}\ x)? \cdot \mathit{foldr}\ \mathit{add}\ [\,]$$

where $hasPreorder\ x = (x\ {==}) \cdot preorder \cdot roll$. The next step is to fuse some constraints into the fold to eliminate its non-determinism. Still, *hasPreorder* is too strong an invariant to enforce. Can we again invent a weaker alternative that can be fused into the fold?

Define *preorderF* to be the preorder traversal of forests:

$$preorderF \quad = \quad concat \cdot map\ preorder$$

Look at Figure 4.7 again. The preorder traversal of the tree on the left-hand side is

$$[e, d, c, b] \mathbin{+\!\!+} preorderF\ [t, u, v, w]$$

that is, to go down along the left spine, then traverse through the subtrees upwards. In general, given a spine tree us, its preorder traversal is

$$reverse\ (map\ fst\ us) \mathbin{+\!\!+} preorderF\ (map\ snd\ us)$$

We will call the part before $\mathbin{+\!\!+}$ the *prefix* and that after $\mathbin{+\!\!+}$ the *suffix* of the traversal. Now look at the tree on the right-hand side. Its preorder traversal is

$$[e, d, a, c, b] \mathbin{+\!\!+} preorderF\ [t, u, v, w]$$

It is not difficult to see that when we add a node a to a spine tree us, the suffix of its preorder traversal does not change. The new node a is always inserted to the prefix.

With this insight, we split *hasPreorder* into two parts:

$$\begin{array}{lcl} hasPreorder & :: & List\ A \rightarrow Spine\ A \rightarrow Bool \\ hasPreorder\ x\ us & = & prefixOk\ x\ us \wedge suffixOk\ x\ us \\ suffixOk\ x\ us & = & preorderF\ (map\ snd\ us)\ \mathbf{isSuffixOf}\ x \\ prefixOk\ x\ us & = & reverse\ (map\ fst\ us)\ {==}\ (x \ominus preorderF\ (map\ snd\ us)) \end{array}$$

where $x \ominus y$ removes y from the tail of x and is defined by:

$$x \ominus y \quad = \quad z \quad \mathbf{where}\ z \mathbin{+\!\!+} y = x$$

The expression x **isSuffixOf** y yields true if x is a suffix of y. The use of boldface font here indicates that it is an infix operator (and binds looser than function application). The plan is to fuse only $suffixOk\ x$ into the fold while leaving $prefixOk\ x$ outside.

There is a slight problem, however. The invariant $suffixOk\ x$ does not prevent the fold from generating, say, a leftist tree with all *null* along the spine, since the empty list is indeed a suffix of any list. Such a tree may be bound to be rejected later. Look again at the right-hand side of Figure 4.7. Assume we know that the preorder traversal of the tree we want is $x = [..\ d, c, b] \mathbin{+\!\!+} preorderF\ [t, u, v, w]$. The tree in the right-hand side of Figure 4.7, although satisfying $suffixOk\ x$, is bound to be wrong because d is the next immediate symbol but a now stands in the way between d and c, and there is no way to change the order afterwards. Thus when we find a proper location to insert a new node, we shall be more aggressive and consume as much suffix of x as possible. The following predicate *lookahead* x ensures that in the constructed tree, the next immediate symbol in x will be consumed:

$$\begin{array}{lcl} \mathit{lookahead} & :: & \mathit{List}\ A \rightarrow \mathit{Spine}\ A \rightarrow \mathit{Bool} \\ \mathit{lookahead}\ x\ us & = & \mathit{length}\ us \leq 1\ \vee\ (\mathit{map}\ \mathit{fst}\ us)\ !!\ 1 \neq \mathit{last}\ x' \\ \quad \textbf{where}\ \ x' & = & x \ominus \mathit{preorderF}\ (\mathit{map}\ \mathit{snd}\ us) \end{array}$$

Apparently *lookahead* x is weaker, and thus can be conjuncted with *hasPreorder* x without changing it. We will use both *suffixOk* x and *lookahead* x as our invariant. Define

$$\mathit{ok}\ x\ us \quad = \quad \mathit{suffixOk}\ x\ us \wedge \mathit{lookhead}\ x\ us$$

The derivation goes:

$$\begin{array}{cl} & \mathit{rebuild} \\ = & \quad \{\text{definition}\} \\ & ((x \mathrel{==}) \cdot \mathit{preorder})? \cdot \mathit{inorder}^{\circ} \\ = & \quad \{\text{inverting } \mathit{inorder} \text{ and moving } \mathit{roll} \text{ to the left}\} \\ & \mathit{roll} \cdot (\mathit{hasPreorder}\ x)? \cdot \mathit{foldr}\ \mathit{add}\ [\,] \\ = & \quad \{\text{since } \mathit{hasPreorder}\ x\ us = \mathit{prefixOk}\ x\ us \wedge \mathit{ok}\ x\ us\} \\ & \mathit{roll} \cdot (\mathit{prefixOk}\ x)? \cdot (\mathit{ok}\ x)? \cdot \mathit{foldr}\ \mathit{add}\ [\,] \\ = & \quad \{\text{fold fusion, assume } \mathit{nodup}\ x\} \\ & \mathit{roll} \cdot (\mathit{prefixOk}\ x)? \cdot \mathit{foldr}\ (\mathit{add}'\ x)\ [\,] \end{array}$$

To justify the fusion step, it can be shown that if x contains no duplicated elements, the following fusion condition holds:

$$(\mathit{ok}\ x)?\ (\mathit{add}\ (a, us)) \quad = \quad \mathit{add}'\ x\ (a, (\mathit{ok}\ x)?\ us)$$

where add' is defined by:

$$\begin{array}{lcl} \mathit{add}' & :: & \mathit{List}\ A \rightarrow (A \times \mathit{Spine}\ A) \rightarrow \mathit{Spine}\ A \\ \mathit{add}'\ x\ (a, us) & = & \mathit{up}\ a\ \mathit{null}\ (us, x \ominus \mathit{preorderF}\ (\mathit{map}\ \mathit{snd}\ us)) \end{array}$$

$$\begin{array}{llcl} \multicolumn{4}{l}{\mathit{up}\ ::\ A \rightarrow \mathit{Tree}\ A \rightarrow (\mathit{Spine}\ A \times \mathit{List}\ A) \rightarrow \mathit{Spine}\ A} \\ \mathit{up}\ a\ v\ ([\,], x) & & = & [(a, v)] \\ \mathit{up}\ a\ v\ ((b, u) : us, x \mathbin{+\!\!+} [b']) & \mid\ b \mathrel{==} b' & = & \mathit{up}\ a\ (\mathit{node}\ (b, (v, u)))\ (us, x) \\ & \mid\ \mathit{otherwise} & = & (a, v) : (b, u) : us \end{array}$$

In words, the function *up* traces the left spine upwards and consumes the values on the spine if they match the tail of x. It tries to roll as much as possible before adding a to the end of the spine.

As a final optimisation, we can avoid computing $x \ominus \mathit{preorderF}\ (\mathit{map}\ \mathit{snd}\ us)$ from scratch each time by applying a tupling transformation, having the fold returning a pair. The Haskell implementation is shown in Figure 4.8. The fold in *rebuild* returns a pair, the first component being a tree and the second component being a list representing

```
data Tree a = Null | Node a (Tree a) (Tree a) deriving (Show,Eq)

rebuild :: Eq a => [a] -> [a] -> Tree a
rebuild x = rollpf . foldr add' ([],reverse x)
  where add' a (us,x) = up a Null (us,x)
        up a v ([],x) = ([(a,v)],x)
        up a v ((b,u):us, b':x)
           | b == b' = up a (Node b v u) (us, x)
           | otherwise = ((a,v):(b,u):us, b':x)

rollpf :: Eq a => ([(a,Tree a)],[a]) -> Tree a
rollpf (us,x) = rp Null (us,x)
 where rp v ([],[]) = v
       rp v ((b,u):us, b':x)
          | b == b' = rp (Node b v u) (us,x)
```

Figure 4.8: Rebuilding a tree from its traversals via a fold.

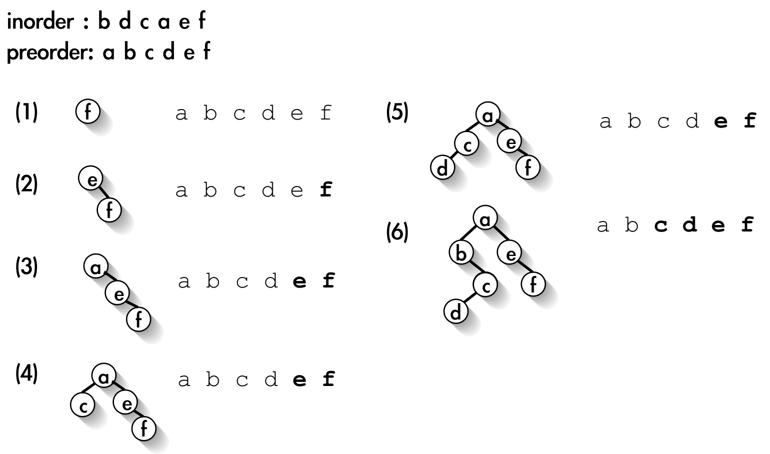

Figure 4.9: Building a tree from its preorder. The preorder traversals of the trees under the spine is written in boldface font.

$x \ominus preorderF\,(map\ snd\ us)$. Since the list is consumed from the end, we represent it in reverse. The function *rollpf* implements $roll \cdot (prefixOk\ x)?$.

Figure 4.9 shows an example of this algorithm in action. The part in boldface font indicates $preorderF\,(map\ snd\ us)$. Notice how the preorder traversals on of the trees under the spine always form a suffix of the given list $[a, b, c, d, e, f]$.

We have actually reinvented the algorithm proposed in [24], but in a functional style. The first step in [24] was to transform the recursive definition of $\langle preorder, inorder\rangle$ into an iteration by introducing a stack. The same effect we achieved by introducing the spine representation.

4.5.3 Building a Tree with a Given Preorder

The reader might justifiably complain that the derivation works because, by luck, we choose to invert *inorder* first. Had we started with $\langle inorder, preorder\rangle^{\circ}$ instead, it would lead us to

$$((x ==) \cdot inorder)? \cdot preorder^{\circ}$$

Now, we would have to invert *preorder*, and then enforce, on the resulting fold, the constraint that the tree built must have a given inorder traversal. Does the alternative still work? In fact, it does, and the result is a new, though complicated, algorithm to the problem of building a tree with given traversals. We sketch an outline of its development in this section.

We first seek to invert *preorder*. For this problem it turns out that it makes more sense to work on forests rather than trees. Abbreviate $List\,(Tree\ A)$ to $Forest\ A$. Recall $preorderF :: Forest\ A \rightarrow List\ A$ defined by $preorderF = concat \cdot map\ preorder$. The reader can easily verify that *preorderF* can be inverted as below:

$$\begin{array}{lll} preorderF^{\circ} & = & foldr\ step\ [\,] \\ \quad \textbf{where} & step\,(a, us) & = \ tip\ a : us \\ & & \quad \Box\ lbr\,(a, head\ us) : tail\ us \\ & & \quad \Box\ rbr\,(a, head\ us) : tail\ us \\ & & \quad \Box\ node(a, (us!!0, us!!1)) : tail\,(tail\ us) \end{array}$$

where the helper functions *tip*, *lbr* and *rbr* respectively creates a tip tree, a tree with only the left branch, and a tree with only the right branch. They are defined by:

$$\begin{array}{lll} tip\ a & = & node\,(a, (null, null)) \\ lbr\,(a, t) & = & node\,(a, (t, null)) \\ rbr\,(a, t) & = & node\,(a, (null, t)) \end{array}$$

In words, the relation *step* extends a forest in one of the four possible ways, when applicable : adding a new tip tree, extending the left-most tree in the forest by making it a left-subtree or a right-subtree, or combining the two left-most trees, if they exist.

The next step is to find out a guideline which of the four operations to perform when adding a new value. We need to invent an invariant to enforce in the body of the fold.

To begin with, we reason:

$$
\begin{array}{ll}
& ((x \mathbin{==}) \cdot \mathit{inorder})? \cdot \mathit{preorder}^{\circ} \\
= & \quad \{\text{since } \mathit{preorder} = \mathit{preorderF} \cdot \mathit{wrap}\} \\
& ((x \mathbin{==}) \cdot \mathit{inorder})? \cdot \mathit{wrap}^{\circ} \cdot \mathit{preorderF}^{\circ} \\
= & \quad \{\text{some trivial manipulation}\} \\
& \mathit{wrap}^{\circ} \cdot ((x \mathbin{==}) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder})? \cdot \mathit{preorderF}^{\circ}
\end{array}
$$

Again, the condition $(x \mathbin{==}) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder}$ is too strong to maintain. Luckily, it turns out that the weaker constraint

$$(\mathbf{isSubSeqOf}\ x) \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inorder}$$

will do, where $(\mathbf{isSubSeqOf}\ x)\ y = y\ \mathbf{isSubSeqOf}\ x$ yields true if y is a subsequence of x. That is, we require that during the construction of the forest, the inorder traversal of each tree shall always form segments of x, in correct order. Figure 4.10 demonstrates the process of constructing the same tree as that in Figure 4.9. This time notice how the inorder traversal of the constructed forest always forms a subsequence of the given list $[b, d, c, a, e, f]$.

After some pencil-and-paper work, it is not difficult to work out the rules to extend the forest while maintaining the invariant. Let the desired inorder traversal of the resulting tree be x. Define skip_x by:

$$
\begin{array}{lll}
\mathit{skip}_x & :: & \mathit{Forest}\ A \rightarrow \mathit{List}\ A \\
\mathit{skip}_x\,(t : u : \mathit{us}) & = & y \\
\quad \mathbf{where}\ \ \_ \mathbin{+\!\!+} \mathit{inorder}\ t \mathbin{+\!\!+} y \mathbin{+\!\!+} \mathit{inorder}\ u \mathbin{+\!\!+} \_ \ = \ x & & \\
\mathit{skip}_x\ \_ & = & [\,]
\end{array}
$$

That is, $\mathit{skip}_x\ \mathit{us}$ is the part of x between the leftmost tree and the next tree of the forest us. They correspond to the underlined segments in Figure 4.10. Also define left_x by

$$
\begin{array}{lll}
\mathit{left}_x & :: & \mathit{Forest}\ A \rightarrow \mathit{List}\ A \\
\mathit{left}_x\,[\,] & = & [\,] \\
\mathit{left}_x\,(t : \mathit{us}) & = & y \\
\quad \mathbf{where}\ \ y \mathbin{+\!\!+} \mathit{inorder}\ t \mathbin{+\!\!+} \_ \ = \ x & &
\end{array}
$$

It is a prefix of x before the leftmost tree in the forest. Some experiments will lead one to the following rules adding a value a to a forest while maintaining the invariant that

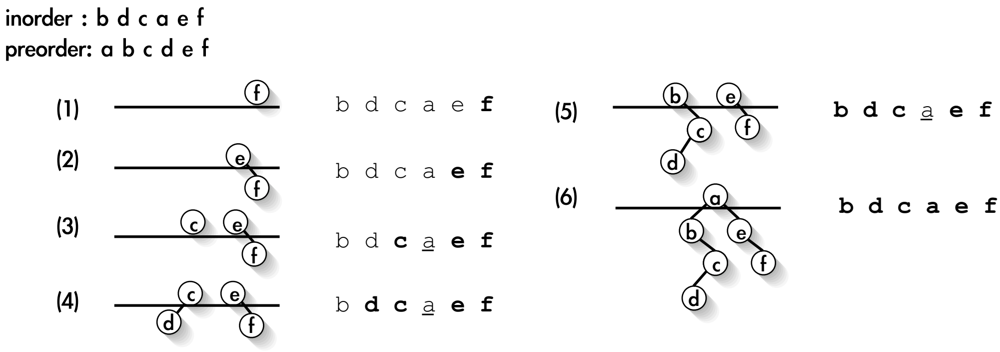

Figure 4.10: Building a tree from its preorder.

the inorder traversal of the forest is a subsequence of x:

$$
\begin{array}{lcl}
\mathit{add} & :: & (A \times \mathit{Forest}\ A) \rightarrow \mathit{Forest}\ A \\
\mathit{add}\ (a, [\,]) & = & [\mathit{tip}\ a] \\
\mathit{add}\ (a, \mathit{us}) & | & \mathit{skip}_x\ \mathit{us} == [\,]\ = \\
 & & \quad \mathbf{if}\ \mathit{isNext}_x\ \mathit{us}\ a\ \mathbf{then}\ \mathit{rbr}\ (a, \mathit{head}\ \mathit{us}) : \mathit{tail}\ \mathit{us} \\
 & & \quad \mathbf{else}\ \mathit{tip}\ a : \mathit{us} \\
\mathit{add}\ (a, \mathit{us}) & | & \mathit{skip}_x\ \mathit{us} == [a]\ = \\
 & & \quad \mathbf{case}\ \mathit{us}\ \mathbf{of} \\
 & & \qquad [t] \rightarrow [\mathit{lbr}\ (a, t)] \\
 & & \qquad (t : u : \mathit{us}) \rightarrow \mathit{node}\ (a, (t, u)) : \mathit{us} \\
\mathit{add}\ (a, t : \mathit{us}) & | & \mathit{head}\ (\mathit{skip}_x\ (t : \mathit{us})) == [a]\ =\ \mathit{lbr}\ (a, t) : \mathit{us} \\
 & | & \mathit{isNext}_x\ (t : \mathit{us})\ a\ =\ \mathit{rbr}\ (a, t) : \mathit{us} \\
 & | & \mathit{otherwise}\ =\ \mathit{tip}\ a : t : \mathit{us} \\
 & & \\
\mathit{isNext}_x\ \mathit{us}\ a & = & \mathit{not}\ (\mathit{null}\ (\mathit{left}_x\ \mathit{us})) \wedge \mathit{last}\ (\mathit{left}_x\ \mathit{us}) == a
\end{array}
$$

The rules above are visualised in Figure 4.11, where *add* is written as $\oplus$ and the values of skip_x is written as a superscript above the trees. The left-hand sides of the arrows indicates the patterns to match, in top-down order. The resulting forest is drawn on the right-hand side, together with new values of skip_x. From the figure, it is easier to see that the rules do maintain the invariant.

To the best of the author's knowledge, this algorithm is new. However, the rules consists of totally eight cases and are relatively complicated comparing to the simpler algorithms in Section 3.3 and Section 4.5.2. It is due to the fact that we have four possible operations to choose from, while in Section 4.5.2 there are only two – either to walk upward along the spine for one node or to stop and attach a new node. For that reason we will not go into more detail but just present the result. A similar approach can be applied to the well-known problem of constructing a binary search tree from its preorder traversal. In this simpler version, the algorithm is much simpler.

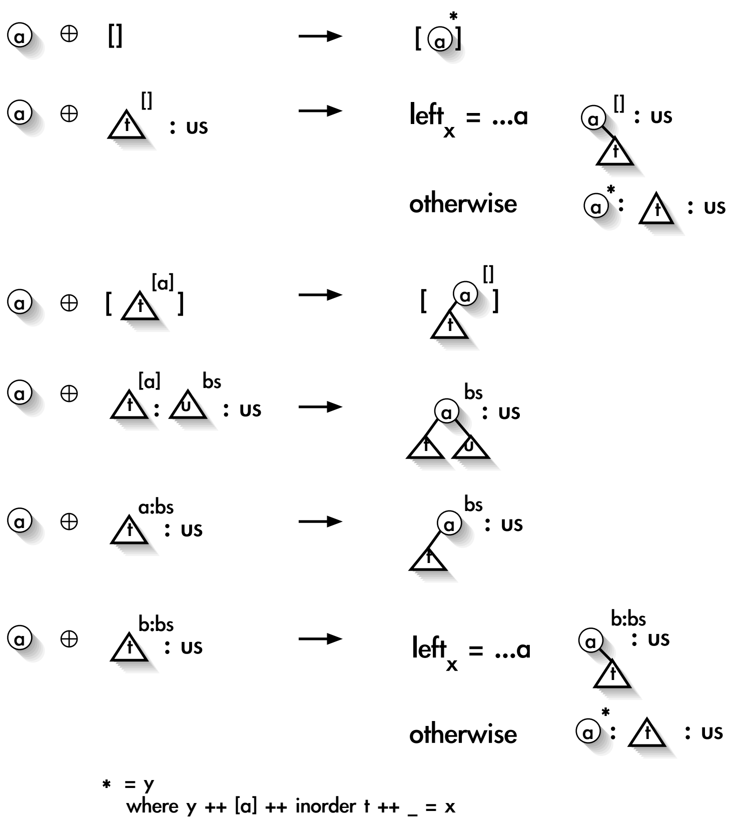

Figure 4.11: Building a tree from its preorder.

```
tip a   = Node a Null Null
rbr a x = Node a Null x
lbr a x = Node a x Null

type AForest a = [(Tree a, [a])]

rebuild :: Eq a => [a] -> [a] -> Tree a
rebuild x = fst . unwrap . snd . foldr add (reverse x, [])
 where add :: Eq a => a -> ([a],AForest a) -> ([a],AForest a)
       add a xu@(x, [])  = newtree a xu
       add a xu@(x, (t,[]):us)
         | isNext x a = (tail x, (rbr a t, []):us)
         | otherwise = newtree a xu
       add a xu@(x,(t,b:bs):us)
         | a == b = (x, join a (t,bs) us)
         | isNext x a = (tail x, (rbr a t, b:bs):us)
         | otherwise = newtree a xu

       join a (t,[]) [] = [(lbr a t,[])]
       join a (t,[]) ((u,y):us) = (Node a t u, y) : us
       join a (t,bs) us = (lbr a t, bs):us

       newtree a (x,us) = (x', (tip a, y):us)
          where (x',y) = skip x a

       isNext [] a = False
       isNext (b:bs) a = a == b

skip x a = locate a [] x
 where locate a y [] = ([],y)
       locate a y (b:x) | a == b = (x,y)
                        | otherwise = locate a (b:y) x
```

Figure 4.12: Another way to rebuild a tree from its traversals via a fold.

A program implementing the algorithm is presented in Figure 4.12. To avoid repeatedly traversing the forest for each computation of *skip*, each tree in the forest is annotated with the *skip* value of the forest starting from itself, represented by the type *AForest A*. Furthermore, to speed up the computation of *isNext*, the value of *left* at each step of computation is reversed and paired with the forest. The function *add* thus takes and returns an annotated forest of type ($List\ A \times AForest\ A$). After this data refinement, the program runs in linear time, but with a bigger constant overhead than that in Section 4.5.2.

4.6 The Generalised Converse-of-a-Function Theorem

By definition, a *hylomorphism* is the composition of a fold after an unfold. The hylomorphism $(\![R]\!)_{\mathsf{F}} \cdot (\![T]\!)_{\mathsf{F}}^{\circ}$ can be characterised as the least solution for X of the inequation $R \cdot \mathsf{F}X \cdot T^{\circ} \subseteq X$. The aim of this section is to prove the following generalisation of Theorem 4.1:

Theorem 4.3 (Generalised converse-of-a-function theorem) Let $S :: B \to A$ be a simple relation. If there exists a relation $R :: \mathsf{F}(C,B) \to B$ and a simple relation $T :: \mathsf{F}(C,A) \to A$ are such that (i) $dom\ S = ran\ R$; (ii) $S \cdot R \subseteq T \cdot \mathsf{F}S$; and (iii) $\delta_{\mathsf{F}} \cdot R^{\circ}$ is inductive, then

$$S^{\circ} \quad = \quad (\![R]\!)_{\mathsf{F}} \cdot (\![T]\!)_{\mathsf{F}}^{\circ}$$

In words, Theorem 4.3 gives conditions under which a simple relation can be inverted as a hylomorphism. The new ingredients in Theorem 4.3 are the *membership* relation δ_{F} of a relator F, and the notion of an *inductive* relation. Both are described below in Section 4.6.1. The main proof is given in Section. 4.6.2.

Theorem 4.1 follows as a special instance of Theorem 4.3 by taking $T = \alpha$ and S to be an entire relation as well as a simple one, that is, a function. An entire relation S is one for which $dom\ S = id$, so condition (i) translates to the requirement that R be a surjective relation. In Section 4.6.2, we will prove that condition (iii) holds if both (i) and (ii) do and if $\delta_{\mathsf{F}} \cdot T^{\circ}$ is inductive. Fact 4.6 below gives us that $\delta_{\mathsf{F}} \cdot \alpha_{\mathsf{F}}^{\circ}$ is inductive. Since $(\![\alpha_{\mathsf{F}}]\!)_{\mathsf{F}} = id$, we then obtain the result $S^{\circ} = (\![R]\!)_{\mathsf{F}}$, the conclusion of Theorem 4.1.

4.6.1 Inductivity and Membership

We say that a relation admits induction, or is inductive, if we can use it to perform induction[31]. Formally, inductivity is defined by:

Definition 4.4 (Inductivity) A relation $R :: A \leadsto A$ is inductive if for all $X :: B \leadsto A$,

$$R \backslash X \subseteq X \quad \Rightarrow \quad \Pi \subseteq X$$

Here Π denotes the largest relation of its type, and the left division operator ($\backslash$) is defined by the Galois connection:

$$S \subseteq R \backslash T \quad \equiv \quad R \cdot S \subseteq T$$

The definition can be translated to the point level to aid understanding. It says that R is inductive if the property

$$(\forall c :: (c, a) \in R \Rightarrow (c, b) \in X) \quad \Rightarrow \quad (a, b) \in X$$

where a and b are arbitrary, implies X contains all the pairs of its type. As an example, take R to be $<$, the ordering on natural numbers, and $P\,a = (a, b) \in X$ to be some property we want to prove for all a and some fixed b. The definition specialises to the claim that if

$$(\forall c :: c < a \Rightarrow P\,c) \quad \Rightarrow \quad P\,a$$

then $P\,a$ holds for all natural numbers a. Thus we can see that inductivity captures the principle of induction.

Inductivity is important to us because it guarantees uniqueness of solutions. The following theorem comes from [17, Theorem 6.3]:

Theorem 4.5 If $\delta_{\mathsf{F}} \cdot R$ is inductive, then then equation $X = T \cdot \mathsf{F}X \cdot R$ has a unique solution $X = (\![T]\!)_{\mathsf{F}} \cdot (\![R^\circ]\!)_{\mathsf{F}}^\circ$.

Three facts concerning inductivity we will need are the following:

Fact 4.6 The relation $\delta_{\mathsf{F}} \cdot \alpha_{\mathsf{F}}^\circ$ is inductive.

Fact 4.7 If R is inductive and $S \subseteq R$, then S is inductive.

Fact 4.8 If R is inductive, so is $S^\circ \cdot R \cdot S$ for any simple relation S.

The other concept we need, due to Hoogendijk and de Moor [42], is the membership relation of a datatype. For example, a membership relation δ_{List} for lists can be specified informally by:

$$(a, [a_0, a_1, \ldots a_n]) \in \delta_{List} \quad \equiv \quad (\exists i :: a = a_i)$$

The formal definition of membership is not at all intuitive, and we refer the reader to [42] for more discussion. A fact about membership we will use is that it is a lax natural transformation, which is to say,

$$\delta_{\mathsf{F}} \cdot \mathsf{F}R \quad \subseteq \quad R \cdot \delta_{\mathsf{F}} \tag{4.7}$$

for all R.

4.6.2 The Proof

Taking converses of both sides, the aim is to prove that $S = (\![T]\!)_{\mathsf{F}} \cdot (\![R]\!)_{\mathsf{F}}^\circ$ under the given conditions. Since, by Theorem 4.5 and assumption (iii) that $\delta_{\mathsf{F}} \cdot R^\circ$ is inductive, we know that $(\![T]\!)_{\mathsf{F}} \cdot (\![R]\!)_{\mathsf{F}}^\circ$ is the unique solution for X of the equation below:

$$X \quad = \quad T \cdot \mathsf{F}X \cdot R^\circ$$

Now we will show that S is also a solution. The proof goes:

$$
\begin{array}{ll}
 & S \\
= & \quad \{\text{since } S = S \cdot \mathit{dom}\, S = S \cdot \mathit{ran}\, R \text{ by assumption (i)}\} \\
 & S \cdot \mathit{ran}\, R \\
\subseteq & \quad \{\text{since } \mathit{ran}\, R \subseteq R \cdot R^\circ\} \\
 & S \cdot R \cdot R^\circ \\
\subseteq & \quad \{\text{by assumption (ii): } S \cdot R \subseteq T \cdot \mathsf{F}S\} \\
 & T \cdot \mathsf{F}S \cdot R^\circ \\
= & \quad \{\text{since } R = \mathit{ran}\, R \cdot R = \mathit{dom}\, S \cdot R \text{ by assumption (i)}\} \\
 & T \cdot \mathsf{F}S \cdot R^\circ \cdot \mathit{dom}\, S \\
\subseteq & \quad \{\text{since } \mathit{dom}\, S \subseteq S^\circ \cdot S\} \\
 & T \cdot \mathsf{F}S \cdot R^\circ \cdot S^\circ \cdot S \\
\subseteq & \quad \{\text{by assumption (ii): } S \cdot R \subseteq T \cdot \mathsf{F}S\} \\
 & T \cdot \mathsf{F}S \cdot (T \cdot \mathsf{F}S)^\circ \cdot S \\
\subseteq & \quad \{\text{since } T \cdot \mathsf{F}S \text{ simple}\} \\
 & S
\end{array}
$$

We will now prove a lemma which shows that condition (iii) of Theorem 4.3 holds if conditions (i) and (ii) do and if $\delta_{\mathsf{F}} \cdot T^\circ$ is inductive. It is this lemma that establishes the connection between Theorem 4.1 and Theorem 4.3. We will make use of the following shunting rule for simple S:

$$S \cdot X \subseteq Y \quad \equiv \quad \mathit{dom}\, S \cdot X \subseteq S^\circ \cdot Y \tag{4.8}$$

For the reader's reference, the above shunting rule is proved in Appendix A. When S is also entire, i.e., $\mathit{dom}\, S = \mathit{id}$, (4.8) reduces to the usual shunting rule for functions.

Lemma 4.9 The relation $\delta_{\mathsf{F}} \cdot R^\circ$ is inductive if (i) $\mathit{ran}\, R \subseteq \mathit{dom}\, S$; (ii) $S \cdot R \subseteq T \cdot \mathsf{F}S$; and (iii) $\delta_{\mathsf{F}} \cdot T^\circ$ is inductive.

Proof. We reason:

$$
\begin{array}{ll}
 & \delta_{\mathsf{F}} \cdot R^\circ \\
\subseteq & \quad \{\text{claim: } R \subseteq S^\circ \cdot T \cdot \mathsf{F}S\} \\
 & \delta_{\mathsf{F}} \cdot \mathsf{F}S^\circ \cdot T^\circ \cdot S \\
\subseteq & \quad \{\text{by (4.7)}\} \\
 & S^\circ \cdot \delta_{\mathsf{F}} \cdot T^\circ \cdot S
\end{array}
$$

Since $\delta_{\mathsf{F}} \cdot T^\circ$ is inductive, so is $S^\circ \cdot \delta_{\mathsf{F}} \cdot T^\circ \cdot S$ by Fact 4.8. We then obtain that $\delta_{\mathsf{F}} \cdot R^\circ$ is inductive by Fact 4.7.

The claim that $R \subseteq S^\circ \cdot T \cdot \mathsf{F}S$ is proved below:

$$
\begin{array}{cl}
 & R \subseteq S^\circ \cdot T \cdot \mathsf{F}S \\
\equiv & \quad \{\text{using } R = \mathit{ran}\, R \cdot R\} \\
 & \mathit{ran}\, R \cdot R \subseteq S^\circ \cdot T \cdot \mathsf{F}S \\
\Leftarrow & \quad \{\text{assumption (i)}\} \\
 & \mathit{dom}\, S \cdot R \subseteq S^\circ \cdot T \cdot \mathsf{F}S \\
\equiv & \quad \{\text{shunting (4.8)}\} \\
 & S \cdot R \subseteq T \cdot \mathsf{F}S
\end{array}
$$

□

4.7 Applications of the Generalised Theorem

Theorem 4.3 can potentially be very powerful since it allows the functor F, which determines the pattern of recursion, to be independent from the input and output types. A much wider class of algorithms can thus be covered. However, the theorem itself offers no clue how F and f could be chosen. It is therefore less useful for program derivation and more helpful in proving the correctness of known algorithms.

One application we have found for Theorem 4.3 is to prove that a loop implements the inverse of some function. A loop can be specified relationally by

$$T \cdot R^* \cdot S$$

The relation S initialises the loop, while R serves as the loop body. The domain of T represents the terminating condition and therefore ought to be disjoint from the domain of R. Given a relation R, the reflexive transitive closure R^* is the smallest reflexive transitive relation containing R. More generally, the relation $R^* \cdot S :: A \to B$, where $S :: A \to B$ and $R :: B \to B$, can be defined as a least fixed-point:

$$R^* \cdot S \quad = \quad \mu(X \mapsto S \cup R \cdot X)$$

A key observation here is that a closure can also be written as a hylomorphism, with the base functor $\mathsf{F}_A X = A + X$:

$$
\begin{array}{cl}
 & R^* \cdot S \\
= & \quad \{\text{definition of closure}\} \\
 & \mu(X : S \cup R \cdot X) \\
= & \quad \{\text{coproduct}\} \\
 & \mu(X : [S, R] \cdot (\mathit{id} + X) \cdot [\mathit{id}, \mathit{id}]^\circ) \\
= & \quad \{\text{hylomorphism, let } \mathsf{F}_A X = A + X\} \\
 & (\!|[S, R]|\!)_{\mathsf{F}} \cdot (\!|[\mathit{id}, \mathit{id}]|\!)^\circ_{\mathsf{F}}
\end{array}
$$

Here the unfolding phase wraps the input value with an *inl*, before wrapping it with an indefinite number of *inr*s. The folding phase then replaces the *inl* with S and each *inr* with an R. The exact number of iterations performed is determined the termination test T.

Given a function f, let us instantiate Theorem 4.3 to discover the conditions under which $f^\circ = (\![S, R]\!)_{\mathsf{F}} \cdot (\![id, id]\!)_{\mathsf{F}}^\circ$:

- Since $dom\, f = id$, condition (i) instantiates to $ran\,[S, R] = id$. That is, S and R shall be jointly surjective.
- Condition (ii) can be divided into two parts:

  $$f \cdot S \subseteq id \quad \wedge \quad f \cdot R \subseteq f$$

  Shunting the functions to the other side, we get:

  $$S \subseteq f^\circ \quad \wedge \quad R \cdot f^\circ \subseteq f^\circ$$

  which looks familiar enough! Think of f° as an invariant. The first half says that the initial values satisfies the invariant, while the second half says that given inputs satisfying the invariant, the loop body R maintains the invariant.
- Since $\delta_{\mathsf{F}} \cdot [S, R] = R$, condition (iii) requires that R be inductive. Intuitively speaking, we want R to "decrease" the loop variables in some sense, so that the loop terminates.

Assume we wish to prove that $T \cdot R^* \cdot S$ correctly implements a specification X. As will be shown in the next two sections, in some occasions X can be quite naturally factored into $T \cdot f^\circ$ for some f. We then just need to check the three conditions above.

4.7.1 Splitting a List revisited

As the first example, let us consider again inverting the function $cat :: (List\ A \times List\ A) \rightarrow List\ A$. The two conditions instantiate to

$$\begin{array}{lcl} cat \cdot S & \subseteq & id \\ cat \cdot R & \subseteq & cat \end{array}$$

The second condition says that given a pair of lists (x, y), the relation R ought to map it to another pair of lists which still concatenates to $x +\!\!+\, y$. One possible way to do that is to move one element from the head of y to the tail of x:

$$R\,(x, y) \quad = \quad (x +\!\!+\, head\ y, tail\ y)$$

The relation R alone is not surjective: it only generates pairs of lists whose first components are not empty. To ensure surjectivity, the following choice of S comes naturally:

$$S\,y \quad = \quad ([\,], y)$$

The relation R reduces the length of y, therefore the loop does eventually terminate when y becomes empty. We therefore conclude that $cat^\circ = R^* \cdot S$.

Note that the following choice of R, moving two elements at a time, does not work:

$$\begin{array}{lcl} R\,(x,[a]) & = & (x \mathbin{+\!\!+} [a],[\,]) \\ R\,(x,(a:b:y)) & = & (x \mathbin{+\!\!+} [a,b],y) \end{array}$$

It fails to satisfy the surjectivity condition because it does not generate pairs of lists (x, y) where x is of odd length and y is not empty.

4.7.2 The String Edit Problem

The string edit problem is a typical example for dynamic programming. Recently it has drawn much attention due to its application in DNA sequence matching. In its simplest form, we are given two strings, one as the source and one as the target, and some available commands. Imagine a cursor positioned to the left of the source string. We assume the following commands:

- $ins\ c$: to insert a character c at the current position. The target string will thus have an extra character c after this operation.
- $del\ c$: to delete the character, c, in the current position. Or, one can think of it as a statement that the source string has an extra c.
- $cpy\ c$: to skip the current character c and move the cursor one position to the right. Some people prefer to view it as copying the character c from the source to the target.

The task is to find the shortest sequence of commands to transform the source string to the target string. In more complicated variations we might be given more commands and their weights may vary.

We represent the three commands with a datatype Op:

$$\mathbf{data}\ Op \quad = \quad ins\ Char \mid del\ Char \mid cpy\ Char$$

To specify the problem, one might attempt to construct a relation taking the pair of strings and return an arbitrary sequence of commands relating the strings. In fact, it is easier to construct its inverse. The function *exec* below executes a sequence of commands, starting from a pair of empty strings, and yields two strings:

$$\begin{array}{lcl} exec & :: & List\ Op \to (String \times String) \\ exec & = & foldl\ step\ ([\,],[\,]) \\ & & \quad \mathbf{where}\quad \begin{array}{lcl} step\,(x,y)\,(ins\ c) & = & (x, y \mathbin{+\!\!+} [c]) \\ step\,(x,y)\,(del\ c) & = & (x \mathbin{+\!\!+} [c], y) \\ step\,(x,y)\,(cpy\ c) & = & (x \mathbin{+\!\!+} [c], y \mathbin{+\!\!+} [c]) \end{array} \end{array}$$

The string edit problem is thus defined by

$$stredit \quad = \quad min\ R \cdot \Lambda exec^{\circ}$$

The ingredient $min\ R$ will be discussed in detail in the next chapter. For now the reader merely needs to know that its type instantiates to $Set\,(List\ Op) \to List\ Op$ and it chooses a shortest sequence of operations from a set of candidates. In [17, Chapter 9], Bird and de Moor derived from this specification a dynamic programming algorithm using their dynamic programming theorem for unfolds.

Yet some others prefer to describe $exec^{\circ}$ as an iterative process. That is, they claim that $exec^{\circ} = end \cdot move^{*} \cdot start$, where

$$\begin{array}{lcl} start\,(x, y) & = & (x, y, [\,]) \\ move\,(x, y, ops) & = & (x, init\,y, ins\,(last\,y) : ops) \\ & & \Box\,(init\,x, y, del\,(last\,x) : ops) \\ & & \Box\,((init\,x, init\,y, cpy\,(last\,x) : ops), if\ last\,x == last\,y) \\ end\,([\,], [\,], ops) & = & ops \end{array}$$

The loop starts with the two strings and an empty list of commands. The non-deterministic loop body *move* then try to recover what the last command might be by trying all possible commands. The iteration repeats until both strings become empty. Notice that *move* is defined as a partial relation which yields value only when not both of x and y are empty. This was the view taken by Curtis in [26]. Once a specification is written in terms of a $min\,R$ after a loop, theories in [26] are ready to transform it to a dynamic programming algorithm, if certain conditions are satisfied.

Optimisation problems will be discussed in the next chapter and this is not the place to discuss how the problem can be solved using the developed theories. Instead we will bridge the gap between the two views on *exec*. In other words, how do we know the claim that $exec^{\circ} = end \cdot move^{*} \cdot start$ is true?

With the discussions in the opening of Section 4.7 in mind, we generalise *exec* to *execWith* such that

$$exec \;=\; execWith \cdot end^{\circ}$$

The function *execWith* has type $(String \times String \times List\ Op) \rightarrow (String \times String)$ and is defined by:

$$execWith\,(x, y, ops) \;=\; foldl\ step\,(x, y)\ ops$$

It is just replacing the constant $([\,], [\,])$ in the definition of *exec* with a given argument (x, y). The task is then to show that $execWith^{\circ} = move^{*} \cdot start$. One may also think of it as that we have just invented and proposed $execWith^{\circ}$ to be the loop invariant, and are about to check whether this invariant works. The invariant says that, denoting the input pair of strings by (x, y), and the intermediate values at any point of computing $move^{*} \cdot start$ by (x', y', ops), executing the commands *ops* on (x', y') shall always yield (x, y).

Now we will check the conditions one by one:

- Condition (i) holds: *start* and *move* are jointly surjective.
- Condition (ii) requires:

$$\begin{array}{rcl} execWith \cdot start & \subseteq & id \\ execWith \cdot move & \subseteq & execWith \end{array}$$

The first one trivially holds. The second inclusion holds because *move* undoes the last step of execution. Thus the domain of the left-hand side is restricted to triples where one of the two strings is not empty. The execution still yields the same result.

- For condition (iii): *move* is well-founded and thus inductive.

Therefore, we conclude that $execWith^\circ = (\![start, move]\!)_{\mathsf{F}} \cdot (\![id, id]\!)^\circ_{\mathsf{F}} = move^* \cdot start$.

4.7.3 Building Trees by Combining Pairs

Recall again the following datatype for leaf-valued binary trees:

$$\mathbf{data}\ Tree\ A \quad = \quad tip\ A \mid bin\ (Tree\ A \times Tree\ A)$$

And yes, we are about to introduce yet another approach to building trees from a given list.

We have briefly mentioned inverting *flatten* to an unfold (we will come back to this unfolding approach later in Chapter 6), and the majority of this chapter has been focusing on inverting *flatten* to a fold. There is yet another alternative way to build a tree from a list: starting from a list of tips, keep combining adjacent trees until only one is left. The process can be characterised by

$$wrap^\circ \cdot join^* \cdot map\ tip$$

where $join\ (x +\!\!+ [a, b] +\!\!+ y) = x +\!\!+ [bin\ (a, b)] +\!\!+ y$.

Our aim is, of course, to show that $flatten^\circ = wrap^\circ \cdot join^* \cdot map\ tip$. Observe that

$$flatten \quad = \quad flattenF \cdot wrap$$

where $flattenF = concat \cdot map\ flatten$. We have just proposed this invariant for the loop: that during the iterations, the forest always flattens to the given list. Now we check that $flattenF^\circ = join^* \cdot map\ tip$:

- Indeed, *map tip* and *join* are jointly surjective. The former covers any lists of tip trees while the latter covers the rest.

- We need to verify that:

$$\begin{array}{rcl} concat \cdot map\ flatten \cdot map\ tip & \subseteq & id \\ concat \cdot map\ flatten \cdot join & \subseteq & concat \cdot map\ flatten \end{array}$$

  The first inclusion trivially holds. The second holds because *join* restricts the domain of the left-hand side to lists with at least two trees, but not affecting the result returned.

- Finally, *join* is well-founded because it reduces the length of the forest.

It then follows that $\mathit{flattenF}^{\circ} = \mathit{join}^{*} \cdot \mathit{map}\ \mathit{tip}$ and, consequently, $\mathit{flatten}^{\circ} = \mathit{wrap}^{\circ} \cdot \mathit{join}^{*} \cdot \mathit{map}\ \mathit{tip}$.

One might relate this small exercise to merge sort. There are two ways to implement merge sort: one is to implement it as a hylomorphism, where the unfolding phase expands a tree and the folding phase performs merging at each node. The other is to implement it as a loop: to start with *map wrap*, converting the input to a list of singleton lists, and then to iteratively merge adjacent lists until only one list is left. The first can be said to be top-down and the second bottom-up. A similar reasoning converts the former to the latter. However, an additional distributivity property of list merging will be needed in the proof. A similar problem was treated in [41], where a top-down algorithm was also transformed to a bottom-up one.

Chapter 5

Optimisation Problems

Optimisation problems, which usually involves choosing a best solution among the set of all legal ones, are suitable to be expressed relationally. In [17], Bird and de Moor have developed theories formalising when and how an optimisation problem, specified as a fold or an unfold, can be solved by a greedy algorithm, a thinning algorithm or a dynamic programming strategy. Curtis [26] further generalised their theories to problems that can be specified in terms of an iterative operator, which covers most optimisation problems we encounter in practice.

In the first two sections of this chapter, we will look at the interplay between the converse-of-a-function theorem and the above theories about optimisation problems. In particular, we utilise the greedy and the thinning theorem developed in [17] for the optimal bracketing problem. A greedy linear-time algorithm is derived for one of its instances — to build trees of minimum height.

In the third section, we will deviate from the theme of inverse functions a bit and pursue further on more knowledge about optimisation problems specified as folds. The greedy theorem is extended to allow mutually defined algebras. The generalised theorem is applied to an interesting class of problems called the *optimal marking problem*. Polynomial-time algorithms are derived for two instances of such problems.

5.1 Building Trees with Minimum Height

Given is a list of trees. The task is to combine them into a single tree, retaining the left-to-right order of the subtrees. How can we make the height of the resulting tree as small as possible? Figure 5.1 illustrates one such tree, of height 11, for given subtrees of heights $[2, 9, 8, 3, 6, 9]$. As the actual content of the subtrees is not important, we can think of them simply as numbers representing the heights. The problem is therefore again one of turning a list of numbers to a tree. A linear-time algorithm to this problem has been proposed in [14]. Here we will demonstrate how a similar algorithm can be derived.

First let us consider now to formalise the problem. We will make use of the same datatype as in Section 4.3 for leaf-valued binary trees:

$$\mathbf{data}\ \mathit{Tree}\ A \quad = \quad \mathit{tip}\ A \mid \mathit{bin}\ (\mathit{Tree}\ A \times \mathit{Tree}\ A)$$

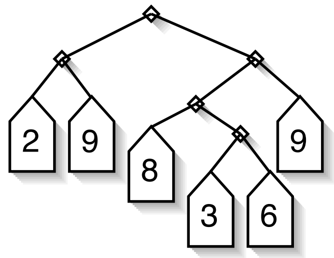

Figure 5.1: A tree with height 11 built from trees with heights $[2, 9, 8, 3, 6, 9]$.

Also recall the familiar function *flatten* defined by

$$\begin{array}{lcl} \mathit{flatten} & :: & \mathit{Tree}\ A \to \mathit{List}_1\ A \\ \mathit{flatten} & = & \mathit{foldTree}\ (+\!\!+)\ \mathit{wrap} \end{array}$$

Given a tip-valued binary tree whose tip values represent the heights of trees below, the function computing the height of the combined tree can be defined as a fold in the obvious way:

$$\begin{array}{lcl} \mathit{height} & :: & \mathit{Tree}\ \mathcal{Z} \to \mathcal{Z} \\ \mathit{height} & = & \mathit{foldTree}\ \mathit{ht}\ \mathit{id} \\ & & \quad \textbf{where}\quad \mathit{ht}\,(a, b) \ = \ (a \sqcup b) + 1 \end{array}$$

where $\sqcup$ returns the larger of its two arguments.

The problem is to find, among all the trees which flatten to the given list, one for which *height* yields the minimal value. To collect all possible solutions, we can make use of the power transpose operator Λ. To extract a value from a set, we will need the relation $\mathit{min}\ R :: \mathit{Set}\ A \to A$ defined by:

$$(xs, x) \in \mathit{min}\ R \equiv x \in xs \wedge (\forall y : y \in xs : x\ R\ y)$$

The relation $\mathit{min}\ R$ chooses among the given set a minimum member under the ordering R. For this definition to be of any use, R has to be a *connected preorder*. A relation is a preorder if it is reflexive and transitive. We call a preorder connected if it compares everything of the type: i.e., for every x and y either $x\,R\,y$ or $y\,R\,x$. However, in such cases x and y need not necessarily be equal, That is, a preorder is not necessarily anti-symmetric. Therefore, $\mathit{min}\ R$ will not in general be a function. Complementarily, we also define:

$$\mathit{max}\ R \ = \ \mathit{min}\ R^{\circ}$$

The relation $\mathit{max}\ R$ chooses among the given set a maximum member. Two properties of *min* (as well as *max*), proved in [17], will be used repeatedly and are thus cited below:

$$\mathit{min}\ R \cdot \mathsf{P}f \ = \ f \cdot \mathit{min}\,(f^{\circ} \cdot R \cdot f) \qquad (5.1)$$

$$\mathit{min}\ R \subseteq \mathit{min}\ Q \ \Leftarrow \ R \subseteq Q \qquad (5.2)$$

For our problem, define $(\preceq)$ to be a comparison between the heights of two trees:

$$x \preceq y \quad \equiv \quad \mathit{height}\, x \leq \mathit{height}\, y$$

Our problem can then be specified as:

$$\mathit{bmh} \quad = \quad \mathit{min}\,(\preceq) \cdot \Lambda(\mathit{flatten}^{\circ})$$

Given a list, $\mathit{flatten}^{\circ}$ maps it to an arbitrary tree that flattens to the list. The Λ operator collects all the trees into a set. Within the set, one with the minimum height is chosen by $\mathit{min}\,(\preceq)$.

To derive an algorithm from the specification, the relations derived in Sect. 4.3.1 can be reused. We borrow from there the spine representation:

$$\textbf{type}\, \mathit{Spine}\ A \quad = \quad (A \times \mathit{List}\,(\mathit{Tree}\ A))$$

and an injective function $\mathit{roll} :: \mathit{Spine}\ A \to \mathit{Tree}\ A$ for converting between the two representations, as well as the algebra add and one are defined by:

$$\begin{array}{lcl} \mathit{one}\ a & = & (a, [\,]) \\ \mathit{add}\,(a, (b, \mathit{us})) & = & (a, \mathit{roll}\,(b, \mathit{vs}) : \mathit{ws}) \\ \quad\quad \textbf{where} \quad \mathit{vs} +\!\!+\, \mathit{ws} \ = \ \mathit{us} & & \end{array}$$

The function $\mathit{flatten}$ was inverted to $\mathit{roll} \cdot \mathit{foldrn}\ \mathit{add}\ \mathit{one}$. Back to our problem, the derivation goes:

$$\begin{array}{ll} & \mathit{bmh} \\ = & \quad \{\text{definition}\} \\ & \mathit{min}\,(\preceq) \cdot \Lambda(\mathit{flatten}^{\circ}) \\ = & \quad \{\text{inverting } \mathit{flatten}\} \\ & \mathit{min}\,(\preceq) \cdot \Lambda(\mathit{roll} \cdot \mathit{foldrn}\ \mathit{add}\ \mathit{one}) \\ = & \quad \{\mathit{roll} \text{ a function}\} \\ & \mathit{min}\,(\preceq) \cdot \mathsf{P}\mathit{roll} \cdot \Lambda(\mathit{foldrn}\ \mathit{add}\ \mathit{one}) \\ = & \quad \{\text{by (5.1), define } \preceq' \text{ below}\} \\ & \mathit{roll} \cdot \mathit{min}\,(\preceq') \cdot \Lambda(\mathit{foldrn}\ \mathit{add}\ \mathit{one}) \end{array}$$

where the ordering $\preceq'$ is the counterpart of $\preceq$ on spine trees:

$$\mathit{xs} \preceq' \mathit{ys} \quad \equiv \quad \mathit{roll}\ \mathit{xs} \preceq \mathit{roll}\ \mathit{ys}$$

Still, $\Lambda(\mathit{foldrn}\ \mathit{add}\ \mathit{one})$ generates an exponential number of possible trees. This time, how are we supposed to enforce some constraints into the fold to reduce the number of possibilities? The answer is to make use of the greedy theorem, to be introduced in the next section.

5.1.1 The Greedy Theorem

We will now briefly review the greedy theorem in [17] and see how it can be applied to our problem. First of all, we will introduce the notion of monotonicity.

Definition 5.1 (Monotonicity) A relation $S :: \mathsf{F}A \to A$ is said to be *monotonic on R* if and only if:

$$S \cdot \mathsf{F}R \quad \subseteq \quad R \cdot S$$

Take non-empty cons-lists for example. The base functor is $\mathsf{F}_A X = A + A \times X$. Without loss of generality we can assume that S has the form $S = [\mathit{base}, \mathit{step}]$. What does it mean for S to be monotonic on a preorder $\preceq$? The above definition translates to[1]:

$$(a, x) \in \mathit{base} \quad \Rightarrow \quad (\exists y : (a, y) \in \mathit{base} : x \preceq y)$$

which is a tautology if $\preceq$ is a preorder since we can take y to be x, and:

$$((a, x), x') \in \mathit{step} \wedge x \preceq y \quad \Rightarrow \quad (\exists y' : ((a, y), y') \in \mathit{step} : x' \preceq y') \tag{5.3}$$

Suppose that we use *foldrn step base* to generate an arbitrary solution. The relation *step* takes a partial solution and extends it. If we have S monotonic on $\preceq$, in effect it means that for two solutions x and y, y being at least as large as x with respect to $\preceq$, no matter how we extend x to x', we can always find a way to extend y to y' such that y' is not smaller than x'. There is thus no point keeping the smaller one, x, in the first place. We need to keep only the *best* solution so far in each stage. This is made precise in the following *greedy theorem*:

Theorem 5.2 (Greedy Theorem) If S is monotonic on connected preorder R, then

$$(\!|\mathit{max}\, R \cdot \Lambda S|\!)_{\mathsf{F}} \quad \subseteq \quad \mathit{max}\, R \cdot \Lambda (\!|S|\!)_{\mathsf{F}}$$

Since $\mathit{min}\, R = \mathit{max}\, R^{\circ}$, the same theorem can also be written as

$$(\!|\mathit{min}\, R \cdot \Lambda S|\!)_{\mathsf{F}} \subseteq \mathit{min}\, R \cdot \Lambda (\!|S|\!)_{\mathsf{F}} \quad \Leftarrow \quad S \text{ monotonic on } R^{\circ}$$

We see that $\mathit{min}\, R$ is promoted into the fold. Rather than looking for a minimum one among all the solutions returned by the fold, a minimum solution is chosen in each step of the fold and becomes the only one to be passed on to the next step.

Back to our problem. Had *add* satisfied the monotonicity condition (5.3) with respect to $\succeq'$ (the converse of $\preceq'$), we would be able to apply the greedy theorem. However, it is not true: a tree with the smallest height does not always remain the smallest after being extended by *add*.

Fortunately, *add* is monotonic on a stronger ordering. We define:

$$\mathit{heights}\,(a, xs) \quad = \quad (\mathit{reverse} \cdot \mathit{map}\ \mathit{height} \cdot \mathit{scanl}\ \mathit{Bin}\,(\mathit{Tip}\ a))\, xs$$

[1] Free identifiers are considered to be universally quantified.

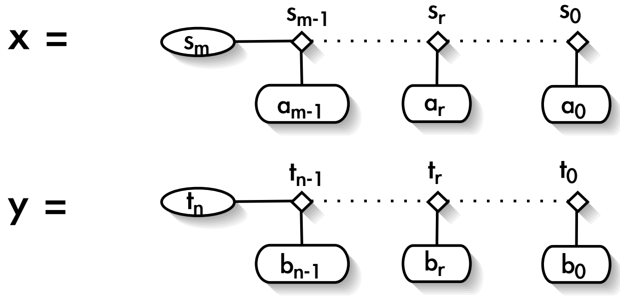

Figure 5.2: Assumption of how x and y look like. Here $x = (s_m, [a_{m-1}, a_{m-2}, \ldots a_0])$ and $y = (t_n, [b_{n-1}, b_{n-2}, \ldots b_0])$. The values $s_{m-1}, s_{m-2}, \ldots s_0$ and $t_{n-1}, t_{n-2}, \ldots t_0$ are the computed hights on the spine.

In words, *heights* returns a list of heights along the left spine, starting from the root. The relation *add* is monotonic on $\gg$, defined by:

$$y \gg x \quad \equiv \quad \mathit{heights}\ y \trianglerighteq \mathit{heights}\ x$$

where $\trianglerighteq$ is the lexicographic ordering on sequences. This choice does make sense: to ensure monotonicity, we need to optimise not only the whole tree, but also all the subtrees on the left spine. The use of the lexicographic ordering is quite common for such problems, for example, it also features in Knuth's axiomatic theory of convex hull algorithms [52].

Once we know that the monotonicity condition holds, we can apply the greedy theorem to refine the specification. We will prove the monotonicity condition in section 5.1.2 and talk about a further refinement necessary to make it a linear-time algorithm in section 5.1.3. The resulting code is shown in 5.1.4.

5.1.2 Proving the Monotonicity Condition

This section is dedicated to proving that *add* is monotonic on $\gg$. Before we go into the details, we will informally explain how we can maintain the monotonicity. For any two spine trees $y \trianglerighteq x$, no matter how x is extended by *add*, we must find a way to extend y such that the resulting tree is not larger under $\gg$. Suppose the spines of x and y look like in Figure 5.2. Since $x \trianglelefteq y$, either the two spines are all the same, or we can find a position where the values (computed by *height*) on the two spines divert from each other. Call the position r. Assume x was extended at position p. If the position is within the area where the two spines are all the same, as in Figure 5.3, we also extend y at the same position p. If p comes after that area, we can always extend y at position r, as in Figure 5.4.

The rest of this section will be devoted to the actual proof. To start with, we notice that the cost function $x \oplus y = (x \sqcup y) + 1$ has the following useful properties:

$$\text{commuting} \quad : \quad a \oplus b = b \oplus a \tag{5.4}$$

$$\begin{aligned}
\text{strictness} &: \quad a \oplus b > a && (5.5)\\
\text{monotonicity} &: \quad a' \geq a \Rightarrow a' \oplus b \geq a \oplus b && (5.6)\\
\text{bimonotonicity} &: \quad a \oplus c = b \oplus d \ \wedge \ a' \geq a \ \wedge \ a' \geq b' \\
&\quad\ \Rightarrow a' \oplus c \geq b' \oplus d && (5.7)\\
\text{ordering} &: \quad a \oplus b \geq b \oplus c \Rightarrow (a \oplus b) \oplus c \geq a \oplus (b \oplus c) && (5.8)
\end{aligned}$$

The purpose of property (5.4) is just to keep other properties brief. We will make use of (5.5) and (5.7) in this section, while the others will be useful in the next section when we talk about an important refinement. Some of the properties above are rather obvious, but we still give a proof for the last two items.

Proof. For bimonotonicity:

$$\begin{aligned}
& a' \oplus c \geq b' \oplus d \\
\equiv \quad & \{\text{definition of } \oplus\} \\
& a' \sqcup c \geq b' \sqcup d \\
\equiv \quad & \{\text{property of } \sqcup\} \\
& a' \sqcup c \geq b' \ \wedge \ a' \sqcup c \geq d \\
\equiv \quad & \{a' \geq b'\} \\
& a' \sqcup c \geq d \\
\Leftarrow \quad & \{\ a' \geq a\} \\
& a \sqcup c \geq d \\
\Leftarrow \quad & \{\text{property of } \sqcup\} \\
& a \oplus c = b \oplus d
\end{aligned}$$

For ordering:

$$\begin{aligned}
& a \oplus b \geq b \oplus c \\
\equiv \quad & \{\text{definition of } \oplus\} \\
& a+1 \sqcup b+1 \geq b+1 \sqcup c+1 \\
\equiv \quad & \{\text{arithmetic}\} \\
& a+2 \sqcup b+2 \geq b+2 \sqcup c+2 \\
\Rightarrow \quad & \{a+2 \geq a+1 \ \wedge \ c+1 \leq c+2\} \\
& a+2 \sqcup b+2 \sqcup c+1 \geq a+1 \sqcup b+2 \sqcup c+2 \\
\equiv \quad & \{\text{definition of } \oplus\} \\
& (a \oplus b) \oplus c \geq a \oplus (b \oplus c)
\end{aligned}$$

□

In fact, the properties above holds for many cost functions commonly seen. What makes this cost function $a \oplus b = (a \sqcup b) + 1$ unique is the following lemma, which says that adding an element to a tree results in a value greater than both of them. The lemma follows from the definition of $\oplus$.

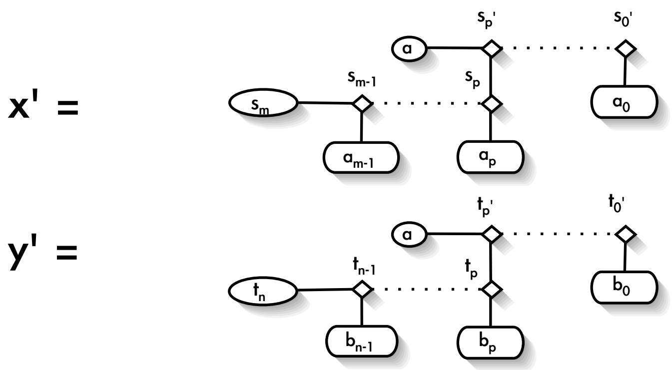

Figure 5.3: How we can extend y when $p < r$.

Lemma 5.3 $((a, x), x') \in add \Rightarrow height\, x' \geq a + 1 \sqcup height\, x$

Now we prove the main proposition.

Proposition 5.4 The relation *add* is monotonic on $\gg$ in the sense that

$$add \cdot (id \times (\gg)) \quad \subseteq \quad (\gg) \cdot add$$

Or restating the monotonicity condition in first-order logic:

$$((a, x), x') \in add \ \wedge \ x \gg y \quad \Rightarrow \quad (\exists y' : ((a, y), y') \in add : x' \gg y')$$

Proof. Suppose the spines of x and y look like in Figure 5.2, with $y \gg x$. Here $x = (s_m, [a_{m-1}, a_{m-2}, \ldots a_0])$ and $y = (t_n, [b_{n-1}, b_{n-2}, \ldots b_0])$. The values $s_{m-1}, s_{m-2}, \ldots s_0$ and $t_{n-1}, t_{n-2}, \ldots t_0$ are computed by *heights*. They are not actually represented in the data structure. Note that the values on each spines are strictly increasing. Furthermore, by bringing in the context, we can assume that $s_m = t_n$. Therefore,

1. either the spines are identical (i.e. $m = n \ \wedge \ \forall i : m \geq i \geq 0 : t_i = s_i$),

2. or we can find the first value on x, starting from the root, strictly greater than the corresponding value on y. That is, exists r, $0 \leq r \leq m \sqcap n$, such that $s_r > t_r$ and $\forall i : r > i \geq 0 : t_i = s_i$.

Assume *add* extends x at position p. We will distinguish between two cases:

- **Case** 1 : $p \leq r$ or when the spine values are the same (i.e. $\forall i : p \geq i \geq 0 : s_i = t_i$). In this case we extend y at the same position p, as in Figure 5.3. We can show that the lexicographic ordering holds by showing the following two properties:

1. $s'_p \geq t'_p$.
By assumption we have $s_p \geq t_p$.It then follows by (5.4) and (5.6) that $s'_p = a \oplus s_p \geq a \oplus t_p = t'_p$.

2. $\forall i : p > i \geq 0 : s'_i \geq t'_i$.
By definition of p we know that $s_{p-1} = t_{p-1}$, or equivalently, $s_p \oplus a_{p-1} = t_p \oplus b_{p-1}$. From (5.5) we know that $s'_p > s_p$. And $s'_p = t'_p$ because $s_p = t_p$. Putting them all together, we start with:

$$\begin{array}{cl} & s_p \oplus a_{p-1} = t_p \oplus b_{p-1} \ \wedge \ s'_p > s_p \ \wedge \ s'_p = t'_p \\ \Rightarrow & \quad \{(5.7)\} \\ & s'_p \oplus a_{p-1} \geq t'_p \oplus b_{p-1} \\ \equiv & \quad \{\text{definition of } s'_{p-1} \text{ and } t'_{p-1}\} \\ & s'_{p-1} = t'_{p-1} \end{array}$$

Also, we have $s_{p-2} = t_{p-2}$ followed by definition of p, $s'_{p-1} > s_{p-1}$ followed by (5.5), and $s'_{p-1} = t'_{p-1}$ proved just now. Therefore, we can apply (5.7) again to prove the next step.

$$\begin{array}{cl} & s_{p-1} \oplus a_{p-2} = t_{p-1} \oplus b_{p-2} \ \wedge \ s'_{p-1} > s_{p-1} \ \wedge \ s'_{p-1} = t'_{p-1} \\ \Rightarrow & \quad \{(5.7)\} \\ & s'_{p-1} \oplus a_{p-2} \geq t'_{p-1} \oplus b_{p-2} \\ \equiv & \quad \{\text{definition of } s'_{p-2} \text{ and } t'_{p-2}\} \\ & s'_{p-2} = t'_{p-2} \end{array}$$

We can repeatedly apply (5.7) this way until we reach

$$\begin{array}{cl} & s_0 = s_1 \oplus a_0 = t_1 \oplus b_0 \geq t_0 \ \wedge \ s'_1 > s_1 \ \wedge \ s'_1 \geq t'_1 \\ \Rightarrow & \quad \{(5.7)\} \\ & s'_0 = s'_1 \oplus a_0 \geq t'_1 \oplus b_0 = t'_0 \end{array}$$

- **Case** 2: $p > r$. In this case we can always extend y at position r, as in Figure 5.4. By the strictness property (5.5) we know that a must be strictly less than s'_{r+1}. Therefore, to retain the lexicographic ordering we just need to show

1. $s'_r \geq t'_r$.
We reason

$$\begin{array}{cl} & s'_r \\ \geq & \quad \{\text{by Lemma 5.3}\} \\ & a + 1 \sqcup s_r \\ \geq & \quad \{s_r > t_r \Rightarrow s_r \geq t_r + 1\} \\ & a + 1 \sqcup t_r + 1 \\ = & \quad \{\text{definition of } \oplus\} \\ & t'_r \end{array}$$

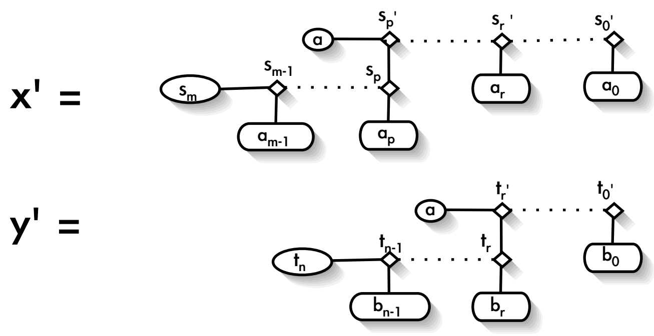

Figure 5.4: How we can extend y when $p \geq r$.

Note that we made use of Lemma 5.3 and the definition of $\oplus$. This is the part of the proof which can not be adapted to other cost functions.

2. $\forall i : p \geq i \geq 0 : s'_i \geq t'_i$.
The same reasoning as in the last case applies.

□

5.1.3 A Further Refinement

Now that we have proved the monotonicity condition, we can apply the greedy theorem:

$$
\begin{array}{ll}
 & bmh \\
= & \quad \{\text{shown in the beginning of Section 5.1}\} \\
 & roll \cdot min\,(\preceq') \cdot \Lambda(foldrn\ add\ one) \\
\supseteq & \quad \{\text{by (5.2)}\} \\
 & roll \cdot min\,(\ll) \cdot \Lambda(foldrn\ add\ one) \\
\supseteq & \quad \{\text{the greedy theorem}\} \\
 & roll \cdot foldrn\,(min\,(\ll) \cdot \Lambda add)\,(min\,(\ll) \cdot \Lambda one)
\end{array}
$$

We still need to further refine the two argument to *foldrn* to functions. Since *one* is a function, Λone always yields a singleton list. Therefore the expression $min\,(\ll) \cdot \Lambda one$ equals *one*. On the other hand, $min\,(\ll) \cdot \Lambda add$ can be implemented as a function by checking through all the possible positions to insert a new node, and choose, say, the lowest position.

In fact, we can do better. We claim that to find the best position to insert node a on spine x of length n, we do not need to actually check through all the $n + 1$ possibilities. A minimum result would always come from extending x at position p, where p is the maximal index satisfying $a \oplus s_p < s_{p-1}$, assuming $s_{-1} = \infty$. We can start from the left of the spine and choose the first p which satisfies the condition.

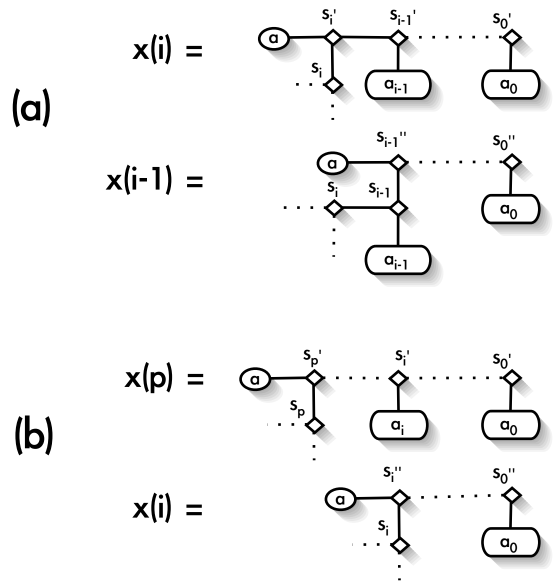

Figure 5.5: Proof for a further refinement.

Let us denote the spine resulting from extending x at position i by $x(i)$. To prove the above claim, we will show that

$$(\forall i \quad : m \geq i > p : \quad x(i) \rhd x(i-1)) \tag{5.9}$$
$$(\forall i \quad : p > i \geq 0 : \quad x(i) \rhd x(p)) \tag{5.10}$$

where $x \rhd y$ denotes that x is strictly greater than y under the reversed lexicographic ordering.

Proof. Figure 5.5(a) compares $x(i)$ and $x(i-1)$. To prove (5.9), we show that $s'_{i-1} \geq s''_{i-1}$.

$$\begin{array}{ll} & m \geq i > p \\ \Rightarrow & \quad \{\text{definition of } p\} \\ & a \oplus s_i \geq s_{i-1} = s_i \oplus a_{i-1} \\ \Rightarrow & \quad \{\text{ordering (5.8)}\} \\ & (a \oplus s_i) \oplus s_{i-1} \geq a \oplus (s_i \oplus s_{i-1}) \\ \equiv & \quad \{\text{definition of } s'_{i-1} \text{ and } s''_{i-1}\} \\ & s'_{i-1} \geq s''_{i-1} \end{array}$$

Then, by the monotonicity property (5.6) , $s'_j \geq s''_j$ for every j between $i-1$ and 0. In addition, we know that $s'_i > a$. Thus we conclude $x(i) \rhd x(i-1)$.

To prove (5.10), look at Figure 5.5(b). We will show that $s'_{p-1} = s_{p-1}$. That will imply $s'_i = s_i$ for all i between $p-1$ and 0, that is, the spine values does not change after position $p-1$. Since $s''_i > s_i$ for any other $x(i)$, we then have $x(i) \rhd x(p)$.

By the definition of p, we know $a \oplus s_p < s_{p-1}$. We reason

$$
\begin{array}{ll}
 & a \oplus s_p < s_{p-1} \\
\equiv & \quad \{\text{definition of } s_{p-1}\} \\
 & a \oplus s_p < s_p \oplus a_{p-1} \\
\equiv & \quad \{\text{definition of } \oplus\} \\
 & a \sqcup s_p < s_p \sqcup a_{p-1} \\
\equiv & \quad \{\text{property of } \sqcup\} \\
 & a < s_p \sqcup a_{p-1} \;\; \wedge \;\; s_p < s_p \sqcup a_{p-1} \\
\equiv & \quad \{s_p = s_p\} \\
 & a < s_p \sqcup a_{p-1} \;\; \wedge \;\; s_p < a_{p-1}
\end{array}
$$

Therefore we have

$$
\begin{array}{ll}
 & s'_{p-1} \\
= & \quad \{\text{definition of } s'_{p-1}\} \\
 & (a \oplus s_p) \oplus a_{p-1} \\
= & \quad \{\text{definition of } \oplus\} \\
 & a + 2 \sqcup s_p + 2 \sqcup a_{p-1} + 1 \\
= & \quad \{\text{arithmetic}\} \\
 & (a \sqcup (s_p \sqcup a_{p-1} - 1)) + 2 \\
= & \quad \{a < s_p \sqcup a_{p-1} \;\Rightarrow\; a \leq s_p \sqcup a_{p-1} - 1\} \\
 & (s_p \sqcup a_{p-1} - 1) + 2 \\
= & \quad \{\text{arithmetic}\} \\
 & (s_p + 1 \sqcup a_{p-1}) + 1 \\
= & \quad \{s_p < a_{p-1} \;\Rightarrow\; s_p + 1 \leq a_{p-1}\} \\
 & (s_p \sqcup a_{p-1}) + 1 \\
= & \quad \{\text{definition of } \oplus \text{ and } s_{p-1}\} \\
 & s_{p-1}
\end{array}
$$

□

5.1.4 The Implementation

As usual, we refine the data structure to avoid recomputing the height of each subtree. A spine is represented by **type** $\mathit{SpineI}\ A = (A \times \mathit{List}(\mathcal{Z} \times \mathit{Tree}\ A))$, annotating each subtree along the spine with its height. Note that the value paired with a tree stands for the height of the subtree (the a_i's in the diagrams), not the value on the spine (the s_i's), because we do not want to update the value all the way to the root each time we attach a new tip.

In this representation, when we are processing the ith subtree on the spine we only have s_i and a_{i-1} at hand. We will now show that the condition we check in each step to decide where to extend the spine, namely $a \oplus s_i < s_{i-1}$, is equivalent to $a < a_{i-1} \wedge s_i < a_{i-1}$. This is not a necessary step but we choose to do so to reflect the close resemblance with the code in [14]:

$$
\begin{array}{cl}
 & a \oplus s_i < s_{i-1} \\
\equiv & \quad \{\text{definition of } s_{i-1}\} \\
 & a \oplus s_i < s_i \oplus a_{i-1} \\
\equiv & \quad \{\text{definition of } \oplus\} \\
 & a \sqcup s_i < s_i \sqcup a_{i-1} \\
\equiv & \quad \{\text{property of } \sqcup\} \\
 & a < s_i \sqcup a_{i-1} \ \wedge \ s_i < s_i \sqcup a_{i-1} \\
\equiv & \quad \{\ s_i = s_i\ \} \\
 & (a < s_i \vee a < a_{i-1}) \ \wedge \ s_i < a_{i-1} \\
\equiv & \quad \{\text{distribution}\} \\
 & (a < s_i \ \wedge \ s_i < a_{i-1}) \vee (a < a_{i-1} \ \wedge \ s_i < a_{i-1}) \\
\equiv & \quad \{\ (a < s_i \ \wedge \ s_i < a_{i-1}) \Rightarrow (a < a_{i-1} \ \wedge \ s_i < a_{i-1})\} \\
 & a < a_{i-1} \ \wedge \ s_i < a_{i-1}
\end{array}
$$

The resulting code is shown in Figure 5.6. Function *minadd* is the result of the refinement described in Section 5.1.3. It's not difficult to see that it is a linear time algorithm, since each call to *minadd* consumes a value, each recursive call to *minsplit* either returns or joins a node, and each node in the resulting tree is built only once.

5.2 Optimal Bracketing Problems

The monotonicity of *add* on the reversed lexicographic ordering, proved in Section 5.1.2, depends on Lemma 5.3, which in turn depends crucially on that particular definition of $\oplus$. For other cost functions, this nice property holds no more. In this section, let us see what we can do if we consider a wider range of cost functions.

We aim to solve problems of the form:

$$obp \quad = \quad min\,(\preceq) \cdot \Lambda \mathit{flatten}^{\circ}$$

where $min\,(\preceq)$ attempts to minimise the value obtained by folding over the tree with cost function $\oplus$:

$$
\begin{array}{lcl}
x \preceq y & \equiv & value\, x \leq value\, y \\
value & = & foldTree\,(\oplus)\; id
\end{array}
$$

```
type SpineI a = (a, [(Int, Tree a)])

bmh :: [Int] -> (Tree Int, Int)
bmh = roll . foldrn minadd one

one a = (a,[])

minadd :: Int -> SpineI Int-> SpineI Int
minadd a (b,xs) = (a, minsplit (tip b) xs)
  where minsplit x [] = [x]
        minsplit x (y:xs) | a < height y
                            && height x < height y = x:y:xs
                          | otherwise = minsplit (bin x y) xs

tip a = (Tip a, a)
bin (x,a) (y,b) = (Bin x y, ht a b)
height = snd

ht a b = (a `max` b) + 1

roll :: SpineI Int -> (Tree Int, Int)
roll (a,x) = foldl bin (tip a) x
```

Figure 5.6: Program for Building Trees with Minimum Height

In other words, we are to solve the optimal bracketing problem with respect to cost function $\oplus$. Given a list of elements, the resulting tree indicates how it should be bracketed.

The cost functions we will investigate into are those satisfying properties (5.4), (5.5) and (5.6). Among the functions belonging to this class are

- $a \oplus b = (a \sqcup b) \times 2$, thus *value* computes $\bigsqcup_{i=1}^{n} a_i \times 2^{d_i}$, where d_i is the depth of element a_i in the tree;
- $a \oplus b = (a + b) \times 2$. Folding it over the tree computes $\sum_{i=1}^{n} a_i \times 2^{d_i}$;
- $(c_1, s_1) \oplus (c_2, s_2) = (c_1 + c_2 + s_1 + s_2, s_1 + s_2)$, computing the pair $(\sum_{i=1}^{n} a_i \times d_i, \sum_{i=1}^{n} a_i)$, which is the cost function used in the optimal alphabetic tree problem;
- $(c_1, (m, l)) \oplus (c_2, (l, n)) = (c_1 + c_2 + (m \times l \times n), (m, n))$, representing the number of multiplications performed to compute the product of a sequence of matrix, together with its dimension,

and many more.

Typically, the optimal bracketing problem is solved with a dynamic programming strategy. Still, we are interested in how the converse-of-a-function theorem suggests another possible approach. As in the previous sections, *flatten* is inverted to *roll* $\cdot$ *foldrn add one*. With properties (5.4), (5.5) and (5.6), it is not difficult to see that *add* is monotonic on the *pairwise* ordering. That is, the tree y in Figure 5.2 is better than x if $m = n$ and $t_i \leq s_i$ for $m \geq i \geq 0$.

However, the pairwise ordering is not connected — not every two spines are comparable. In such cases we do not know which lead to a better solution and we have to keep both of them. The greedy theorem is thus not applicable. We can instead make use of the thinning theorem.

An introduction to the thinning theorem and its implementation will be given in Section 5.2.1 and 5.2.2. The techniques learnt in these two sections will be applied to the optimal bracketing problem in in Section 5.2.3. Unfortunately, the algorithm is exponential in the worst case. Section 5.2.4 explains the reason by comparing it with the traditional dynamic programming approach.

5.2.1 The Thinning Theorem

The greedy theorem is only of use for connected preorders. Otherwise *min* may be partial on non-empty arguments.. For unconnected preorders, the best we can do is to keep all the solutions which we cannot compare, while throwing away those which we know are inferior to some others. This is the motivation of a *thinning* algorithm.

Let $Q :: A \to A$ be a preorder. The relation *thin* $Q :: \mathit{Set}\ A \to \mathit{Set}\ A$ is defined by

$$(xs, ys) \in \mathit{thin}\ Q \quad \equiv \quad (ys \subseteq xs) \wedge (\forall x : x \in xs : \exists y : y \in ys : yQx)$$

That is, ys is a streamlined subset of xs. It cannot be a arbitrary subset, however. The second term ensures that for every x in xs there must be something at least as good in ys. So x itself must survive to ys if nothing else in xs as good.

Given a specification $min\ R \cdot \Lambda(\!|S|\!)_{\mathsf{F}}$, where S is monotonic on an ordering Q which is a sub-relation of R. If Q is connected, we might just go for applying the greedy theorem. Otherwise, rather than keep only the best solution, we will have to keep a set of solutions. In each step we try reduce the size of the set by applying $thin\ Q$ to it, throwing away some useless solutions. This is what the thinning theorem says:

Theorem 5.5 (Thinning Theorem) If S is monotonic on preorder Q, and $Q \subseteq R$, then

$$min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F}{\in})|\!)_{\mathsf{F}} \quad \subseteq \quad min\ R \cdot \Lambda(\!|S|\!)_{\mathsf{F}}$$

Note that $thin\ Q$ is a relation mapping a set of solutions to any set satisfying the constraints in its definition. Neither the thinning theorem nor the definition of $thin$ specify how the set is to be thinned, which is left for the programmer to decide. In other words, we are still left with the work of refining $min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!)_{\mathsf{F}}$ to a function. One possible approach will be discussed in the next section.

5.2.2 Implementing Thinning

In this section we will talk about how to further refine to a function the result delivered by the thinning theorem

$$min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F}{\in})|\!) \tag{5.11}$$

In one extreme, refining $thin\ Q$ to id does satisfy the requirement for $thin\ Q$. In this case nothing gets thinned at all and all the work is just left to $min\ R$. In practice, we hope to throw away as many useless solutions as possible to improve the efficiency. Yet we do not want to perform a full comparison between each pair of elements in the set, whose quadratic overhead usually outweighs the benefit of down-sizing the set of solutions. The programmer thus often faces the dilemma between not thinning enough or wasting too much time thinning.

In this section we will present a common solution: to sort the set of solutions such that we only need to compare adjacent elements. The derivation in this section is a generalisation of the binary thinning theorem in Section 8.3 of [17]. The new theorem allows thinning to be performed before as well as after merging.

First we go from sets to lists. Let $setify :: List\ A \rightarrow Set\ A$ be the function converting a list to a set. It is a lax natural transformation in that for all R:

$$setify \cdot map\ R \quad \subseteq \quad \mathsf{P}R \cdot setify \tag{5.12}$$

Here the functor P is generalised to a relator. Given a relation $R :: A \rightarrow B$, the relation $\mathsf{P}R$ has type $Set\ A \rightarrow Set\ B$, defined by:

$$\begin{aligned}(x, y) \in \mathsf{P}R \quad \equiv \quad & a \in x \Rightarrow (\exists b : b \in y : (a, b) \in R) \ \wedge \\ & b \in y \Rightarrow (\exists a : a \in x : (a, b) \in R)\end{aligned}$$

Property (5.12) can be proved by defining $setify = \Lambda\delta_{List}$, and using the naturality of δ_{List}. In fact, (5.12) is true for any type functor with membership. For the reader's reference, it is proved in Appendix A.

The relation $min\ R :: Set\ A \rightarrow A$ has an obvious functional counterpart $minlist\ R :: List\ A \rightarrow A$, which should satisfy

$$minlist\ R \quad \subseteq \quad min\ R \cdot setify \tag{5.13}$$

To simulate $\Lambda\mathsf{F}{\in} :: \mathsf{F}(Set\ A) \rightarrow Set\ A$, we need a function $cplist_{\mathsf{F}} :: \mathsf{F}(List\ A) \rightarrow List\ A$ satisfying

$$setify \cdot cplist_{\mathsf{F}} \quad \subseteq \quad \Lambda\mathsf{F}{\in} \cdot \mathsf{F}setify \tag{5.14}$$

We hope to represent the set of solutions as a list such that we only need to compare adjacent elements, which is a linear time operation. We will sort the solution set according to some ordering P, hoping that it can bring comparable elements together. Sorting a set into a list with respect to a connected preorder P is specified by

$$sort\ P \quad = \quad (ordered\ P)? \cdot setify^{\circ} \tag{5.15}$$

where $ordered\ P$ is a predicate yielding true for a list if it is sorted with respect to P. Given $sort\ P$, we assume the existence of a function *thinlist*, the list counterpart of *thin*, satisfying

$$thinlist\ Q \cdot sort\ P \quad \subseteq \quad sort\ P \cdot thin\ Q \tag{5.16}$$

We will not overspecify either *minlist* or *thinlist*, but just assume that they can be implemented as functions taking time linear in the size of the input list. Problem specific choices for P and *thinlist* will be discussed in the next section.

It is also useful to see how *sort* interacts with *min* and *cplist*. Given (5.13), (5.14), (5.15), $ordered\ P \subseteq id$, and that *setify* is a function, we can prove that

$$minlist\ R \cdot sort\ P \quad \subseteq \quad min\ R \tag{5.17}$$

$$setify \cdot cplist_{\mathsf{F}} \cdot \mathsf{F}(sort\ P) \quad \subseteq \quad \Lambda\mathsf{F}{\in} \tag{5.18}$$

Finally, we also need the property of *thin* below, which states that if we are about to thin a union of sets, we can also thin each them separately before thinning them again as a whole:

$$thin\ Q \cdot union \cdot \mathsf{P}(thin\ Q) \quad \subseteq \quad thin\ Q \cdot union \tag{5.19}$$

This property is also proved in Appendix A.

Finished with all the properties we need, the aim is to refine (5.11) to a function. We start with:

$$\begin{array}{ll}
 & min\ R \cdot (\![thin\ Q \cdot \Lambda(S \cdot \mathsf{F}{\in})]\!) \\
\supseteq & \quad \{(5.17)\} \\
 & minlist\ R \cdot sort\ P \cdot (\![thin\ Q \cdot \Lambda(S \cdot \mathsf{F}{\in})]\!) \\
\supseteq & \quad \{\text{fold fusion. See below.}\} \\
 & minlist\ R \cdot (\![T]\!)
\end{array}$$

We want to use the fold fusion theorem to derive T. The fusion condition is

$$\mathit{sort}\ P \cdot \mathit{thin}\ Q \cdot \Lambda(S \cdot \mathsf{F}{\in}) \quad \supseteq \quad T \cdot \mathsf{F}(\mathit{sort}\ P)$$

To construct T, we derive:

$$\begin{array}{ll}
& \mathit{sort}\ P \cdot \mathit{thin}\ Q \cdot \Lambda(S \cdot \mathsf{F}{\in}) \\
= & \quad \{\text{since } \Lambda(S \cdot T) = \mathsf{E}S \cdot \Lambda T \text{ and } \mathsf{E}S = \mathit{union} \cdot \mathsf{P}(\Lambda S)\} \\
& \mathit{sort}\ P \cdot \mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\Lambda S) \cdot \Lambda\mathsf{F}{\in} \\
\supseteq & \quad \{\text{by } (5.19)\} \\
& \mathit{sort}\ P \cdot \mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q \cdot \Lambda S) \cdot \Lambda\mathsf{F}{\in} \\
\supseteq & \quad \{\text{by } (5.16)\} \\
& \mathit{thinlist}\ Q \cdot \mathit{sort}\ P \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q \cdot \Lambda S) \cdot \Lambda\mathsf{F}{\in} \\
\supseteq & \quad \{\text{by } (5.18)\} \\
& \mathit{thinlist}\ Q \cdot \mathit{sort}\ P \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q \cdot \Lambda S) \cdot \mathit{setify} \cdot \mathit{cplist}_{\mathsf{F}} \cdot \mathsf{F}(\mathit{sort}\ P) \\
\supseteq & \quad \{\text{by } (5.12)\} \\
& \mathit{thinlist}\ Q \cdot \mathit{sort}\ P \cdot \mathit{union} \cdot \mathit{setify} \cdot \mathit{map}\ (\mathit{thin}\ Q \cdot \Lambda S) \cdot \mathit{cplist}_{\mathsf{F}} \cdot \mathsf{F}(\mathit{sort}\ P) \\
= & \quad \{\text{let } \mathit{merge} \text{ satisfy } \mathit{merges}\ P \cdot \mathit{map}\ \mathit{setify}^{\circ} = \mathit{sort}\ P \cdot \mathit{union} \cdot \mathit{setify}\} \\
& \mathit{thinlist}\ Q \cdot \mathit{merges}\ P \cdot \mathit{map}\ (\mathit{setify}^{\circ} \cdot \mathit{thin}\ Q \cdot \Lambda S) \cdot \mathit{cplist}_{\mathsf{F}} \cdot \mathsf{F}(\mathit{sort}\ P) \\
\supseteq & \quad \{\text{since } \mathit{setify}^{\circ} \supseteq \mathit{sort}\ P\} \\
& \mathit{thinlist}\ Q \cdot \mathit{merges}\ P \cdot \mathit{map}\ (\mathit{sort}\ P \cdot \mathit{thin}\ Q \cdot \Lambda S) \cdot \mathit{cplist}_{\mathsf{F}} \cdot \mathsf{F}(\mathit{sort}\ P) \\
\supseteq & \quad \{\text{by } (5.16)\} \\
& \mathit{thinlist}\ Q \cdot \mathit{merges}\ P \cdot \mathit{map}\ (\mathit{thinlist}\ Q \cdot \mathit{sort}\ P \cdot \Lambda S) \cdot \mathit{cplist}_{\mathsf{F}} \cdot \mathsf{F}(\mathit{sort}\ P)
\end{array}$$

The function *merges* will have to merge a list of sorted lists into one, which can be done in time $O(\log k \times n + k)$ for k lists of length n. Fig. 5.7 shows one possible implementation of *merge* and two of *thinlist* satisfying the specifications above. More possibilities can be found in Section 8.3 of [17].

Usually we can implement ΛS such that it generates the solutions in the correct order. That is, we can implement $f = \mathit{sort}\ P \cdot \Lambda S$. In summary, in this section we have proved the following theorem:

Theorem 5.6 Let R and Q be connected preorder such that R is connected and $Q \subseteq R$. Also given are functions *minlist*, $\mathit{cplist}_{\mathsf{F}}$, *thinlist* and *merge* characterised by the following axioms that for all connected preorder X:

$$\begin{array}{rcl}
\mathit{minlist}\ X & \subseteq & \mathit{min}\ X \cdot \mathit{setify} \\
\mathit{setify} \cdot \mathit{cplist}_{\mathsf{F}} & \subseteq & \Lambda\mathsf{F}{\in} \cdot \mathsf{F}\mathit{setify} \\
\mathit{merges}\ X \cdot \mathit{map}\ \mathit{setify}^{\circ} & = & \mathit{sort}\ X \cdot \mathit{union} \cdot \mathit{setify}
\end{array}$$

If we can find a connected preorder P such that

$$\mathit{thinlist}\ Q \cdot \mathit{sort}\ P \quad \subseteq \quad \mathit{sort}\ P \cdot \mathit{thin}\ Q$$

```
merges :: (a -> a -> Bool) -> [[a]] -> [a]
merges p = foldr mrg []
  where mrg x [] = x
        mrg [] y = y
        mrg (a:x) (b:y)
            | a `p` b = a : mrg x (b:y)
    | otherwise = b : mrg (a:x) y

thinlist :: (a -> a -> Bool) -> [a] -> [a]
thinlist q [] = []
thinlist q [x] = [x]
thinlist q (a:b:x)
      | a `q` b = thinlist q (a:x)
      | b `q` a = thinlist q (b:x)
      | otherwise  = a : thinlist q (b:x)

thinlist' :: (a -> a -> Bool) -> [a] -> [a]
thinlist' q = foldr (bump q) []
bump q (a,[]) = [a]
bump q (a,(b:x)) | a `q` b = a:x
                 | b `q` a = b:x
         | otherwise = a:b:x
```

Figure 5.7: Possible Implementations of *merges* and *thinlist*

we then have:

$$\begin{aligned} & \mathit{minlist}\ R \cdot (\![\mathit{thinlist}\ Q \cdot \mathit{merges}\ P \cdot \mathit{map}\,(\mathit{thinlist}\ Q \cdot f) \cdot \mathit{cplist}_{\mathsf{F}}]\!) \\ & \quad \subseteq \mathit{min}\ R \cdot (\![\mathit{thin}\ Q \cdot \Lambda(S \cdot \mathsf{F}\!\in)]\!) \end{aligned}$$

where $f = \mathit{sort}\ P \cdot \Lambda S$.

Therefore, before solving specification in the form of $\mathit{min}\, R \cdot \Lambda(\![S]\!)$ with the thinning theorem, we will need to find a sub-relation Q of R on which S is monotonic, and a relation P grouping together the partial solutions comparable under Q.

5.2.3 Solving the Optimal Bracketing Problem

Back to the optimal bracketing problem. We have chosen Q to be the pairwise ordering. Now we need a suitable ordering P with which we sort the list of solutions such that comparable elements are brought to adjacent positions. The choice of P can dramatically change the efficiency of the program. After some experiments we choose

$$\begin{aligned} xs\ P\ ys \quad \equiv \quad & \mathit{value}\,(\mathit{roll}\ xs) < \mathit{value}\,(\mathit{roll}\ ys) \vee \\ & (\mathit{value}\,(\mathit{roll}\ xs) = \mathit{value}\,(\mathit{roll}\ ys) \wedge \mathit{length}\,(\mathit{snd}\ xs) \geq \mathit{length}\,(\mathit{snd}\ ys)) \end{aligned}$$

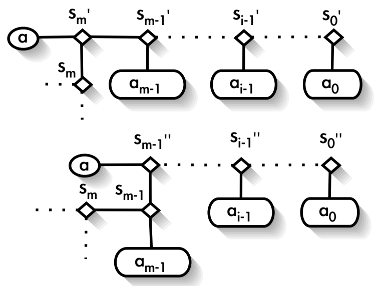

Figure 5.8: Creating trees and thinning at the same time.

That is, we choose to sort the list of solutions firstly in *ascending cost*, then in *descending spine length*. The implementation of *thinlist* also has strong influence on the efficiency. In our experience, the first *thinlist* in Figure 5.7 outperforms all the others for this particular problem. A possible reason is that the first element of the resulting list is available earlier.

Another little refinement can be done. We can implement Λadd such that it generates the extended spines in descending spine length. Figure 5.8 compares two extended spine trees which differ in length only by one. The longer spine has no chance to be better than the shorter one under the pairwise ordering: by the strictness property (5.5), a is strictly smaller than s'_m. If s''_{m-1} is also smaller than or equal to s'_{m-1}, we know immediately that the longer one is worse because they will both be joined to the same list of a_is rightwards. We can thus drop it immediately. Otherwise they become incomparable and we have to keep them both. In this manner we can fuse $thinlist\ Q \cdot sort\ P \cdot \Lambda add$ into one function. If we further assume that property (5.8) holds, comparing s''_{m-1} and s'_{m-1} can be done by comparing $a \oplus s_m$ and $s_m \oplus a_{m-1}$.

The code is shown in Figure 5.9. The function *obp* is the main program. The type *SpineI* in Figure 5.6 has been reused. To ease the task of merging and thinning, we extend *SpineI* to *SpineV*, attaching to each spine its length and the list of values (the *s*s) on the spine. The function *spineval*, parametrised by a cost function, generates the list of values on the spine of the given spine tree.

The function *thinaddI* is the fusion of $thinlist\ Q \cdot sort\ P \cdot \Lambda add$ which, as described above, performs spine extension and thinning together. The function *thinaddV* merely acts as a wrapper, unwrapping and wrapping a spine tree with its length and spine values. Predicates *pwleq* and *acdl* (which stands for "ascending cost and descending length") are the orderings we use in the thinning and merging phases, respectively. Finally, since the list of solutions has been sorted in ascending cost, the outermost *minlist* can be simply replaced by *head*.

Unfortunately, the worst case running time of this algorithm is exponential. For such an example, try the cost function $a \oplus b = (a + b) \times 2$. The sequence $a_1, a_2, \ldots a_n$ with $a_1 = 1$ and a_i greater than the maximal cost of trees built from $a_1 \ldots a_{i-1}$ will force the thinning phrases to process lists of exponential sizes, no matter how we sort or thin the list. This unfortunately complicates matters by making the elements to the right of the

```
type SpineV a = (SpineI a, Int, [a])
type CostFun a = (a, a) -> a

obp :: Ord a => CostFun a -> [a] -> Tree a
obp op =
   rollV . head . foldrn (step op) base
  where base a = [(one a,0,[])]
        step op = thinlist pwleq . merges acdl .
                  map(thinaddV op) . cplist

thinaddV :: Ord a => CostFun a -> (a,SpineV a) -> [SpineV a]
thinaddV op (a,(x,_,_)) = map (sval op) (thinaddI op (a,x))
  where sval op (a,x) = ((a,x), length x+1, spineval op (a,x))

thinaddI :: Ord a => CostFun a -> (a, SpineI a) -> [SpineI a]
thinaddI op (a,(b,x)) = add (b,Tip b) x
  where add x [] = [(a,[x])]
        add x (y:ys)
          | op (a,fst x) >= op (fst x,fst y) = add (bin(x,y)) ys
          | otherwise = (a,x:y:ys) : add (bin (x,y)) ys
        bin ((b,x),(c,y)) = (op (b,c), Bin (x,y))

spineval :: CostFun a -> SpineI a -> [a]
spineval op (a,x) = foldl step [a] x
  where step y@(a:_) t = op (a,fst t) : y

rollV :: SpineV a -> Tree a
rollV (x,_,_) = roll x

(_,_,x) `pwleq` (_,_,y) = and $ zipWith (<=) x y

(_,m,x) `acdl` (_,n,y) = head x < head y ||
                         (head x == head y && m >= n)

cplist :: (a,[b]) -> [(a,b)]
cplist (a,x) = [(a,b) | b <- x]
```

Figure 5.9: The code solving the general optimal bracketing problem.

spine so heavy that every attempt to deepen an element will incur some penalty.

5.2.4 A Comparison with Dynamic Programming

How did the algorithm become exponential? Is it possible, say, by choosing a better Q, to expose more possibility of thinning and thereby make the algorithm polynomial? In this section we will compare our approach to the traditional dynamic programming approach to optimal bracketing problems. We claim that the inefficiency lies not in the ordering Q, but in the data structure representing the set of solutions.

Many optimisation problems can be solved in two dual approaches. We can either specify it in terms of a fold and solve it using a thinning strategy, or write the specification in terms of an unfold and turn to dynamic programming. In most cases, the two approaches represent very similar computations. We also have some examples for which the thinning approach performs slightly better.

Take, for example, the 0-1 knapsack problem for example, which in many textbooks constitutes a typical example of dynamic programming: we maintain a table, one entry for each weight, to store the best value we can pack within that weight. The algorithm proceeds by adding the items one by one, updating the table, until all the items has been considered.

Alternatively, we can also apply the thinning approach to the 0-1 knapsack problem. A packing is a subsequence of the list of items. The subsequences can be generated by a fold. The ordering in the thinning phrase is chosen such that one packing is worse than another when it is neither more valuable nor lighter. Maintaining the list of solutions has the same effect as maintaining the table. The restriction of the ordering on the weight ensures that we only keep the best packing for each weight. This approach represent a very similar computation to the dynamic programming approach. Even better, the list may have a smaller size than the table because we do not keep packings for weights that are not constructible. The thinning approach is therefore slightly more efficient than the dynamic programming approach. In [65], de Moor has developed a program for 0-1 knapsack problem which outperforms all other algorithms.

Back to the optimal bracketing problem. We also expect the ordering we have chosen to bear some resemblance to what we did in the traditional approach. In the dynamic programming strategy, we compute the best subtree for each segment of the input list, and then choose a best combination among them. What about the thinning approach?

The pairwise ordering does guarantee that all the subtrees along the spine are optimal. A subtree, *once it is rolled*, can survive on the spine only if it is the best one among all the *rolled* subtrees having the same set of leafs. This is illustrated in Figure 5.10. Assume $(b \oplus c) \oplus d$ is the better way to bracket b, c and d than $b \oplus (c \oplus d)$. Both x and y will remain in the set, before $(b \oplus c) \oplus d$ is rolled. However, after being extended with a, x' will definitely be better than y'. This resembles the dynamic programming way of keeping only the optimal subtree for each segments. We therefore believe that pairwise ordering is the right ordering to choose.

The problem lies in the data representation. See Figure 5.11. All the tips b, c, d, e and f standing alone is the best subtree representing themselves. The best subtree for

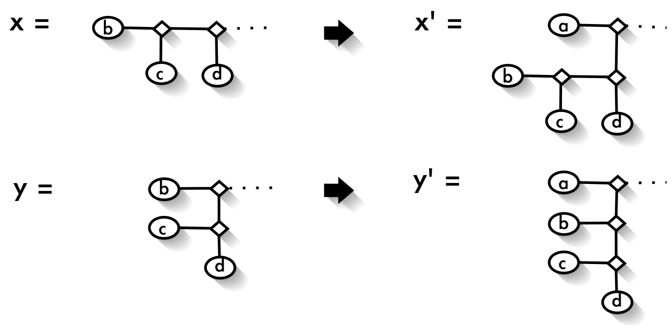

Figure 5.10: All rolled subtrees must be optimal. In this figure, y' will be ruled out by x'.

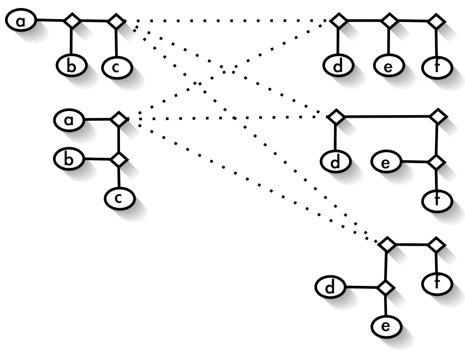

Figure 5.11: How the number of solutions become exponential

any two elements, say b, c, is of course $bin\,(tip\,b, tip\,c)$. We do not know what element will be appended to the spine later, so we do have reason to keep them all, in case we may need one of them. In the dynamic programming approach, the optimal subtrees are kept in separate entries in the table. In our approach, however, due to our data representation, the two trees for a, b, c must be repeated for each instance of d, e, f. We have to keep all their combinations. That is where the exponential number of trees come from.

It is possible to refine the data structure of the set of solutions to avoid repeating the tails. Eventually, we will probably arrive at an algorithm very similar to the traditional dynamic programming approach. It may help to clarify the relationship between thinning and dynamic programming. This is subject to further research.

5.3 The Generic Greedy Theorem

This section deviates from our theme of inverse functions and investigate into the interplay between minimisation and folds. In the previous sections we have been dealing with problems of this form:

$$min\ R \cdot \Lambda(\![S]\!)$$

where the fold generates all the solutions and $min\ R$ chooses one among them. However, there are occasions when it is not possible to have the fold returning just the set of valid candidates. For instance, the fold would have to construct the solutions with the help of the banana-split transform [17, Chapter 3] and return the candidates in a pair. The problem might thus be be specified as:

$$bmin\ R \cdot (min\ R \times min\ R) \cdot (\![S]\!)$$

where S has type $\mathsf{F}(Set\ A \times Set\ A) \rightarrow (Set\ A \times Set\ A)$ and $bmin\ R :: (A \times A) \rightarrow A$ chooses a preferred member from a pair. In general, the solutions might need to be classified into many kinds in order to generate new ones. That leads us to consider problems of this general form:

$$min_{\mathsf{G}}\ R \cdot \mathsf{G}(min\ R) \cdot (\![S]\!)$$

where S has type $\mathsf{FG}(Set\ A) \rightarrow \mathsf{G}(Set\ A)$ and $min_{\mathsf{G}}\ R$ chooses a minimum member from a G-structure.

In this section we will discuss a generalisation of the greedy and theorem to promote $\mathsf{G}(min\ R)$ into a fold. We will first discuss, as a motivating example, a generalisation of the famous maximum segment sum problem to rose trees in Section 5.3.1. We will then present in Section 5.3.2 our extended greedy theorem and see the theorem in action in Section 5.3.3. and 5.3.4. We finally make a comparison between the generalised theorem and the ordinary thinning approach in Section 5.3.5.

5.3.1 The Maximum Subtree Problem

Assume the following datatype definition of a rose tree:

$$\textbf{data}\ Rose \quad = \quad null \mid node\ (\mathcal{Z} \times List\ Rose)$$

It is the type defined by the base functor $\mathsf{F}X = 1 + \mathcal{Z} \times List\ X$. The fold function coming with it is defined by:

$$\begin{array}{lcl} foldRose\ f\ e\ null & = & e \\ foldRose\ f\ e\ (node\ (a, xs)) & = & f\ (a, map\ (foldRose\ f\ e\ xs)) \end{array}$$

The maximum subtree problem is to find a subtree whose sum of the values in the nodes is maximal. By a “subtree”, we mean a contiguous set of nodes such that for any two marked nodes, all the nodes along the paths to their common parents must be chosen as well. For example, for the tree shown in Figure 5.12, the nodes with bold borders indicates its maximum subtree. The node with value -1 under 8 need not be chosen. In general, the subtree need not start from the root.

This problem is a generalisation of the famous *maximum segment sum* [33] problem to rose trees, mentioned as an example in [13], the very first paper that introduced generic

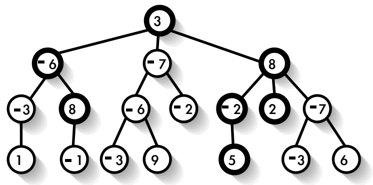

Figure 5.12: An example of the maximum subtree problem.

programming. In [76], Sasano and Hu calculated a linear-time algorithm for a family of such *optimal marking problems* – given the name because nodes in the tree are marked according to a given predicate. Bird [16] further showed how their algorithm is actually an instance of the thinning strategy. In this section we are going to show that the same problems can also be seen as a greedy algorithm, derivable using a generalised greedy theorem.

One possible approach to tackle this problem is to follow the example of maximum segment sum on lists: to decompose segments into inits of tails. To apply the same approach to trees, one would need a notion of initial and tail segments for rose trees. Another possibility is to follow the theme of previous chapters: to express the problem in the form of $max\ R \cdot \Lambda cont$, where $cont :: Rose \rightarrow Rose$ relates a rose tree to one of its legal marking – a contiguous subtree. If *cont* can be written as a fold, we can then apply either the greedy or the thinning theorem.

However, *cont* can not be a fold of that type. In each iteration, it can not simply return a subtree, since we need more information about what kind of subtree it is. Instead, function *conts* which returns the set of all the contiguous subtrees of a tree can be written as a fold over rose trees by further dividing the subtrees into two kinds: those which *starting from the root*, and those which *do not include the root*, and return them in a tuple:

$$
\begin{array}{lcl}
conts & :: & Rose \rightarrow (Set\ Rose \times Set\ Rose) \\
conts & = & foldRose\ \langle \Lambda(incl \cdot (\in \times \in)), \Lambda(excl \cdot (\in \times \in))\rangle\ (\{null\}, \{null\})
\end{array}
$$

$$
\begin{array}{lcl}
incl, excl & :: & \mathcal{Z} \times List\ (Rose \times Rose) \rightarrow Rose \\
incl\ (a, xs) & = & node\ (a, subseq\ (map\ fst\ xs)) \\
excl\ (a, xs) & = & fst\ (\delta_{List}\ xs) \,\Box\, snd\ (\delta_{List}\ xs)
\end{array}
$$

Relations *incl* and *excl* of type $(\mathcal{Z} \times List\ (Rose \times Rose)) \rightarrow Rose$ return a contiguous subtree that includes or excludes the root of a given tree, respectively. The relation $subseq :: List\ A \rightarrow List\ A$, having the usual definition:

$$
subseq \;=\; foldr\ (cons \cup snd)\ [\,]
$$

relates a list to one of its subsequences. Membership relations in general have been introduced in Section 4.6.1. Here δ_{List} can be thought of as a variant of $\in$ defined on list: it non-deterministically relates a list to any of its members. The relation *incl* picks any combination of subtrees (represented by the expression $subseq\,(map\,fst\,xs)$), all of them including the roots of those direct children, and attaches them to the current node, thus forming a new tree starting from the root. The relation $excl$, on the other hand, just combines the results of previous calls to $incl$ and $excl$. A fold defined this way is usually called a *mutumorphism*, as the two (or more) relations are mutually defined in terms of each other.

The problem specification can then be written as

$$\begin{array}{lcl} mstree & = & bmax\ R \cdot (max\ R \times max\ R) \cdot conts \\ x\ R\ y & \equiv & val\ x \leq val\ y \\ val & = & foldRose\ sumF\ 0 \\ \quad \textbf{where} & & sumF\,(a,x) \;=\; a + sumlist\ x \end{array}$$

where function $bmax\ R :: (A \times A) \rightarrow A$ takes a pair and returns the greater one with respect to R. Function $sumlist$, as the name suggests, sums up a given list of numbers.

We thus need a theorem enabling us to refine $(max\ R \times max\ R) \cdot conts$, where $conts$ is a fold returning a pair of sets.

Before proceeding to the next section, let us rewrite the definition of $conts$ in the banana-bracket notation, as it is easier to relate to the general theorem to be presented in the next section. Relations $incl$ and $excl$ are also rephrased in point-free style:

$$\begin{array}{lcl} conts & = & (\!|\langle \Lambda([null, incl] \cdot \mathsf{F}(\in \times \in)), \Lambda([null, excl] \cdot \mathsf{F}(\in \times \in))\rangle|\!)_{\mathsf{F}} \\ incl & = & node \cdot (id \times subseq \cdot map\,fst) \\ excl & = & (fst \cup snd) \cdot \delta_{List} \cdot snd \end{array}$$

5.3.2 Introducing the Theorem

It is not difficult to see that both $incl$ and $excl$ are monotonic with respect to R in the sense that

$$incl \cdot (id \times map\,(R \times R)) \;\subseteq\; R \cdot incl \tag{5.20}$$

$$excl \cdot (id \times map\,(R \times R)) \;\subseteq\; R \cdot excl \tag{5.21}$$

In effect, that means we only need to keep the best result returned by $incl$ and $excl$, respectively. Denoting $[null, incl]$ by S and $[null, excl]$ by T, it follows from (5.20) and (5.21) that:

$$\langle S, T\rangle \cdot \mathsf{F}(R \times R) \;\subseteq\; (R \times R) \cdot \langle S, T\rangle$$

In general, when faced with a fold defined in terms of two relations S and T as below:

$$(max\ R \times max\ R) \cdot (\!|\langle \Lambda(S \cdot \mathsf{F}(\in \times \in)), \Lambda(T \cdot \mathsf{F}(\in \times \in))\rangle|\!)_{\mathsf{F}} \tag{5.22}$$

We need a theorem enabling us to turn it into

$$(\!|\langle max\ R \cdot \Lambda S, max\ R \cdot \Lambda T\rangle|\!)_{\mathsf{F}} \tag{5.23}$$

given that the monotonic condition below holds.

$$\langle S, T\rangle \cdot \mathsf{F}(R \times R) \;\subseteq\; (R \times R) \cdot \langle S, T\rangle \tag{5.24}$$

$\mathsf{G}A = (A \times A)$	G generalised
$(max\ R \times max\ R)$ $::\ (Set\ A \times Set\ A) \to (A \times A)$	$\mathsf{G}(max\ R)$ $::\ \mathsf{G}(Set\ A) \to \mathsf{G}A$
$\langle \mathsf{E}S, \mathsf{E}T \rangle$ $::\ Set(\mathsf{F}(A \times A)) \to (Set\ A \times Set\ A)$	h $::\ Set(\mathsf{FG}A) \to \mathsf{G}(Set\ A)$
$\langle \Lambda S, \Lambda T \rangle$ $::\ \mathsf{F}(A \times A) \to (Set\ A \times Set\ A)$	$h \cdot wrap$ $::\ \mathsf{FG}A \to \mathsf{G}(Set\ A)$
$\langle \Lambda(S \cdot \mathsf{F}(\in \times \in)), \Lambda(T \cdot \mathsf{F}(\in \times \in)) \rangle$ $::\ \mathsf{F}(Set\ A \times Set\ A) \to (Set\ A \times Set\ A)$	$h \cdot \Lambda \mathsf{FG} \in$ $::\ \mathsf{FG}(Set\ A) \to \mathsf{G}(Set\ A)$

Table 5.1: Comparison of the case when G is a product and when G is generalised to an arbitrary regular functor.

In fact, we can prove a more general theorem not only for the product, but for any regular functors. Before we can state the theorem, however, we need to find out this generic form. We start from finding out the general form of the algebra in the fold. Supposing both S and T have type $\mathsf{F}(A \times A) \to A$, we write down the algebra in (5.22) and its type below:

$$\langle \Lambda(S \cdot \mathsf{F}(\in \times \in)), \Lambda(T \cdot \mathsf{F}(\in \times \in)) \rangle \quad :: \quad \mathsf{F}(Set\ A \times Set\ A) \to (Set\ A \times Set\ A)$$

A sub term $\Lambda \mathsf{F}(\in \times \in)$ can be factored out of the split, resulting in:

$$\langle \mathsf{E}S, \mathsf{E}T \rangle \cdot \Lambda \mathsf{F}(\in \times \in) \quad :: \quad \mathsf{F}(Set\ A \times Set\ A) \to (Set\ A \times Set\ A)$$

We then abstract the product to a general functor G and replace $\langle \mathsf{E}S, \mathsf{E}T \rangle$ by a function h (which covers $\langle \mathsf{E}S, \mathsf{E}T \rangle$ as a special case because E delivers functions from sets to sets), yielding:

$$h \cdot \Lambda \mathsf{FG}{\in} \quad :: \quad \mathsf{FG}(Set\ A) \to \mathsf{G}(Set\ A)$$

Now $h \cdot \Lambda \mathsf{FG}{\in}$ is an F-algebra with carrier $\mathsf{G}(Set\ A)$. Note that h has type $Set\,(\mathsf{FG}A) \to \mathsf{G}(Set\ A)$, which is generalised from $Set\,(\mathsf{F}(A \times A)) \to (Set\ A \times Set\ A)$, the type of $\langle \mathsf{E}f, \mathsf{E}g \rangle$.

The generic form of expression (5.22) is therefore

$$\mathsf{G}(max\ R) \cdot (\!|h \cdot \Lambda \mathsf{FG}{\in}\,|\!)_{\mathsf{F}} \quad :: \quad \mathsf{T} \to \mathsf{G}A$$

where T is the datatype defined by base functor F. The generic counterpart of $\langle \Lambda S, \Lambda T \rangle$ is $h \cdot wrap$ because

$$\begin{array}{cl}
 & h \cdot wrap \\
= & \quad \{\text{since we choose } h = \langle \mathsf{E}S, \mathsf{E}T \rangle\ \} \\
 & \langle \mathsf{E}S, \mathsf{E}T \rangle \cdot wrap \\
= & \quad \{\text{since functions distributes into splits and } \mathsf{E}R \cdot wrap = \Lambda R\} \\
 & \langle \Lambda S, \Lambda T \rangle
\end{array}$$

The expression (5.23) thus generalises to:

$$(\!|\mathsf{G}(\mathit{max}\ R) \cdot h \cdot \mathit{wrap}|\!)_{\mathsf{F}}$$

Since h now operates on set of values, we want the monotonic condition to be:

$$h \cdot \mathsf{EFG}R \quad \subseteq \quad \mathsf{GE}R \cdot h$$

Finally, halfway in the proof we will find ourselves needing this property:

$$h \cdot \mathit{subset} \quad = \quad \mathsf{G}\mathit{subset} \cdot h$$

where *subset* relates a set of one of its subsets. The above condition ensures that h is made of functions lifted by existential functor E — that the result of applying h to a subset can also be obtained by taking the subset after applying h to the whole set.

Table 5.1 summarises the conversion from pairs to the generic form. Now we can present our main theorem in this section:

Theorem 5.7 (The Generic Greedy Theorem) Let G be a regular functor. If a function $h :: \mathit{Set}\,(\mathsf{FG}A) \rightarrow \mathsf{G}(\mathit{Set}\ A)$ satisfies

$$h \cdot \mathit{subset} \ = \ \mathsf{G}\mathit{subset} \cdot h \tag{5.25}$$

and is monotonic with respect to preorder R in the sense that

$$h \cdot \mathsf{EFG}R \subseteq \mathsf{GE}R \cdot h \tag{5.26}$$

then

$$(\!|\mathsf{G}(\mathit{max}\ R) \cdot h \cdot \mathit{wrap}|\!)_{\mathsf{F}} \quad \subseteq \quad \mathsf{G}(\mathit{max}\ R) \cdot (\!|h \cdot \Lambda\mathsf{FG}{\in}\ |\!)_{\mathsf{F}}$$

The proof of the theorem will be presented in Appendix B.

5.3.3 Application

For the contiguous tree marking problem, we take $\mathsf{G}A = A \times A$ and

$$h \quad = \quad \langle \mathsf{E}[\mathit{null}, \mathit{incl}], \mathsf{E}[\mathit{null}, \mathit{excl}] \rangle$$

The monotonic condition (5.26) specialises to:

$$\begin{array}{l} \langle \mathsf{E}[\mathit{null}, \mathit{incl}], \mathsf{E}[\mathit{null}, \mathit{excl}] \rangle \cdot \mathsf{EF}(R \times R) \\ \quad \subseteq \ (\mathsf{E}R \times \mathsf{E}R) \cdot \langle \mathsf{E}[\mathit{null}, \mathit{incl}], \mathsf{E}[\mathit{null}, \mathit{excl}] \rangle \end{array}$$

which follows directly from (5.20) and (5.21), as shown below:

$$\begin{array}{ll} & \langle \mathsf{E}[\mathit{null}, \mathit{incl}], \mathsf{E}[\mathit{null}, \mathit{excl}] \rangle \cdot \mathsf{EF}(R \times R) \\ \subseteq & \quad \{\text{splits, } \mathsf{E} \text{ a functor}\} \\ & \langle \mathsf{E}([\mathit{null}, \mathit{incl}] \cdot \mathsf{F}(R \times R)), \mathsf{E}([\mathit{null}, \mathit{excl}] \cdot \mathsf{F}(R \times R)) \rangle \\ = & \quad \{\text{coproduct, letting } \mathsf{F}_1 A = \mathit{id} \times \mathit{List}\ A \text{ for brevity}\} \end{array}$$

$$
\begin{array}{ll}
& \langle \mathsf{E}[null, incl \cdot \mathsf{F}_1(R \times R)], \mathsf{E}[null, excl \cdot \mathsf{F}_1(R \times R)]\rangle \\
\subseteq & \quad \{\text{by (5.20) and (5.21)}\} \\
& \langle \mathsf{E}[null, R \cdot incl], \mathsf{E}[null, R \cdot excl]\rangle \\
= & \quad \{null = R \cdot null, \text{ coproduct}\} \\
& \langle \mathsf{E}(R \cdot [null, incl]), \mathsf{E}(R \cdot [null, excl])\rangle \\
= & \quad \{\text{product}\} \\
& (\mathsf{E}R \times \mathsf{E}R) \cdot \langle \mathsf{E}[null, incl], \mathsf{E}[null, excl]\rangle
\end{array}
$$

We can thus apply the generic greedy theorem to refine our specification:

$$
\begin{array}{ll}
& mstree \\
= & \quad \{\text{definition}\} \\
& bmax\ R \cdot (max\ R \times max\ R) \cdot \\
& \quad (\!|\langle \Lambda([null, incl] \cdot \mathsf{F}(\in \times \in)), \Lambda([null, excl] \cdot \mathsf{F}(\in \times \in))\rangle|\!) \\
\supseteq & \quad \{\text{generic greedy theorem}\} \\
& bmax\ R \cdot (\!|\langle max\ R \cdot \Lambda[null, incl], max\ R \cdot \Lambda[null \cdot excl]\rangle|\!)
\end{array}
$$

We now have a chance to optimise $max\ R \cdot \Lambda[null, incl]$ and $max\ R \cdot \Lambda[null, excl]$ separately. Since post composition and the Λ operator distribute into joins, and $max\ R \cdot \Lambda null$ equals $null$, we are left with optimising $max\ R \cdot \Lambda incl$ and $max\ R \cdot \Lambda excl$. The optimisation will in turn make use of the (ordinary) greedy theorem. Notice that the generic greedy theorem itself does not provide immediate enhancement in speed. Its role is to promote max into the fold, so that further chances of refinement can be exposed.

We will demonstrate how $max\ R \cdot \Lambda incl$ can be refined using the greedy theorem. The case for $max\ R \cdot \Lambda excl$ is similar. Using (5.1), $max\ R \cdot \Lambda incl$ is equivalent to

$$node \cdot max\ (node^\circ \cdot R \cdot node) \cdot \Lambda(id \times subseq \cdot map\ fst)$$

We need some mechanism to promote $max\ (node^\circ \cdot R \cdot node)$ into Λ. The following property of min, given as an exercise in [17], becomes handy:

$$\langle max\ R \cdot \Lambda S, max\ Q \cdot \Lambda T\rangle \quad \subseteq \quad max\ (R \times Q) \cdot \Lambda\langle S, T\rangle \tag{5.27}$$

To make use of it, however, we need to convert $node^\circ \cdot R \cdot node$ into a product. We calculate

$$
\begin{array}{ll}
& (node\ (a, xs))\ R\ (node\ (b, ys)) \\
\equiv & \quad \{\text{definition of } R\} \\
& (val\ (node\ (a, xs)))\ R\ (val\ (node\ (b, ys))) \\
\equiv & \quad \{\text{expand } val \cdot node, \text{ let } sumvals = sumlist\ map\ val\} \\
& a + sumvals\ xs \leq b + sumvals\ ys \\
\Leftarrow & \quad \{\text{arithmetic}\} \\
& a \leq b \wedge sumvals\ xs \leq sumvals\ ys \\
\equiv & \quad \{\text{functors}\} \\
& (a, xs)\ ((\leq) \times (sumvals^\circ \cdot (\leq) \cdot sumvals))\ (b, ys)
\end{array}
$$

Let $Q = sumvals^\circ \cdot (\leq) \cdot sumvals$, we have just shown that $((\leq) \times Q) \subseteq node^\circ \cdot R \cdot node$. We then derive:

$$
\begin{array}{cl}
& max\,(node^\circ \cdot R \cdot node) \cdot \Lambda(id \times subseq \cdot map\,fst) \\
\supseteq & \quad \{\text{since } node^\circ \cdot R \cdot node \supseteq ((\leq) \times Q),\ (5.2)\} \\
& max\,((\leq) \times Q) \cdot \Lambda(id \times subseq \cdot map\,fst) \\
\supseteq & \quad \{(5.27)\} \\
& (max\,(\leq) \cdot \Lambda id \times max\,R' \cdot \Lambda(subseq \cdot map\,fst)) \\
= & \quad \{\text{since } max\,Q \cdot \Lambda id = id\} \\
& (id \times max\,Q \cdot \Lambda(subseq \cdot map\,fst)) \\
\supseteq & \quad \{\text{fusing } subseq \cdot map\,snd \text{ to a fold, applying the greedy theorem}\} \\
& (id \times foldr\,(max\,Q \cdot \Lambda((cons \cdot (fst \times id)) \cup snd))\,[\,]) \\
= & \quad \{\text{further simplification}\} \\
& (id \times foldr\,(((t, \_), x) \mapsto \mathbf{if}\ val\,t \leq 0\ \mathbf{then}\ x\ \mathbf{else}\ t : x)\,[\,])
\end{array}
$$

The corresponding Haskell code is shown in Figure 5.13. After using a tupling transformation to pair a tree together with its value, the program runs in time linear to the size of the tree.

5.3.4 The Maximum Sub-Rectangle Problem

Consider a similar problem on a different datatype. This time we are given a matrix represented by a list of lists:

$$\mathbf{newtype}\ Rect = List_1\,(List\ \mathcal{Z})$$

We also assume that the lists are all of the same length, that is, they represent a rectangle. The *maximum sub-rectangle* problem is to find a sub-rectangle within the given rectangle such that the sum of the values in the rectangle is maximal. As an example, the maximum sub-rectangle in Figure 5.14 is the area surrounded by the lines. The problem appeared in various texts, for example, [11]. In [45], Hu, Iwasaki and Takeichi talked about the same problem in the context of parallel programming.

To solve the rectangle problem we can follow a route similar to that in the previous section. One possible approach is to choose $\mathsf{G}A = A \times A$, and express the problem in terms of a fold on non-empty lists returning a pair of sets of rectangles:

$$
\begin{array}{lcl}
msrect & = & bmax\,R \cdot (max\,R \times max\,R) \cdot rects \\
rects & = & foldrn\,\langle \Lambda(incl \cdot \mathsf{F}_1(\in \times \in), \Lambda(excl \cdot \mathsf{F}_1(\in \times \in)\rangle\,\langle \mathsf{P}wrap \cdot segs, const\,\emptyset\rangle
\end{array}
$$

such that *incl* relates a rectangle to a sub-rectangle including some part of the first row, while *excl* relates it to a sub-rectangle excluding the first row. The function $segs :: List\,A \rightarrow Set\,(List\,A)$ returns the set of all segments of the given list. Both relations have type $\mathsf{F}_1(Rect \times Rect) \rightarrow Rect$, where $\mathsf{F}_1 A = id \times A$, the second component of the base functor defining non-empty lists.

```
data Rose = Null | Node Int [Rose]  deriving Show

foldRose f e Null = e
foldRose f e (Node a us) = f a (map (foldRose f e) us)

null = (Null,0)

mstree :: Rose -> Rose
mstree = fst . bmaxR . foldRose step (null,null)
  where step a us = (incl a us, excl a us)
        incl a us = (Node a (foldr op1 [] us), a + foldr op2 0 us)
          where op1 ((t,v),_) us = if v <= 0 then us else (t:us)
                op2 ((t,v),_) n  = if v <= 0 then n  else (v+n)
        excl _ [] = null
        excl _ us = (bmaxR . (maxR `cross` maxR) . unzip) us

maxR x = foldr1 (curry bmaxR) x
bmaxR ((t,a), (u,b)) | a >= b = (t,a)
                     | otherwise = (u,b)

(f `cross` g) (a,b) = (f a, g b)
```

Figure 5.13: Program for the maximum subtree problem

0	1	−8	6	−5	−4	3	1	−6	−7
7	2	−9	1	−1	−3	7	9	5	2
−9	−8	0	−9	8	0	1	−2	7	−3
−8	−6	9	2	−9	8	−9	0	3	−9
3	−4	4	6	1	−6	4	−1	−5	−9

Figure 5.14: An example of the maximum sub-rectangle problem.

This time *incl* is not monotonic on R. It is instead monotonic on a sub-relation of R under which two rectangles are comparable only when their first row start and end in the same columns. That means we cannot keep only one optimal instance among the solutions returned by *incl*. Instead we have to keep the optimal rectangle for each starting and ending position. We will therefore have to appeal to some generic thinning theorem. The problem becomes one thinning algorithm (the *incl* part) and one greedy algorithm (the *excl* part) running together. The size of the solution set we have to keep for the thinning part is at most $(n \times (n+1))/2$. We thus have a cubic time algorithm.

Alternatively, we could have chosen to manipulate the $(n \times (n+1))/2$ instances in the solution set explicitly by choosing $\mathsf{G}A = \mathit{List}\ A \times A$. We rephrase the problem as:

$$\begin{array}{lcl} \mathit{msrect} & = & \mathit{max}_{\mathit{snoc}}\ R \cdot (\mathit{map}\ (\mathit{max}\ R) \times \mathit{max}\ R) \cdot \\ & & \mathit{foldrn}\ \langle \Lambda(\mathit{incl} \cdot \mathsf{F}_1\mathsf{G}{\in}), \Lambda(\mathit{excl} \cdot \mathsf{F}_1\mathsf{G}{\in})\rangle\ \langle \mathit{segs}, \mathit{const}\ \emptyset\rangle \end{array}$$

where $\mathit{max}_{\mathit{snoc}}\ R\,(x,a) = \mathit{max}\ R\,(\mathit{setify}\ x \cup \{a\})$.

This time *incl* has to process a whole list of rectangles instead of just one. The list contains the optimal solutions so far for each starting-ending positions. As the base case, we need *segs* to have type $\mathit{List}\ A \rightarrow \mathit{List}\,(\mathit{List}\ A)$ and return the *list* of all segments of the given list in a fixed order. We use a list of lists rather than a set of lists because the ordering helps: we store the best solutions for each segments in the list in a fixed order, therefore we can then zip the list with the existing list of solutions to generate the new solutions, without having to keep track of the actual positions. The relations *incl* and *excl* are defined below:

$$\begin{array}{lcl} \mathit{incl} & :: & \mathsf{F}_1(\mathsf{G}\mathit{Rect}) \rightarrow \mathit{List}\ \mathit{Rect} \\ \mathit{incl}\,(x,(\mathit{xss},\mathit{ys})) & = & \mathit{zipWith}\ \mathit{above}\,(\mathit{segs}\ x,\ \mathit{xss}) \\ \quad \mathbf{where}\quad \mathit{above}(x,\mathit{xs}) & = & [x] \,\square\, x : \mathit{xs} \end{array}$$

$$\begin{array}{lcl} \mathit{excl} & :: & \mathsf{F}_1(\mathsf{G}\mathit{Rect}) \rightarrow \mathit{Rect} \\ \mathit{excl}\,(x,(\mathit{xss},\mathit{ys})) & = & \delta_{\mathit{List}}\ \mathit{xss} \,\square\, \mathit{ys} \end{array}$$

Here *incl* takes a list of rectangles *xss* and, for each *xs* in *xss*, the relation *incl* has the choice between either adding a new row x above it, or just throw it away.

Now we can apply the generic greedy theorem. Fusing $(\mathit{map}\ (\mathit{max}\ R) \times \mathit{max}\ R)$ into the fold, we get:

$$\mathit{foldrn}\ \langle \mathit{map}\ (\mathit{max}\ R) \cdot \Lambda\mathit{incl}, \mathit{max}\ R \cdot \Lambda\mathit{excl}\rangle\ \langle \mathit{segs}, \mathit{const}\ \emptyset\rangle$$

Further calculations needs to be done to simplify $\mathit{map}\ (\mathit{max}\ R) \cdot \Lambda\mathit{incl}$ and $\mathit{max}\ R \cdot \Lambda\mathit{excl}$. We will then find it becomes $1 + (n \times (n+1))/2$ greedy algorithms running in parallel! Some routine calculations results in the code in Figure 5.15. It is a cubic-time algorithm.

The algorithm derived in [45] was in the same spirit, except for that their algorithm, aiming at parallel execution, was based on join-lists rather than cons-lists. The resulting algorithm has a $O((\log n)^2)$ parallel complexity. The *Almost Fusion* theorem generalises the fold fusion theorem in a way different from the generic greedy theorem: it addresses fusion in general rather than just minimals, while it dealt with only tuples rather than arbitrary functors.

```
type Rect = [[Int]]

msrect :: Rect -> Rect
msrect = fst . bmaxR . (maxR `cross` id) . foldrn step base
  where base = (map (wrap `fork` sum) . segs) `fork` const ([],0)
        step x (xss,ys) = (incl x (xss,ys), excl x (xss,ys))
        incl x (xss,_) = zipWith above (segs x) xss
           where above x (xs,v) | v <= 0    = ([x],sum x)
                                | otherwise = (x:xs, sum x + v)
        excl _ (xss,xs) = bmaxR (maxR xss, xs)

segs = concat . map tails . inits
```

Figure 5.15: A program solving the maximum sub-rectangle problem.

5.3.5 Comparison

We have seen two applications of the generic greedy theorem. Why do we need this beast? How does it compare to the ordinary thinning theorem?

The thinning approach of solving 0-1 knapsack problem, as presented in [65] and [17], beats the traditional dynamic programming solution both in efficiency and clarity. In [16], Bird derived an elegant generic algorithm solving marking problems, where he separated the phrases of extending a solution and shrinking the solution set. The derived program runs nicely in polynomial time for problems defined on datatypes with a polynomial base functor — that is, functors defined in terms of the identity functor, product, and coproduct. Lists and binary trees, among many useful datatypes, both belong to this category.

When it comes to non-polynomial based datatypes such as rose trees, however, the algorithm takes exponential time. Take the maximum subtree problem for example. To compute the best tree starting from the root, the first step in a thinning phrase, namely $\mathsf{F}(id, \in)$ (or $cplist_{\mathsf{F}}$ in the implementation), automatically generate an exponential number of possibilities because there is a linear number of children and all the children can be either taken or dropped. This is not necessary. We all know that the best policy is simply take all the children yielding positive value.

The solution in [76] and [16] is to transform non-polynomial based datatypes into their polynomial based embeddings in a systematic way. When it comes to *Rect*, for which the relationship between the polynomial and the non-polynomial types are more obscure, we lose the clarity.

The problem was that, during the separation of phases, we lost the chance to perform customised refinement for each problem. What we need is a mechanism enabling us to promote $max\ R$ into the fold, like what was done in the previous sections. We can then further refine $max\ R \cdot \Lambda incl$ and $max\ R \cdot \Lambda excl$ in a way customised for each definitions of *incl* and *excl*, eliminating the exponential number of choices we encounter when we treat them uniformly. This is what the generic greedy theorem enables us to do.

Chapter 6

Countdown: A Case Study

We have seen many techniques to invert a function. One may look into its definition and invert it via the compositional approach. When the function happens to be a fold, the result would be an unfold. The resulting program thus runs in a top-down manner. One may also attempt to invert it as a fold, thereby computing the results bottom-up. Yet another bottom-up approach, similar to that in Section 4.7.3, will be discussed later in this chapter. It works by turning a top-down specification to a closure. In practice it is difficult to predict which one will turn out to be better, as each of them might expose different chances for further optimistaion, or interact with the rest of the algorithm in different ways.

A very nice illustration of all these ideas is provided by Hutton's functional pearl on the Countdown problem [46]. In the Countdown problem one is given a bag of positive integers, and the aim is to construct an arithmetic expression out of some of these numbers to get as close as possible to a given target integer. The name of the problem derives from a popular British television programme in which contestants are given six source numbers and a time limit of 30 seconds to construct a solution. In the Computing Laboratory, the Countdown problem has been used as a topic for a programming competition held by Spivey[82], who also conducted some initial research on the problem. In [46], Hutton developed a straightforward but, at least for bags of size six, reasonably effective top-down algorithm that nevertheless repeated a lot of work. In a concluding section he proposed the investigation of a bottom-up algorithm to see whether it would be superior. It turns out to be an attractive problem for comparing the various approaches to function inversion, as well as being ideal both for presenting and illustrating some general theory about tabulation, thinning and closure algorithms, we take up Hutton's proposal in this chapter. Specifically, we will derive about half a dozen algorithms for Countdown, both top-down and bottom-up, and compare their performance.

6.1 The Specification

To specify Countdown we define the following datatype for expressions:

$$\begin{aligned}
\textbf{data}\, \mathit{Expr} &= \mathit{val}\, \mathcal{Z} \mid \mathit{app}\, (\mathit{Op} \times (\mathit{Expr} \times \mathit{Expr})) \\
\textbf{data}\, \mathit{Op} &= \mathit{add} \mid \mathit{sub} \mid \mathit{mul} \mid \mathit{div}
\end{aligned}$$

The fold function for *Expr* is naturally defined by:

$$\begin{aligned}
\mathit{foldExpr}\, f\, g\, (\mathit{val}\, n) &= g\, n \\
\mathit{foldExpr}\, f\, g\, (\mathit{app}\, (\mathit{op}, (l, r))) &= f\, (\mathit{op}, (\mathit{foldExpr}\, f\, g\, l, \mathit{foldExpr}\, f\, g\, r))
\end{aligned}$$

We call the bag of numbers used in an expression it *basis*. It can be defined in terms of *foldExpr* as below:

$$\begin{aligned}
\mathit{basis} &:: \mathit{Expr} \to \mathit{Bag}\, \mathcal{Z} \\
\mathit{basis} &= \mathit{foldExpr}\, \mathit{wrap}\, (\mathit{plus} \cdot \mathit{snd})
\end{aligned}$$

The function *wrap* is overloaded to bags, and *plus* takes the bag-union of two non-empty bags. Note that *wrap* and *plus* have disjoint ranges: *wrap* returns a singleton bag, while *plus* returns a bag of size at least two.

The function *value* evaluates an expression and is defined by

$$\begin{aligned}
\mathit{value} &:: \mathit{Expr} \to \mathcal{Z} \\
\mathit{value} &= \mathit{foldExpr}\, \mathit{id}\, \mathit{apply}
\end{aligned}$$

The subsidiary function *apply* applies an operator to two numbers and is defined in the obvious way, with *div* interpreted as integer division. According to the rules of the game, whenever division is used, the denominator must exactly divide the numerator. Also, the result of a subtraction must always be positive. We therefore build only those expressions that are valid according to the rules. The validity of expressions is determined by the coreflexive $\mathit{valid} :: \mathit{Expr} \to \mathit{Expr}$ defined by $\mathit{valid} = \mathit{foldExpr}\, \mathit{val}\, \mathit{app}'$, where app' is a partial function defined by:

$$\begin{aligned}
\mathit{app}' &:: (\mathit{Op} \times (\mathit{Expr} \times \mathit{Expr})) \to \mathit{Expr} \\
\mathit{app}' &= \mathit{app} \cdot \mathit{legal}?
\end{aligned}$$

and the boolean-valued function *legal* is defined by

$$\begin{aligned}
\mathit{legal}\, (\mathit{add}, (x, y)) &\equiv \mathit{true} \\
\mathit{legal}\, (\mathit{sub}, (x, y)) &\equiv (\mathit{value}\, x > \mathit{value}\, y) \\
\mathit{legal}\, (\mathit{mul}, (x, y)) &\equiv \mathit{true} \\
\mathit{legal}\, (\mathit{div}, (x, y)) &\equiv (\mathit{value}\, x)\, \textbf{mod}\, (\mathit{value}\, y) == 0
\end{aligned}$$

We know that *valid* is a coreflexive because $\mathit{app}' \subseteq \mathit{app}$, and as a consequence we have $\mathit{valid} \subseteq \mathit{foldExpr}\, \mathit{val}\, \mathit{app} = \mathit{id}$.

To continue with the specification, let $\mathit{subbag} :: \mathit{Bag}\, \mathcal{Z} \to \mathit{Bag}\, \mathcal{Z}$ be a relation that takes a non-empty bag to one of its non-empty sub-bags, including possibly the bag itself. The expression $\mathit{valid} \cdot \mathit{basis}^{\circ} \cdot \mathit{subbag}$ therefore takes a bag of numbers to a valid expression mentioning only the numbers in the bag. The problem is to find an expression whose value is as close to a chosen target number as possible. Hence we define

$$\begin{array}{lcl} \mathit{countdown} & :: & \mathcal{Z} \to \mathit{Bag}\,\mathcal{Z} \to \mathit{Expr} \\ \mathit{countdown}\; n & = & \mathit{min}\; R_n \cdot \Lambda(\mathit{valid} \cdot \mathit{basis}^\circ \cdot \mathit{subbag}) \end{array}$$

where the parameterised ordering R_n is defined by $u\, R_n\, v \equiv \mathit{dist}\; n\; u \leq \mathit{dist}\; n\; v$ and the function dist by $\mathit{dist}\; x\; y = \mathit{abs}\,(x - y)$.

6.2 The Top-Down Approach

Let us first review Hutton's top-down solution to the Countdown problem. The first thing to say is that the problem reminds one of bin packing[60], but is much more complicated. The main complication is that the principle of optimality does *not* hold for Countdown: expressions that are closest to the target are not built out of subexpressions that are closest to the target. This is due to the presence of operators like minus, division and multiplication. Consequently the main focus is on how to compute the full set of possible expressions. True, once one finds an expression whose value matches the target exactly, further computation can be abandoned. We will return to this point after seeing how to transform $\Lambda(\mathit{valid} \cdot \mathit{basis}^\circ \cdot \mathit{subbag})$ into an implementable form.

The second thing to say is that with the given definition of *valid* there is a great deal of redundancy in the set of expressions one can build. For example, $x + y$ and $y + x$ are essentially the same expression, as are $(x - y) + z$ and $x + (z - y)$ (and so on), and x, $x \times 1$ and $x/1$. On the other hand, while possessing the same value, $(7 + 4) + 3$ and 7×2 are essentially different expressions. One approach to restraining the redundancy is to strengthen the definition of *valid* with the aim of excluding all but a single representative of each set of essentially similar expressions. In fact, Hutton uses a definition of *valid* based on the following definition of *legal*:

$$\begin{array}{lcl} \mathit{legal}\,(\mathit{add}, (x, y)) & \equiv & \mathit{value}\; x \leq \mathit{value}\; y \\ \mathit{legal}\,(\mathit{sub}, (x, y)) & \equiv & \mathit{value}\; x > \mathit{value}\; y \\ \mathit{legal}\,(\mathit{mul}, (x, y)) & \equiv & \mathit{value}\; x \neq 1 \wedge \mathit{value}\; y \neq 1 \wedge \mathit{value}\; x \leq \mathit{value}\; y \\ \mathit{legal}\,(\mathit{div}, (x, y)) & \equiv & \mathit{value}\; y \neq 1 \wedge (\mathit{value}\; x)\, \mathbf{mod}\, (\mathit{value}\; y) = 0 \end{array}$$

Stronger still is the following definition:

$$\begin{array}{lcl} \mathit{legal}\,(\mathit{add}, (x, y)) & \equiv & \mathit{value}\; x \leq \mathit{value}\; y \wedge \mathit{not\; sub}\; x \wedge \mathit{not\; add}\; y \wedge \mathit{not\; sub}\; y \\ \mathit{legal}\,(\mathit{sub}, (x, y)) & \equiv & \mathit{value}\; x > \mathit{value}\; y \wedge \mathit{not\; sub}\; x \wedge \mathit{not\; sub}\; y \\ \mathit{legal}\,(\mathit{mul}, (x, y)) & \equiv & 1 < \mathit{value}\; x \leq \mathit{value}\; y \wedge \mathit{not\; div}\; x \wedge \mathit{not\; mul}\; y \wedge \mathit{not\; div}\; y \\ \mathit{legal}\,(\mathit{div}, (x, y)) & \equiv & 1 < \mathit{value}\; y \wedge (\mathit{value}\; x)\, \mathbf{mod}\, (\mathit{value}\; y) = 0 \\ & & \wedge \mathit{not\; div}\; x \wedge \mathit{not\; div}\; y \end{array}$$

where $\mathit{not\; op}\,(\mathit{val}\; n) = \mathit{true}$ and $\mathit{not\; op1}\,(\mathit{app\; op2}\; x\; y) = (\mathit{op1} \neq \mathit{op2})$. One can add yet more refinements, and it is orthogonal to the developments in the sections to come. However, it is quite tricky to devise a test that ensures a single representative of essentially similar expressions. An alternative and more systematic approach to the eliminate redundant expressions is described in Section 6.3.4 below.

Turning to the derivation of the top-down algorithm. Observe that the function *basis* is defined as a fold, so its converse $basis^\circ$ is an unfold. The expression $valid \cdot basis^\circ$ is thus a hylomorphism. Define $expr = valid \cdot basis^\circ$, we have:

$$
\begin{array}{cl}
 & expr \\
= & \quad \{\text{hylomorphism}\} \\
 & \mu(X \mapsto [val, app'] \cdot (id + (id \times (X \times X))) \cdot [wrap, plus \cdot snd]^\circ) \\
= & \quad \{\text{join and coproduct}\} \\
 & \mu(X \mapsto val \cdot wrap^\circ \cup app' \cdot (id \times (X \times X)) \cdot (plus \cdot snd)^\circ) \\
= & \quad \{\text{since } snd \cdot (f \times R) = R \cdot snd\} \\
 & \mu(X \mapsto val \cdot wrap^\circ \cup app' \cdot snd^\circ \cdot (X \times X) \cdot plus^\circ)
\end{array}
$$

The resulting expression has a simple reading: to compute *expr* on a singleton bag, just apply *val* to its sole inhabitant. To compute *expr* on a bag of size at least two, split the bag into two, recursively compute *expr* on each sub-bag, choose an operator and, if the result is a valid expression, combine them. The relation snd° invents an operator to fill in. Note that $wrap^\circ$ is defined only on singleton bags and $plus^\circ$ only on bags of size at least two, so it is legitimate to interpret $\cup$ as a conditional. Also note that *expr* is in fact the *unique* fixed-point of the mapping. The reason is that $(fst \cup snd) \cdot plus^\circ$, which has type $Bag\ A \to Bag\ A$, is an inductive relation.

Fusing Λ into the fixed-point using an approach similar to that in Section 3.1, we obtain the following recursive definition for $exprs = \Lambda expr$, in which $ops = \{add, sub, mul, div\}$:

$$
\begin{array}{lllll}
exprs\ xb & \mid\ singleton\ xb & = & \{val\,(wrap^\circ\ xb)\} & \\
 & \mid\ otherwise & = & \{app\,(op, (e_1, e_2)) & \mid\ (yb, zb) \leftarrow (\Lambda plus^\circ)\ xb, \\
 & & & & \quad e_1 \leftarrow exprs\ yb, e_2 \leftarrow exprs\ zb, \\
 & & & & \quad op \leftarrow ops, legal\,(op, (e_1, e_2))\}
\end{array}
$$

Hence, using Λ-composition, we have:

$$countdown\ n \quad = \quad min\ R_n \cdot union \cdot \mathsf{P} exprs \cdot \Lambda subbag$$

We are now at a stage where the overall structure of the program has been determined, and it seems that we are just one step away from an implementation. However, there are still a number of decisions one can make in this last step, some of them having phenomenal influence on the efficiency. We will explore some of the alternatives in the incoming sections.

6.2.1 Choosing a Representation for Bags

The specification of *countdown* involves both bags and sets, and in the implementation one might choose to represent both these types by lists. The approach taken in [46] can be seen as to represent a bag of integers by a list of *all* its permutations. The function $\Lambda subbag$ is therefore represented by a function *subbags* defined by

$$
\begin{array}{lcl}
subbags & :: & List\ A \to List\,(List\ A) \\
subbags\ xs & = & [zs \mid ys \leftarrow subseqs\ xs, zs \leftarrow perms\ ys]
\end{array}
$$

The redundancy in the representation of bags means that the function $\Lambda plus^\circ$ in the definition of *exprs* can be implemented by $splits :: List\ A \to List\ ([List\ A, List\ A])$ that splits a list into two non-empty sublists in all possible ways, mentioned in Section 3.1. Finally, the set comprehension in the definition of *exprs* can be replaced by a list comprehension. As usual, we can pair expressions with their values to avoid repeated *value* computations in the evaluation of *legal* and $min\ R_n$.

Another choice is to represent a bag of integers by a single list in ascending order. Then $\Lambda subbag$ can be implemented simply as *subseqs*. Under this representation, *plus* will be implemented by a function *merge* which merges two sorted lists into one. Consequently, to implement $\Lambda plus^\circ$, we have to "unmerge" a sorted list into two sorted lists. For example, the list $[1, 2, 3]$ can be decomposed in six ways:

$$([1], [2, 3]), ([2], [1, 3]), ([3], [1, 2]), ([1, 2], [3]), ([1, 3], [2]), ([2, 3], [1])$$

Half of these unmerges are pairwise swaps of the other half. We can define one half by

$$\begin{array}{lcl} unmerges & :: & List\ A \to List\ (List\ A \times List\ A) \\ unmerges\ [\,] & = & [\,] \\ unmerges\ [a] & = & [\,] \\ unmerges\ (a : x) & = & ([a], x) : concat\ [[(a : y, z), (y, a : z)] \mid (y, z) \leftarrow unmerges\ x] \end{array}$$

and then add in the pairwise swaps. A better idea is to take account of the swaps in the definition of *exprs* by strengthening the definition of *legal* so that at most one of $app\ (op, (e_1, e_2))$ and $app\ (op, (e_2, e_1))$ is a valid expression. This is certainly the case when *op* is *div* or *sub*, and it causes no harm to extend it to *add* and *mul* because both operations are commutative. Thus we can define

$$\begin{array}{lll} exprs\ xb & \mid & singleton\ xb\ =\ \{val\ (wrap^\circ\ xb)\} \\ & \mid & otherwise\ = \\ & & \quad union\{combine\ (op, (e_1, e_2)) \mid (yb, zb) \leftarrow (\Lambda plus^\circ)\ xb, \\ & & \qquad\qquad e_1 \leftarrow exprs\ yb, e_2 \leftarrow exprs\ zb, \\ & & \qquad\qquad op \leftarrow ops\} \end{array}$$

$$\begin{array}{lll} combine\ (op, (e_1, e_2)) & \mid & legal\ (op, (e_1, e_2))\ =\ \{app\ (op, (e_1, e_2))\} \\ & \mid & legal\ (op, (e_2, e_1))\ =\ \{app\ (op, (e_2, e_1))\} \\ & \mid & otherwise\ =\ \{\,\} \end{array}$$

and implement $\Lambda plus^\circ$ by *unmerge*, *union* by *concat*, and set comprehension by list comprehension.

Yet another slight modification on *legal* can be done. The reason for computing the set of valid expressions in the first place is so that we can apply $min\ R_n$ to it. As we said at the start, this relation cannot be fused with the generation of expressions since the optimality principle does not hold. However, once we have found an expression with value n there is no point in continuing to construct further expressions. We can therefore strengthen the definition of *legal* to exclude expressions that contain a subexpression with value n. This gives a modest performance improvement of about 12% for the naive top-down approach. We will use this definition of *legal* through out this chapter.

6.2.2 Building Trees First

Surprisingly, the most time and space efficient non-memoising implementation of the pure top-down approach comes from introducing, rather than eliminating, an intermediate datatype. Recall the tip-valued binary tree and its fold function:

$$\textbf{data}\ \mathit{Tree} \ = \ \mathit{tip}\ \mathcal{Z} \mid \mathit{bin}\,(\mathit{Tree} \times \mathit{Tree})$$

$$\begin{array}{lcl} \mathit{foldTree} & :: & ((A \times A) \to A) \to (\mathcal{Z} \to A) \to \mathit{Tree} \to A \\ \mathit{foldTree}\ f\ g\ (\mathit{tip}\ n) & = & f\ n \\ \mathit{foldTree}\ f\ g\ (\mathit{bin}\,(x,y)) & = & g\,(\mathit{foldTree}\ f\ g\ x, \mathit{foldTree}\ f\ g\ y) \end{array}$$

We introduce a function $\mathit{basisT} :: \mathit{Tree} \to \mathit{Bag}\ \mathcal{Z}$ and a relation $\mathit{toExpr} :: \mathit{Tree} \to \mathit{Expr}$ such that

$$\mathit{basis} \ = \ \mathit{basisT} \cdot \mathit{toExpr}^{\circ}$$

The function basisT returns the basis of a tree rather than an expression, while the relation toExpr maps a Tree to an Expr by filling in arbitrary operators at the nodes. They can be defined respectively as:

$$\begin{array}{lcl} \mathit{basisT} & = & \mathit{foldTree}\ \mathit{plus}\ \mathit{wrap} \\ \mathit{toExpr} & = & \mathit{foldTree}\,(\mathit{app} \cdot \mathit{snd}^{\circ})\ \mathit{val} \end{array}$$

Consequently, we have:

$$\begin{array}{lcl} \mathit{countdown}\ n & = & \mathit{min}\ R_n \cdot \Lambda(\mathit{valid} \cdot \mathit{toExpr} \cdot \mathit{basisT}^{\circ} \cdot \mathit{subbag}) \\ & = & \mathit{min}\ R_n \cdot \mathsf{E}(\mathit{valid} \cdot \mathit{toExpr}) \cdot \mathsf{E}(\mathit{basisT}^{\circ}) \cdot \Lambda \mathit{subbag} \\ & = & \mathit{min}\ R_n \cdot \mathit{union} \cdot \mathsf{P}(\Lambda \mathit{valid} \cdot \mathit{toExpr}) \cdot \mathit{union} \cdot \mathsf{P}(\Lambda \mathit{basisT}^{\circ}) \cdot \Lambda \mathit{subbag} \end{array}$$

Let $\mathit{trees} = \Lambda \mathit{basisT}^{\circ}$. After some reasoning, one can come up with the following recursive definition for trees:

$$\begin{array}{lllcl} \mathit{trees}\ \mathit{xb} & \mid & \mathit{singleton}\ \mathit{xb} & = & \{\mathit{tip}\,(\mathit{wrap}^{\circ}\ \mathit{xb})\} \\ & \mid & \mathit{otherwise} & = & \{\mathit{bin}\,(y,z) \mid (\mathit{yb},\mathit{zb}) \leftarrow (\Lambda \mathit{plus}^{\circ})\ \mathit{xb}, \\ & & & & \qquad y \leftarrow \mathit{trees}\ \mathit{yb}, z \leftarrow \mathit{trees}\ \mathit{zb}\} \end{array}$$

The function $\mathit{toValidExprs} = \Lambda(\mathit{valid} \cdot \mathit{toExpr})$ converts a tree into a set of valid expressions by inserting operators in all legal ways:

$$\begin{array}{lcl} \mathit{toValidExprs}\,(\mathit{tip}\ m) & = & \{\mathit{val}\ m\} \\ \mathit{toValidExprs}\,(\mathit{bin}\,(t_1,t_2)) & = & \mathit{union}\,\{\mathit{combine}\,(\mathit{op},(e_1,e_2)) \mid \\ & & \qquad e_1 \leftarrow \mathit{toValidExprs}\ t_1, \\ & & \qquad e_2 \leftarrow \mathit{toValidExprs}\ t_2, \mathit{op} \leftarrow \mathit{ops}\} \end{array}$$

The function $\mathit{combine}$ is the same one defined in Section 6.2.1. The function $\Lambda \mathit{plus}^{\circ}$ in the definition of trees can be implemented by $\mathit{unmerges}$ because the order of the subtrees in a tree is immaterial: $\mathit{toExprs}\,(\mathit{bin}\,(x,y)) = \mathit{toExprs}\,(\mathit{bin}\,(y,x))$. The type Tree therefore implements the abstract type of *oriented* binary trees.

The reason why this approach is efficient (at least, once *toExprs* is modified to return both expressions and their values) is that it is very economical in its use of space: there are 1881 oriented binary trees with a basis included in six given numbers, compared typically to about 70,000 valid expressions (depending on the precise definition of valid), so the resident space is smaller.

6.2.3 Summary and Comparisons

For the purposes of comparison we implemented three versions of the top-down algorithm:

hutton Hutton's algorithm with a strengthened validity test, modified to return a closest match;

td1 Like *hutton* except that a bag is represented by a list in ascending order;

td2 The version in which trees are used as an intermediate data structure.

Each program was compiled using the Glasgow Haskell Compiler (version 4.08.2) with the `-O` flag, run on a Sun Blade 100 workstation (with a 500 MHZ UltraSPARC IIe processor). Each program was run on the following set of test cases:

Run1 100 test cases, each consists of 6 sources numbers. The source and target numbers are randomly generated.

Run2 30 test cases each with 6 randomly generated sources. The target number is set to -1, an unreachable number, in order to test the worst case performance.

Run3 100 test cases, each with 8 sources and a randomly generated target number.

Run4 10 cases, each consists of 7 sources with target number set to -1.

For Run1 and Run2, the programs were run with a standard 64 megabytes heap. For Run3 and Run4, they were allocated with a 500 megabyte heap. The result of timing in seconds is shown below:

	hutton	td1	td2
Run1	1.165	0.463	0.320
Run2	2.139	0.868	0.546
Run3	22.954	7.586	3.807
Run4	87.753	30.628	14.661

Heap profiles of the programs running on the source numbers $\{\!|13, 7, 18, 187, 475, 217|\!\}$ and target number 4117, which resulted in a miss, are shown in Figure 6.1. The heap profile is generated by the utility `hp2ps`. Chunks in memory are classified according to the function that generated the data, with the one occupying the largest area on the top.

The program *td*1 is about 2.5 to 3 times faster than *hutton*1, which shows that the choice of representing bags as ascending lists does save lots of computation. The cost, however, is a larger heap residency. If we just measure the elapse time, *td*2, with *Tree*

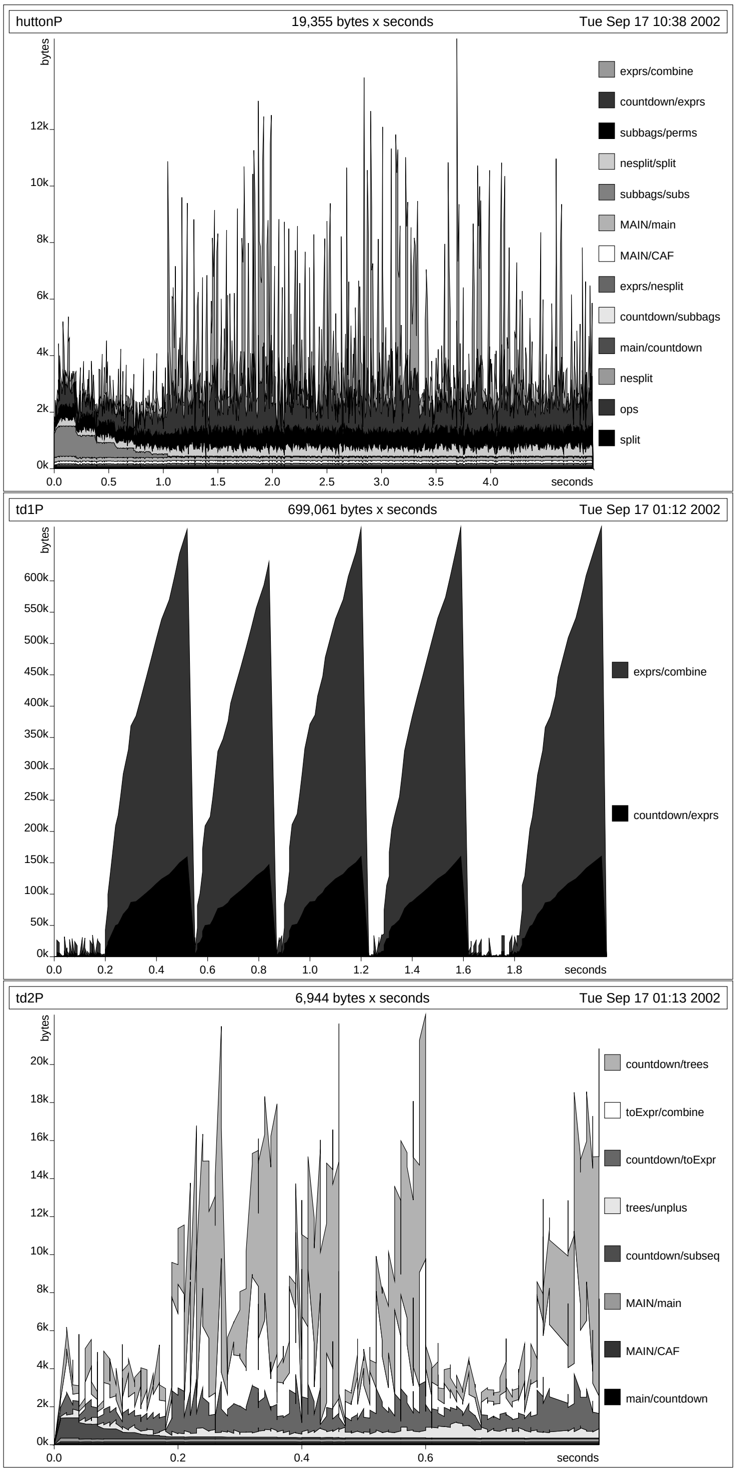

Figure 6.1: Heap profiles of *hutton*, *td*1, and *td*2.

introduced, is around 1.5 times faster than $td1$ for 6 source numbers and nearly twice as fast for 7 and 8 source numbers. In their time profile generated by GHC, however, $td1$ is actually around 8 percent faster than $td2$ if the time spent on garbage collection is not counted. The overall better performance of $td2$ is clearly contributed by its economic use of memory, resulting in less garbage collections. For 6 source numbers, the maximal heap residency of $td2$ is around 24 kilobytes of memory, as opposed to around 700 kilobytes required by $td1$. The reason why it is more memory-economic can be seen from the heap profile. The phase occupying the most memory is not the generation of expressions (chunks created by *combine*), but the generation of trees, which is much smaller in number than the expressions.

6.3 The Closure Algorithm

Despite the above refinements, the problem with all of the above top-down solutions is that computations are repeated. For example, take the input bag $[1, 2, 3, 4, 5]$ and the two splits $([1, 2, 3], [4, 5])$ and $([1, 2, 3, 4], [5])$. For the first, we compute all expressions that can be formed from $[1, 2, 3]$. But this work is repeated when we take the split $([1, 2, 3], [4])$ of the first component of the second split.

One way to avoid these repeated computations is to memoise the computation of *exprs*. That way we will have to either rely on specific language features or implement our own mechanism or memoisation. Or we can tabulate *exprs*. Consider again the structure of Hutton's original algorithm:

$$countdown\ n \quad = \quad min\ R_n \cdot union \cdot \mathsf{P}exprs \cdot \Lambda subbag$$

Suppose we implement bags as lists in ascending order and $\Lambda subbag$ as *subseqs*, taking care to generate the list of subsequences of a list in such a way that, for all *xs* and *ys*, if *xs* is a subsequence of *ys* then *xs* appears before *ys*. Then in the evaluation of $exprs\ xs$, which involves the evaluation of $exprs\ ys$ and $exprs\ zs$ for all $(ys, zs) \in unmerges\ xs$, we can arrange that all these sets of expressions will have already been computed. Consequently, for some suitable type *Basis* that describes the set of possible bases for expressions, we can represent the set of expressions currently computed by an element of $FiniteMap\ Basis\ (List\ Expr)$ and simply look up previously computed expressions. No recomputation is necessary. Though similar in effect to memoisation, tabulation is different in that it proceeds by assembling solutions to larger problems out of smaller ones.

A good definition of *Basis* is a bit sequence; for example, the subsequence $[x_1, x_2, x_4]$ of $[x_0, \ldots x_4]$ can be represented as the bit sequence 01101. Apart from implementing *unmerges* as a function with type $Basis \rightarrow List\,(Basis \times Basis)$ there is little more to be said about the tabulation approach as far as Countdown is concerned. It can benefit from the optimisiation described in Section 6.3.4, but it is simple and available without much effort. The overhead is the space required to store the finite map and the cost of looking up an entry. The former is significant, but the latter is not since there is at most 2^n entries for n source numbers, so looking up an entry costs $\log 2^n = n$ steps.

Yet another approach to avoid repeated computation is to transform the specification to a closure algorithm. We will discuss this approach in finer details.

6.3.1 Generating Subbags within the Recursion

Let us return to the expression $valid \cdot basis^\circ \cdot subbag$ and fuse all three relations. The fusion theorem for $foldExpr$ says that $R \cdot foldExpr\ S_1\ S_2 = foldExpr\ T_1\ T_2$ if the following conditions hold:

$$\begin{aligned} R \cdot S_1 &= T_1 \cdot (id \times (R \times R)) \\ R \cdot S_2 &= T_2 \end{aligned}$$

The relation $subbag^\circ \cdot basis$ can therefore be expressed as a fold if we can find T_1 and T_2 so that

$$\begin{aligned} subbag^\circ \cdot plus \cdot snd &= T_1 \cdot (id \times (subbag^\circ \times subbag^\circ)) \\ subbag^\circ \cdot wrap &= T_2 \end{aligned}$$

Writing $pick = wrap^\circ \cdot subbag$, so $pick$ picks an element from a non-empty bag, it is clear that we can take $T_2 = pick^\circ$. We can also take $T_1 = plus \cdot snd$ since it is easy to check that

$$subbag^\circ \cdot plus = plus \cdot (subbag^\circ \times subbag^\circ)$$

if $plus$, as mentioned before, is a partial relation taking only sets of more than two elements as its input. Consequently, $valid \cdot basis^\circ \cdot subbag$ is a hylomorphism and can be written as a fixed-point:

$$valid \cdot basis^\circ \cdot subbag = \mu(X \mapsto val \cdot pick\ \cup\ app' \cdot snd^\circ \cdot (X \times X) \cdot plus^\circ)$$

The fixed-point is also unique because $\delta_{\mathsf{F}} \cdot [pick, plus \cdot snd]^\circ$ is inductive. Note, however, that the union in the body of the recursion shall not be interpreted as a conditional anymore: the domains of the two alternatives are not disjoint and we can nondeterministically choose to terminate via $val \cdot pick$ at any time. Consequently, in the breadth of its recursive solution, written as $subexprs$ below, the two alternatives shall both be explored and their union taken:

$$\begin{array}{lcl} countdown\ n &=& min\ R_n \cdot subexprs \\ subexprs\ xb & | & singleton\ xb\ =\ \{val\,(wrap^\circ\ xb)\} \\ & | & otherwise\ =\ \mathsf{P}val\ xb\ \cup \\ && \qquad union\{combine\,(op,(e_1, e2))\ | \\ && \qquad\qquad (yb, zb) \leftarrow (\Lambda plus^\circ)\ xb, \\ && \qquad\qquad e_1 \leftarrow subexprs\ yb, e_2 \leftarrow subexprs\ zb, \\ && \qquad\qquad op \leftarrow ops\} \end{array}$$

Whereas $exprs\ xb$ denotes the set of valid expressions with basis xb, the value of $subexprs\ xb$ is the set of valid expressions with a basis which is a subbag of xb. The main advantage of $subexprs$ is that we no longer have to compute $subbags$ explicitly.

The problem with this approach is, however, that when sets are implemented by lists and union by concatenation, we generate lots of repetitions in the list. Say, $subexprs\ [1, 2, 3]$ would generate the expression $app\ (op, (2, 3))$ three times for each operator op. One might attempt to define a clever variation of $plus$ to avoid the repetition. Another possibility, however, is to turn to a bottom-up algorithm, and thereby systematically generate all the expressions without repetition.

6.3.2 Transforming to a Closure

We know from the previous section that

$$valid \cdot basis^\circ \cdot subbag \quad = \quad \mu(X \mapsto val \cdot pick \ \cup \ app' \cdot snd^\circ \cdot (X \times X) \cdot plus^\circ)$$

Suppose there are no duplicated elements in the bag, an assumption we can take care of by tagging identical numbers with distinct basis values to ensure uniqueness. Then bags can be replaced by sets, *subbag* by *subset* (the relation returning a non-empty subset), and *plus* by *cup*, the disjoint union of two non-empty sets. The point of this change is that we can then exploit the following identity:

$$(subset \times subset) \cdot cup^\circ \quad = \quad disjoint? \cdot \langle subset, subset\rangle$$

where *disjoint* is a predicate testing whether a pair of sets have no members in common, The relation cup° splits the set into two disjoint sets. In words, the identity says that we can select two disjoint non-empty subsets from a set (the right-hand side) by splitting the set into two (disjoint, proper, non-empty) subsets and taking non-empty subsets from each half (the left-hand side).

Having that in mind, we derive that applied to sets of size at least two:

$$\begin{aligned}
& (valid \cdot basis^\circ \cdot subset \times valid \cdot basis^\circ \cdot subset) \cdot cup^\circ \\
= & \quad \{\text{products}\} \\
& (valid \cdot basis^\circ \times valid \cdot basis^\circ) \cdot (subset \times subset) \cdot cup^\circ \\
= & \quad \{\text{above identity}\} \\
& (valid \cdot basis^\circ \times valid \cdot basis^\circ) \cdot disjoint? \cdot \langle subset, subset\rangle \\
= & \quad \{\text{let } disjointE \text{ be the counterpart of } disjoint \text{ on expressions}\} \\
& disjointE? \cdot (valid \cdot basis^\circ \times valid \cdot basis^\circ) \cdot \langle subset, subset\rangle \\
= & \quad \{\text{splits absorbs products}\} \\
& disjointE? \cdot \langle valid \cdot basis^\circ \cdot subset, valid \cdot basis^\circ \cdot subset\rangle
\end{aligned}$$

The predicate *disjointE* determines whether two expressions have disjoint bases.

Since we also have $val \cdot pick \subseteq valid \cdot basis^\circ \cdot subset$, it follows that $valid \cdot basis^\circ \cdot subset$ is a solution for X of the inequation

$$val \cdot pick \ \cup \ app' \cdot snd^\circ \cdot disjointE \cdot \langle X, X\rangle \quad \subseteq \quad X \tag{6.1}$$

We have mentioned in Section 4.7 what a closure is. In brief, the relation $R^* \cdot S$ is defined as a least fixed-point:

$$R^* \cdot S \quad = \quad \mu(X \mapsto S \cup R \cdot X)$$

The least fixed-point is the unique fixed-point if and only if R° is an inductive relation. Here R has type $B \to B$.

Now look at (6.1) again. Its least solution is also the least fixed-point of the corresponding equation. Furthermore, $(fst \cup snd) \cdot snd \cdot app'^\circ \cdot disjointE?$ is an inductive relation, so the least fixed-point is the only fixed-point. In summary, we have shown:

$$valid \cdot basis^\circ \cdot subset \quad = \quad \mu(X \mapsto val \cdot pick \ \cup \ app' \cdot snd^\circ \cdot disjointE? \cdot \langle X, X\rangle)$$

Substitute S for $val \cdot pick$ and R for $app' \cdot snd^{\circ} \cdot disjointE?$, the right-hand side abbreviates to:

$$\mu(X \mapsto S \ \cup \ R \cdot \langle X, X \rangle)$$

It is a generalisation of a closure to binary relations $R :: (B \times B) \to B$!

In the next section we will talk about, in general, how to compute $\mu(X \mapsto S \ \cup \ R \cdot \langle X, X \rangle)$ for arbitrary R and S.

6.3.3 Computing Closures

Given an initial value a, a naive way to compute $\Lambda(R^* \cdot S)\, a$ is to apply ΛS to a, and then repeatedly apply $\mathsf{E}R$ to the resulting set until we reach a fixed-point, i.e., until the set does not change after an application of $\mathsf{E}R$. This way, however, we might end up with unnecessarily applying R many times to those members persisting in the set. A better approach is to distinguish, say, by keeping two sets, the newly computed members and the older ones, and apply the next iteration of $\mathsf{E}R$ to those new members only. In [17, Chapter 6] such an algorithm was derived. In this section, we are going to generalise the result to binary R, i.e., to compute:

$$\mu(X \mapsto S \ \cup \ R \cdot \langle X, X \rangle) \tag{6.2}$$

Let θ_R, parameterised by a relation R, be a function from pairs of relations to relations, defined by:

$$\theta_R(P, Q) \quad = \quad P \cup \mu(X \mapsto Q \cup (R \cdot (\langle X, X \rangle \cup \langle X, P \rangle \cup \langle P, X \rangle) - P))$$

where the relations P and Q have type $A \to B$, and R has type $(B \times B) \to B$. When we set Q to S and P to $\emptyset$, the right-hand side reduces to (6.2). Therefore, to compute (6.2) we just need to compute $\theta_R(\emptyset, S)$. We now aim at deriving an recursive characterisation of $\theta_R(P, Q)$.

When $Q = \emptyset$, it is clear that the empty set is a solution, thus the least, of the fixed-point. Therefore $\theta_R(P, \emptyset) = P$. To derive the general case, we will make use of the *rolling rule* for fixed-points, stated below:

$$\mu(f \cdot g) \quad = \quad f(\mu(g \cdot f)) \tag{6.3}$$

Abbreviate the chain of unions $\langle X, X \rangle \cup \langle X, P \rangle \cup \langle P, X \rangle$ to $prods(X, P)$. We derive:

$$\begin{array}{ll}
& \theta_R(P, Q) \\
= & \quad \{\text{definition}\} \\
& P \cup \mu(X : Q \cup (R \cdot prods(X, P) - P)) \\
= & \quad \{\text{since } X \cup Y = X \cup (Y - X)\} \\
& P \cup \mu(X : Q \cup (R \cdot prods(X, P) - (P \cup Q))) \\
= & \quad \{\text{rolling rule (6.3), with } f = (Q \cup) \text{ and } g\, X = R \cdot prods(X, P) - (P \cup Q)\} \\
& P \cup Q \cup \mu(X : R \cdot prods(Q \cup X, P) - (P \cup Q))
\end{array}$$

$$
\begin{array}{ll}
= & \{\text{claim: } \mathit{prods}(Q \cup X, P) = \mathit{prods}(Q, P) \cup \mathit{prods}(X, P \cup Q)\} \\
& P \cup Q \cup \mu(X : R \cdot (\mathit{prods}(Q, P) \cup \mathit{prods}(X, P \cup Q)) - (P \cup Q)) \\
= & \{\text{since } (R\cdot) \text{ and } (-(P \cup Q)) \text{ distribute into } \cup\} \\
& P \cup Q \cup \mu(X : (R \cdot \mathit{prods}(Q, P) - (P \cup Q)) \cup \\
& \qquad\qquad (R \cdot \mathit{prods}(X, P \cup Q) - (P \cup Q))) \\
= & \{\text{folding the definition of } \theta_R\} \\
& \theta_R(P \cup Q, R \cdot \mathit{prods}(Q, P) - (P \cup Q))
\end{array}
$$

To prove the claim, we will make use of the following fact:

$$
\langle A \cup B, C \cup D \rangle \;=\; \langle A, C \rangle \cup \langle A, D \rangle \cup \langle B, C \rangle \cup \langle B, D \rangle \tag{6.4}
$$

The claim will be proved as below:

$$
\begin{array}{ll}
& \langle Q \cup X, Q \cup X \rangle \cup \langle Q \cup X, P \rangle \cup \langle P, Q \cup X \rangle \\
= & \{\text{expanding all the terms by (6.4)}\} \\
& \langle Q, Q \rangle \cup \langle Q, X \rangle \cup \langle X, Q \rangle \cup \langle X, X \rangle \\
& \cup \langle Q, P \rangle \cup \langle X, P \rangle \cup \langle P, Q \rangle \cup \langle P, X \rangle \\
= & \{\text{rearranging the terms}\} \\
& \langle P, Q \rangle \cup \langle Q, P \rangle \cup \langle Q, Q \rangle \\
& \cup \langle X, X \rangle \cup \langle Q, X \rangle \cup \langle P, X \rangle \cup \langle X, P \rangle \cup \langle X, Q \rangle \\
= & \{\text{folding the last four terms with (6.4)}\} \\
& \langle P, Q \rangle \cup \langle Q, P \rangle \cup \langle Q, Q \rangle \\
& \cup \langle X, X \rangle \cup \langle P \cup Q, X \rangle \cup \langle X, P \cup Q \rangle
\end{array}
$$

In summary, we have derived the following definition for θ_R:

$$
\begin{array}{lcl}
\theta_R(P, \emptyset) & = & P \\
\theta_R(P, Q) & = & \theta_R(P \cup Q, R \cdot (\langle P, Q \rangle \cup \langle Q, P \rangle \cup \langle Q, Q \rangle) - (P \cup Q))
\end{array}
$$

which itself can be written as a closure:

$$
\begin{array}{lcl}
\theta_R & = & \mathit{stop} \cdot \mathit{step}_R^* \\
\mathit{stop}\,(P, \emptyset) & = & P \\
\mathit{step}_R\,(P, Q) & \mid & Q \neq \emptyset \;=\; (P \cup Q, R \cdot (\langle P, Q \rangle \cup \langle Q, P \rangle \cup \langle Q, Q \rangle) - (P \cup Q))
\end{array}
$$

It is therefore an iterative algorithm.

The function θ_R constructs a relation defined as a least fixed-point. The set-theoretic counterpart to θ_R is a function *close* defined by:

$$
\begin{array}{lcl}
\mathit{close} & :: & ((\mathit{Set}\ A \times \mathit{Set}\ A) \to \mathit{Set}\ A) \to (\mathit{Set}\ A \times \mathit{Set}\ A) \to \mathit{Set}\ A \\
\mathit{close}\,f\,(ps, \emptyset) & = & p \\
\mathit{close}\,f\,(ps, qs) & = & \mathit{close}\,f\,(ps \cup qs, f\,(ps, qs) \cup f\,(qs, ps) \cup f\,(qs, qs) - (ps \cup qs))
\end{array}
$$

which can also be written as a closure:

$$
\begin{array}{lcl}
\mathit{close}\,f & = & \mathit{stop} \cdot (\mathit{step}\,f)^* \\
\mathit{stop}\,(ps, \emptyset) & = & ps \\
\mathit{step}\,f\,(ps, qs) & \mid & qs \neq \emptyset \;=\; (ps \cup qs, f\,(ps, qs) \cup f\,(qs, ps) \cup f\,(qs, qs) - (ps \cup qs))
\end{array}
$$

The relationship between θ and close is given by:

$$\Lambda(\theta_R(P,Q)) \quad = \quad close\,\Lambda(R \cdot (\in \times \in)) \cdot \langle \Lambda P, \Lambda Q \rangle$$

We have derived an algorithm to compute (6.2) in general. For Countdown, however, some simplification can be done. Firstly, we can make f commutative by constructing both $app\ (op,(x,y))$ and $app\ (op,(y,x))$, thus $f\ (ps,qs) \cup f\ (qs,ps) \cup f\ (qs,qs)$ is equivalent to $f\ (ps \cup qs, qs)$. Secondly, each tree we construct are deeper than those in the previous iteration, therefore $ps \cup qs$ are disjoint with $f\ (ps \cup qs, qs)$. The subtraction is thus not necessary. Consequently, the recursive case of *close* can be simplified to:

$$close\,f\ (ps,qs) \quad = \quad close\,f\ (ps \cup qs, f\ (ps \cup qs, qs))$$

Instantiating *close* for the Countdown problem we find that

$$\begin{array}{lcl} countdown\ n & = & min\ R_n \cdot close\,\Lambda(join \cdot (\in \times \in)) \cdot \langle const\,\emptyset, \mathsf{P}val \rangle \\ join & = & app \cdot snd^{\circ} \cdot (id \cup swap) \cdot disjointE? \end{array}$$

The term $(id \cup swap)$ is inserted into *join* to ensure commutativity.

To understand what is going on with this algorithm, let (ps_n, qs_n) be the arguments of *close* at the nth recusive call, beginning with $n = 0$. It is easy to show by induction that ps_n is the set of valid expressions of height less than n and qs_n is the set of expressions with height n. The algorithm therefore stops after $k-1$ iterations, where k is the size of the input set.

As a side note: had we started off with this definition:

$$\theta'_R(P,Q) \quad = \quad \mu(X \mapsto Q \cup (R \cdot (\langle X,X \rangle \cup \langle X,P \rangle \cup \langle P,X \rangle) - P))$$

we would have come up with this recursive definition for θ_R:

$$\begin{array}{lcl} \theta'_R(P,\emptyset) & = & \emptyset \\ \theta'_R(P,Q) & = & Q \cup \theta'_R(P \cup Q, R \cdot (\langle P,Q \rangle \cup \langle Q,P \rangle \cup \langle Q,Q \rangle) - (P \cup Q)) \end{array}$$

This “online” algorithm produces results as soon as they are available, which may be advantageous in a lazy language. The derivation of θ'_R is recoded in Appendix C.1. However, in our experiment, this variant is not significantly better, so we will just stick with the first, tail-recursive definition.

6.3.4 Thinning

There are about 33 million expressions on 6 numbers, of which about 4.5 million satisfy the basic validity test, 250,000 that satisfy the first strengthened validity test given in Section 6.2, and 70,000 that satisfy the second (of course, the figures vary depending on the actual source numbers). But we can go further in reducing the number of expressions that have to be considered. If two expressions x and y have the same value but the basis of x is contained in the basis of y, then there is no point keeping y. Whatever expressions we further construct using y, we can construct with x instead. We will make this informal observation precise below. By ‘thinning’ of the set of possible expressions we can eliminate duplicates of essentially the same expression, expressions such as $(x+y)+z$ and $x+(z+y)$ which have exactly the same basis, or expressions such as x and $x*(y/y)$ in which the former has a smaller basis than the latter. We will also eliminate expressions such as $7+5+2$ in favour of $7*2$, an expression with the same value but a smaller basis. Thinning can therefore dramatically cut down the number of expressions we need to consider. The downside, of course, is that thinning takes time.

The relation *thin* has been introduced in Section 5.2.1. Let $Q :: A \to A$ be a preorder. The relation $\mathit{thin}\ Q :: \mathit{Set}\ A \to \mathit{Set}\ A$ is defined by

$$(xs, ys) \in \mathit{thin}\ Q \quad \equiv \quad (ys \subseteq xs) \wedge (\forall x : x \in xs : \exists y : y \in ys : yQx)$$

The resulting set ys is a streamlined subset of xs. For the Countdown problem take Q to be the preorder

$$x\ Q\ y \quad \equiv \quad (\mathit{value}\, x = \mathit{value}\, y)\ \wedge\ (\mathit{basis}\, x \subseteq \mathit{basis}\, y)$$

we have $x\ Q\ y \Rightarrow \mathit{dist}\, n\, x = \mathit{dist}\ n\ y \Rightarrow x\ R_n\ y$, so it is legitimate to introduce the term $\mathit{thin}\ Q$ after $\mathit{min}\ R_n$. The next step is to fuse the relation $\mathit{thin}\ Q$ into the main computation, thereby thinning at each stage.

The proof of the following theorem is relegated to Appendix C.2:

Theorem 6.1 Let R be monotonic on Q° in the sense that $R \cdot (Q^\circ \times Q^\circ) \subseteq Q^\circ \cdot R$. Then

$$\mathit{thinclose}\ Q\, f \quad \subseteq \quad \mathit{thin}\ Q \cdot \mathit{close}\, f$$

where $f = (\Lambda R \cdot (\in \times \in))$ and *thinclose* is defined by:

$$\mathit{thinclose}\ Q\, f \quad = \quad \mathit{stop} \cdot ((\mathit{thin}\ Q \times \mathit{thin}\ Q) \cdot \mathit{step}\, f)^* \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q)$$

In earlier experiments in Oberon, with efficient hash table and sophisticated linking data structure, this approach turned out to be very efficient [82]. In order to install the thinning refinement into Haskell, we made use of a supplementary data structure **type** $\mathit{Table} = \mathit{FiniteMap}\, \mathcal{Z}\, (\mathit{List}\, (\mathit{Expr} \times \mathit{Basis}))$ that organises computed expressions according to their value. The entry associated with value v in the table consists of a set of expressions with disjoint bases each with value v. We will not spell out the details because the bottom line is that the thinning stage turns out not to be worth the candle, at least not in Haskell without the use of destructive data structures. Our experiments show that a thinning algorithm with a basic validity test is outperformed by a non-thinning algorithm with the more sophisticated validity test described in Section 6.2. Nevertheless, the efficiency of the thinning algorithms could be dramatically improved if arrays were used, at the expense of having to program in a monadic style. We have recorded the thinning theorem in this section primarily because they do not appear in either [17] or [26].

6.4 A Fold Algorithm

The top-down approach is based on the compositional approach to function inversion. Naturally, we would like to give the converse-of-a-function theorem a try.

Define the datatype for oriented binary trees by:

$$\textbf{data}\ \mathit{OTree} \quad = \quad \mathit{tip}\ \mathcal{Z} \mid \mathit{bin}\,(\nabla\,\mathit{Tree})$$

where ∇ stands for the type of unordered pairs, or sets with two elements. A value of type ∇A can be constructed by $\otimes : (A \times A) \to \nabla A$, and it is assumed that $x \otimes y = y \otimes x$ for all x and y. The fold function for OTree will be denoted by $\mathit{foldOTree} :: ((A \times A) \to A) \to (\mathcal{Z} \to A) \to \mathit{OTree} \to A$. It is defined the same as that for ordinary, ordered binary trees, with one extra constraint — that its first argument, as a consequence of using ∇, must be a commutative function. We overload the function basisT to oriented trees and bags. It has type $\mathit{OTree} \to \mathit{Bag}\,\mathcal{Z}$ and is be defined by $\mathit{basisT} = \mathit{foldOTree}\ \mathit{wrap}\ \mathit{bcup}$, where bcup is the counterpart of cup for bags.

Similarly, the fold function for non-empty bags, written $\mathit{foldBag} :: ((A \times B) \to B) \to (A \to B) \to \mathit{Bag}\,A \to B$, is defined the same as that for non-empty lists, except that we require a healthiness condition on its first argument — denoting it by $\oplus$, we need $a \oplus (b \oplus x)) = b \oplus (a \oplus x)$ to hold.

According to the converse-of-a-function theorem, in order to show that:

$$\mathit{basisT}^{\circ} \quad = \quad \mathit{foldBag}\ \mathit{add}\ \mathit{tip}$$

for some definition of add, we need to show that the following premises holds:

$$\begin{aligned} \mathit{basisT} \cdot \mathit{tip} &\subseteq \mathit{wrap} \\ \mathit{basisT} \cdot \mathit{add} &\subseteq \mathit{bcons} \cdot (\mathit{id} \times \mathit{basisT}) \end{aligned}$$

where bcons is the counterparts of cons on bags. The condition on tip is obviously true. For the second condition, we claim the following definition of add suffices:

$$\begin{aligned} \mathit{add}\,(a, \mathit{tip}\ b) &= \mathit{bin}\,(\mathit{tip}\ a \otimes \mathit{tip}\ b) \\ \mathit{add}\,(a, \mathit{bin}\,(x \otimes y)) &= \mathit{bin}\,(\mathit{tip}\ a \otimes \mathit{bin}\,(x \otimes y)) \\ &\quad\ \square\ \mathit{bin}\,(\mathit{add}\,(a, x) \otimes y) \end{aligned}$$

Since $\mathit{bin}\,(x \otimes y)$ is a nondeterministic pattern, the recursion only need to be performed on one of the branches.

We will need to prove that add does satisfy the premise for converse-of-a-function theorem. Besides, we need to show that add satisfies the healthiness condition to be an argument to $\mathit{foldBag}$. Since add is defined recursively, a proof that add does satisfy the above conditions involves manipulating with fixed-points. A good exercise as it is, guiding the reader through the proof is not the purpose of this chapter. The full proof will be be relegated to Appendix C.

In the implementation, OTree can be simulated by Tree. To implement the nondeterministic pattern $\mathit{bin}\,(x \otimes y)$ in add, however, we have to perform the swapping ourselves. The following function simulates $\Lambda \mathit{add}$:

$$\begin{aligned} \mathit{padd} &:: (\mathcal{Z} \times \mathit{Tree}) \to \mathit{Set}\ \mathit{Tree} \\ \mathit{padd}\,(a, \mathit{tip}\ b) &= \{\mathit{bin}\,(\mathit{tip}\ a, \mathit{tip}\ b)\} \\ \mathit{padd}\,(a, \mathit{bin}\,(x, y)) &= \{\mathit{bin}\,(\mathit{tip}\ a, \mathit{bin}\,(x, y))\} \\ &\quad\ \cup \{\mathit{bin}\,(x', y) \mid x' \leftarrow \mathit{padd}\,(a, x)\} \\ &\quad\ \cup \{\mathit{bin}\,(x, y') \mid y' \leftarrow \mathit{padd}\,(a, y)\} \end{aligned}$$

We now have:

$$\Lambda(basisT^{\circ}) = foldBag\,(union \cdot \mathsf{P}padd \cdot \Lambda(id \times \in))\,(wrap \cdot tip)$$

This implementation of $\Lambda(basisT^{\circ})$ can readily be used in place of that in Section 6.2.2.

In the development of the top-down algorithms, we tried to fuse *subbag* into the body of the computation. It turned out to be a bad idea because we end up building lots of repeated trees. The problem, however, does not occur for our folding algorithm here. Since *subbag* is a fold on bags:

$$\begin{array}{lll} subbag & = & foldBag\ step\ \{\!|\,|\!\} \\ \quad \textbf{where} & step\,(a,x) & = \{\!|\,|\!\} \,\square\, (\{\!|a|\!\} \cup x) \end{array}$$

Simple fold fusion suffices to show that *foldBag paddsub* $(wrap \cdot tip)$ implements $\Lambda(basisT^{\circ} \cdot subbag)$, where *paddSub* is defined by:

$$\begin{array}{lll} paddsub & :: & (\mathcal{Z} \times Set\ Tree) \rightarrow Set\ Tree \\ paddsub\,(a, xs) & = & \{tip\ a\} \cup xs \cup \{add\,(a,x) \mid x \leftarrow xs\} \end{array}$$

We still need to refine sets to lists. One naive way is just to replace each $\cup$ by $+\!\!+$. But look at the definition of *paddsub* above. Take $xs = [x, y, z]$, we would get

$$[tip\ a] +\!\!+ [x,y,z] +\!\!+ padd\,(a,x) +\!\!+ padd\,(a,y) +\!\!+ padd\,(a,z)$$

as a result. Recall that the list is then piped to $min\ R_n$ lazily. The order of the list means that x needs to reside in memory until y and z are processed! Similarly with y. Learning from previous experiences how heap residency can effect the efficiency, we would prefer this order:

$$[tip\ a] +\!\!+ [x] +\!\!+ padd\,(a,x) +\!\!+ [y] +\!\!+ padd\,(a,y) +\!\!+ [z] +\!\!+ padd\,(a,z)$$

such that x and y can both be thrown away earlier. Define:

$$\begin{array}{lll} shuffle & :: & (List\ A \times List\,(List\ A)) \rightarrow List\ A \\ shuffle & = & concat \cdot map\ cons \cdot zip \end{array}$$

where $zip :: (List\ A \times List\ B) \rightarrow List\,(A \times B)$, we can get the order we want by implementing the second $\cup$ in the definition of *paddsub* by *shuffle*. This trick effectively reduces the heap residency.

6.5 Comparisons

We implemented the following variations of the closure and folding algorithms:

close1 The closure algorithm of Section 6.3.3;

close2 The closure algorithm with thinning;

cft1 The fold algorithm of Section 6.4, *subbag* not fused;

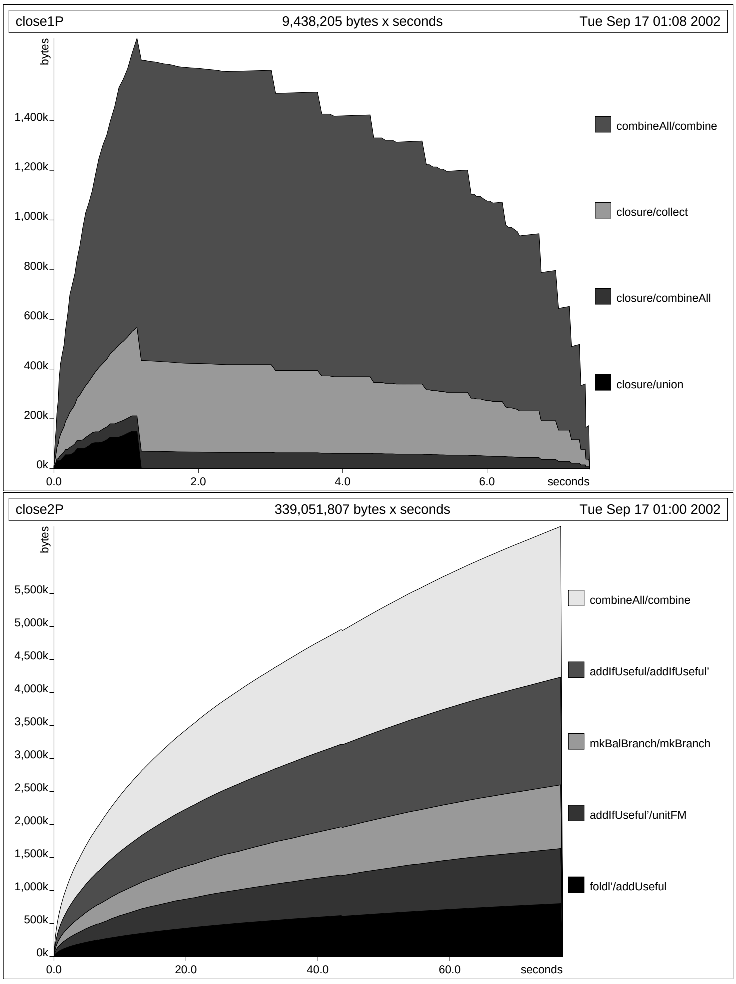

Figure 6.2: Heap profiles of *close*1 and *close*2.

cft2 The same as *cft*1 except that *subbag* is fused;

cft3 The same as *cft*2 but with shuffling.

Each program was compiled and run as for the top-down algorithms. The results, with the first and second columns repeating the statistics for *hutton* and *td*2, were:

	hutton	td2	close1	close2	cft1	cft2	cft3
Run1	1.165	0.320	0.593	1.744	0.293	0.219	0.212
Run2	2.139	0.546	1.008	2.559	0.484	0.413	0.414
Run3	22.954	3.807	10.856	-	3.311	3.035	1.772
Run4	87.753	14.661	37.261.0	87.261	13.445	13.382	13.462

The performance of *close*1, the closure algorithm without thinning, lies between the most naive and the most sophisticated top-down algorithm. Not indexing on values, the

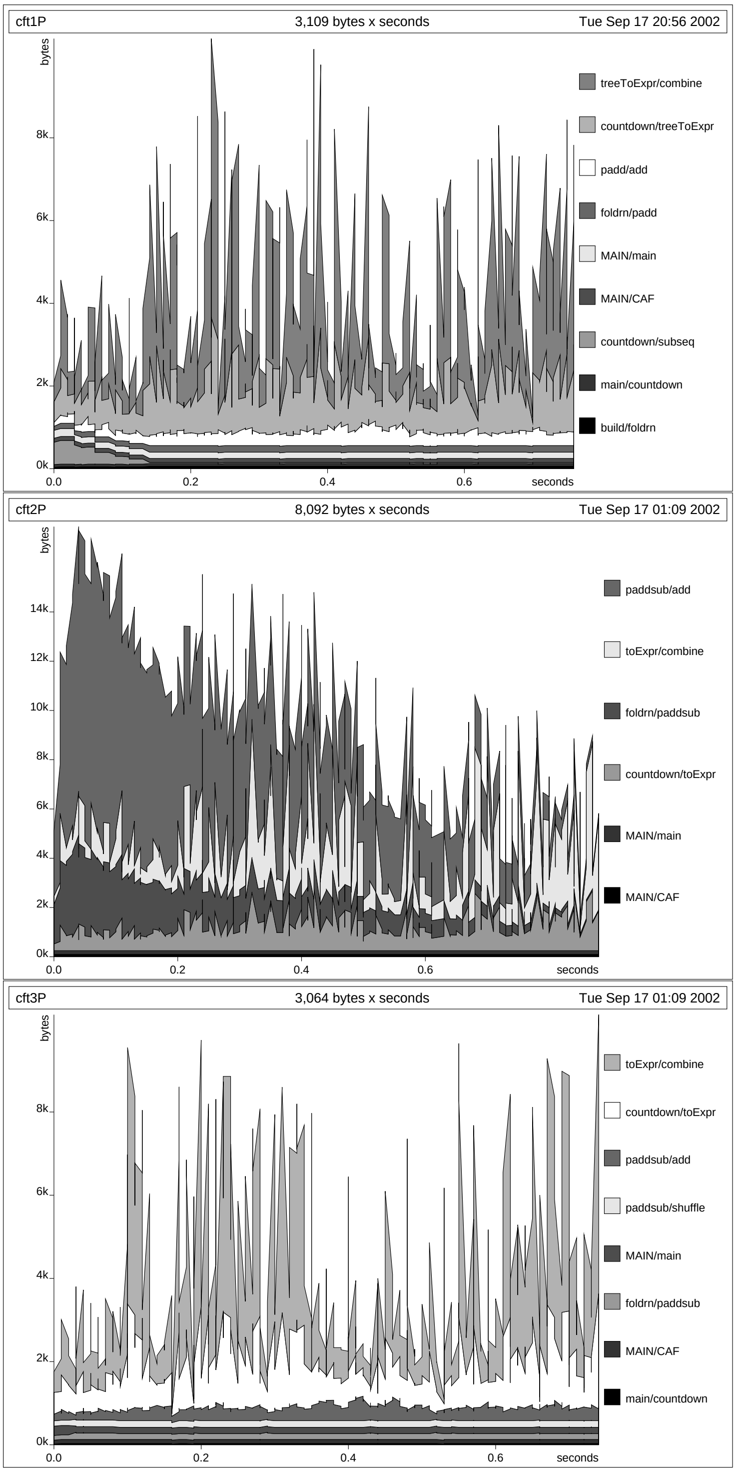

Figure 6.3: Heap profiles of *cft*1, *cft*2, and *cft*3.

answer is extracted by flattening the *FiniteMap* to a list and performing a linear search, which results in the down slope in the heap profile. Thinning, however, does not pay in our experiments. We believe that both of them will do better with nicer support of destructive data structures.

The folding programs are very fast and memory economic. Comparing *cft*1 and *cft*2, we see that fusing *subbag* helps to gain fringe speed, while increasing the heap residency. By looking at the heap profile of *cft*2, however, we find that the dominant chunks are from *add* called by *paddsub*, leading one to doubt that some chunks might have stayed in memory longer than necessary. It leads to the idea of shuffling, which proved to be very effective – the memory requirement of *cft*3 is reduced back to around 10 kilobytes. The effort pays in Run3, where *cft*3 performed 170 percent faster than *cft*2 and 12 times faster than *hutton*.

6.6 Conclusions

By now the reader has probably been counted out by Countdown. All the algorithms we have described have been derived from a simple specification, using a variety of ideas drawn from the growing body of mathematics for program construction. There are many ways to solve the problem, some more or less closely related to others. The structure of an algorithm is partly determined by different strategies for inversion. However, we also find that in transforming a specification to a program, various other decisions, especially those involving implementing sets in lists such as different representations of sets, different ways to merge lists, etc, play important roles in improving the performance of the resulting program.

Chapter 7

The Burrows-Wheeler Transform

The Burrows-Wheeler Transform [21] is a method for permuting a string with the aim of bringing repeated characters together. As a consequence, the permuted string can be compressed effectively using simple techniques such as move-to-front or run-length encoding. In [69], the article that brought the BWT to the world's attention, it was shown that the resulting compression algorithm could outperform many commercial programs available at the time. The BWT has now been integrated into a high-performance utility `bzip2`, available from [79].

Clearly the best way of bringing repeated characters together is just to sort the string. But this idea has a fatal flaw as a preliminary to compression: there is no way to recover the original string unless the complete sorting permutation is produced as part of the output. Instead, the BWT achieves a more modest permutation, one that aims to bring some but not all repeated characters into adjacent positions. Moreover, the transform can be inverted using a single additional piece of information, namely an integer b in the range $0 \leq b < n$, where n is the length of the output (or input) string.

It often puzzles people, at least on a first encounter, why the BWT is invertible and how the inversion is actually carried out. Our objective in this chapter is to prove a fundamental property which made the inversion possible and, based on that, derive the inverse transform from its specification, all by equational reasoning. As a reward, we can further derive the inverse of two variations of the BWT transform, one proposed in [77], another in [23].

7.1 Defining the BWT

The BWT is specified by two functions: $bwp :: String \rightarrow String$, which permutes the string and $bwn :: String \rightarrow \mathcal{Z}$, which computes the supplementary integer. The restriction to strings is not essential to the transform, and we can take bwp to have type $Ord\ A \Rightarrow List\ A \rightarrow List\ A$, so lists of any type will do provided there is a total ordering relation on the elements. The function bwp is defined by

$$bwp \quad = \quad map\ last \cdot sort \cdot rots \tag{7.1}$$

```
0  y o k o h a m a          5  a m a y o k o h
1  o k o h a m a y          7  a y o k o h a m
2  k o h a m a y o          4  h a m a y o k o
3  o h a m a y o k          2  k o h a m a y o
4  h a m a y o k o          6  m a y o k o h a
5  a m a y o k o h          3  o h a m a y o k
6  m a y o k o h a          1  o k o h a m a y
7  a y o k o h a m          0  y o k o h a m a
```

Figure 7.1: Computation of *recreate*

The function $\mathit{sort} :: \mathit{Ord}\,A \Rightarrow \mathit{List}\,(\mathit{List}\,A) \rightarrow \mathit{List}\,(\mathit{List}\,A)$ sorts a list of lists into lexicographic order and is considered in greater detail in the following section. The function *rots* returns the rotations of a list and is defined by

$$\begin{array}{lcl} \mathit{rots} & :: & \mathit{List}\,A \rightarrow \mathit{List}\,(\mathit{List}\,A) \\ \mathit{rots}\,x & = & \mathit{take}\,(\mathit{length}\,x)\,(\mathit{iterate}\,\mathit{lrot}\,x) \end{array}$$

where $\mathit{lrot}\,x = \mathit{tail}\,x +\!\!+ [\mathit{head}\,x]$, so *lrot* performs a single left rotation. The definition of *bwp* is constructive, but we won't go into details – at least, not in this chapter – as to how the program can be made more efficient.

The function *bwn* is specified by

$$\mathit{sort}\,(\mathit{rots}\,x)\,!!\,\mathit{bwn}\,x \quad = \quad x \tag{7.2}$$

where $x\,!!\,k$ applied to a list returns the element of x in position k, starting from 0. In words, $\mathit{bwn}\,x$ returns some position at which x occurs in the sorted list of rotations of x. If x is a repeated string, then $\mathit{rots}\,x$ will contain duplicates, so $\mathit{bwn}\,x$ is not defined uniquely by (7.2).

As an illustration, consider the string `yokohama`. The rotations and the lexicographically sorted rotations are as in Figure 7.1. The output of *bwp* is the string `hmooakya`, the last column of the second matrix, and *bwn* `"yokohama"` = 7 because row number 7 in the sorted matrix of rotations is the input string.

The BWT helps compression because it brings together characters with a common context. To give a brief illustration, an English text may contain many occurrences of words such as “this”, “the”, “that” and some occurrences of “where”, “when”, “she”, “ he” (with a space), etc. Consequently, many of the rotations beginning with “h” will end with a “t”, some with a “w”, an “s” or a space. The chance is smaller that a rotation beginning with “h” would end in a “x”, a “q”, or an “u”, etc. Thus the BWT brings together a smaller subset of alphabets, say, those “t”s, “w”s and “s”s. A move-to-front encoding phase is then able to convert the characters into a series of small-numbered indexes, which improves the effectiveness of entropy-based compression techniques such as Huffman or arithmetic coding. For a fuller understanding of the role of the BWT in data compression, consult [21, 69].

For us, however, the BWT is interesting because it is not obvious how to convert the string back. The inverse transform $unbwt :: Ord\ A \Rightarrow \mathcal{Z} \to List\ A \to List\ A$ is specified by

$$unbwt\,(bwn\ x)\,(bwp\ x) \quad = \quad x \tag{7.3}$$

To compute *unbwt* we have to show how the matrix of lexicographically sorted rotations, or at least its tth row where $t = bwn\ x$, can be recreated solely from the knowledge of its last column. To do so we need to examine lexicographic sorting in more detail.

7.2 Lexicographic sorting

Let $(\leq) :: A \to A \to Bool$ be a linear ordering on A. Define $(\leq_k) :: List\ A \to List\ A \to Bool$ inductively by

$$\begin{array}{lcl} x \leq_0 y & = & true \\ (a : x) \leq_{k+1} (b : y) & = & a < b\ \vee\ (a = b\ \wedge\ x \leq_k y) \end{array}$$

The value $x \leq_k y$ is defined whenever the lengths of x and y are both no smaller than k.

Now, let $sort\,(\leq_k) :: List\,(List\ A) \to List\,(List\ A)$ be a stable sorting algorithm that sorts an $n \times n$ matrix, given as a list of lists, according to the ordering $\leq_k$. Thus $sort\,(\leq_k)$, which we henceforth abbreviate to $sort_k$, sorts a matrix on its first k columns. Stability means that rows with the same first k elements appear in their original order in the output matrix. By definition, $sort = sort_n$.

Define $cols\,j = map\,(take\,j)$, so $cols\,j$ returns the first j columns of a matrix. Our aim in this section is to establish the following fundamental relationship, which is the key property establishing the existence of an algorithm for inverse BWT. Provided $1 \leq j \leq k$ we have

$$cols\,j \cdot sort_k \cdot rots \quad = \quad sort_1 \cdot cols\,j \cdot map\ rrot \cdot sort_k \cdot rots \tag{7.4}$$

where *rrot* denotes a single right rotation, defined by $rrot\ xs = last\ xs : init\ xs$. Equation (7.4) looks daunting, but take $j = n$ (so $cols\,j$ is the identity) and $k = n$ (so $sort_k$ is a complete lexicographic sorting), the above reduces to:

$$sort_n \cdot rots \quad = \quad sort_1 \cdot map\ rrot \cdot sort_n \cdot rots$$

It says that given a matrix of sorted rotations, if we move the last column to the right, and stably sort them on the first character, we get the same matrix back again. More generally, (7.4) states that the following transformation on the sorted rotations is the identity: move the last column to the front and resort the rows on the new first column. As we will see, this implies that the (stable) permutation that produces the first column from the last column is the same as that which produces the second from the first, and so on.

To prove (7.4) we will need some basic properties of rotations and sorting. For rotations, one identity suffices:

$$map\ rrot \cdot rots \quad = \quad rrot \cdot rots \tag{7.5}$$

More generally, applying a rotation to the columns of a matrix of rotations has the same effect as applying the same rotation to the rows.

For sorting we will need

$$sort_k \cdot map\ rrot^k \quad = \quad (sort_1 \cdot map\ rrot)^k \tag{7.6}$$

where f^k is the composition of f with itself k times. Equivalently, equation (7.6) can be read as $sort_k = (sort_1 \cdot map\ rrot)^k \cdot map\ lrot^k$. This identity formalises the fact that one can sort a matrix on its first k columns by first rotating the matrix to bring these columns into the last k positions, and then repeating k times the process of rotating the last column into first position and *stable* sorting according to the first column only. Since $map\ rrot^n = id$, the initial processing is omitted in the case $k = n$, and we have the standard definition of *radix sort*. In this context see [34] which deals with the derivation of radix sorting in a more general setting.

Substituting $k+1$ for k in (7.6) and expanding the right-hand side, we obtain

$$sort_{(k+1)} \cdot map\ rrot^{k+1} \quad = \quad sort_1 \cdot map\ rrot \cdot sort_k \cdot map\ rrot^k$$

Since $rrot^k \cdot rrot^{n-k} = rrot^n = id$ we can compose both sides with $map\ rrot^{n-k}$ to obtain

$$sort_{(k+1)} \cdot map\ rrot \quad = \quad sort_1 \cdot map\ rrot \cdot sort_k \tag{7.7}$$

Finally, we will need the following two properties of columns. Firstly, for arbitrary j and k:

$$cols\ j \cdot sort_k \quad = \quad cols\ j \cdot sort_{j \sqcap k} \quad = \quad sort_{j \sqcap k} \cdot cols\ j \tag{7.8}$$

where $\sqcap$ returns the smaller of its two arguments. In particular, $cols\ j \cdot sort_k = cols\ j \cdot sort\ j$ whenever $j \leq k$. Furthermore, since *sort k* sorts the list of strings by the first k characters only, we have:

$$cols\ j \cdot sort_k \cdot perm \quad = \quad cols\ j \cdot sort_k \tag{7.9}$$

whenever $j \leq k$ and *perm* is any function that permutes its argument.

Having introduced the fundamental properties (7.5), (7.7), (7.8) and (7.9), we can now prove (7.4). With $1 \leq j \leq k$ we reason:

$$\begin{aligned}
& sort_1 \cdot cols\ j \cdot map\ rrot \cdot sort_k \cdot rots \\
= \quad & \{\text{by } (7.8)\} \\
& cols\ j \cdot sort_1 \cdot map\ rrot \cdot sort_k \cdot rots \\
= \quad & \{\text{by } (7.7)\} \\
& cols\ j \cdot sort_{k+1} \cdot map\ rrot \cdot rots \\
= \quad & \{\text{by } (7.8)\} \\
& cols\ j \cdot sort_k \cdot map\ rrot \cdot rots \\
= \quad & \{\text{by } (7.5)\} \\
& cols\ j \cdot sort_k \cdot rrot \cdot rots \\
= \quad & \{\text{by } (7.9)\} \\
& cols\ j \cdot sort_k \cdot rots
\end{aligned}$$

Thus, (7.4) is established.

7.3 Recreating the Matrix

Our aim is to develop a program that reconstructs the sorted matrix from its last column. In other words, we aim to construct $\mathit{sort}\ n \cdot \mathit{rots} \cdot \mathit{unbwt}\ t$. In fact, we will try to construct a more general expression $\mathit{cols}\ j \cdot \mathit{sort}\ k \cdot \mathit{rots} \cdot \mathit{unbwt}\ t$ (of which the former expression is the case $j = k = n$) because the more general expression is used in the two variants of the BWT described in Sections 7.5 and 7.6.

First observe that for $0 \leq j$:

$$\mathit{cols}\ (j+1) \cdot \mathit{map}\ \mathit{rrot} \quad = \quad \mathit{join} \cdot \langle \mathit{map}\ \mathit{last}, \mathit{cols}\ j \rangle \tag{7.10}$$

where $\mathit{join}\ (x, xs) = \mathit{zipWith}\ (:)\ x\ xs$, the matrix xs with x adjoined as a new first column. Hence:

$$\begin{array}{ll}
& \mathit{cols}\ (j+1) \cdot \mathit{sort}_k \cdot \mathit{rots} \cdot \mathit{unbwt}\ t \\
= & \quad \{\text{by } (7.4)\} \\
& \mathit{sort}_1 \cdot \mathit{cols}\ (j+1) \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{sort}_k \cdot \mathit{rots} \cdot \mathit{unbwt}\ t \\
= & \quad \{\text{by } (7.10)\} \\
& \mathit{sort}_1 \cdot \mathit{join} \cdot \langle \mathit{map}\ \mathit{last}, \mathit{cols}\ j \rangle \cdot \mathit{sort}_k \cdot \mathit{rots} \cdot \mathit{unbwt}\ t \\
= & \quad \{\text{products: } \langle f, g \rangle \cdot h = \langle f \cdot h, g \cdot h \rangle\} \\
& \mathit{sort}_1 \cdot \mathit{join} \cdot \langle \mathit{map}\ \mathit{last} \cdot \mathit{sort}_k \cdot \mathit{rots} \cdot \mathit{unbwt}\ t, \mathit{cols}\ j \cdot \mathit{sort}_k \cdot \mathit{rots} \cdot \mathit{unbwt}\ t \rangle
\end{array}$$

In particular, consider $t = \mathit{bwn}\ xs$ for an input xs and $k = n$, the length of xs. Since $\mathit{bwp} = \mathit{map}\ \mathit{last} \cdot \mathit{sort}\ n \cdot \mathit{rots}$, and $\mathit{bwp}\ (\mathit{unbwt}\ t\ xs) = xs$, the equality shown above reduces to:

$$\begin{array}{l}
(\mathit{cols}\ (j+1) \cdot \mathit{sort}\ n \cdot \mathit{rots} \cdot \mathit{unbwt}\ t)\ xs \\
\qquad = \ (\mathit{sort}\ 1 \cdot \mathit{join} \cdot \langle \mathit{id}, \mathit{cols}\ j \cdot \mathit{sort}\ n \cdot \mathit{rots} \cdot \mathit{unbwt}\ t \rangle)\ xs
\end{array}$$

Setting $\mathit{recreate}\ j = \mathit{cols}\ j \cdot \mathit{sort}\ n \cdot \mathit{rots} \cdot \mathit{unbwt}\ t$, we have just constructed a recursive definition for $\mathit{recreate}$:

$$\begin{array}{lcl}
\mathit{recreate}\ 0 & = & \mathit{map}\ (\mathit{const}\ [\,]) \\
\mathit{recreate}\ (j+1) & = & \mathit{sort}\ 1 \cdot \mathit{join} \cdot \mathit{fork}\ (\mathit{id}, \mathit{recreate}\ j)
\end{array}$$

The Haskell code for $\mathit{recreate}$ is given in Figure 7.2. A function call to $\mathit{recreate}\ j$ reconstructs the first j columns of the sorted matrix of rotations.

The function $\mathit{sortby} :: (A \rightarrow A \rightarrow \mathit{Bool}) \rightarrow \mathit{List}\ A \rightarrow \mathit{List}\ A$ is a stable sort. Its type slightly varies from the standard function sortBy.

Now that $\mathit{recreate}$ reconstructs the matrix, we just need to pick a particular row. Taking $j = n$, we have $\mathit{unbwt}\ t = (!!\ t) \cdot \mathit{recreate}\ n$. This implementation of $\mathit{recreate}$ involves computing sort_1 a total of j times. To avoid repeated sorting, observe that $\mathit{recreate}\ 1\ y = \mathit{sort}\ y$, where sort now sorts a list rather than a matrix of one column. That is, each time in the recursive call we really just repeatedly perform the same permutation as sorting the string y. We can thus just sort y once, remember the permutation, and re-apply it afterwards.

```
recreate :: Ord a => Int -> [a] -> [[a]]
recreate 0     = map (const [])
recreate (j+1) = sortby leq . join . fork (id, recreate j)
    where leq us vs = head us <= head vs
          join = uncurry (zipWith (:))
          fork (f,g) x = (f x, g x)
```

Figure 7.2: Computation of *recreate*

We represent a permutation by a function of type $\mathcal{Z} \to \mathcal{Z}$ and define a function *permby* rearranging a list according to a given permutation, such that:

$$permby\,p\,[x_0,\ldots,x_{n-1}] \quad = \quad [x_{p(0)},\ldots,x_{p\,(n-1)}]$$

More precisely, *permby* can be defined by:

$$permby\,p\,x \quad = \quad map\,((x!!)\cdot p)\,[0..length\,x-1]$$

For each y, we can construct a permutation sp as below:

$$\begin{array}{lcl} sp_y & = & snd \cdot spl_y \\ spl_y\,i & = & sort\,(enum\,y)\,!!\,i \\ enum\,y & = & zip\,y\,[0..length\,y-1] \end{array}$$

It is then obvious that

$$sort\,(\leq)\,y \quad = \quad permby\,sp_y\,y$$

In other words, sp_y remembers the permutation sorting y. Now we just need to sort y once to find out the permutation sp_y, and reuse it in the body of *recreate*. Furthermore, *permby* sp_y is a natural transformation, which is important for the next section. We will omit the subscript of sp_y when it is clear from the context.

Denoting function application by $\bullet$, which binds looser than function composition. The recursive case for *recreate* can be written as:

$$recreate\,(j+1)\,y \quad = \quad join\cdot permby\,sp_y\cdot\langle id, recreate\,j\rangle \bullet y$$

7.4 Picking a Row from the Matrix

We are now able to rebuild the sorted matrix from its last column. However, we do not need the entire matrix, but demand only one specific row of it. In this section, we are going to perform an optimisation such that we can build just that row. The derivation makes heavy use of naturality and even higher-order naturality.

The first step is to build the matrix iteratively, rather than recursively. Recall the Prelude function $transpose :: List\,(List\,A) \to List\,(List\,A)$ for transposing a matrix, which we will abbreviate to *trans* below. Also define:

$$iter_1\,f\,a \quad = \quad f\,a : iter_1\,f\,(f\,a)$$

It is a variation of the Haskell Prelude function *iterate*. It can be shown by the approximaton lemma [15, Chapter 7] that:

$$iter_1\, f \quad = \quad cons \cdot \langle f, map\, f \cdot iter_1\, f \rangle$$

and it therefore follows that:

$$take\,(j+1) \cdot iter_1\, f \quad = \quad cons \cdot \langle f, map\, f \cdot take\, j \cdot iter_1\, f \rangle \tag{7.11}$$

The first aim of this section is to prove that the sorted matrix can be rebuilt using $iter_1$. More precisely, we aim at showing the following equality:

$$trans \cdot recreate\, j \bullet y \quad = \quad take\, j \cdot iter_1\,(permby\ sp) \bullet y$$

Once it is established, we have the following alternative definition of *recreate* in terms of $iter_1$:

$$recreate\, j\, y \quad = \quad trans \cdot take\, j \cdot iter_1\,(permby\ sp) \bullet y \tag{7.12}$$

The base case is easily established: when $j = 0$ both sides reduce to empty lists. We reason for the inductive case:

$$\begin{array}{cl}
 & trans \cdot recreate\,(j+1) \bullet y \\
= & \quad \{\text{definition}\} \\
 & trans \cdot sort_1 \cdot join \cdot \langle id, recreate\, j \rangle \bullet y \\
= & \quad \{\text{find } sp \text{ such that } sort\ y = permby\ sp\ y\} \\
 & trans \cdot permby\ sp \cdot join \cdot \langle id, recreate\, j \rangle \bullet y \\
= & \quad \{\text{higher-order naturality: } \eta \cdot zip = zip \cdot \langle \eta, \eta \rangle\} \\
 & trans \cdot join \cdot \langle permby\ sp, permby\ sp \cdot recreate\, j \rangle \bullet y \\
= & \quad \{\text{higher-order naturality: } \eta = map\, \eta \cdot trans\} \\
 & trans \cdot join \cdot \langle permby\ sp, map\,(permby\ sp) \cdot trans \cdot recreate\, j \rangle \bullet y \\
= & \quad \{\text{since } trans \cdot join = cons\} \\
 & cons \cdot \langle permby\ sp, map\,(permby\ sp) \cdot trans \cdot recreate\, j \rangle \bullet y \\
= & \quad \{\text{induction}\} \\
 & cons \cdot \langle permby\ sp, map\,(permby\ sp) \cdot take\, j \cdot iter_1\,(permby\ sp) \rangle \bullet y \\
= & \quad \{\text{by (7.11), with } f = permby\ sp\} \\
 & take\,(j+1) \cdot iterate\,(permby\ sp) \bullet y
\end{array}$$

We have thus established (7.12).

How does that help to develop an algorithm picking a particular row in the matrix? Let t be the row of interest, we wish to some how fuse $(!!t)$ into *recreate*, and come up with an algorithm which efficiently builds just that row. Let us start with:

$$\begin{array}{cl}
 & (!!t) \cdot recreate\, j \bullet y \\
= & \quad \{\text{definition}\}
\end{array}$$

$$
\begin{array}{ll}
& (!!t) \cdot \mathit{trans} \cdot \mathit{take}\, j \cdot \mathit{iter}_1\, (\mathit{permby}\; \mathit{sp}) \bullet y \\
= & \quad \{\text{since } (!!t) \cdot \mathit{trans} = \mathit{map}\, (!!t)\} \\
& \mathit{map}\, (!!\, t) \cdot \mathit{take}\, j \cdot \mathit{iter}_1\, (\mathit{permby}\; \mathit{sp}) \bullet y \\
= & \quad \{\text{naturality: } \mathit{map}\, f \cdot \mathit{take}\, j = \mathit{take}\, j \cdot \mathit{map}\, f\} \\
& \mathit{take}\, j \cdot \mathit{map}\, (!!t) \cdot \mathit{iter}_1\, (\mathit{permby}\; \mathit{sp}) \bullet y
\end{array}
$$

To push $\mathit{map}\, (!!t)$ further to the right, one naturally recalls the following property which can be easily proved by the approximation lemma:

$$\mathit{map}\, h \cdot \mathit{iter}_1\, f = \mathit{iter}_1\, g \cdot h \quad \Leftarrow \quad h \cdot f = g \cdot h \tag{7.13}$$

To make use of it, however, we have to find a function g such that $(!!t) \cdot \mathit{permby}\; \mathit{sp} = g \cdot (!!t)$.

Recall the definition of spl in the last section:

$$\mathit{spl}_y\, i \quad = \quad \mathit{sort}\, (\mathit{enum}\, y)\; !!\; i$$

Consider, for a fixed y, the list constructed in spl_y. The elements in y are (stably) sorted. Furthermore, each of the elements is augmented with an index, indicating where it came from. For example, take the string `hmooakya`, we get

```
a a h k m o o y
4 7 0 5 1 2 3 6
```

If we apply the same permutation to `aahkmooy` again, we get

```
m y a o a h k o
1 6 4 2 7 0 5 3
```

Now notice that each of the numbers `47051236` also indicates which item is going to occupy its place. For example, the place of the first `a` is occupied by the element indexed 4 in the list, `m`. The same happens when you apply the permutation again. The above formalises to the following lemma:

Lemma 7.1 Let x be a list, p a permutation, and define $\mathit{pl}_x\, i = \mathit{permby}\, p\, (\mathit{enum}\, x)\; !!\; i$. We then have:

$$\mathit{iter}_1\, (\mathit{permby}\, p)\, x \quad = \quad \mathit{map}\, (\mathit{map}\, \mathit{fst}) \cdot \mathit{iter}_1\, (\mathit{map}\, (\mathit{pl}_x \cdot \mathit{snd})) \cdot \mathit{enum} \bullet x$$

Proof of Lemma 7.1 will be given in Appendix A. Furthermore, by the naturality property $(!!t) \cdot \mathit{map}\, f = f \cdot (!!t)$, we have

$$(!!t) \cdot \mathit{map}\, (\mathit{spl} \cdot \mathit{snd}) = \mathit{spl} \cdot \mathit{snd} \cdot (!!t)$$

That is indeed what we need to make use of (7.13)! We reason:

$$
\begin{array}{ll}
& \mathit{take}\, j \cdot \mathit{map}\, (!!t) \cdot \mathit{iter}_1\, (\mathit{permby}\; \mathit{sp}) \bullet y \\
= & \quad \{\text{by Lemma 7.1}\} \\
& \mathit{take}\, j \cdot \mathit{map}\, (!!t) \cdot \mathit{map}\, (\mathit{map}\, \mathit{fst}) \cdot \mathit{iter}_1\, (\mathit{map}\, (\mathit{spl} \cdot \mathit{snd})) \cdot \mathit{enum} \bullet y \\
= & \quad \{\text{naturality: } (!!t) \cdot \mathit{map}\, f = f \cdot (!!t)\} \\
& \mathit{take}\, j \cdot \mathit{map}\, \mathit{fst} \cdot \mathit{map}\, (!!t) \cdot \mathit{iter}_1\, (\mathit{map}\, (\mathit{spl} \cdot \mathit{snd})) \cdot \mathit{enum} \bullet y \\
= & \quad \{(7.13), \text{ since } \mathit{enum}\, y!!t = (y!!t, t)\ \} \\
& \mathit{take}\, j \cdot \mathit{map}\, \mathit{fst} \cdot \mathit{iter}_1\, (\mathit{spl} \cdot \mathit{snd}) \bullet (y!!t, t)
\end{array}
$$

```
unbwt :: Ord a => Int -> [a] -> [a]
unbwt t y = take (length y) (thread t)
  where spl i = sort (zip y [0..]) !! i
        thread i = x : thread j
           where (x,j) = spl i
```

Figure 7.3: Computation of *unbwt*

Finally, *map fst* and $iter_1$ can be combined into one loop, which leads to the following algorithm:

$$\begin{array}{rcl} \mathit{unbwt}\; t\; y & = & \mathit{take}\,(\mathit{length}\; y)\,(\mathit{thread}\; t) \\ \textbf{where} & & \mathit{thread} \;=\; \mathit{cons} \cdot (\mathit{id} \times \mathit{thread}) \cdot \mathit{spl} \\ & & \mathit{spl}\; i \;=\; \mathit{sortby}\,(\mathit{zip}\; y\,[0..])\; !!\; i \end{array}$$

Its Haskell implementation is given in Figure 7.3. In a real implementation, the sorting in *spl* would be performed by counting the histogram of the input, which can be done in linear time using a mutable array. The "threading" part can be performed in linear time, assuming constant-time array looking up. Seward [80] observed that the main inefficiency lies in the cache misses involved in the threading, as a result of accessing the big array in a non-sequential order.

7.5 Schindler's variation

The main variation of BWT is to exploit the general form of (7.4) rather than the special case $k = n$. Suppose we define

$$\mathit{bwpS}\; k \quad = \quad \mathit{map}\; \mathit{last} \cdot \mathit{sort}_k \cdot \mathit{rots}$$

This version, which sorts only on the first k columns of the rotations of a list, was considered in [77]. The derivation of the previous section shows how we can recreate the first k columns of the sorted rotations from $y = \mathit{bwp}\; k\; x$, namely by computing $\mathit{recreate}\; k\; y$.

The remaining columns cannot be computed in the same way. However, we can reconstruct the tth row, where $t = \mathit{bwn}\; k\; x$ and

$$\mathit{sort}_k\,(\mathit{rots}\; x)\; !!\; t \quad = \quad x$$

The first k elements of x are given by $\mathit{recreate}\; k\; y\; !!\; t$, and the last element of x is $y!!t$. Certainly we know

$$\mathit{take}\; k\,(\mathit{rrot}\; x) \quad = \quad [x_n, x_1, \ldots, x_{k-1}]$$

This list begins with the *last* row of the unsorted matrix, and consequently, since sorting is stable, will be the *last* occurrence of the list in $\mathit{recreate}\; k\; y$. If this occurrence is at position p, then $y!!p = x_{n-1}$. Having discovered x_{n-1}, we know $\mathit{take}\; k\,(\mathit{rrot}^2\; x)$. This list

```
unbwt :: Ord a => Int -> Int -> [a] -> [a]
unbwt k p y = us ++ reverse (take (length y - k) v)
  where u  = ys !! p
        ys = recreate k y
        v  = a:search k (reverse (zip ys y)) (take k (a:u))
        a   = y !! p

search :: Eq a => Int -> [([a],a)] -> [a] -> [a]
search k table x = a:search k table' (take k a:x))
                   where (a,table') = dlookup table x

dlookup :: Eq a => [(a,b)] -> a -> (b,[(a,b)])
dlookup ((a,b):abs) d = if a==d then (b,abs)
                        else (c,(a,b):cds)
                        where (c,cds) = dlookup abs d
```

Figure 7.4: Computation of Schindler's variation

begins the penultimate row of the unsorted matrix, and will be either the last occurrence of the list in the sorted matrix, or the penultimate one if it is equal to the previous list. We can continue this process to discover all of $[x_{k+1}, \ldots, x_n]$ in reverse order. Efficient implementation of this phase of the algorithm requires building an appropriate data structure for repeatedly looking up elements in reverse order in the list $zip\,(recreate\;k\;y)\;y$ and removing them when found. A simple implementation is given in Figure 7.4.

7.6 Chapin and Tate's variation

Primarily for the purpose of showing that the pattern of derivation in this chapter can be adapted to other cases, we will consider another variation. Define the following alternative of BWT:

$$bwpCT\;k \quad = \quad map\;last \cdot twists\;k \cdot sort \cdot rots$$

Here the function $twists\;k$ rearranges the rows of the matrix and is defined as a sequence of steps:

$$\begin{aligned} twists\;0 \quad &= \quad id \\ twists\;(k+1) \quad &= \quad tstep\;(k+1) \cdot twists\;k \end{aligned}$$

One possible choice of $tstep$ is shown in Figure 7.5. As an example, consider the rotations of the string `aabab`:

```
tstep :: Eq a => Int -> [[a]] -> [[a]]
tstep k = concat . mapEven (map reverse). groupby (take k)

mapEven, mapOdd :: (a->a) -> [a] -> [a]
mapEven f [] = []
mapEven f (x:xs) = f x : mapOdd f xs
mapOdd f [] = []
mapOdd f (x:xs) = x : mapEven f xs
```

Figure 7.5: One possible choice of *tstep*

```
aabab    ababa    abaab
abaab    abaab    ababa
ababa    aabab    aabab
baaba    baaba    babaa
babaa    babaa    baaba
```

Shown on the left is the sorted matrix of rotations. The matrix in the middle is the result of applying *tstep* 1. The rows are first partitioned into groups by *groupby* according to their first characters. The even numbered groups (counting from zero) are then reversed. In the example, the group starting with `a` is reversed. Shown on the right is the result of applying *tstep* 2 to the matrix in the middle. The rows are partitioned into three groups, starting with `ab`, `aa`, and `ba` respectively. The noughth and the second group are reversed.

The idea of twisting the matrix of sorted rotations was proposed in [23], where a similar but slightly more complicated version of *tstep* was considered based on the Gray code. Chapin and Tate's generalisation can marginally improve the compression ratio of the transformed text.

What we require from *twists* to be invertible, however, is not specific to any particular *tstep*: we need only the property that for $0 < j \leq k$,

$$\mathit{cols}\, j \cdot \mathit{twists}\, k \quad = \quad \mathit{cols}\, j \cdot \mathit{twists}\, (j-1) \tag{7.14}$$

In words, further twisting (*twists* k where $j \leq k$) does not change the first j columns after they have been set by *twists* $(j-1)$. In the example above, for instance, the call to *tstep* 2 does not change the first two columns of the matrix in the middle, nor do successive calls to *tstep* k where $k \geq 2$. Any *tstep* allowing *twists* to satisfy (7.14) suffices to make *bwtCT* invertible. This separation of concerns on compression rate and invertibility means that one can try many possible choices satisfying (7.14) and experiment with the effect on compression.

To derive an algorithm for the reverse transform we need the following analogue of (7.4):

$$\begin{aligned} & \mathit{col}\, j \cdot \mathit{twists}\, k \cdot \mathit{sort} \cdot \mathit{rots} \\ = \quad & \mathit{twists}\, k \cdot \mathit{sort}\, 1 \cdot \mathit{untwists}\, k \cdot \mathit{cols}\, j \cdot \mathit{map}\, \mathit{rrot} \cdot \mathit{twists}\, k \cdot \mathit{sort} \cdot \mathit{rots} \end{aligned} \tag{7.15}$$

where $\mathit{untwist}\,k$ is inverse to $\mathit{twists}\,k$. The proof of (7.15) follows a similar path to the derivation in Section 7.2. When $k = 0$ (so $\mathit{twists}\,k = \mathit{id}$) equation (7.15) reduces to a special case of (7.4). In words, (7.15) means that the following operation is an identity on a matrix generated by $\mathit{twists}\,k \cdot \mathit{sort} \cdot \mathit{rots}$: move the last column to the first, untwist it, sort it by the first character, and twist it again.

Based on (7.15) one can now derive an algorithm similar to that of Section 7.3. Defining

$$\mathit{recreateCT}\;j\;k \quad = \quad \mathit{col}\,j \cdot \mathit{twists}\,k \cdot \mathit{sort} \cdot \mathit{rots} \cdot \mathit{unbwtCT}\;t$$

we can construct a recursive definition for *recreateCT* which is similar to (7.12), but with the permutation *sp* simulating $\mathit{twists}\,i \cdot \mathit{sort}\,1 \cdot \mathit{untwists}\,i$ for appropriate i, rather than just $\mathit{sort}\,1$. The details are more complicated than for the corresponding definition of *recreate* (which builds one column in each step) because in *recreateCT* the permutation *sp* changes each time a new column is built. So the algorithm has to construct a new permutation as well as a new column at each step. The resulting algorithm will thus return a pair whose first component is the reconstructed matrix and the second component is a permutation representating *sp*. In the first step we build the first column and a permutation simulating $\mathit{twists}\,1 \cdot \mathit{sort}\,1 \cdot \mathit{untwists}\,1$; in the second step we build the second column and a permutation for $\mathit{twists}\,2 \cdot \mathit{sort}\,1 \cdot \mathit{untwists}\,2$, and so on. Further details are omitted.

7.7 Conclusions

We have shown how the inverse Burrows-Wheeler transform can be derived by equational reasoning. The derivation can be re-used to invert the more general versions proposed by Schindler and by Chapin and Tate.

Other aspects of the BWT also make interesting topics. The BWT can be modified to sort the tails of a list rather than its rotations, and in [59] it is shown how to do this in $O(n \log n)$ steps using suffix arrays. How efficiently it can be done in a functional setting remains unanswered, though we conjecture that $O(n(\log n)^2)$ steps is the best possible.

Chapter 8

Conclusion

Looking back at the promises we made in the beginning of the thesis, we have shown that the inverse function is a useful tool for specification. Indeed, tasks like compression and decompression certainly imply connection with inverse functions. Even for some tasks in which we do not immediately see such a connection, such as breadth-first search or the string edit problem, the presence of inverse functions in their specifications may come as a surprise. It shows that program derivation involving inverse functions certainly deserves more attention.

The converse-of-a-function theorem plays a central role in this thesis. The compositional approach to function inversion, presented in Chapter 3, inverts a fold to an unfold and vice versa. The converse-of-a-function theorem, on the other hand, inverts any function that satisfies its premises to a fold. To invert a function with the theorem, what matters is not how it is defined but what properties it satisfies. This technique is not new. Similar techniques have been adopted in, for example, [50] and [72]. However, to the best our knowledge, it was de Moor [17, 67] who first presented the technique as a theorem, suggesting a wider range of application. The problem dealt with in [67] was precedence parsing, leading to a derivation of Floyd's algorithm. Recently, Hinze [41] solved the problem again by a different approach avoiding the introduction of a spine representation.

We have applied the converse-of-a-function theorem to a number of examples. The inversion usually results in a non-deterministic fold. It is often composed before some other function which acts as a filter. The fold fusion theorem is then applied to fuse the filter into the fold to remove its non-determinism, refining the specification to an implementable function. This pattern of derivation turned out to be useful in solving many problems.

In the sections to follow, we will discuss some related work and future directions.

8.1 Relations and Non-determinism

Encapsulating non-determinism, relations provide a natural and concise framework to extend inversion to non-injective functions. The non-determinism can either be eliminated later in the specification, or by taking the breadth of the constructed relation. The price we pay, however, is having to bring in a heavy machinery: the algebra of relations is notorious for having too many rules, and program transformation based on relational inclusion rather than simple equivalence adds to the complexity. We have shown, at least for the examples in this thesis, that the complexity is still within a manageable scale.

An alternative approach is to use set-valued functions [40, 41]. The pro is, besides getting back to simple and nice equational reasoning, that the resulting program is closer to its functional (or in particular, Haskell) implementation – there is no need to take the breadth. The con, on the other hand, is having to take care of the bookkeeping details of maintaining a set of results, which is implicit in the relational approach.

We use relations to model non-determinism by allowing one item in the domain to be mapped to more than one items in the range. This mapping, however, does not distinguish between angelic and demonic non-determinism. Dijkstra proposed in [28], as a healthiness condition, that predicate transformers should be conjunctive. The guarded command language of Dijkstra thus captures demonic non-determinism. Back and von Wright [84, 3, 4] released the restriction and considered disjunctive as well, adding angelic non-determinism to the language. In [4], they related this calculus to program inversion and showed that the inverse of a demonic program is angelic, and vice versa.

Much less has been done on modelling both style of non-determinism using relations. An recent attempt was made by Rewitzky [73], where she proposed using upclosed multirelations to capture both angelic and demonic non-determinism. More examples and applications are in high demand, and it is interesting to see whether program inversion provides good applications.

8.2 The Converse-of-a-Function Theorem

One natural question is how widely the converse-of-a-function theorem can be applied. In other words, how to determine whether the converse-of-a-function theorem can be applied to a particular function. Part of the answer is given by Gibbons and Hutton in [36]. If the converse of a function can be written as a fold, the function itself must be an unfold. The necessary and sufficient conditions for a function to be an unfold given in [36] can thus be used as a test before applying the converse-of-a function theorem.

We have proved a generalisation of the converse-of-the-function theorem, which inverts a simple relation to a hylomorphism. The generalised converse-of-a-function theorem extends the original one in two ways: it inverts partial as well as total functions, and the result can be a hylomorphism rather than a fold. For all the examples we currently have, it suffices to invert a total function before fusing a constraint into it. There is thus less need to invert partial functions. On the other hand, being able to construct hylomorphisms does cover a much wider range of algorithms. It also allows one to introduce a base functor independent from the input or output types. However, this extra degree of freedom also means there is less help on how it could be used. We have applied the theorem to a special case, choosing a particular base functor such that we can express a loop as a hylomorphism. The author is keen to see more examples for which the general theorem is necessary.

The theorem is formulated and proved with the concept of inductivity. In [36], on the other hand, the central concept is the *kernel* of a function. It would be useful to have a variation of the converse-of-a-function theorem based on kernels rather than the more obscure concept of inductivity. This would make an interesting future work.

8.3 Tree Construction and the Spine Representation

Many examples in this thesis involves building trees, and in many of them we did so by introducing a spine representation. One might complain that it is too inventive a step, if not itself the answer to the problem. Our defence is that the spine representation is invented to enable traversing the tree upwards from the left-most tip, which is more a concern of efficiency than an algorithmic one. Indeed, in Chapter 6 where we discussed the Countdown problem, we attempted to solve the sub-problem of constructing all oriented trees from a list. Since all trees are needed, being able to traverse from the bottom does not give one too many advantages and the spine representation is not used.

Encouragingly, the converse-of-a-function seems to provide just the right tool for this particular task. It is superior to the top-down approach to construct oriented trees both in clarity and efficiency.

The actions we perform on a spine (rolling a subtree down the spine and attaching a new leaf to the left) resemble reducing and shifting in a shift-reduce parser. Indeed, the motivating application for the invention of the converse-of-a-function theorem was precedence parsing [67] in the abstract form of constructing heaps. The result was a derivation of Floyd's algorithm. It is certainly possible to derive a full shift-reduce parser using the theorem, although it may be a laborious exercise.

Some earlier algorithms solve problems similar to those in Chapter 4 and 5 without the use of the spine representation, at least not explicitly. The problem of rebuilding a binary tree from its traversals has been discussed by, among many, Chen and Udding [24] and van de Snepscheut [81]. The derivation of Chen and Udding started with converting the recursive characterisation of prefix and infix to an iterative one. As as result he explicitly introduced a stack, which served the same purpose of the spine we use. Van de Snepscheut's algorithm, whose functional counterpart is presented in Chapter 3, evolves from the recursive definitions directly. The problem of constructing heaps was also dealt with Schoenmakers [78] and Hinze [41]. Hinze come up with an algorithm by first performing a tupling transformation, then turning the top-down algorithm bottom-up. In those algorithms without explicit use of the spine representation, however, one can still see the result as implicitly storing the spine in the stack.

It has been pointed out by Backhouse that the problem of building trees of minimum height can be seen as an instance of Knuth's generalised shortest path problem [51]. The problem addressed was, given a context-free grammar and a cost function on parse trees, to construct a word and a parse tree whose cost is minimum. Given a list of numbers, we can construct an ambiguous grammar whose only word is the list, while the possible parse trees includes all binary trees. The cost of a parse tree would simply be its height.

Knuth's algorithm can thus be applied to find the best parse yielding the minimum height. It would be interesting to investigate whether the linear time algorithm in Chapter 5 is an optimised special case and how they relate to each other.

8.4 More on Compression and Decompression

Compression and decompression are natural candidates of examples of inverse functions. Some compression methods, such as the run-length encoding or the simple dictionary look-up method, are rather trivial considering constructing the reverse algorithm from the forward algorithm. In this thesis we talked about the Burrows-Wheeler transform, which acts as a preprocessor to compression. The transformed string is not compressed, but is put in a form making the compression phase more effective. The transform attracted our attention because it is not obvious at all at the first glance how to perform the inverse transform.

Arithmetic coding makes another interesting case. The most naive version of the encoding phase takes a string and outputs a rational number in the interval [0..1). It starts with the interval [0..1) and successively narrows the interval, with respect to a model designed by statistics, while processing the input string from left to right. Finally a rational number is chosen from the resulting interval. The decoding phase, on the other hand, takes the rational number and recovers the string starting from the leftmost character. One can see that the encoding process is a *foldl*, while decoding *unfoldr*. It is a pattern not covered in this thesis.

In their recent work, Bird and Stratford [20] showed how to formally specify arithmetic encoding and derive from it the decoding algorithm. They took into account the change of model (which, interestingly, turned out to be necessary to justify further optimisations) and specified the interval-narrowing process in the form $\mathit{foldl}\ (\otimes)\ e \cdot \mathit{unfoldr}\ \mathit{gen}$. A novel theorem was presented addressing on how to invert functions of the form. The idea was that the reverse algorithm simulates each step of the forward algorithm.

Compression of structured data can be more effective if its structure information can be exploited. In [47, 48], Jansson and Jeuring extended compositional program inversion to polytypic data. They ensured that an generic operation and its inverse are always constructed in pairs. Generic, structure-specific compression and decompression were among their examples. It is largely orthogonal with conventional, bit-stream compressors and they can be used together to achieve better compression rate.

8.5 Mechanised Approaches to Inverse Computation

This thesis is about program derivation. Consequently, our aim is not, say, to answer which binary tree yields a particular pair of prefix and infix traversals; rather, we are interested in producing algorithms that answer the question.

Researchers from the field of partial evaluation took complementary approaches [74, 49, 2]. In [2], Abramov and Glück attempted at a universal method to *inverse computation* via an *inverse interpreter*. At the core of their approach is a flexible, finite representation of possibly infinite sets. Here is a sketch how the inverse interpreter works. Initially, the input is unconstrained or specified by the user. All the paths of the program to invert are systemically traced with the help of a partial evaluation technique called the *universal resolving algorithm*. Eventually, the input/output pairs of the program are stored in a table. The user can then query the system with questions like "For what numbers n would *even* n yield *True*?", "What trees have breadth-first traversal $[1, 2, 3, 4, 5]$?", or even "What strings do *not* contain `AAA` as a substring?"

Many interesting results were obtained by exploiting partial evaluation techniques. Firstly, after one implements an inverse interpreter as above for some programming language L, one does not have to repeat the work for another language N. One just needs to write an ordinary interpreter for N in L, and partial evaluation produces an inverse interpreter for N for free. Secondly, partially evaluating an inverse interpreter with a given program as input produces a *program* that performs the inverted task, which bridges the gap between inverse computation and *inverse compilation*. Generalised versions of these results are presented in [1].

They also advocated an interesting view on the necessity of function inversion: there are mainly three operations we can perform on a function – application, composition and inversion. We know a lot about the former two, but comparatively little about the last, which justifies more efforts to be put on the research on function inversion.

8.6 Reversible Computation and Quantum Computing

On a micro level, the interests in reversibility of computation comes from the desire to reduce heat dissipation and achieve higher density and speed of computing machinery. It is known in thermodynamics as the Landauer's Principle [56] that erasure of information has a non-zero thermodynamics cost, that is, it always generates an increase of the entropy of the universe. Our ordinary model of computation may involve many-to-one functions that lose information. The computing task must be realised by means of digital network and, at some point, this loss of information results in a work-to-heat conversion. It is thus desirable to have a model of computation where irreversibility is restricted, or at least made more explicit. Many such models has been proposed, some based on a Turing machine recording its history [57, 9], some based on logic gates that have extra "garbage lines" [83].

Reversibility is also an interesting topic for quantum computing because quantum computation, obeying the the microscopic laws of physics, is always reversible. This has given rise to the question whether it is possible to develop a suitable programming language for quantum computers, which we know are inherently reversible devices. Efforts in this direction have been reported in, amongst many others, [85, 86].

Bibliography

[1] S. M. Abramov and R. Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12(2):171–211, 2001.

[2] S. M. Abramov and R. Glück. The universal resolving algorithm: inverse computation in a functional language. *Science of Computer Programming*, 43:193–299, 2002.

[3] R. J. R. Back and J. von Wright. Combining angels, demons and miracles in program specifications. *Theoretical Computer Science*, 100:365–383, 1992.

[4] R. J. R. Back and J. von Wright. Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.

[5] R. C. Backhouse. Fixed point calculus. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, number 2297 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[6] R. C. Backhouse, P. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers, 1991.

[7] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. A. Partsch, and S. A. Schuman, editors, *Formal Program Development. Proc. IFIP TC2/WG 2.1 State of the Art Seminar.*, number 755 in Lecture Notes in Computer Science, pages 7–42. Springer-Verlag, January 1992.

[8] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1985.

[9] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[10] R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoretical Computer Science*, 43:123–147, 1986.

[11] R. S. Bird. Lectures on Constructive Functional Programming. In M. Broy, editor, *Constructive Methods in Computing Science*, number 55 in NATO ASI Series F, pages 151–216. Springer-Verlag, 1989.

[12] R. S. Bird. A calculus of functions for program derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison-Wesley, 1990.

[13] R. S. Bird. Generic programming with relations and functors. *Journal of Functional Programming*, 6(1):1–28, 1996.

[14] R. S. Bird. On building trees with minimum height. *Journal of Functional Programming*, 7(4):441–445, 1997.

[15] R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[16] R. S. Bird. Maximum marking problems. Submitted to Functional Pearl, Journal of Functional Programming, 2000.

[17] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[18] R. S. Bird, J. Gibbons, and S.-C. Mu. Algebraic methods for optimization problems. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, number 2297 in Lecture Notes in Computer Science, pages 281–307. Springer-Verlag, January 2002.

[19] R. S. Bird and S.-C. Mu. Inverting the Burrows-Wheeler transform. In R. Hinze, editor, *ACM SIGPLAN 2001 Haskell Workshop*, number 59.2 in Electronic Notes in Theoretical Computer Science, pages 33–40. Elsevier Science Publishers, September 2001.

[20] R. S. Bird and B. Stratford. Arithmetic coding with folds and unfolds. Work in progress.

[21] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994. Research Report 124. Available online at `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html`.

[22] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[23] B. K. Chapin and S. Tate. Higher compression from the Burrows-Wheeler transform by modified sorting. In *Data Compression Conference 1998*, page 532. IEEE Computer Society Press, March 1998. (Poster Session).

[24] W. Chen and J. T. Udding. Program inversion: more than fun! *Science of Computer Programming*, 15:1–13, 1990.

[25] T.-R. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[26] S. Curtis. *A Relational Approach to Optimization Problems*. PhD thesis, Oxford University Computing Laboratory, 1995.

[27] J. Darlington. The structured description of algorithm derivations. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 221–250. Elsevier Science Publishers, 1981.

[28] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[29] E. W. Dijkstra. Program inversion. Technical Report EWD671, Eindhoven University of Technology, 1978.

[30] H. Doornbos. *Reductivity Arguments and Program Construction*. PhD thesis, Eindhoven University of Technology, 1996.

[31] H. Doornbos and R. C. Backhouse. Induction and recursion on datatypes. In B. Möller, editor, *Mathematics of Program Construction, 3rd International Conference*, number 947 in Lecture Notes in Computer Science, pages 242–256. Springer-Verlag, July 1995.

[32] H. Doornbos and R. C. Backhouse. Reductivity. *Science of Computer Programming*, 26:217–236, 1996.

[33] J. Gibbons. An introduction to the Bird-Meertens formalism. *New Zealand Formal Program Development Colloquium Seminar, Hamilton*, November 1994.

[34] J. Gibbons. A pointless derivation of radixsort. *Journal of Functional Programming*, 9(3):339–346, May 1999.

[35] J. Gibbons. Lecture notes on algebraic and coalgebraic methods for calculating functional programs. In *Estonian Winter School on Computer Science*, 1999.

[36] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? In A. Corradini, M. Lenisa, and U. Montanari, editors, *Coalgebraic Methods in Computer Science*, number 44.1 in Electronic Notes in Theoretical Computer Science, April 2001.

[37] J. Gibbons and G. Jones. Linear-time breadth-first tree algorithms: an exercise in the arithmetic of folds and zips. Technical report, University of Auckland, 1993. University of Auckland Computer Science Report No. 71, and IFIP Working Group 2.1 working paper 705 WIN-2.

[38] D. Gries. *The Science of Programming*. Springer Verlag, 1981.

[39] D. Gries and J. L. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 37–42. Addison Wesley, 1990.

[40] P. G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.

[41] R. Hinze. Constructing tournament representations: An exercise in pointwise relational programming. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, Dagstuhl, Germany, July 2002. Springer-Verlag.

[42] P. F. Hoogendijk and O. de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, March 2000.

[43] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[44] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science*, number 1113 in Lecture Notes in Computer Science, pages 407–418, Cracow, September 1996. Springer-Verlag.

[45] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *The Annual European conference on Parallel Processing (Euro-Par'96)*, number 1123 in Lecture Notes in Computer Science, pages 553–562, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[46] G. Hutton. The countdown problem. *Journal of Functional Programming*, 12(6):609–616, 2002.

[47] P. Jansson and J. T. Jeuring. Polytypic compact printing and parsing. In S. D. Swierstra, editor, *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, number 1576 in Lecture Notes in Computer Science, pages 273–287. Springer-Verlag, 1999.

[48] P. Jansson and J. T. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002. Technical report Utrecht University UU-CS-2001-34, 2001.

[49] H. Khoshnevisan and K. M. Stephton. InvX: an automatic function inverter. In N. Dershowitz, editor, *Rewriting Techniques and Applications (RTA '89)*, number 792 in Lecture Notes in Computer Science, pages 564–568. Springer-Verlag, 1989.

[50] E. Knapen. Relational Programming, Program Inversion, and the Derivation of Parsing Algorithms. Master's thesis, Eindhoven University of Technology, 23 November 1993.

[51] D. E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[52] D. E. Knuth. *Axioms and Hulls. Lecture Notes in Computer Science, no. 606.* Springer-Verlag, 1992.

[53] D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms, 3rd Edition.* Addison Wesley, 1997.

[54] D. E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching, 3rd Edition.* Addison Wesley, 1997.

[55] R. E. Korf. Inversion of applicative programs. In *Proceedings of the Seventh Intern. Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 1007–1009. William Kaufmann, Inc., 1981.

[56] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[57] Y. Lecerf. Machines de Turing réversibles. Récursive insolubilité en $n \in N$ de l'équation $u = \theta^n$, où θ est un "isomorphisme de codes". In *Comptes Rendus*, volume 257, pages 2597–2600, 1963.

[58] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Groningen University, The Netherlands, 1990.

[59] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[60] S. Martello and P. Toth. *Knapsack Problems.* Wiley, 1990.

[61] L. Meertens. Algorithmics - towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and L. J. K., editors, *Mathematics and Computer Science*, number 1 in CWI Monographs, pages 3–42. North-Holland Publishing Co., New York, 1987.

[62] L. Meertens. Constructing a calculus of programs. In J. L. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 66–90. Springer-Verlag, 1989.

[63] A. Mili. A relational approach to the design of deterministic programs. *Acta Informatica*, 20:315–328, 1983.

[64] O. de Moor. *Categories, Relations and Dynamic Programming.* PhD thesis, Oxford University Computing Laboratory, 1992.

[65] O. de Moor. A generic program for sequential decision processes. In *PLILP*, pages 1–23, 1995.

[66] O. de Moor and J. Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1), September 1999. Revised version of Technical Report CMS-TR-97-03, School of Computing and Mathematical Sciences, Oxford Brookes University.

[67] O. de Moor and J. Gibbons. Pointwise relational programming. In *Proceedings of Algebraic Methodology and Software Technology 2000*, number 1816 in Lecture Notes in Computer Science, pages 371–390. Springer-Verlag, May 2000.

[68] S.-C. Mu and R. S. Bird. Inverting functions as folds. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, pages 209–232. Springer-Verlag, July 2002.

[69] M. Nelson. Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal*, September 1996.

[70] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[71] C. Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 131–136. ACM Press, September 2000.

[72] C. Pareja-Flores and J. Á. Velázquez-Iturbide. Synthesis of functions by transformations and constraints. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, page 317, Amsterdam, The Netherlands, June 1997. ACM Press.

[73] I. Rewitzky. Binary multirelations. Work in progress.

[74] A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjorner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 422–744. North-Holland Publishing Co., New York, 1988.

[75] B. J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing Journal*, 9:331–348, 1997.

[76] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it Practical: A Generic Linear-Time Algorithm for Solving Maximum-Weightsum Problems. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 137–149. ACM Press, September 2000.

[77] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Data Compression Conference 1997*, page 469. IEEE Computer Society Press, March 1997. (Poster Session).

[78] B. Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In *Mathematics of Program Construction 1992*, number 669 in Lecture Notes in Computer Science, pages 291–301. Springer-Verlag, 1993.

[79] J. Seward. `bzip2`, 2000. `http://sources.redhat.com/bzip2/`.

[80] J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Data Compression Conference 2001*, pages 439–448, 2001.

[81] J. L. van de Snepscheut. Inversion of a recursive tree traversal. Technical Report JAN 171a, California Institute of Technology, May 1991. Available online at `ftp://ftp.cs.caltech.edu/tr/cs-tr-91-07.ps.Z` .

[82] M. Spivey. . Personal communications.

[83] T. Toffoli. Reversible computing. In J. W. d. Bakker, editor, *Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.

[84] J. von Wright. Program inversion in the refinement calculus. *Information Processing Letters*, 37:95–100, 1991.

[85] P. Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 46(6):807–818, 2001. Available online at `http://www.research.ibm.com/journal/rd45-6.html`.

[86] P. Zuliani. *Quantum Programming*. PhD thesis, Oxford University Computing Laboratory, 2001.

Appendix A

Proof of Minor Lemmas

Property (3.1)

The functions *assocl* and *assocr* can be written in point-free style as:

$$\begin{array}{lcl} assocl & = & \langle fst \cdot fst, \langle snd \cdot fst, snd \rangle \rangle \\ assocr & = & \langle \langle fst, fst \cdot snd \rangle, snd \cdot snd \rangle \end{array}$$

For simple S, we have

$$(R \cap T) \cdot S \quad = \quad (R \cdot S) \cap (T \cdot S) \qquad \Leftarrow \qquad S \,\text{simple} \tag{A.1}$$

The proof goes:

$$\begin{array}{cl} & assocl^{\circ} \\ = & \quad \{\text{by definition}\} \\ & (\langle fst \cdot fst, \langle snd \cdot fst, snd \rangle \rangle)^{\circ} \\ = & \quad \{\text{by definition of fork}\} \\ & ((fst^{\circ} \cdot fst \cdot fst) \cap (snd^{\circ} \cdot \langle snd \cdot fst, snd \rangle))^{\circ} \\ = & \quad \{\text{by definition of fork}\} \\ & ((fst^{\circ} \cdot fst \cdot fst) \cap (snd^{\circ} \cdot ((fst^{\circ} \cdot snd \cdot fst) \cap (snd^{\circ} \cdot snd))))^{\circ} \\ = & \quad \{\text{converse distributes into intersection}\} \\ & (fst^{\circ} \cdot fst^{\circ} \cdot fst) \cap (((fst^{\circ} \cdot snd^{\circ} \cdot fst) \cap (snd^{\circ} \cdot snd)) \cdot snd) \\ = & \quad \{(\text{A.1})\} \\ & (fst^{\circ} \cdot fst^{\circ} \cdot fst) \cap (fst^{\circ} \cdot snd^{\circ} \cdot fst \cdot snd) \cap (snd^{\circ} \cdot snd \cdot snd) \\ = & \quad \{(\text{A.1})\} \\ & (fst^{\circ} \cdot ((fst^{\circ} \cdot fst) \cap (snd^{\circ} \cdot fst \cdot snd))) \cap (snd^{\circ} \cdot snd \cdot snd) \\ = & \quad \{\text{definition of fork}\} \\ & \langle (fst^{\circ} \cdot fst) \cap (snd^{\circ} \cdot fst \cdot snd), snd \cdot snd \rangle \\ = & \quad \{\text{definition of fork}\} \\ & \langle \langle fst, fst \cdot snd \rangle, snd \cdot snd \rangle \end{array}$$

$$\begin{array}{ll} = & \{\text{definition of } assocr\} \\ & assocr \end{array}$$

Property (3.2)

Proof.

$$\begin{array}{ll} & cup \cdot \langle \Lambda R, \Lambda S \rangle \\ = & \{\text{definition of } cup\} \\ & \Lambda(\in \cdot (fst \cup snd)) \cdot \langle \Lambda R, \Lambda S \rangle \\ = & \{\text{since } \langle \Lambda R, \Lambda S \rangle \text{ is a function}\} \\ & \Lambda(\in \cdot (fst \cup snd) \cdot \langle \Lambda R, \Lambda S \rangle) \\ = & \{\text{composition distributes into union}\} \\ & \Lambda((\in \cdot fst \cdot \langle \Lambda R, \Lambda S \rangle) \cup (\in \cdot snd \cdot \langle \Lambda R, \Lambda S \rangle)) \\ = & \{\text{since } \Lambda R \text{ and } \Lambda S \text{ are total}\} \\ & \Lambda((\in \cdot \Lambda R) \cup (\in \cdot \Lambda S)) \\ = & \{\text{cancellation}\} \\ & \Lambda(R \cup S) \end{array}$$

□

Lemma 4.2

Proof. The proof makes use of the modular law:

$$(S \cdot R) \cap T \quad = \quad S \cdot (R \cap (S^\circ \cdot T)) \qquad \Leftarrow \qquad S \text{ simple} \tag{A.2}$$

and the following property concerning forks:

$$\langle R, S \rangle^\circ \cdot \langle X, Y \rangle \quad = \quad (R^\circ \cdot X) \cap (S^\circ \cdot Y) \tag{A.3}$$

Also note that $((e ==) \cdot R)?$ is equivalent to $(\Pi \cdot (const\ e)^\circ \cdot R) \cap id$. The proof of Lemma 4.2 goes:

$$\begin{array}{ll} & \langle R, f \rangle^\circ \cdot \langle const\ e, id \rangle \\ = & \{\text{by (A.3) }\} \\ & (R^\circ \cdot const\ e) \cap f^\circ \\ = & \{\text{by modular law (A.2)}\} \\ & ((R^\circ \cdot const\ e \cdot f) \cap id) \cdot f \\ = & \{\text{since } const\ e \cdot S = const\ e \cdot \Pi \text{ if } S \text{ is entire}\} \\ & ((R^\circ \cdot const\ e \cdot \Pi) \cap id) \cdot f \\ = & \{\text{since } C = C^\circ \text{ for coreflexives } C\} \\ & ((\Pi \cdot (const\ e)^\circ \cdot R) \cap id) \cdot f \end{array}$$

$$\begin{array}{ll} = & \{\text{since } ((e\ \texttt{==}) \cdot R)? = (\Pi \cdot (const\ e)^{\circ} \cdot R) \cap id\} \\ & ((e\ \texttt{==}) \cdot R)? \cdot f \end{array}$$

□

Property 4.4

Proof. We will prove a stronger property:

$$flatten\,(foldl\ Bin\ u\ us) \quad = \quad concat\,(map\,flatten\,(u : us))$$

The proof is a simple induction on the length of us.

Case $[\,]$:

$$\begin{array}{ll} & flatten\,(foldl\ Bin\ u\,[\,]) \\ = & \quad \{\text{definition of } foldl\} \\ & flatten\ u \\ = & \quad \{\text{definition of } concat \text{ and } map\} \\ & concat\,(map\,flatten\,[u]) \end{array}$$

Case $v : us$:

$$\begin{array}{ll} & flatten\,(foldl\ Bin\ u\,(v : us)) \\ = & \quad \{\text{definition of } roll\} \\ & flatten\,(foldl\ Bin\,(Bin\,(u, v))\ us) \\ = & \quad \{\text{induction}\} \\ & concat\,(map\,flatten\,(Bin\,(u, v) : us)) \\ = & \quad \{\text{definition of } map\} \\ & concat\,(flatten\,(Bin\,(u, v)) : map\,flatten\ us) \\ = & \quad \{\text{definition of } flatten\} \\ & concat\,(flatten\ u \mathbin{+\!\!+} flattenv : map\,flatten\ us) \\ = & \quad \{\text{definition of } concat \text{ and } map, \mathbin{+\!\!+} \text{ associative}\} \\ & concat\,(map\,flatten\,(u : v : us)) \end{array}$$

□

Property 4.8

$$\begin{array}{ll} & S \cdot X \subseteq Y \\ \Rightarrow & \quad \{\text{since composition is monotonic}\} \\ & S^{\circ} \cdot S \cdot X \subseteq S^{\circ} \cdot Y \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{\text{since } \mathit{dom}\, S \subseteq S^\circ \cdot S\} \\
& \mathit{dom}\, S \cdot X \subseteq S^\circ \cdot Y \\
\Rightarrow & \{\text{since composition is monotonic}\} \\
& S \cdot \mathit{dom}\, S \cdot X \subseteq S \cdot S^\circ \cdot Y \\
\Rightarrow & \{\text{since } S \cdot \mathit{dom}\, S = S \text{ and } S \cdot S^\circ \subseteq \mathit{id}\} \\
& S \cdot X \subseteq Y
\end{array}
$$

Property 5.12

The aim is to prove the inclusion:

$$\mathit{setify} \cdot \mathsf{T}R \quad \subseteq \quad \mathsf{P}R \cdot \mathit{setify}$$

for some type T. We will make use of the point-free definition of P:

$$\mathsf{P}R \quad = \quad \in\backslash(R\cdot \in) \cap (\ni \cdot R)/\ni$$

Furthermore, $\mathit{setify} :: \mathsf{T}A \rightarrow \mathit{Set}\ A$ for any T with membership can be defined by:

$$\mathit{setify} \quad = \quad \Lambda\delta_{\mathsf{T}}$$

The proof goes:

$$
\begin{array}{ll}
& \mathit{setify} \cdot \mathsf{T}R \subseteq \mathsf{P}R \cdot \mathit{setify} \\
\equiv & \{\text{definition of } \mathsf{P}\text{, since } \mathit{setify} \text{ is a function}\} \\
& \mathit{setify} \cdot \mathsf{T}R \subseteq (\in\backslash(R\cdot \in) \cdot \mathit{setify}) \cap ((\ni \cdot R)/\ni \cdot\, \mathit{setify}) \\
\equiv & \{\text{meet}\} \\
& \mathit{setify} \cdot \mathsf{T}R \subseteq \in\backslash(R\cdot \in) \cdot \mathit{setify} \ \wedge \\
& \mathit{setify} \cdot \mathsf{T}R \subseteq (\ni \cdot R)/\ni \cdot\, \mathit{setify}
\end{array}
$$

The first premise can be proved by:

$$
\begin{array}{ll}
& \mathit{setify} \cdot \mathsf{T}R \subseteq \in\backslash(R\cdot \in) \cdot \mathit{setify} \\
\equiv & \{\text{since } \mathit{setify} \text{ is a function, division}\} \\
& \mathit{setify} \cdot \mathsf{T}R \subseteq \in\backslash(R\cdot \in \cdot \mathit{setify}) \\
\equiv & \{\text{division}\} \\
& \in \cdot \mathit{setify} \cdot \mathsf{T}R \subseteq R\cdot \in \cdot \mathit{setify} \\
\equiv & \{\text{definition of } \mathit{setify}\} \\
& \delta_{\mathsf{T}} \cdot \mathsf{T}R \subseteq R \cdot \delta_{\mathsf{T}} \\
\equiv & \{\text{by } (4.7)\} \\
& \mathit{true}
\end{array}
$$

The second premise is proved by:

$$\mathit{setify} \cdot \mathsf{T}R \subseteq (\ni \cdot R)/\ni \cdot\, \mathit{setify}$$

$$
\begin{array}{ll}
\equiv & \{\text{since } \mathit{setify} \text{ is a function, division}\} \\
& \mathit{setify} \cdot \mathsf{T}R \subseteq (\ni \cdot R)/(\in \cdot \mathit{setify})^\circ \\
\equiv & \{\text{definition of } \mathit{setify}\} \\
& \delta_{\mathsf{T}} \cdot \mathsf{T}R \subseteq (\ni \cdot R)/\delta_{\mathsf{T}}^\circ \\
\equiv & \{\text{division}\} \\
& \delta_{\mathsf{T}} \cdot \mathsf{T}R \cdot \delta_{\mathsf{T}}^\circ \subseteq \ni \cdot R \\
\equiv & \{\text{since } \Lambda T \cdot T^\circ \subseteq \ni\} \\
& \mathit{true}
\end{array}
$$

Property 5.19

The aim is to prove:

$$\mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q) \quad \subseteq \quad \mathit{thin}\ Q \cdot \mathit{union}$$

The function *union* can be defined as $\mathit{union} = \mathsf{E} \in = \Lambda(\in \cdot \in)$. We will also make use of the point-free definition of the relator P:

$$\mathsf{P}R \quad = \quad \in\backslash(R\cdot \in) \cap (\ni \cdot R)/\ni$$

and the universal property of *thin* Q:

$$X \subseteq \mathit{thin}\ Q \cdot \Lambda S \quad \equiv \quad \in \cdot X \subseteq S \ \wedge \ X \cdot S^\circ \subseteq \ni \cdot Q$$

According to the universal property, (5.19) follows from:

$$
\begin{array}{rcl}
\in \cdot \mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q) & \subseteq & \in \cdot \in \\
\mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q) \cdot \ni \cdot \ni & \subseteq & \ni \cdot Q
\end{array}
$$

The first premise is proved by:

$$
\begin{array}{ll}
& \in \cdot \mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q) \\
\subseteq & \{\text{since } \mathit{thin}\ Q \subseteq \in\backslash\in, \text{ division}\} \\
& \in \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q) \\
= & \{\text{since } \mathit{union} = \Lambda(\in \cdot \in), \text{ power transpose}\} \\
& \in \cdot \in \cdot \mathsf{P}(\mathit{thin}\ Q) \\
\subseteq & \{\text{since } \mathsf{P}(\mathit{thin}\ Q) \subseteq \in\backslash(\mathit{thin}\ Q \cdot \in), \text{ division}\} \\
& \in \cdot \mathit{thin}\ Q \cdot \in \\
\subseteq & \{\text{since } \mathit{thin}\ Q \subseteq \in\backslash\in, \text{ division}\} \\
& \in \cdot \in
\end{array}
$$

while the second is proved by:

$$\mathit{thin}\ Q \cdot \mathit{union} \cdot \mathsf{P}(\mathit{thin}\ Q) \cdot \ni \cdot \ni$$

$$
\begin{array}{ll}
\subseteq & \{\text{since } \mathsf{P}(\mathit{thin}\ Q) \subseteq (\ni \cdot \mathit{thin}\ Q)/\ni, \text{division}\} \\
& \mathit{thin}\ Q \cdot \mathit{union} \cdot \ni \cdot \mathit{thin}\ Q \cdot \ni \\
\subseteq & \{\text{since } \mathit{thin}\ Q \subseteq (\ni \cdot Q)/\ni, \text{ division}\} \\
& \mathit{thin}\ Q \cdot \mathit{union} \cdot \ni \cdot \ni \cdot Q \\
\subseteq & \{\text{since } \mathit{union} = \Lambda(\in \cdot \in) \text{ and } \Lambda R \cdot R^\circ \subseteq \ni\} \\
& \mathit{thin}\ Q \cdot \ni \cdot Q \\
\subseteq & \{\text{since } \mathit{thin}\ Q \subseteq (\ni \cdot Q)/\ni, \text{ division}\} \\
& \ni \cdot Q \cdot Q \\
\subseteq & \{\text{assumption: } Q \text{ transitive}\} \\
& \ni \cdot Q
\end{array}
$$

Lemma 7.1

To save space, we abbreviate *apply* p to π. Also, we adopt the good old squiggle notation writing $\mathit{map}\, f$ as f^* and $\mathit{zip}\ x\ y$ as $x \Upsilon y$.

We will prove the property for *iterate*, that is,

$$\mathit{iterate}\, \pi\, x \quad = \quad \mathit{fst}^{**} \cdot \mathit{iterate}\, (\mathit{pl}_x \cdot \mathit{snd})^* \bullet x \Upsilon [0..m]$$

where $m = \mathit{length}\, x$ and $\mathit{pl}_x\ i = \pi\, (x \Upsilon [0..m])\ !!\ i$. The lemma then follows from $\mathit{iter}_1 = \mathit{tail} \cdot \mathit{iterate}$. The proof proceeds by using the approxmiation lemma and the following rule:

$$\mathit{map}\, h \cdot \mathit{iterate}\, f = \mathit{iterate}\, g \cdot h \quad \Leftarrow \quad h \cdot f = g \cdot h \tag{A.4}$$

The key property, however, is the following equality

$$\mathit{pl}_y^*\, (\pi\, \mathit{ns}) \quad = \quad (\mathit{id} \times \pi)\, (\mathit{pl}_{\pi\, y}^*\, \mathit{ns}) \tag{A.5}$$

We reason:

$$
\begin{array}{ll}
& \mathit{approx}\, (n+1)\, (\mathit{fst}^{**} \cdot \mathit{iterate}\, (\mathit{pl}_x \cdot \mathit{snd})^* \bullet x \Upsilon [0..m]) \\
= & \{\text{definition of } \mathit{iterate} \text{ and } \mathit{map}\} \\
& x : \mathit{approx}\, n\, (\mathit{fst}^{**} \cdot \mathit{iterate}\, (\mathit{pl}_x \cdot \mathit{snd})^* \bullet \mathit{pl}^*\, (\mathit{snd}^*\, (x \Upsilon [0..m]))) \\
= & \{\text{since } \mathit{snd}^*\, (x \Upsilon y) = y\} \\
& x : \mathit{approx}\, n\, (\mathit{fst}^{**} \cdot \mathit{iterate}\, (\mathit{pl} \cdot \mathit{snd})^* \bullet \mathit{pl}_x^*\, [0..m]) \\
= & \{\text{definition of } \mathit{pl}_x\} \\
& x : \mathit{approx}\, n\, (\mathit{fst}^{**} \cdot \mathit{iterate}\, (\mathit{pl}_x \cdot \mathit{snd})^* \bullet \pi (x \Upsilon [0..m])) \\
= & \{\text{higher-order naturality: } \eta\, (x \Upsilon y) = \eta\, x \Upsilon \eta\, y\} \\
& x : \mathit{approx}\, n\, (\mathit{fst}^{**} \cdot \mathit{iterate}\, (\mathit{pl}_x \cdot \mathit{snd})^* \bullet \pi\, x \Upsilon \pi\, [0..m]) \\
= & \{\text{by (A.5) and (A.4)}\} \\
& x : \mathit{approx}\, n\, (\mathit{fst}^{**} \cdot (\mathit{id} \times \pi)^* \cdot \mathit{iterate}\, (\mathit{pl}_{\pi\, x} \cdot \mathit{snd})^* \bullet \pi\, x \Upsilon [0..m]) \\
= & \{\text{since } \mathit{fst}^* \cdot (\mathit{id} \times f) = \mathit{fst}^*\}
\end{array}
$$

$$
\begin{array}{ll}
 & x : approx\ n\,(fst^{**} \cdot iterate\,(pl_{\pi\,x} \cdot snd)^{*} \bullet \pi\,x\,\Upsilon[0..m]) \\
= & \quad \{\text{induction}\} \\
 & x : approx\ n\,(iterate\,\pi\,(\pi\,x)) \\
= & \quad \{\text{definition of } approx \text{ and } iterate\} \\
 & approx\,(n+1)\,(iterate\,\pi\,x)
\end{array}
$$

Appendix B

Proof of the Generic Greedy Theorem

To prove the mutual greedy theorem, we will need the following lemma

Lemma B.1 Let G be a regular functor, we have

$$min\, \mathsf{G}R \cdot \Lambda\mathsf{G}{\in} \quad = \quad \mathsf{G}(min\, R)$$

Proof. We reason

$$\begin{array}{ll}
 & min\, \mathsf{G}R \cdot \Lambda\mathsf{G}{\in} \\
= & \quad \{\text{definition of } min\} \\
 & ({\in} \cap (\mathsf{G}R/{\ni})) \cdot \Lambda\mathsf{G}{\in} \\
= & \quad \{\Lambda\mathsf{G}{\in} \text{ function}\} \\
 & ({\in} \cdot \Lambda\mathsf{G}{\in}) \cap ((\mathsf{G}R/{\ni}) \cdot \Lambda\mathsf{G}{\in}) \\
= & \quad \{\text{since } R/S \cdot f = R/f^{\circ} \cdot S, \text{ power transpose}\} \\
 & \mathsf{G}{\in} \cap (\mathsf{G}R/\mathsf{G}{\ni}) \\
= & \quad \{\text{see below}\} \\
 & \mathsf{G}({\in} \cap (R/{\ni})) \\
= & \quad \{\text{definition of } min\} \\
 & min\, \mathsf{G}R
\end{array}$$

The property used in the penultimate step:

$$\mathsf{G}((R/S) \cap S^{\circ}) \quad = \quad (\mathsf{G}R/\mathsf{G}S) \cap \mathsf{G}S^{\circ}$$

is proved as Lemma 8.3.1.2 of [64].

□

Now we prove the theorem itself.

Proof. We reason:

$$
\begin{array}{cl}
& (\!|(\mathsf{G}(min\ R) \cdot h \cdot wrap)|\!)_{\mathsf{F}} \subseteq \mathsf{G}(min\ R) \cdot (\!|(h \cdot \Lambda\mathsf{FG}\in)|\!)_{\mathsf{F}} \\
\Leftarrow & \quad \{\text{fold fusion}\} \\
& \mathsf{G}(min\ R) \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \subseteq \mathsf{G}(min\ R) \cdot h \cdot \Lambda\mathsf{FG}\in \\
\equiv & \quad \{\text{Lemma B.1}\} \\
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \subseteq min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in \\
\equiv & \quad \{\text{since } h \cdot \Lambda\mathsf{FG}\in \text{ is a function}\} \\
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \subseteq min\,\mathsf{G}R \cdot \Lambda(\mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in) \\
\equiv & \quad \{\text{universal property of } min\} \\
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \subseteq \mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in \\
& \quad \wedge \\
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \cdot (\mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in)^{\circ} \subseteq \mathsf{G}R
\end{array}
$$

The first of the two premises can be proved by:

$$
\begin{array}{cl}
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \\
\subseteq & \quad \{\text{since } min\,\mathsf{G}R \subseteq \in\} \\
& \mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}\in \\
= & \quad \{\text{since } f = \in \cdot \Lambda f\} \\
& \mathsf{G}\in \cdot h \cdot wrap \cdot \in \cdot \Lambda\mathsf{FG}\in \\
\subseteq & \quad \{\text{since } wrap \cdot \in \subseteq subset\} \\
& \mathsf{G}\in \cdot h \cdot subset \cdot \Lambda\mathsf{FG}\in \\
= & \quad \{\text{by (5.25)}\} \\
& \mathsf{G}(\in \cdot subset) \cdot h \cdot \Lambda\mathsf{FG}\in \\
\subseteq & \quad \{\text{since } subset = \in\backslash\in\} \\
& \mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in
\end{array}
$$

and the second by:

$$
\begin{array}{cl}
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot \mathsf{FG}(min\ R) \cdot (\mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in)^{\circ} \\
\subseteq & \quad \{\text{claim}: \mathsf{G}\in \cdot h \cdot \Lambda\mathsf{FG}\in \cdot \mathsf{FG}(min\ R)^{\circ} \subseteq \mathsf{G}(R^{\circ} \cdot \in) \cdot h \cdot wrap\} \\
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G}\in \cdot h \cdot wrap \cdot wrap^{\circ} \cdot h^{\circ} \cdot \mathsf{G}(\ni \cdot R) \\
\subseteq & \quad \{wrap \text{ and } h \text{ functions}\} \\
& min\,\mathsf{G}R \cdot \Lambda\mathsf{G} \in \cdot \mathsf{G}(\ni \cdot R) \\
\subseteq & \quad \{min\,\mathsf{G}R \subseteq \mathsf{G}R/\ni\} \\
& (\mathsf{G}R/\ni) \cdot \Lambda\mathsf{G}\in \cdot \mathsf{G}(\ni \cdot R) \\
= & \quad \{\text{since } R/S \cdot f = R/f^{\circ} \cdot S\} \\
& (\mathsf{G}R/\mathsf{G}\ni) \cdot \mathsf{G}(\ni \cdot R)
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \quad \{\text{since } (R/S) \cdot S \subseteq R\} \\
& \mathsf{G}R \cdot \mathsf{G}R \\
\subseteq & \quad \{R \text{ preorder}\} \\
& \mathsf{G}R
\end{array}
$$

The claim can be proved below.

$$
\begin{array}{ll}
& \mathsf{G}{\in} \cdot h \cdot \Lambda\mathsf{FG}{\in} \cdot \mathsf{FG}(min\ R)^{\circ} \\
\subseteq & \quad \{\text{since } \mathsf{FG}{\in} \cdot \mathsf{FG}(min\ R)^{\circ} \subseteq \mathsf{FG}R^{\circ} \text{ and } R \cdot S \subseteq T \Rightarrow \Lambda R \cdot S \subseteq subset \cdot \Lambda T\} \\
& \mathsf{G}{\in} \cdot h \cdot subset \cdot \Lambda\mathsf{FG}R^{\circ} \\
= & \quad \{\text{by } (5.25)\} \\
& \mathsf{G}({\in} \cdot subset) \cdot h \cdot \Lambda\mathsf{FG}R^{\circ} \\
\subseteq & \quad \{\text{since } subset = {\in}\backslash{\in}\} \\
& \mathsf{G}{\in} \cdot h \cdot \Lambda\mathsf{FG}R^{\circ} \\
= & \quad \{\text{since } \Lambda R = \mathsf{E}R \cdot wrap\} \\
& \mathsf{G}{\in} \cdot h \cdot \mathsf{EFG}R^{\circ} \cdot wrap \\
\subseteq & \quad \{\text{by } (5.26)\} \\
& \mathsf{G}({\in} \cdot \mathsf{E}R^{\circ}) \cdot h \cdot wrap \\
= & \quad \{\text{power functor}\} \\
& \mathsf{G}(R^{\circ} \cdot {\in}) \cdot h \cdot wrap
\end{array}
$$

$\Box$

Appendix C

Missing Proofs in Chapter 6

C.1 An Online Algorithm for Binary Closure

We start with the following definition for θ_R:

$$\theta_R(P, Q) \quad = \quad \mu(X \mapsto Q \cup (R \cdot (\langle X, X\rangle \cup \langle X, P\rangle \cup \langle P, X\rangle) - P))$$

Substituting Q for $\emptyset$, we have $\theta_R(P, \emptyset) = \emptyset$. For the non-empty case we derive:

$$\begin{array}{ll}
 & \theta_R(P, Q) \\
= & \quad \{\text{definition}\} \\
 & \mu(X : Q \cup (R \cdot \mathit{prods}(X, P) - P)) \\
= & \quad \{\text{since } X \cup Y = X \cup (Y - X)\} \\
 & \mu(X : Q \cup (R \cdot \mathit{prods}(X, P) - (P \cup Q))) \\
= & \quad \{\text{rolling rule, letting } f = (Q\cup) \text{ and } g\,X = R \cdot \mathit{prods}(X, P) - (P \cup Q)\} \\
 & Q \cup \mu(X : R \cdot \mathit{prods}(Q \cup X, P) - (P \cup Q)) \\
= & \quad \{\text{claim in Section 6.3.3: } \mathit{prods}(Q \cup X, P) = \mathit{prods}(Q, P) \cup \mathit{prods}(X, P \cup Q)\text{, see below}\} \\
 & Q \cup \mu(X : R \cdot ((\mathit{prods}(Q, P) \cup \mathit{prods}(X, P \cup Q)) - (P \cup Q)) \\
= & \quad \{\text{since composition and subtraction distributes into union}\} \\
 & Q \cup \mu(X : (R \cdot \mathit{prods}(Q, P) - (P \cup Q)) \cup (R \cdot \mathit{prods}(X, P \cup Q) - (P \cup Q))) \\
= & \quad \{\text{definition}\} \\
 & Q \cup \theta_R(P \cup Q, R \cdot \mathit{prods}(Q, P) - (P \cup Q))
\end{array}$$

We thus come up with this definition for θ_R:

$$\begin{array}{lcl}
\theta_R(P, \emptyset) & = & \emptyset \\
\theta_R(P, Q) & = & Q \cup \theta(P \cup Q, R \cdot (\langle P, Q\rangle \cup \langle Q, P\rangle \cup \langle Q, Q\rangle) - (P \cup Q))
\end{array}$$

C.2 Proof of Theorem 6.1

Our task is to refine $\mathit{thin}\ Q \cdot \mathit{close}\, f = \mathit{thin}\ Q \cdot \mathit{stop} \cdot (\mathit{step}\, f)^*$. Introducing $\Delta R = R \times R$ for brevity, we have $\mathit{thin}\ Q \cdot \mathit{stop} = \mathit{stop} \cdot \Delta(\mathit{thin}\ Q)$, so we can move $\mathit{thin}\ Q$ past stop.

For the closure we use the rule

$$S^* \cdot R \subseteq R \cdot T^* \quad \Leftarrow \quad S \cdot R \subseteq R \cdot T$$

For our problem we aim to show

$$\Delta(\mathit{thin}\ Q) \cdot \mathit{step}\, f \cdot \Delta \mathit{thin} \subseteq \Delta(\mathit{thin}\ Q) \cdot \mathit{step}\, f \tag{C.1}$$

That is, we have to somehow push $\mathit{thin}\ Q$ through step.

The crucial part of the proof, however, is to show that

$$\Lambda(R \cdot (\in \times \in)) \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q) \quad \subseteq \quad \mathit{thin}\ Q \cdot \Lambda(R \cdot (\in \times \in))$$

The proof is typical of that involving thin. Given the definition $\mathit{thin}\ Q = \in \backslash \in \cap (\ni \cdot Q)/\ni$, the above inclusion is equivalent to:

$$\begin{array}{rcl} \in \cdot \Lambda(R \cdot (\in \times \in)) \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q) & \subseteq & R \cdot (\in \times \in) \\ \Lambda(R \cdot (\in \times \in)) \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q) \cdot (\ni \times \ni) \cdot R^\circ & \subseteq & \ni \cdot Q \end{array}$$

The first inclusion can be proved by:

$$\begin{array}{ll} & \in \cdot \Lambda(R \cdot (\in \times \in)) \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q) \\ = & \quad \{\text{breadth}\} \\ & R \cdot (\in \times \in) \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q) \\ \subseteq & \quad \{\text{functor, definition of } \mathit{thin}\text{, division}\} \\ & R \cdot (\in \times \in) \end{array}$$

while the second can be proved by:

$$\begin{array}{ll} & \Lambda(R \cdot (\in \times \in)) \cdot (\mathit{thin}\ Q \times \mathit{thin}\ Q) \cdot (\ni \times \ni) \cdot R^\circ \\ \subseteq & \quad \{\text{functor, definition of } \mathit{thin}\text{, division}\} \\ & \Lambda(R \cdot (\in \times \in)) \cdot (\ni \cdot Q \times \ni \cdot Q) \cdot R^\circ \\ \subseteq & \quad \{\text{monotonicity: } R \cdot (Q^\circ \times Q^\circ) \subseteq Q^\circ \cdot R\} \\ & \Lambda(R \cdot (\in \times \in)) \cdot (\ni \times \ni) \cdot R^\circ \cdot Q \\ \subseteq & \quad \{\text{since } \Lambda f \cdot f^\circ \subseteq \ni\} \\ & \ni \cdot Q \end{array}$$

The rest of the proof is just laboriously but boring pushing $\mathit{thin}\ Q$ through step. Observe that step can be written in point-free notation as

$$\begin{array}{lcl} \mathit{step}\, f & = & \mathit{step}'\, f \cdot (\mathit{id} \times \mathit{nonempty?}) \\ \mathit{step}'\, f & = & \langle \mathit{cup}, \mathit{sub} \cdot \langle \mathit{prods}\, f, \mathit{cup} \rangle \rangle \\ \mathit{prods}\, f & = & \mathit{cup} \cdot \langle f, \mathit{cup} \cdot \langle f \cdot \mathit{swap}, f \cdot \langle \mathit{snd}, \mathit{snd} \rangle \rangle \rangle \end{array}$$

Abbreviating $\mathit{thin}\ Q$ to thin, we calculate:

$$\Delta \mathit{thin} \cdot \mathit{step}'\, f$$

$$
\begin{array}{ll}
= & \quad \{\text{definition}\} \\
& \Delta thin \cdot \langle cup, sub \cdot \langle prods\, f, cup \rangle\rangle \\
= & \quad \{\text{products}\} \\
& \langle thin \cdot cup, thin \cdot sub \cdot \langle prods\, f, cup \rangle\rangle \\
= & \quad \{\text{since } thin \cdot cup = thin \cdot cup \cdot \Delta thin \text{ and } thin \cdot sub = thin \cdot sub \cdot \Delta thin\} \\
& \langle thin \cdot cup \cdot \Delta thin, thin \cdot sub \cdot \Delta thin \cdot \langle prods\, f, cup \rangle\rangle \\
= & \quad \{\text{products}\} \\
& \Delta thin \cdot \langle cup \cdot \Delta thin, sub \cdot \langle thin \cdot prods\, f, thin \cdot cup \rangle\rangle \\
\supseteq & \quad \{\text{claim: } prods\, f \cdot \Delta thin \subseteq thin \cdot prods\, f\} \\
& \Delta thin \cdot \langle cup \cdot \Delta thin, sub \cdot \langle prods\, f \cdot \Delta\, thin, thin \cdot cup \rangle\rangle \\
\supseteq & \quad \{\text{since } cup \cdot \Delta thin \subseteq thin \cdot cup\} \\
& \Delta thin \cdot \langle cup \cdot \Delta thin, sub \cdot \langle prods\, f \cdot \Delta\, thin, cup \cdot \Delta thin \rangle\rangle \\
\supseteq & \quad \{\text{using } \langle R, S\rangle \cdot T \subseteq \langle R \cdot T, S \cdot T\rangle \text{ twice}\} \\
& \Delta thin \cdot \langle cup, sub \cdot \langle prods\, f, cup \rangle\rangle \cdot \Delta thin \\
= & \quad \{\text{definition}\} \\
& \Delta thin \cdot step'\, f \cdot \Delta thin
\end{array}
$$

And since $\Delta thin \cdot (id \times nonempty?) = (id \times nonempty?) \cdot \Delta thin$, we have shown that (C.1) holds. The missing claim is proved below:

$$
\begin{array}{ll}
& prods\, f \cdot \Delta thin \\
= & \quad \{\text{definition}\} \\
& cup \cdot \langle f, cup \cdot \langle f \cdot swap, f \cdot \langle snd, snd \rangle\rangle\rangle \cdot \Delta thin \\
\subseteq & \quad \{\text{using } \langle R, S\rangle \cdot T \subseteq \langle R \cdot T, S \cdot T\rangle \text{ three times}\} \\
& cup \cdot \langle f \cdot \Delta thin, cup \cdot \langle f \cdot swap \cdot \Delta thin, f \cdot \langle snd \cdot \Delta thin, snd \cdot \Delta thin \rangle\rangle\rangle \\
= & \quad \{\text{since } swap \cdot \Delta thin = \Delta thin \cdot swap \\
& \qquad\qquad \text{and } snd \cdot \Delta thin = thin \cdot snd, \text{ products}\} \\
& cup \cdot \langle f \cdot \Delta thin, cup \cdot \langle f \cdot \Delta thin \cdot swap, f \cdot \Delta thin \cdot \langle snd, snd \rangle\rangle\rangle \\
\subseteq & \quad \{\text{since } f \cdot \Delta thin \subseteq thin \cdot f\} \\
& cup \cdot \langle thin \cdot f, cup \cdot \langle thin \cdot f \cdot swap, thin \cdot f \cdot \langle snd, snd \rangle\rangle\rangle \\
\subseteq & \quad \{\text{products and } cup \cdot \Delta thin \subseteq thin \cdot cup, \text{ twice}\} \\
& thin \cdot cup \cdot \langle f, cup \cdot \langle f \cdot swap, f \cdot \langle snd, snd \rangle\rangle\rangle \\
= & \quad \{\text{definition}\} \\
& thin \cdot prods\, f
\end{array}
$$

The condition is established.

C.3 Building Oriented Trees by a Fold

The definition of *add* can also be written as a least fixed-point:

$$add \quad = \quad \mu(X : bin \cdot (tip \times id) \cup bin \cdot (X \times id) \cdot assocl \cdot (id \times bin^{\circ}))$$

where $assocl\,(a, (x, y)) = ((a, x), y)$. According to the converse-of-a-function theorem, in order to show that:

$$basisT^{\circ} \quad = \quad foldBag\; add\; tip$$

for some *add*, we need to show:

$$\begin{array}{rcl} basisT \cdot tip & \subseteq & bwrap \\ basisT \cdot add & \subseteq & bcons \cdot (id \times basisT) \end{array}$$

where *bcons* is the counterparts of *cons* on bags. The condition on *tip* is obviously true. For the second condition, we reason:

$$\begin{array}{ll} & basisT \cdot add \subseteq bcons \cdot (id \times basisT) \\ \equiv & \quad \{\text{division}\} \\ & add \subseteq basisT \backslash (bcons \cdot (id \times basisT)) \\ \Leftarrow & \quad \{\text{least-fixed point, let } Y = basisT \backslash (bcons \cdot (id \times basisT))\} \\ & bin \cdot (tip \times id) \cup bin \cdot (Y \times id) \cdot assocl \cdot (id \times bin^{\circ}) \subseteq Y \\ \equiv & \quad \{\text{union}\} \\ & bin \cdot (tip \times id) \subseteq Y \;\; \wedge \;\; bin \cdot (Y \times id) \cdot assocl \cdot (id \times bin^{\circ}) \subseteq Y \\ \equiv & \quad \{\text{division}\} \\ & basisT \cdot bin \cdot (tip \times id) \subseteq bcons \cdot (id \times basisT) \\ & \quad \wedge\, basisT \cdot bin \cdot (Y \times id) \cdot assocl \cdot (id \times bin^{\circ}) \subseteq bcons \cdot (id \times basisT) \end{array}$$

The first of the premises can be proved by:

$$\begin{array}{ll} & basisT \cdot bin \cdot (tip \times id) \\ = & \quad \{\text{definition of } basisT\} \\ & bcup \cdot (basisT \times basisT) \cdot (tip \times id) \\ = & \quad \{\text{product, definition of } basisT\} \\ & bcup \cdot (bWrap \times id) \cdot (id \times basisT) \\ = & \quad \{\text{since } bcup \cdot (bwrap \times id) = bcons\} \\ & bcons \cdot (id \times basisT) \end{array}$$

To prove the second premise we reason:

$$\begin{array}{ll} & basisT \cdot bin \cdot (Y \times id) \cdot assocl \cdot (id \times bin^{\circ}) \\ = & \quad \{\text{definition of } basisT\text{, product}\} \\ & bcup \cdot (basisT \cdot Y \times basisT) \cdot assocl \cdot (id \times bin^{\circ}) \end{array}$$

$$
\begin{array}{ll}
\subseteq & \{\text{definition of } Y\} \\
& bcup \cdot (bcons \cdot (id \times basisT) \times basisT) \cdot assocl \cdot (id \times bin^\circ) \\
= & \{\text{since } ((id \times f) \times f) \cdot assocl = assocl \cdot (id \times (f \times f))\} \\
& bcup \cdot (bcons \times id) \cdot assocl \cdot (id \times (basisT \times basisT) \cdot bin^\circ) \\
= & \{\text{since } bcup \cdot (bcons \times id) = bcons \cdot (id \times bcup) \cdot assocl^\circ\} \\
& bcons \cdot (id \times bcup) \cdot assocl^\circ \cdot assocl \cdot (id \times (basisT \times basisT) \cdot bin^\circ) \\
= & \{\text{since } assocl^\circ \cdot assocl = id, \text{ product}\} \\
& bcons \cdot (id \times bcup \cdot (basisT \times basisT) \cdot bin^\circ) \\
= & \{\text{definition of } basisT\} \\
& bcons \cdot (id \times basisT)
\end{array}
$$

Certainly *add* and *tip* are jointly surjective, as any non-tip tree can be generated by *add*.

We still need to prove that *add* does satisfy the healthiness condition to be an argument to *foldBag*: that $add\ a \cdot add\ b = add\ b \cdot add\ a$ for all a and b. For the case the tree is a tip, we reason:

$$
\begin{array}{ll}
& add\ a\,(add\ b\,(tip\ c)) \\
= & \{\text{definition of } add\} \\
& add\ a\,(bin\,(tip\ b \otimes tip\ c)) \\
= & \{\text{definition of } add \text{ and } bin\} \\
& bin\,(tip\ a \otimes bin\,(tip\ b \otimes tip\ c)) \\
& \Box\ bin\,(bin\,(tip\ a \otimes tip\ b) \otimes tip\ c) \\
& \Box\ bin\,(bin\ tip\ b \otimes (tip\ a \otimes tip\ c)) \\
= & \{\text{since } x \otimes y = y \otimes x\} \\
& bin\,(tip\ a \otimes bin\,(tip\ b \otimes tip\ c)) \\
& \Box\ bin\,(bin\,(tip\ b \otimes tip\ a) \otimes tip\ c) \\
& \Box\ bin\,(bin\ tip\ b \otimes (tip\ a \otimes tip\ c)) \\
= & \{\text{definition of } add \text{ and } bin\} \\
& add\ b\,(bin\,(tip\ a \otimes tip\ c)) \\
= & \{\text{definition of } add\} \\
& add\ b\,(add\ a\,(tip\ c))
\end{array}
$$

For the case the tree is a non-tip:

$$
\begin{array}{ll}
& add\ a\,(add\ b\,(bin\ x\ y)) \\
= & \{\text{definition of } add \text{ and } bin\} \\
& add\ a\,(bin\,(tip\ b \otimes bin\,(x \otimes y)) \\
& \qquad \Box bin\,(add\ b\ x \otimes y) \\
& \qquad \Box bin\,(x \otimes add\ b\ y)) \\
= & \{\text{definition of } add \text{ and } bin\}
\end{array}
$$

$$
\begin{aligned}
& \quad bin\,(tip\;a \otimes bin\,(tip\;b \otimes bin\,(x \otimes y))) \\
& \quad \Box\; bin\,(bin\,(tip\;a \otimes tip\;b) \otimes bin\,(x \otimes y)) \\
& \quad \Box\; bin\,(tip\;b \otimes add\;a\,(bin\,(x \otimes y))) \\
& \quad \Box\; bin\,(tip\;a \otimes bin\,(add\;b\;x \otimes y)) \\
& \quad \Box\; bin\,(add\;a\,(add\;b\;x) \otimes y) \\
& \quad \Box\; bin\,(add\;b\;x \otimes add\;a\;y) \\
& \quad \Box\; bin\,(tip\;a \otimes bin\,(x \otimes add\;b\;y)) \\
& \quad \Box\; bin\,(add\;a\;x \otimes add\;b\;y) \\
& \quad \Box\; bin\,(x \otimes add\;a\,(add\;b\;y)) \\
= & \quad \{\text{expand } add \text{ in the third case}\} \\
& \quad bin\,(tip\;a \otimes bin\,(tip\;b \otimes bin\,(x \otimes y))) \\
& \quad \Box\; bin\,(bin\,(tip\;a \otimes tip\;b) \otimes bin\,(x \otimes y)) \\
& \quad \Box\; bin\,(tip\;b \otimes bin\,(tip\;a \otimes bin\,(x \otimes y))) \\
& \quad \Box\; bin\,(tip\;b \otimes bin\,(add\;a\;x \otimes y)) \\
& \quad \Box\; bin\,(tip\;b \otimes bin\,(x \otimes add\;a\;y)) \\
& \quad \Box\; bin\,(tip\;a \otimes bin\,(add\;b\;x \otimes y)) \\
& \quad \Box\; bin\,(add\;a\,(add\;b\;x) \otimes y) \\
& \quad \Box\; bin\,(add\;b\;x \otimes add\;a\;y) \\
& \quad \Box\; bin\,(tip\;a \otimes bin\,(x \otimes add\;b\;y)) \\
& \quad \Box\; bin\,(add\;a\;x \otimes add\;b\;y) \\
& \quad \Box\; bin\,(x \otimes add\;a\,(add\;b\;y))
\end{aligned}
$$

If we expand $add\;b\,(add\;a\,(bin\;x\;y))$, we get another 11 possibilities.

$$
\begin{aligned}
& bin\,(tip\;b \otimes bin\,(tip\;a \otimes bin\,(x \otimes y))) \\
& \Box\; bin\,(bin\,(tip\;b \otimes tip\;a) \otimes bin\,(x \otimes y)) \\
& \Box\; bin\,(tip\;a \otimes bin\,(tip\;b \otimes bin\,(x \otimes y))) \\
& \Box\; bin\,(tip\;a \otimes bin\,(add\;b\;x \otimes y)) \\
& \Box\; bin\,(tip\;a \otimes bin\,(x \otimes add\;b\;y)) \\
& \Box\; bin\,(tip\;b \otimes bin\,(add\;a\;x \otimes y)) \\
& \Box\; bin\,(add\;b\,(add\;a\;x) \otimes y) \\
& \Box\; bin\,(add\;a\;x \otimes add\;b\;y) \\
& \Box\; bin\,(tip\;b \otimes bin\,(x \otimes add\;a\;y)) \\
& \Box\; bin\,(add\;b\;x \otimes add\;a\;y) \\
& \Box\; bin\,(x \otimes add\;b\,(add\;a\;y))
\end{aligned}
$$

By the community of $\otimes$ and the inductive premise we have established, we can check through each of them and conclude that they are equivalent.

善數不用籌策

老子 道德經

A good computer needs no aid from machines.

Lau Tzu, *Tao Te Ching*