Theories for Algorithm Calculation

Theorieën voor het berekenen van algoritmen

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Rijksuniversiteit te Utrecht
op gezag van de Rector Magnificus, Prof. Dr. J.A. van Ginkel
ingevolge het besluit van het College van Dekanen
in het openbaar te verdedigen
op donderdag 18 februari 1993 des namiddags te 2.30 uur

door

Johan Theodoor Jeuring

geboren op 12 augustus 1965
te Oude Pekela

Promotor: Prof. L.G.L.T. Meertens
Faculteit Wiskunde en Informatica

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Jeuring, Johan Theodoor

Theories for algorithm calculation / Johan Theodoor
Jeuring. - Utrecht : Universiteit Utrecht, Faculteit
Wiskunde. - Ill.
Proefschrift Rijksuniversiteit Utrecht. - Met lit. opg. -
Met samenvatting in het Nederlands.
ISBN 90-393-0235-9
Trefw.: algoritmen.

Support has been received from the Dutch organisation for scientific research NWO under NFI project STOP, project number NF 62/63-518.

Contents

I've not said a word.

Arthur Miller, *The Crucible*

voor mijn vader en moeder

Acknowledgements

It is a pleasure to thank the many people who supported me in writing this thesis.

My promotor, Lambert Meertens, offered me an OIO position in the NFI STOP-project and taught me the basics of ‘Constructive Algorithmics’.

Eddy Boeve, Maarten Fokkinga, Lambert Meertens, Sjoerd Mullender, Steven Pemberton, and Jaap van der Woude answered many questions concerning Constructive Algorithmics, LaTeX, editors, mailers, newsreaders, windowmanagers, etcetera.

Maarten Fokkinga, Jeremy Gibbons, Lambert Meertens, and Jaap van der Woude read all or substantial parts of this thesis. The form and contents of the thesis have been greatly influenced by their comments.

Jaap van der Woude made a big effort in correcting my sometimes very unwieldy proofs and suggesting many alternative approaches. I appreciate his friendly way of commenting very much.

My visits to Oxford in 1989 and 1990 were very fruitful; I thank Richard Bird, Jeremy Gibbons, Wayne Luk, and Oege de Moor for providing an inspiring environment and for taking care of me.

The STOP Workshops and Summerschool on Ameland provided an idyllic environment for discussing Algorithmics and I learned a lot from the other attendants. Beside the people mentioned above these included Roland Backhouse, Paul Chisholm, Netty van Gasteren, Grant Malcolm, Erik Meijer, Doaitse Swierstra, Norbert Völker, and many others.

Another forum where I learned a lot was the weekly Constructive Algorithmics Meeting at Utrecht University. The meetings were attended by many of the people mentioned above, and by Jeroen Fokker, Carroll Morgan, Nico Verwer, and Hans Zantema.

I thank my colleagues from the department AA of the CWI for a pleasant working environment.

Finally, I thank my family and friends for their moral support without which it would have been impossible to finish this thesis. They helped me to put writing a thesis in perspective and they did not (or pretended not to) get bored with me struggling with my thesis. In particular, I want to thank my parents for taking care of me, and stimulating me to learn something and to finish what I've started. To them I dedicate this thesis.

Chapter 1

Introduction

During the years that I have worked on this thesis I've often wandered around the office, looking for people who could tell me why my computer or the programs I was running on it weren't working as I expected them to work. For example, some time ago I sent a message by electronic mail to about eighty people, and then went to lunch. When I came back from lunch I found that the mailer had just started sending the message for the eighth time to the first fifty people on the address-list, and if I hadn't found somebody who could stop the mailer, it would have gone on sending the message forever. The reason why the mailer kept sending the message was that it worked as follows. It sent the message to the first fifty addresses on the address-list, then it got an error message stating that some buffer could not contain more than fifty addresses, and as a consequence, it started sending the message again. Clearly, there was an error in the mailer.

Anybody who has worked for some time with computers can tell you this kind of story. For me, obviously, the experience described above was annoying, but not disastrous. However, few people will find it difficult to recall disasters caused by malfunctioning computers. It is generally acknowledged that reliable computers (hardware) and reliable computer programs (software) are very desirable. Since hardware and software are constructed and used by people, the construction and use of reliable software and reliable hardware depends on the existence of 'reliable people': sensible people that are well educated in the performance of their task. This thesis is concerned with one of the important tasks that can be distinguished in the process of developing reliable software.

Developing reliable software for a given problem is usually a complex task. First of all, the given problem has to be transformed into a formal specification. The formal specification is then divided into manageable pieces, for each of which software is constructed. The obtained software is combined to obtain the final product. Finally, the product is tested to see whether or not it has the desired behaviour. At the heart of this development, in fact during the phase where software is constructed for a manageable piece, programs for adding numbers, sorting lists, parsing inputs, etcetera are used or constructed. This thesis uses and further develops a method that is ideally suited for constructing just such programs.

Two prominent concerns in program design and implementation are correctness and efficiency. Given the formal specification of a (small) programming problem, it is required to construct an efficient program that satisfies the requirements listed in the specification. Evidently correct programs are often not of an acceptable efficiency, and very often efficient correct programs are not obtained easily. Ever since the 'software crisis' was recognised, already in the sixties, the necessity of using formal methods for program construction has been widely acknowledged. Hoare and Dijkstra have done much pioneering work in this area, in particular on the formal derivation of imperative programs. Various approaches to the design of correct and efficient algorithms partly based on this work are described by Dijkstra and Feijen [37], Gries [59], Morgan [106] and many others.

A specific approach to the construction of correct and efficient programs is *Transformational Programming*. A specification, described in some formal language, is considered to be an evidently correct but likely very inefficient program. By definition, the meaning of a program is its input-output behaviour. If a program is deterministic its meaning is a function; if a program is non-deterministic its meaning is a relation. By means of a series of meaning-preserving (in case the meaning is a function) or meaning-refining (in case the meaning is a relation) transformation steps, the specification is transformed into a correct and efficient program. Since each transformation step preserves or refines the meaning of the program, the result of a series of transformation steps is necessarily correct, and therefore the only remaining concern is that of the efficiency of the program. Thus the concerns of correctness and of efficiency are separated in Transformational Programming. A Transformational Programming methodology should possess at least the following two properties. First, to perform calculations, it should be possible to express both specifications and algorithms in the language used by the methodology, and second, to perform understandable calculations, the notation used for algorithms should be concise. Investigations in Transformational Programming started just before the seventies, and a big impetus to the field was given by Burstall and Darlington [27] and Backus [9]. Nowadays, research in Transformational Programming is widespread, and various approaches can be found in the Proceedings of the IFIP WG 2.1 Working Conferences [99, 101]. This thesis adopts a particular Transformational Programming paradigm, known as 'Constructive Algorithmics' or 'the Bird-Meertens Formalism', in which the transformation steps resemble high school algebra.

1.1 Constructive Algorithmics

Constructive Algorithmics is a Transformational Programming calculus that has a number of desirable properties. Important aspects of the calculus are its emphasis on algebraic manipulation and its means for expressing programs concisely. It uses equational reasoning for obtaining an efficient program from a specification.

The aspiration of the constructive-algorithmics approach to program construction is to cover, eventually, large parts of the 'tricks of the trade' of the practice of computing, and to provide a body of concepts, notations and theories with which the methods and results for certain data types or classes of problems can be described in a systematic way.

In Constructive Algorithmics one derives a program from its specification by means of algebraic manipulation. Ideally, we start off with a specification and using equational reasoning, replacing equals for equals, we arrive at an efficient program by means of a straight-line calculation. Replacing equals for equals is done on the basis of laws. The only intellectual activity required is the selection of a law. The following example shows a calculation of a well-known equality in arithmetic.

$$\begin{array}{ll} & (x-a)(x+a) \\ = & \quad \text{multiplication distributes over minus} \\ & x(x+a) - a(x+a) \\ = & \quad \text{multiplication distributes over plus} \\ & x^2 + xa - ax - a^2 \\ = & \quad \text{multiplication is commutative, definition of minus} \\ & x^2 - a^2 \ . \end{array}$$

This is the ideal situation indeed; sometimes more than pattern matching and replacing equals for equals is needed. The proof format used above and throughout this thesis is the format that consists of a series of expressions separated by a relation symbol and a comment why the two consecutive expressions are thus related. So if expression 1 equals expression 3 via expression 2 and two laws, then we write

$$\begin{array}{ll} & \text{expression 1} \\ = & \quad \text{law 1} \\ & \text{expression 2} \\ = & \quad \text{law 2} \\ & \text{expression 3 .} \end{array}$$

Derivations in Constructive Algorithmics are conducted in a formal language. This language is suitable for calculation: the elements in it, some of which correspond to the functional forms described by Backus [9], satisfy many useful laws. Furthermore, the elements in the language can be easily implemented in or translated into existing computer languages. This justifies the terminology 'calculating with programs' for deriving algorithms in this calculus. Since an expression in Constructive Algorithmics is an abstraction of a program, we call such an expression an algorithm, and from now on we will talk about the derivation of algorithms instead of the derivation of programs. Strictly speaking, it is impossible to talk about the efficiency of an algorithm. When we talk about the efficiency of an algorithm, we mean the efficiency of a straightforward implementation of the algorithm.

The formal language in which calculations are conducted is based on a specific theory of data types, which is described by means of Category Theory. A functor is a notion in Category Theory that we use in the description of an algebra. Given a functor F, the category of F-algebras has as objects all F-algebras, and as morphisms all homomorphisms between F-algebras. A data type such as *list*, *tree*, *array*, etc. is defined as an initial object in a category of functor-algebras. Consider the function which sums a list of natural numbers. This recursive function is a so-called 'catamorphism' on the data type *list* of natural numbers. A catamorphism is the unique homomorphism from an initial F-algebra to a given target F-algebra. The uniqueness can be equationally characterised, which makes this approach ideal for the kind of calculation aimed at. The Fusion Theorem, which gives the condition that has to be satisfied in order to 'fuse' the composition of a function with a catamorphism into another catamorphism, is one of the very useful consequences of the uniqueness of catamorphisms. The Fusion Theorem is the main means with which programs are derived. There exist classes of special catamorphisms (functional forms) that satisfy extra laws. An example of such a class is the class of structure-preserving map operators. On the data type *list*, the map operator takes a function and a list and applies the function to each element in the list. It satisfies for example the map-distributivity law: the composition of two maps is a map again.

It is to be expected that by considering different example problems recurring patterns in derivations give rise to the formulation of new laws and theorems. The number of laws and theorems in the calculus rises with the number of problems for which algorithms have been derived. Important tasks are to organise the laws and theorems such that for specific instances it is clear where to look for applicable laws, and to formulate and abstract from the laws and theorems such that similar laws are captured into one. Most of the laws and theorems applied in the construction of an algorithm are organised in specialised theories. We call a body of laws, theorems, and corollaries a specialised theory if the results apply to a specific class of problems, such as the class of combinatorial algorithms on words, or if the results apply to a specific data type, such as the data type array or the data type rose tree, or if the results give a specific kind of algorithms, such as incremental algorithms or parallel algorithms. A specialised theory is developed by abstracting from one or more derivations for specific problems. For example, the development of a theory of combinatorial algorithms on words, algorithms for so-called segment problems, started with the derivation of an algorithm for finding the maximum segment sum. A useful specialised theory covers a broad range of problems, and gives solutions to both simple problems and difficult problems. The derivation of an algorithm for a given problem consists of the following steps. First, the problem is transformed into a formal specification. Then the specialised theory pertaining to this specification is looked up, and the applicability conditions of the results of this specialised theory are verified for the given specification. If the applicability conditions of some of the results hold for the given specification, and the result returns an efficient algorithm we are done, otherwise an inventive step has to be made. In the inventive step either new theory has to be developed for the specification, or the specification has to be extended with or changed into a function such that some theory is applicable to the extended or changed specification. It follows that programming in Constructive Algorithmics is not always merely applying the appropriate theorem. This thesis develops several different specialised theories for algorithm calculation.

To a limited extent, a computer can be useful in the derivation of algorithms. No existing program transformation system can derive efficient programs for new and difficult problems, but a computer program can alleviate the tedious task of editing formulae, provided it is equipped with a friendly user-interface. Furthermore, a computer program can support looking up specialised theories and applying results if it has a library mechanism. An example of such a 'proofeditor' has been built at the University of Groningen, see Chisholm [31, 30].

Research in Constructive Algorithmics was started in the eighties by Bird [16, 17] and Meertens [96]. Since then, many researchers have contributed to many directions within the field, see [6, 7, 8, 11, 20, 47, 58, 80, 90, 89, 98, 102, 122].

1.2 Theories for algorithm calculation

This thesis consists of two parts. The first part contains an overview of the 'general theory' of Constructive Algorithmics, and the second part contains the developments of several 'specialised theories'. The distinction between general theory and specialised theory is of course vague. One can argue that the theory of data types presented in the part on general theory is special to the development of recursive sequential algorithms on finite data types.

In the part on general theory, the theory of data types developed by Meertens [95, 98], Malcolm [90, 89], Fokkinga [47], and others, is presented, and several auxiliary functions and operators defined on these data types are given. Most of these auxiliary functions appear in various chapters in the part on specialised theory.

The part on specialised theories is built upon the part on general theory, and develops several specialised theories. This part consists of four chapters, and in each of these chapters, a specific specialised theory is developed. The chapters are not completely orthogonal, and especially the results from the chapter on generator fusion theorems, the first chapter in this part, are used in the subsequent chapters. We give a brief overview of the chapters in this part.

Chapter 4 discusses generator fusion theorems. A specification is often of the form 'find the mimsiest borogove that is slithy'. Borogove is some noun, and mimsy and slithy are some adjectives, the meanings of which are not important for this discussion. Moreover, we assume two things can be compared for mimsiness. For example, if we replace *mimsiest* by shortest, *borogove* by path, and *is slithy* by visits all cities from set A, then we obtain a specification of the Travelling Salesman Problem. Another example is obtained if we replace mimsiest by some, borogove by permutation and is slithy by is ascending. The result is a specification of sorting. As a last example, let mimsiest be largest, let borogove be natural number, and slithy be prime, then we obtain a specification of finding the largest prime number. Most of the specifications considered in this thesis follow this scheme.

We want to construct an efficient algorithm for finding the mimsiest borogove that is slithy, using Constructive Algorithmics. Since calculations are conducted in a formal language, the informal specification has to be translated into this language. Generally, translating an informal specification into a formal specification is one of the most difficult tasks in the construction of software. In our case, however, the translation is almost trivial. We distinguish three parts in the specification: borogove, is slithy, and mimsiest. Let *borogoves* be a function that enumerates all borogoves as a set. We assume that function *borogoves* takes a list as argument, and that it is defined as a catamorphism on the data type *list*. Function *borogoves* is a so-called generator. Given an element of a data type as argument, a generator is usually a polymorphic function that generates a set of elements that are related in a specific manner to the argument. A generator may be any function, but very often it is a catamorphism or the composition of a projection function with a catamorphism. In the specification of sorting *borogoves* is the function *perms* that takes a bag (also called a multiset), and returns the set of all permutations of the bag. Let *slithy* be a boolean function defined on borogoves, that determines whether or not a borogove is slithy. For example, in the specification of sorting *slithy* is the function *asc* which determines whether or not a list is ascending. Given the function *slithy*, we construct the function $\mathit{slithy}\triangleleft$ which, given a set of borogoves, retains the set of borogoves that are slithy. Finally, let *mimsier* be a function that given two borogoves returns the mimsier of the two. We construct the function $\mathit{mimsier}/$ which given a set of borogoves, returns the mimsiest borogove, by repeatedly comparing two borogoves and taking the mimsier of the two. For function $\mathit{mimsier}/$ to be well-defined, operator *mimsier* has to be associative, commutative and idempotent (note that the maximum operator on natural numbers is such an operator). We have described three functions that are the translations of the informal specification into the formal language. The composition of these three functions is the desired formal specification.

$$\mathit{mimsier}/ \cdot \mathit{slithy}\triangleleft \cdot \mathit{borogoves}\ .$$

We have assumed that the constituents of the informal specification can be described in the formal language. This specification is an instance of the class of *borogoves* problems. The three examples described above can be specified thus. Occasionally, the straightforward implementation of the formal specification is an efficient algorithm. However, in almost all cases the program obtained will be very inefficient. Consider for example the specification of sorting. Given a bag with n elements, there may be $n!$ permutations, so the straightforward implementation of sorting is of an unacceptable efficiency.

For the derivation of an efficient algorithm for the formal specification we proceed by abstracting from the composition of functions $\mathit{mimsier}/ \cdot \mathit{slithy}\triangleleft$ and replacing it by an arbitrary catamorphism on the data type *set*. Since we assume that function *borogoves* is a catamorphism, Fusion can be applied to the specification. To fuse the composition of a catamorphism with the function *borogoves*, the catamorphism has to satisfy the applicability conditions of Fusion. We derive conditions on the components of the catamorphism that are sufficient to fuse the composition of the catamorphism with the function *borogoves* into a catamorphism. These conditions are formulated in a *borogoves*-Fusion Theorem. The *borogoves*-Fusion Theorem is an example of a generator fusion theorem. The *borogoves*-Fusion Theorem is applied to the instance of the class of *borogoves* problems given above, and if the components satisfy the conditions of the *borogoves*-Fusion Theorem, we obtain an algorithm for our problem. If the components of the specification do not satisfy the conditions of the *borogoves*-Fusion Theorem, then we try to extend the problem, i.e., to tuple it with another function, such that it can be applied.

Chapter 5 discusses the derivation of algorithms for problems on segments of lists. Function *segs* returns all segments (subwords) of a list. The *segs*-Fusion Theorem we derive has a number of corollaries that are used in the derivation of algorithms for several example segment problems. Furthermore, these corollaries provide a starting point for some involved problems such as pattern matching on lists and finding the lexicographically least rotation of a list. For the construction of an efficient algorithm for finding a longest palindrome of a list, the theory developed is extended with a theory of segment problems in which the context of a segment is taken into account.

Chapter 6 discusses the definition and derivation of algorithms on the data type array. A hierarchical data type is an initial algebra in a category of functor-algebras where the functor 'imports' another initial algebra. An example of a hierarchical data type is the data type list of lists. By means of a theory for hierarchical data types, the data type n-dimensional array is defined as a hierarchical data type. An n-dimensional array is considered to be a list of $(n-1)$-dimensional arrays. A catamorphism on the data type array is called a hierarchical left-reduction. We prove a number of theorems for manipulating hierarchical left-reductions, and we show how several well-known functions defined on arrays can be expressed as hierarchical left-reductions. The notion of subarray of an array corresponds to the notion of segment of a list. The family of functions suba_n, which returns the subarrays of an n-dimensional array, is defined by means of a hierarchical left-reduction. We derive a suba_n-Fusion Theorem, and using this theorem, we derive a hierarchy of algorithms for finding the maximum subarray sum. Furthermore, we derive a hierarchy of algorithms for pattern matching on n-dimensional arrays.

Chapter 7 discusses the definition and derivation of incremental algorithms. Suppose a computation is performed repeatedly on data that is slightly changing. An incremental algorithm describes how to compute a required value depending on the slightly changed data from the old value, the changes in the data, and perhaps some other information. Examples of computations that are performed repeatedly on slightly changed data can be found in interactive programs such as program development environments and spreadsheet programs. The form of an incremental algorithm depends on the data type on which it is defined, and the edit model used. We propose a method for the description and derivation of incremental algorithms on several data types; the emphasis will be on the data type list. We derive several incremental algorithms, for example for coding a text with respect to a static dictionary, and formatting a text.

Part I

General theory

Chapter 2

Language and laws

An algorithm is a prescription of how to compute output given some input. It can be viewed as a function or a relation from the set of possible inputs to the set of possible outputs. Usually, the elements in the set of possible inputs of an algorithm share a similar structure. For example, quicksort sorts lists of elements on which some total order is defined such as natural numbers, but quicksort would not be able to sort a tree (whatever might be meant by a sorted tree). Similarly, the factorial function is only defined on natural numbers. Elements that share a similar structure are assembled in a data type. The notion of data type has pervaded many theories of algorithm design and implementation. The approach to algorithm design sketched in this thesis is based on a specific theory of data types.

The theory of data types we consider is purely mathematical in nature, and the activity of program design amounts to deriving equalities in the theory. The reason why we speak of a theory of data types for program design is that the components of the theory, the elements in the language of the theory, can easily be translated into computer programs. All functions can be implemented straightforwardly in a functional language such as Miranda[1], see Turner [133], or Haskell, see Hudak and Wadler [67], and with a little more effort in an imperative language such as Pascal. We use a paradigm of data type-definition due to Hagino [60], an important aspect of which is that data types are characterised by a universal property, which prescribes the construction of some specific recursive functions on the defined data type. These recursive functions are called 'catamorphisms'. Catamorphisms play a prominent role in the theory of data types introduced in this chapter. The data types that can be defined using this paradigm include familiar ones such as *list*, *tree*, and *set*. The universal property by means of which data types are characterised provides a basis for succinct notations and elegant equational proofs. A corollary of this property that is called 'fusion' turns out to be particularly useful. Fusion provides a means for proving equalities of functions avoiding the application of induction in the development of algorithms; it can be viewed as 'canned induction'. An elegant treatment of Hagino data

[1]'Miranda' is a trademark of Research Software Ltd

types and properties like fusion is given by Malcolm [90, 89]. The data types we consider are finite data types such as finite lists and trees. Such a data type is an initial object in a specific category of algebras. An infinite data type such as the data type infinite lists is a terminal object in a specific category of co-algebras, see Malcolm [89]. Data types comprising both finite and infinite elements are described by Fokkinga and Meijer [48].

This chapter discusses the theory of data types due to Hagino [60]. Section 2.1 introduces the basic components of our language, such as function, set, cartesian product, etc. Section 2.2 introduces some of the basic concepts of category theory, that are used in Section 2.3, which describes categories of functor-algebras, Section 2.4, in which the definition of a data type is given, and Section 2.5, which discusses map-functors. Section 2.6 describes the Boom-hierarchy, a collection of four data types that is frequently used in the calculation of algorithms: the data types *set*, *bag*, *list*, and *binary tree*. Section 2.7 discusses the data types *snoc-list* and *cons-list*, and Section 2.8, finally, gives some conclusions.

2.1 Notational conventions

This section introduces some parts of set theory. Most of our notational conventions are standard. Deviations from standard notation occur when standard notation is not well suited for calculation and manipulation. For example, the case construct, usually denoted by $[f, g]$ in category theory, is denoted by $f \triangledown g$. Thus the case construct is a real infix operator, and the number of symbols that have to be written is reduced. Most of the notational conventions we adopt have been introduced by Bird [16, 17], Meertens [95, 98], and Fokkinga and Meijer [48]. This section is divided in four subsections on, respectively, sets, functions and relations, cartesian product, disjoint sum, and booleans.

2.1.1 Sets, functions, and relations

Sets

A *set* is a collection of different members. Some sets are called types. A set may be defined by the *set-comprehension* notation $\{x \mid P\,x\}$, where P is some property enjoyed by all members of the set, but not by anything outside the set. We use conventional set theory notation such as $\{\,\}$ for the empty set, $\in$ for the containment relation, $\subseteq$ for the subset relation, etc.

Functions

A *function* is an object with three components written $f : s \to t$, where s is a set called the *source* of the function, t is a set called the *target* of the function, and f maps each member x

of s to a member of t. This member is denoted $f\,x$, using simple juxtaposition and a little white space to denote application of a function f to an argument x. When s and t are clear from the context we write just f instead of $f : s \rightarrow t$. We use the letters f, g, h, etc., as variables standing for arbitrary functions. Function application is right-associative, i.e., we have

$$f\,(g\,(h\,x)) \quad = \quad f\,g\,h\,x \ .$$

The *image* of a function $f : s \rightarrow t$ is the set of elements of the target of f for which there exists a source element, that is,

$$\mathit{image}\,f \quad = \quad \{e \mid \exists u \in s : f\;u = e\} \ ,$$

so *image f* $\subseteq t$. The *composition* of two functions $f : s \rightarrow t$ and $g : r \rightarrow s$ is written $f \cdot g : r \rightarrow t$. Composition is associative, that is, for all f, g, and h we have $f \cdot (g \cdot h) = (f \cdot g) \cdot h$. Taking advantage of associativity, chains of compositions are usually written without brackets, $f \cdot g \cdot h$. Examples of functions are the successor function on natural numbers (the set of natural numbers is denoted by *nat*) $\mathit{succ} : \mathit{nat} \rightarrow \mathit{nat}$, which is defined by

$$\mathit{succ}\,n \quad = \quad n + 1 \ ,$$

the identity function: for each set s, the function $\mathit{id}_s : s \rightarrow s$ is the identity function on the set s, that is, for all x in s

$$\mathit{id}_s\,x \quad = \quad x \ ,$$

and the constant function $\underline{c} : s \rightarrow \{c\}$ which, given a constant c, returns c on each argument, that is, for all x in s

$$\underline{c}\,x \quad = \quad c \ .$$

The identity function and the constant function are the first examples of polymorphic functions, in the sense that $\mathit{id}_s : s \rightarrow s$ and $\underline{c} : s \rightarrow \{c\}$ for all sets s. Many more polymorphic functions will be encountered in the sequel. Two functions $f : s \rightarrow t$ and $g : s \rightarrow t$ are equal if they coincide on all arguments, that is,

$$f = g \qquad \Leftarrow \quad (\forall x \in s : f\;x = g\,x) \ . \tag{2.1}$$

This implication is called the *extensionality axiom*. The 'converse' of this axiom is called *Leibniz' rule*:

$$(\forall x \in s : f\;x = g\,x) \qquad \Leftarrow \qquad f = g \ . \tag{2.2}$$

Two consequences of Leibniz' rule and the extensionality axiom are

$$f \cdot h = g \cdot h \qquad \Leftarrow \qquad f = g \tag{2.3}$$

$$h \cdot f = h \cdot g \qquad \Leftarrow \qquad f = g \ . \tag{2.4}$$

Relations

A *relation* is a set of pairs. Let R be a relation. We write $x\ R\ y$ for $(x, y) \in R$. Statements of the form $(f\ x)\ R\ (f\ y)$ occur very often. We introduce a shorthand for these expressions:

$$x\ R_f\ y \quad \equiv (f\ x)\ R\ (f\ y)\ . \tag{2.5}$$

For example, for the statement $24\ \mathit{mod}\ 10 = 34\ \mathit{mod}\ 10$ we may write $24 =_{\mathit{mod}\ 10} 34$.

2.1.2 Cartesian product

If A and B are sets, then, by definition, their *cartesian product*, $A \times B$, is a set whose members are all pairs (a, b), where $a \in A$ and $b \in B$. We have

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

A function with a cartesian product as its source is referred to as a *binary operator*. The operator $+ : \mathit{nat} \times \mathit{nat} \to \mathit{nat}$ is such a function. Typical variable names for binary operators are $\oplus$, $\otimes$, $\odot$, etc. We will frequently use infix notation for the application of a binary operator $\oplus$ to an argument (a, b), so instead of writing $\oplus\,(a, b)$ we write $a \oplus b$. Here and throughout, we adopt the convention that function application is more binding than infix binary application, so the expression $f\ a \oplus g\ b$ should be parsed as $(f\ a) \oplus (g\ b)$.

Sections

Binary operators can be parameterised, i.e., if $\oplus$ is a binary operator of type $A \times B \to C$, and $a \in A$, we consider the expression $(a\oplus)$ to be a unary function of type $B \to C$. It is defined by

$$(a\oplus)\ b \quad = \quad a \oplus b\ .$$

The function $(\oplus b) : A \to C$ is defined similarly. These parameterised operators are also known as *sections*. For example, the function $(+3) : \mathit{nat} \to \mathit{nat}$ takes a natural number as argument and returns the number increased by three, so $(+3)\,5 = 8$. Other examples of sections are the function $(\mathit{mod}\ 10)$ and the function $(1-)$.

Projection functions

The cartesian product has two primitive operations, the projections $\mathit{exl} : A \times B \to A$ ('first' or left-projection) and $\mathit{exr} : A \times B \to B$ ('second' or right-projection). The projection functions are defined by

$$\mathit{exl}\,(a, b) \quad = \quad a \tag{2.6}$$

$$\mathit{exr}\,(a, b) \quad = \quad b\ . \tag{2.7}$$

Note that the projection functions are polymorphic functions.

Units and zeros

If operator $\oplus$ has a unit or identity element, then it is written $\nu_\oplus$. An identity element is also called a neutral element. The element $\nu_\oplus$ satisfies by definition

$$\nu_\oplus \oplus a = a \oplus \nu_\oplus = a$$

for all a. If operator $\oplus$ has a zero or absorbing element, then it is written $\alpha_\oplus$. The element $\alpha_\oplus$ satisfies by definition

$$\alpha_\oplus \oplus a = a \oplus \alpha_\oplus = \alpha_\oplus$$

for all a. If the unit and/or zero element exist, they are unique. If the unit and/or zero element do not exist we may introduce a 'fictitious' element with the same property, though one should be careful when doing this, since some equalities may not hold anymore. Consider for example the operator *exl*. It does not have a unit element, so let ν_{exl} be a fictitious unit element of *exl*. Operator *exl* is now defined by

$$exl\,(a,b) \quad = \quad \begin{cases} a & a \neq \nu_{exl} \\ b & a = \nu_{exl}\ , \end{cases}$$

and it follows that equation (2.6) does not hold anymore.

Products of functions

The operator $\times$ is defined on functions as well as sets. For functions $f : A \to B$ and $g : C \to D$, function $f \times g : A \times C \to B \times D$ is defined for all pairs $(a, c) \in A \times C$ by

$$(f \times g)\,(a, c) \quad = \quad (f\ a, g\ c)\ . \tag{2.8}$$

Operator $\times$ binds stronger than composition, and satisfies the following law.

$$f \times g \cdot h \times j \quad = \quad (f \cdot h) \times (g \cdot j)\ . \tag{2.9}$$

This law is proved, using the extensionality axiom, equation (2.1), by taking an arbitrary element (a, b) of the type $A \times B$, where A is the source type of h and B is the source type of j, and showing that the results of both sides applied to (a, b) are equal. We have

$$\begin{aligned}
& (f \times g \cdot h \times j)\,(a, b) \\
= & \\
& (f \times g)\,(h\ a, j\ b) \\
= & \\
& (f\ h\ a, g\ j\ b) \\
= & \\
& ((f \cdot h) \times (g \cdot j))\,(a, b)\ .
\end{aligned}$$

Split

If we swap the components of the type of binary operators we obtain the type $C \to A \times B$. Functions of this type can be constructed as follows. Let $f : C \to A$ and $g : C \to B$ be functions, then the function $f \vartriangle g : C \to A \times B$ (pronounced 'f split g') is defined by

$$(f \vartriangle g)\, c \quad = \quad (f\, c, g\, c)\ . \tag{2.10}$$

The usual categorical notation for $f \vartriangle g$ is $\langle f, g \rangle$. Just like operator $\times$, operator $\vartriangle$ binds stronger than composition. An example of a function defined by means of split is the function $\mathit{exr} \vartriangle \mathit{exl}$, which swaps the components of a pair. This function is called *swap*:

$$\mathit{swap} \quad = \quad \mathit{exr} \vartriangle \mathit{exl}\ .$$

Laws for split and product

The operators $\times$ and $\vartriangle$, and the projection functions satisfy the following laws. The proofs of these laws are similar to the proof of law (2.9) and are therefore omitted.

$$\begin{aligned}
f \times g &= (f \cdot \mathit{exl}) \vartriangle (g \cdot \mathit{exr}) && (2.11)\\
f \times g \cdot h \vartriangle j &= (f \cdot h) \vartriangle (g \cdot j) && (2.12)\\
(f \cdot h) \vartriangle (g \cdot h) &= f \vartriangle g \cdot h && (2.13)\\
\mathit{exl} \cdot f \times g &= f \cdot \mathit{exl} && (2.14)\\
\mathit{exr} \cdot f \times g &= g \cdot \mathit{exr} && (2.15)\\
\mathit{exl} \cdot f \vartriangle g &= f && (2.16)\\
\mathit{exr} \cdot f \vartriangle g &= g && (2.17)\\
\mathit{exl} \vartriangle \mathit{exr} &= \mathit{id}_{A \times B} && (2.18)\\
f \vartriangle g = h \vartriangle j &\equiv (f = h)\ \wedge\ (g = j)\ . && (2.19)
\end{aligned}$$

The $n{+}1$-fold cartesian product

The construction of the cartesian product can be generalised to types whose members are triples, quadruples, etc. Let $n > 1$ be a natural number. If $A_0, \ldots, A_n$ are types, then their $n{+}1$-*fold cartesian product*, denoted by $A_0 \times \ldots \times A_n$, is a type whose members are all $n{+}1$-tuples $(a_0, \ldots, a_n)$, where for all i with $0 \le i \le n$ we have $a_i \in A_i$. There are $n{+}1$ projection functions defined on an $n{+}1$-tuple. For all i with $0 \le i \le n$ the projection function π_i is defined by

$$\begin{aligned}
\pi_i &: A_0 \times \ldots \times A_n \to A_i\\
\pi_i\,(a_0, \ldots, a_n) &= a_i\ .
\end{aligned}$$

The extension of the combinators $\times$ and $\vartriangle$ to the n+1-fold cartesian product and the laws satisfied by these combinators are omitted.

Operator $\propto_i$, with $i \geq 1$, takes an i-tuple and an element and returns an (i+1)-tuple with the element as its last component. We only use the operators $\propto_2$ and $\propto_1$. Operator $\propto_2$ is defined by

$$(x, y) \propto_2 z \quad = \quad (x, y, z)\ . \tag{2.20}$$

The function $\natural_i$, with $i \geq 1$ takes an (i+1)-tuple and removes the last element of the (i+1)-tuple to obtain an i-tuple, so operator $\natural_2$ is defined by

$$\natural_2 (x, y, z) \quad = \quad (x, y)\ . \tag{2.21}$$

It follows that $\natural_2 \cdot (\propto_2 z) = id_{A\times B}$ for all z.

2.1.3 Disjoint sum

The *disjoint sum*, also called coproduct or discriminated union, is dual to the cartesian product. If A and B are sets, there exists a set $A+B$, whose elements are the union of the elements of A and B, tagged with the origin of the element (left or right). By definition, interpreting 0 as left and 1 as right,

$$A + B \quad = \quad \{(c, n) \,|\, (c \in A \text{ and } n = 0) \text{ or } (c \in B \text{ and } n = 1)\}\ .$$

The operator + is also defined on functions. For functions $f : A \rightarrow B$ and $g : C \rightarrow D$, the function $f + g : A + C \rightarrow B + D$ is the function that for a left-tagged value a returns the left-tagged value $f\ a$, and for a right-tagged value b the right-tagged value $g\ b$. The dual of the operator split is called 'junc'. Junc combines two functions $f : A \rightarrow C$ and $g : B \rightarrow C$ into one function $f \triangledown g : A + B \rightarrow C$. Function $f \triangledown g$ applies f to left-tagged, and g to right-tagged values, thereby loosing the tag information. Function $f \triangledown g$ is also called the case construct, and it is often denoted by $[f, g]$. We have two injection functions $inl : A \rightarrow A + B$ and $inr : B \rightarrow A + B$. The dual versions of laws (2.9) to (2.18) read as follows.

$$f + g \cdot h + j \quad = \quad (f \cdot h) + (g \cdot j) \tag{2.22}$$
$$f + g \quad = \quad (inl \cdot f) \triangledown (inr \cdot g) \tag{2.23}$$
$$f \triangledown g \cdot h + j \quad = \quad (f \cdot h) \triangledown (g \cdot j) \tag{2.24}$$
$$(h \cdot f) \triangledown (h \cdot g) \quad = \quad h \cdot f \triangledown g \tag{2.25}$$
$$f + g \cdot inl \quad = \quad inl \cdot f \tag{2.26}$$
$$f + g \cdot inr \quad = \quad inr \cdot g \tag{2.27}$$
$$f \triangledown g \cdot inl \quad = \quad f \tag{2.28}$$
$$f \triangledown g \cdot inr \quad = \quad g \tag{2.29}$$
$$inl \triangledown inr \quad = \quad id_{A+B} \tag{2.30}$$
$$f \triangledown g = h \triangledown j \quad \equiv \quad (f = h)\ \wedge\ (g = j)\ . \tag{2.31}$$

Let $\phi : A + B \to C$. Then

$$\phi \;=\; (\phi \cdot inl) \triangledown (\phi \cdot inr) \;,$$

so any function defined on a disjoint sum can be written as a junc of two functions. Finite sum can be defined as a generalisation of the disjoint sum just as the finite cartesian product is defined as a generalisation of the cartesian product.

2.1.4 Booleans

The set *bool* contains two elements: *false* and *true*. Negation is denoted by $\neg : bool \to bool$, and implication, equivalence, conjunction, and disjunction, all of type $bool \times bool \to bool$, are denoted respectively by $\Rightarrow$, $\equiv$, $\wedge$, and $\vee$. These operators bind weaker than all binary operators introduced until now, and $\Rightarrow$ and $\equiv$ bind weaker than $\vee$ and $\wedge$.

We summarise the precedences of the binary operators we have introduced from highest to lowest.

(function application)
$\times$, $\vartriangle$, $+$, $\triangledown$, $\oplus$, $\otimes$ ...
$\cdot$ (function composition)
$\wedge$, $\vee$
$\Rightarrow$, $\equiv$, $=$.

In case this table does not give a unique parsing of an expression, for example in equation (2.31) above, white space is used to determine the precedence of operators. Since $\equiv$ has more white space around it than $=$ in equation (2.31), it has lower precedence.

2.2 Category theory

Category theory is one of the most abstract and general branches of mathematics. It is used, among others, to provide a general framework for theories of programming. Such theories form the basis of the design and definition of programming languages and their associated software engineering methods. Its importance in the field of Computing Science is witnessed by the many introductions to Category Theory for computing scientists, see for example Hoare [63], Fokkinga [46] and Barr and Wells [12]. In this thesis, following Hagino [60] and Malcolm [90, 89], category theory is used to describe a theory of data types. This theory of data types comprises the language in which we present our algorithms and the laws we use to derive the algorithms. The notion of functor is of particular importance in the definition of a data type. A data type is defined as a specific algebra, and an algebra can be described by means of a functor that captures the structure of the operations of the algebra in a single definition. Thus we avoid many distracting details such as subscripts, signatures, and families of operator symbols and operations, which are usually present in publications on initial algebras. Another important concept we borrow from category theory is definition by means of a unique extension property, also called a universal mapping property, an example of which is the definition of an initial object. Given a functor F, a data type will be defined as an initial object in the category of F-algebras. The main purpose of this section is to introduce our notation of categorical concepts, and not to explain category theory. We suppose the reader has some knowledge of category theory, which may be obtained from any of the introductions to category theory listed above. This section is divided into four subsections, in which we introduce categories, unique extension properties, functors, and natural transformations, respectively.

2.2.1 Categories

By definition, a *category* consists of six components: a collection of *objects*, a collection of *morphisms* (also called arrows), two functions from morphisms to objects, called *source* and *target*, a binary partial operator on morphisms called *composition*, and for each object A a distinguished morphism called the *identity* on A. We write $f : A \to B$ when $source\, f = A$ and $target\, f = B$. These components satisfy the following axioms. For each object A,

$$id_A \quad : \quad A \to A \; .$$

If $f : A \to B$ and $g : B \to C$, then

$$g \cdot f \quad : \quad A \to C \; .$$

Furthermore, composition is associative, and whenever $f : A \to B$, then $id_B \cdot f = f = f \cdot id_A$. We use boldface capital letters like $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{E}$ to denote categories.

An example of a category is the category **Set**, which is obtained by taking as objects arbitrary sets and as morphisms arbitrary typed total functions. Another category we shall have occasion to use is the product category. If $\mathbf{C}$ and $\mathbf{D}$ are categories, then the product category $\mathbf{C} \times \mathbf{D}$ is the category whose objects are all ordered pairs (C, D) with C an object of category $\mathbf{C}$ and D an object of category $\mathbf{D}$, and in which a morphism $(f, g) : (C, D) \to (E, F)$ is a pair of morphisms $f : C \to E$ in $\mathbf{C}$ and $g : D \to F$ in $\mathbf{D}$. The identity of object (C, D) is the morphism (id_C, id_D). Composition in $\mathbf{C} \times \mathbf{D}$ is defined componentwise. Let $(f, g) : (C, D) \to (E, F)$ and $(h, j) : (E, F) \to (G, H)$ be morphisms in $\mathbf{C} \times \mathbf{D}$. We define

$$(h, j) \cdot (f, g) \quad = \quad (h \cdot f, j \cdot g) : (C, D) \to (G, H) \; .$$

Note that the composition at the left of the equality symbol is the composition in $\mathbf{C} \times \mathbf{D}$, whereas the compositions at the right of the equality symbol are, respectively, the composition in $\mathbf{C}$ and the composition in $\mathbf{D}$.

A morphism $f : A \to B$ is an *isomorphism* if there exists a morphism $g : B \to A$ such that $f \cdot g = id_B$ and $g \cdot f = id_A$. If $f : A \to B$ is an isomorphism, we call A and B isomorphic.

2.2.2 Unique extension properties

An object A of a category $\mathbf{C}$ is *initial* in $\mathbf{C}$ if there is exactly one morphism of type $A \to B$ for all objects B of $\mathbf{C}$. In other words, A is initial in $\mathbf{C}$ if there exists a morphism $(\!|B|\!)$ such that

$$x = (\!|B|\!) \quad \equiv \quad x : A \to B \ . \tag{2.32}$$

Here $(\!|B|\!)$ is just a notation, a name, for a morphism. All variables in line (2.32), except A, are universally quantified. The dual notion, an object of a category that has a unique morphism from each object (including itself), is called a *terminal* object. In **Set**, the only initial object is the empty set. The terminal objects in **Set** are all sets with one element, the unitsets, which are defined by $\{x \mid x = y\}$. The concept of terminality (and initiality) is defined up to isomorphism: all unitsets are isomorphic. 'The' terminal object is denoted by $\mathit{1}$, and its element is denoted by o.

The definition of an initial object is the first explicit example of a characterisation by means of a unique extension property, or a universal mapping property. The calculational properties of initiality are investigated by Fokkinga [43]. A second example of such a characterisation is the following characterisation of the operator split. For functions $f : A \to B$ and $g : A \to C$ in the category **Set**, the function $f \vartriangle g : A \to B \times C$ is uniquely characterised by the following property. For all $h : A \to B \times C$ we have

$$h = f \vartriangle g \quad \equiv \quad (\mathit{exl} \cdot h = f) \ \wedge \ (\mathit{exr} \cdot h = g) \ . \tag{2.33}$$

A similar characterisation can be given for the function $f \triangledown g$. By taking $h = f \vartriangle g$ in the above definition we obtain laws (2.16) and (2.17). Using these laws, we can prove law (2.13) as follows.

$$\begin{array}{ll}
 & (f \cdot h) \vartriangle (g \cdot h) = f \vartriangle g \cdot h \\
\equiv & \qquad \text{characterisation (2.33)} \\
 & (\mathit{exl} \cdot f \vartriangle g \cdot h = f \cdot h) \ \wedge \ (\mathit{exr} \cdot f \vartriangle g \cdot h = g \cdot h) \\
\equiv & \qquad \text{laws (2.16) and (2.17)} \\
 & \mathit{true} \ .
\end{array}$$

Since operator $\times$ can be defined in terms of $\vartriangle$, laws (2.14) and (2.15) can be proved similarly. Law (2.18) is obtained if we take exl for f and exr for g in (2.33).

2.2.3 Functors

A *functor* is a structure preserving map between categories. It is defined on objects as well as morphisms, and preserves identities and composition. We use symbols F, G, H for functors. The application of a functor F to an argument x, which may be an object or a morphism, is written, following Malcolm [89], using reverse juxtaposition, $x\mathsf{F}$. Functor application binds just as strong as function application. Formally, let $\mathbf{C}$ and $\mathbf{D}$ be categories; then a functor from $\mathbf{C}$ to $\mathbf{D}$ is a mapping F that sends every object in $\mathbf{C}$ to an object in $\mathbf{D}$, and every morphism in $\mathbf{C}$ to a morphism in $\mathbf{D}$ in such a way that $f\mathsf{F} : A\mathsf{F} \to B\mathsf{F}$ whenever $f : A \to B$. Furthermore, for all objects A in $\mathbf{C}$,

$$id_A\mathsf{F} \;=\; id_{A\mathsf{F}} \; .$$

Finally, if the composition $g \cdot f$ is defined in $\mathbf{C}$, then $(g \cdot f)\mathsf{F}$ is a morphism in $\mathbf{D}$ that satisfies

$$(g \cdot f)\mathsf{F} \;=\; g\mathsf{F} \cdot f\mathsf{F} \; .$$

As an example, define for arbitrary category $\mathbf{C}$ the identity functor $\mathsf{I}_\mathbf{C} : \mathbf{C} \to \mathbf{C}$. We omit the subscript of functor $\mathsf{I}_\mathbf{C}$ whenever it is clear in which category I is defined. Let x be a variable standing for an object or a morphism. Functor I is defined by

$$x\mathsf{I} \;=\; x \; .$$

We verify the two conditions. Since for all objects and morphisms x we have $x\mathsf{I} = x$, it follows that $id_A\mathsf{I} = id_A = id_{A\mathsf{I}}$. If the morphism $g \cdot f$ is defined in category $\mathbf{C}$ we have $(g \cdot f)\mathsf{I} = g \cdot f = g\mathsf{I} \cdot f\mathsf{I}$. It follows that I is a functor indeed. Given an object A, the constant functor $\underline{A}$ is defined as follows. For all objects B and for all morphisms f

$$\begin{aligned} B\,\underline{A} &= A \\ f\,\underline{A} &= id_A \; . \end{aligned}$$

The constant functor $\underline{1}$ will appear frequently in the sequel.

Binary functors

A *binary functor* is a functor the source of which is a product category. In contrast, a functor with a single category as source is sometimes called a *monofunctor*. Examples of binary functors, defined on the category $\mathbf{Set} \times \mathbf{Set}$, are the cartesian product and the disjoint sum. The definitions of $\times$ and $+$ on sets and functions imply that $(\times) : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$ and $(+) : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$ are binary functors. Another binary functor is the functor $exl : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$. On the pair of sets (C, D) functor exl is defined by $(C, D)\; exl = C$, and on the pair of functions (f, g) functor exl is defined by $(f, g)\; exl = f$. The functor-conditions are easily verified.

Polynomial functors

Let $\dagger$ be a binary functor, and let F and G be monofunctors. Then monofunctors $\mathsf{F} \dagger \mathsf{G}$ and FG are defined by

$$\begin{aligned} x(\mathsf{F} \dagger \mathsf{G}) &= x\mathsf{F} \dagger x\mathsf{G} \\ x(\mathsf{FG}) &= (x\mathsf{F})\mathsf{G} \; . \end{aligned}$$

Note that binary functor † is 'lifted' to act on functors instead of morphisms and objects. All examples of monofunctors we have given until now belong to the class of *polynomial functors*. Polynomial functors are generated by

$$\mathsf{F} \quad ::= \quad \mathsf{I} \mid \mathsf{A} \mid \mathsf{F} \times \mathsf{G} \mid \mathsf{F} + \mathsf{G} \mid \mathsf{FG} \; .$$

Examples of a polynomial functors are the functors II, the doubling functor, and III, the tripling functor, which are defined by

$$\begin{aligned} \mathsf{II} &= \mathsf{I} \times \mathsf{I} \\ \mathsf{III} &= \mathsf{I} \times \mathsf{I} \times \mathsf{I} \; . \end{aligned}$$

2.2.4 Natural transformations

A *natural transformation* from a functor F to a functor G is a polymorphic function η such that for all types A, $\eta_A : A\mathsf{F} \rightarrow A\mathsf{G}$ and for all functions $f : A \rightarrow B$, η satisfies

$$\eta_B \cdot f\mathsf{F} \quad = \quad f\mathsf{G} \cdot \eta_A \; .$$

We write this as $\eta : \mathsf{F} \stackrel{.}{\rightarrow} \mathsf{G}$. A natural transformation is in fact a family of morphisms; for each type we have one morphism. For example, we have $id : \mathsf{I} \stackrel{.}{\rightarrow} \mathsf{I}$, since for all $f : A \rightarrow B$ we have $id_B \cdot f = f = f \cdot id_A$. The polymorphic function *exl* is another example of a natural transformation $exl : \times \stackrel{.}{\rightarrow} exl$ (note that the occurrence of *exl* in the type of this natural transformation is the functor *exl*). This follows from the fact that for all pairs of functions $(f, g) : (C, D) \rightarrow (E, F)$ we have $exl \cdot f \times g = f \cdot exl$. In this thesis, all polymorphic functions are natural transformations. Wadler [135] and De Bruin [26] prove that any 'reasonable' polymorphic function is a natural transformation.

2.3 Algebras and homomorphisms

In the algebraic calculus for program construction we use algebras play an important role. Data types are modelled by initial algebras or terminal algebras. The data types considered in this thesis are initial algebras.

Functor-algebras

Let $\mathbf{C}$ be a category and F a functor on $\mathbf{C}$. By definition, an F-*algebra* is a pair (A, ϕ) where A is an object from $\mathbf{C}$, called the *carrier* of the algebra, and $\phi : A\mathsf{F} \rightarrow A$ is a morphism, called the operation of the algebra. For example, if ϕ is a binary operator of type $A \times A \rightarrow A$, then (A, ϕ) is a II-algebra. The carrier of the algebra of booleans is the set *bool* $= \{0, 1\}$. The two constants 0 and 1 are modelled with the nullary functions $\mathit{false} : \mathit{1} \to \mathit{bool}$ and $\mathit{true} : \mathit{1} \to \mathit{bool}$. Taking

$$\mathsf{G} \;=\; \underline{\mathit{1}} + \underline{\mathit{1}}\ ,$$

we have that $\mathit{false} \triangledown \mathit{true} : \mathit{bool}\mathsf{G} \to \mathit{bool}$, so $(\mathit{bool}, \mathit{false} \triangledown \mathit{true})$ is a G-algebra. Another example is the algebra of natural numbers. The carrier of this algebra is the set $\mathit{nat} = \{0, 1, 2, \ldots\}$. The constant 0 is modelled with the nullary function $\mathit{zero} : \mathit{1} \to \mathit{nat}$, and the successor function is modelled with $\mathit{succ} : \mathit{nat} \to \mathit{nat}$. Taking

$$\mathsf{H} \;=\; \underline{\mathit{1}} + \mathsf{I}\ ,$$

we have that $\mathit{zero} \triangledown \mathit{succ} : \mathit{nat}\mathsf{H} \to \mathit{nat}$ and $(\mathit{nat}, \mathit{zero} \triangledown \mathit{succ})$ is an H-algebra.

Functor-homomorphisms

Homomorphisms are special functions between algebras. Given two F-algebras (A, ϕ) and (B, ψ), an F-*homomorphism* from (A, ϕ) to (B, ψ) is, by definition, a morphism $h : A \to B$ satisfying

$$h \cdot \phi \;=\; \psi \cdot h\mathsf{F}\ .$$

Consider the function $(\times y) : \mathit{nat} \to \mathit{nat}$ mapping a natural number n to $n \times y$. It is defined by

$$\begin{aligned} (\times y) \cdot \mathit{zero} &\;=\; \mathit{zero} \\ (\times y) \cdot \mathit{succ} &\;=\; (+y) \cdot (\times y)\ , \end{aligned}$$

where $+$ and $\times$ have the usual meaning on natural numbers. Function $(\times y)$ is a $(\underline{\mathit{1}} + \mathsf{I})$-homomorphism from $(\mathit{nat}, \mathit{zero} \triangledown \mathit{succ})$ to $(\mathit{nat}, \mathit{zero} \triangledown (+y))$, since

$$\begin{aligned} & (\times y) \cdot \mathit{zero} \triangledown \mathit{succ} \\ = & \quad \text{law (2.25)} \\ & ((\times y) \cdot \mathit{zero}) \triangledown ((\times y) \cdot \mathit{succ}) \\ = & \quad \text{definition of } (\times y) \\ & (\mathit{zero} \cdot \mathit{id}_{\mathit{1}}) \triangledown ((+y) \cdot (\times y)) \\ = & \quad \text{law (2.24)} \\ & \mathit{zero} \triangledown (+y) \cdot \mathit{id}_{\mathit{1}} + (\times y) \\ = & \quad \text{definition of } \underline{\mathit{1}} + \mathsf{I} \\ & \mathit{zero} \triangledown (+y) \cdot (\times y)(\underline{\mathit{1}} + \mathsf{I})\ . \end{aligned}$$

Similarly, function $(+y)$ is a $(\underline{\mathit{1}}+\mathsf{I})$-homomorphism from $(\mathit{nat}, \mathit{zero} \triangledown \mathit{succ})$ to $(\mathit{nat}, y \triangledown \mathit{succ})$.

Categories of functor-algebras

Let $\mathbf{C}$ be a category and F a functor on $\mathbf{C}$. By definition, the *category of* F*-algebras* has as objects the F-algebras (A, ϕ), and as morphisms all F-homomorphisms between F-algebras. Composition in the category of F-algebras is taken from $\mathbf{C}$, and so are the identities. We have to show that the composition of two F-homomorphisms is an F-homomorphism. Let $f : (A, \phi) \to (B, \psi)$ and $g : (B, \psi) \to (C, \chi)$ be F-homomorphisms. We have to prove that for $g \cdot f : (A, \phi) \to (C, \chi)$

$$g \cdot f \cdot \phi \;=\; \chi \cdot (g \cdot f)\mathsf{F} \; .$$

We have

$$\begin{array}{ll} & g \cdot f \cdot \phi \\ = & \quad f : (A, \phi) \to (B, \psi) \text{ is an } \mathsf{F}\text{-homomorphism} \\ & g \cdot \psi \cdot f\mathsf{F} \\ = & \quad g : (B, \psi) \to (C, \chi) \text{ is an } \mathsf{F}\text{-homomorphism} \\ & \chi \cdot g\mathsf{F} \cdot f\mathsf{F} \\ = & \quad \mathsf{F} \text{ is a functor} \\ & \chi \cdot (g \cdot f)\mathsf{F} \; . \end{array}$$

Initiality in the category of F-algebras turns out to be an important notion; the initial object in the category of F-algebras possesses useful calculational properties.

2.4 Data type theory

This section introduces data types as initial algebras. By definition of initiality, an F-algebra is initial in the category of F-algebras if there is a unique F-homomorphism from it to each algebra in the category. All initial algebras in a category are isomorphic, since initial objects are isomorphic. An initial object in the category of F-algebras exists provided F satisfies some conditions. These conditions are quite loose, see for example Chapter 5 of Malcolm's thesis [89]. For all polynomial functors F an initial F-algebra exists. Most of the functors that we will introduce are polynomial functors. The representative we fix for the initial algebra is denoted by $\mu(\mathsf{F})$. Let $(L, in) = \mu(\mathsf{F})$. We call $in : L\mathsf{F} \to L$ the *constructor* of the initial algebra. Since the algebra (L, in) is initial in the category of F-algebras, we have for all objects A and for all $\phi : A\mathsf{F} \to A$ that there exists precisely one $f : L \to A$ such that

$$f \cdot in \;=\; \phi \cdot f\mathsf{F} \; . \tag{2.34}$$

The equation can be seen as an inductive definition of function f that says that the result of f on an element of L equals the result obtained by applying $f\mathsf{F}$ to the constituents of the element and subjecting these to ϕ. We denote the unique solution for f of equation (2.34) by $(\![\phi]\!)_{\mathsf{F}}$. The F-homomorphism $(\![\phi]\!)_{\mathsf{F}}$ is called an F-*catamorphism*. Initiality of (L, in) is fully captured by the law

(2.35) Theorem (Catamorphism Characterisation)

$$f = (\![\phi]\!)_{\mathsf{F}} \quad \equiv \quad f \cdot in = \phi \cdot f\mathsf{F} \ .$$

If the functor F is clear from the context, we omit the subscript F in $(\![\phi]\!)_{\mathsf{F}}$. Catamorphisms play a very important role in program calculation. They satisfy nice calculational properties. Furthermore, a catamorphism is easily translated to a program in some computer language. If the components of ϕ can be implemented, in some language and on some machine, such that they require constant time for their evaluation, then the catamorphism $(\![\phi]\!)$ can be implemented such that the time to evaluate an argument is linear in the number of constructors appearing in the argument. Among the class of algorithms that at least scan their input, catamorphisms with constant-time computable ϕ are optimal. Therefore, given a specification of a problem, we usually strive to construct a catamorphism $(\![\phi]\!)$ with constant-time computable ϕ that is equal to the specification.

The data types *boolean* and *natural number*

We give two examples of data types: the data types *boolean* and *natural number*. Note that names of data types are written in slanted typeface. Other examples are given in the subsequent sections and chapters. The data type *boolean* is an initial object of the category of G-algebras, where G is the functor $\mathsf{G} = \underline{1} + \underline{1}$. Let $(bool, false \triangledown true) = \mu(\mathsf{G})$. Negation, $\neg$, can be written as a catamorphism on this data type. Negation satisfies

$$\begin{aligned} \neg \cdot false &= true \\ \neg \cdot true &= false \ . \end{aligned}$$

We have by Theorem 2.35

$$\neg = (\![\phi]\!)_{\mathsf{G}} \quad \equiv \quad \neg \cdot false \triangledown true = \phi \cdot \neg\mathsf{G} \ .$$

(Note that $\neg\mathsf{G}$ does not mean the negation of G but instead the application of functor G to $\neg$.) The definition of ϕ is calculated as follows.

$$\begin{aligned} & \neg \cdot false \triangledown true \\ = & \quad \text{equation (2.25)} \end{aligned}$$

$$\begin{array}{ll}
 & (\neg \cdot \mathit{false}) \triangledown (\neg \cdot \mathit{true}) \\
= & \quad \text{definition of } \neg \\
 & \mathit{true} \triangledown \mathit{false} \\
= & \quad \text{definition of } \mathsf{G}\text{, definition of } \mathit{id}_{\mathit{1}}\text{, equation (2.24)} \\
 & \mathit{true} \triangledown \mathit{false} \cdot \neg\mathsf{G} \ .
\end{array}$$

It follows that

$$\neg \ = \ (\![\mathit{true} \triangledown \mathit{false}]\!)_{\mathsf{G}} \ .$$

The data type *natural number* is defined as follows. Let $\mathsf{H} = \underline{\mathit{1}} + \mathsf{I}$, then $(\mathit{nat}, \mathit{zero} \triangledown \mathit{succ})$ is an initial H-algebra. Function *even* : $\mathit{nat} \rightarrow \mathit{bool}$ determines whether or not a natural number is even. We want to render it as a catamorphism. Function *even* satisfies

$$\begin{array}{lcl}
\mathit{even} \cdot \mathit{zero} & = & \mathit{true} \\
\mathit{even} \cdot \mathit{succ} & = & \neg \cdot \mathit{even} \ .
\end{array}$$

According to Theorem (2.35) we need a H-algebra $\phi : \mathit{bool}\mathsf{H} \rightarrow \mathit{bool}$ such that

$$\mathit{even} = (\![\phi]\!)_{\mathsf{H}} \quad \equiv \quad \mathit{even} \cdot \mathit{zero} \triangledown \mathit{succ} = \phi \cdot \mathit{even}\mathsf{H} \ .$$

We calculate the definition of ϕ as follows.

$$\begin{array}{ll}
 & \mathit{even} \cdot \mathit{zero} \triangledown \mathit{succ} \\
= & \quad \text{equation (2.25)} \\
 & (\mathit{even} \cdot \mathit{zero}) \triangledown (\mathit{even} \cdot \mathit{succ}) \\
= & \quad \text{properties of } \mathit{even} \\
 & \mathit{true} \triangledown (\neg \cdot \mathit{even}) \\
= & \quad \text{definition of } \mathit{id}_{\mathit{1}}\text{, equation (2.24)} \\
 & \mathit{true} \triangledown \neg \cdot \mathit{id}_{\mathit{1}} + \mathit{even} \\
= & \quad \text{definition of } \mathsf{H} \\
 & \mathit{true} \triangledown \neg \cdot \mathit{even}\mathsf{H} \\
= & \quad \text{define } \phi = \mathit{true} \triangledown \neg \\
 & \phi \cdot \mathit{even}\mathsf{H} \ .
\end{array}$$

It follows that

$$\mathit{even} \ = \ (\![\mathit{true} \triangledown \neg]\!)_{\mathsf{H}} \ .$$

Other catamorphisms have been given in the previous section: since function $(\times y)$ is a $(\underline{\mathit{1}} + \mathsf{I})$-homomorphism from $(\mathit{nat}, \mathit{zero} \triangledown \mathit{succ})$ to $(\mathit{nat}, \mathit{zero} \triangledown (+y))$, it is the following catamorphism

$$(\![\mathit{zero} \triangledown (+y)]\!)_{\mathsf{H}} \ .$$

Properties of catamorphisms

A first consequence of the Catamorphism Characterisation Theorem is the fact that each function with a left inverse (in **Set** each injective function) is a catamorphism.

(2.36) Corollary *If f has a left inverse g, then*

$$f \quad = \quad (\![f \cdot in \cdot g\mathsf{F}]\!) \ .$$

Proof

$$\begin{array}{ll} & f = (\![f \cdot in \cdot g\mathsf{F}]\!) \\ \equiv & \qquad \text{Catamorphism Characterisation Theorem} \\ & f \cdot in = f \cdot in \cdot g\mathsf{F} \cdot f\mathsf{F} \\ \equiv & \qquad \mathsf{F} \text{ is a functor, } g \cdot f = id \\ & true \ . \end{array}$$

□

The following result, which is an easy consequence of the Catamorphism Characterisation Theorem, states that the identity function is a catamorphism.

(2.37) Corollary (Identity Catamorphism)

$$id \quad = \quad (\![in]\!) \ .$$

Another consequence of the Catamorphism Characterisation Theorem is the fact that tupling two catamorphisms yields a catamorphism (see Fokkinga [41]).

(2.38) Corollary (Tupling Catamorphisms)

$$(\![\phi]\!) \vartriangle (\![\psi]\!) \quad = \quad (\![\phi \times \psi \cdot exl\mathsf{F} \vartriangle exr\mathsf{F}]\!) \ .$$

Proof An easy calculation using the Catamorphism Characterisation Theorem.

$$
\begin{array}{ll}
 & (\!|\phi|\!) \vartriangle (\!|\psi|\!) \cdot \mathit{in} \\
= & \\
 & ((\!|\phi|\!) \cdot \mathit{in}) \vartriangle ((\!|\psi|\!) \cdot \mathit{in}) \\
= & \\
 & (\phi \cdot (\!|\phi|\!)\mathsf{F}) \vartriangle (\psi \cdot (\!|\psi|\!)\mathsf{F}) \\
= & \\
 & \phi \times \psi \cdot (\!|\phi|\!)\mathsf{F} \vartriangle (\!|\psi|\!)\mathsf{F} \\
= & \\
 & \phi \times \psi \cdot (\mathit{exl} \cdot (\!|\phi|\!) \vartriangle (\!|\psi|\!))\mathsf{F} \vartriangle (\mathit{exr} \cdot (\!|\phi|\!) \vartriangle (\!|\psi|\!))\mathsf{F} \\
= & \\
 & \phi \times \psi \cdot \mathit{exl}\mathsf{F} \vartriangle \mathit{exr}\mathsf{F} \cdot ((\!|\phi|\!) \vartriangle (\!|\psi|\!))\mathsf{F} \; .
\end{array}
$$

An alternative proof is obtained if the characterisation of split, equation (2.33), is applied. □

The following result is again a corollary of the Catamorphism Characterisation Theorem. It is called the *Unique Extension Property*, see Meertens [98], and it is used to prove equality of two functions that have the same recursive characterisation.

(2.39) Corollary (Unique Extension Property)

$$f \cdot \mathit{in} = \phi \cdot f\mathsf{F} \;\wedge\; g \cdot \mathit{in} = \phi \cdot g\mathsf{F} \quad\Rightarrow\quad f = g \; .$$

Proof From the Catamorphism Characterisation Theorem and the two conjuncts in the premise we have $f = (\!|\phi|\!)$ and $g = (\!|\phi|\!)$, and hence that $f = g$. □

An equality we can prove using the Unique Extension Property is $\mathit{even} = \mathit{odd} \cdot \mathit{succ}$. We have shown that $\mathit{even} = (\!|\mathit{true} \triangledown \neg|\!)_{\mathsf{H}}$ by means of the equation

$$\mathit{even} \cdot \mathit{zero} \triangledown \mathit{succ} = \mathit{true} \triangledown \neg \cdot \mathit{even}\mathsf{H} \; . \qquad (2.40)$$

Similarly, odd can be defined by $(\!|\mathit{false} \triangledown \neg|\!)_{\mathsf{H}}$. It follows that

$$
\begin{array}{ll}
 & \mathit{odd} \cdot \mathit{succ} \cdot \mathit{zero} \triangledown \mathit{succ} \\
= & \quad \text{equation (2.25)} \\
 & (\mathit{odd} \cdot \mathit{succ} \cdot \mathit{zero}) \triangledown (\mathit{odd} \cdot \mathit{succ} \cdot \mathit{succ}) \\
= & \quad \text{evaluation of } \mathit{odd} \\
 & \mathit{true} \triangledown (\neg \cdot \mathit{odd} \cdot \mathit{succ}) \\
= & \quad \text{definition of } \mathsf{H} \\
 & \mathit{true} \triangledown \neg \cdot (\mathit{odd} \cdot \mathit{succ})\mathsf{H} \; .
\end{array}
$$

From this calculation, equation (2.40), and the Unique Extension Property we have that $\mathit{even} = \mathit{odd} \cdot \mathit{succ}$.

Fusion

One of the most important consequences of the Catamorphism Characterisation Theorem is so-called *Fusion*. Fusion gives the condition that has to be satisfied in order to 'fuse' the composition of a function with a catamorphism into a catamorphism. The left-hand side of the equivalence is a functional equality that has to be satisfied on just a subset of the source of the functions involved. The following corollary uses the after construct, which is defined by

$$f = g \text{ after } h \quad \equiv \quad f \cdot h = g \cdot h \ .$$

(2.41) Corollary (Fusion)

$$f \cdot \phi = \psi \cdot f\mathsf{F} \textit{ after } (\!|\phi|\!)\mathsf{F} \quad \equiv \quad f \cdot (\!|\phi|\!) = (\!|\psi|\!) \ .$$

Proof

$$
\begin{array}{cl}
 & f \cdot (\!|\phi|\!) = (\!|\psi|\!) \\
\equiv & \quad \text{Catamorphism Characterisation} \\
 & f \cdot (\!|\phi|\!) \cdot \mathit{in} = \psi \cdot (f \cdot (\!|\phi|\!))\mathsf{F} \\
\equiv & \quad \text{Catamorphism Characterisation} \\
 & f \cdot \phi \cdot (\!|\phi|\!)\mathsf{F} = \psi \cdot (f \cdot (\!|\phi|\!))\mathsf{F} \\
\equiv & \quad \mathsf{F} \text{ is a functor} \\
 & f \cdot \phi \cdot (\!|\phi|\!)\mathsf{F} = \psi \cdot f\mathsf{F} \cdot (\!|\phi|\!)\mathsf{F} \\
\equiv & \quad \text{definition of after} \\
 & f \cdot \phi = \psi \cdot f\mathsf{F} \text{ after } (\!|\phi|\!)\mathsf{F} \ .
\end{array}
$$

□

Fusion, a name first used by Fokkinga and Meijer [48], is called Promotion by Backhouse [6], Malcolm [89] and Meertens [98]. The term promotion has been coined by Darlington [34] and Bird [14] to describe specific instances of Fusion. The verb fuse reflects better what is actually happening when applying the corollary from left to right. Occasionally, the corollary is applied in the other direction. Fusing a function and a catamorphism into a catamorphism may cause a dramatic increase in efficiency of the corresponding implemented programs. The notion of fusion is ubiquitous in our proofs; it is our main means for the construction of efficient algorithms. Usually, Fusion is formulated as the following implication.

$$f \cdot \phi = \psi \cdot f\mathsf{F} \;\;\Rightarrow\;\; f \cdot (\![\phi]\!) = (\![\psi]\!) \ ,$$

where the premise of the implication is a total equality. We will encounter many situations in which we want to apply Fusion and in which it is only possible to prove the partial equality. For that reason we included the domain restriction in our formulation of Fusion.

As a first example we show that negating a boolean value twice has the same effect as the identity function:

$$\neg \cdot \neg \quad = \quad id \ .$$

This equality can be rewritten as follows

$$\neg \cdot (\![\mathit{true} \triangledown \mathit{false}]\!)_{\mathsf{G}} \quad = \quad (\![\mathit{false} \triangledown \mathit{true}]\!)_{\mathsf{G}} \ .$$

We obtain $\neg \cdot \neg = id$ by applying Fusion, provided

$$\neg \cdot \mathit{true} \triangledown \mathit{false} \quad = \quad \mathit{false} \triangledown \mathit{true} \cdot \neg\mathsf{G} \ .$$

The verification of this equality is left to the reader.

Another example of an application of Fusion is

$$\neg \cdot \mathit{even} \quad = \quad \mathit{odd} \ .$$

Since $\mathit{even} = (\![\mathit{true} \triangledown \neg]\!)_{\mathsf{H}}$ and $\mathit{odd} = (\![\mathit{false} \triangledown \neg]\!)_{\mathsf{H}}$, we can apply Fusion if we can prove that

$$\neg \cdot \mathit{true} \triangledown \neg \quad = \quad \mathit{false} \triangledown \neg \cdot \neg\mathsf{H} \ .$$

This equality is easily verified.

The function *out*

The constructor $in : L\mathsf{F} \rightarrow L$ of a data type $\mu(\mathsf{F})$ has an inverse, namely $(\![in\mathsf{F}]\!) : L \rightarrow L\mathsf{F}$. This inverse is called *out*:

$$out \quad = \quad (\![in\mathsf{F}]\!) \ . \tag{2.42}$$

Function *out* destructs its argument, and is therefore called a *destructor*. For example, on the data type *natural number* we have $out\ succ\ n = inr\ n$, and $out\ zero = inl\ o$.

Paramorphisms

Consider the factorial function *fac*. A standard definition of *fac* reads

$$\begin{array}{lcl} \mathit{fac}\,0 & = & 1 \\ \mathit{fac}\,(n{+}1) & = & \mathit{fac}\,n \times (n{+}1)\ , \end{array}$$

where $+$ and $\times$ have their usual meaning on natural numbers. Defining *fac* as a catamorphism that can be implemented as an efficient program, defined on the data type *natural number* without using fac^{-1} and Corollary 2.36 is difficult. This is caused by the second occurrence of n in the right-hand side of the last equation. A function f that satisfies the recursive pattern

$$\begin{array}{lcl} f\,0 & = & e \\ f\,(n{+}1) & = & (f\,n) \otimes n\ , \end{array}$$

as does *fac*, is called a *paramorphism* on the data type *natural number*. Paramorphisms correspond to primitive recursive functions. Paramorphisms have been introduced by Meertens [98] as a kind of generalisation of catamorphisms. They have calculational properties very similar to those of catamorphisms. The following definition of paramorphisms is a generalisation to arbitrary data types of the above definition of paramorphisms on the data type *natural number*. Given a function $\phi : (A \times L)\mathsf{F} \to A$, the paramorphism of type $L \to A$ is written $[\![\phi]\!]_{\mathsf{F}}$. It is defined by

$$[\![\phi]\!]_{\mathsf{F}} \quad = \quad \mathit{exl} \cdot (\!|\phi \vartriangle (\mathit{in} \cdot \mathit{exr}\mathsf{F})|\!)\ . \tag{2.43}$$

If the subscript F of $[\![\phi]\!]_{\mathsf{F}}$ is clear from the context it is omitted. A paramorphism is characterised by

(2.44) Theorem (Paramorphism Characterisation)

$$f = [\![\phi]\!] \quad \equiv \quad f \cdot \mathit{in} = \phi \cdot (f \vartriangle \mathit{id})\mathsf{F}\ .$$

A proof of this theorem and the properties of paramorphisms mentioned below can be found in Meertens [98]. As an example we show that $\mathit{fac} = [\![\underline{1} \triangledown \otimes]\!]_{\mathsf{H}}$, where operator $\otimes$ is defined by $m \otimes n = m \times (\mathit{succ}\,n)$.

$$\begin{array}{ll} & \mathit{fac} = [\![\underline{1} \triangledown \otimes]\!]_{\mathsf{H}} \\ \equiv & \qquad \text{Paramorphism Characterisation} \\ & \mathit{fac} \cdot \mathit{in} = \underline{1} \triangledown \otimes \cdot (\mathit{fac} \vartriangle \mathit{id})\mathsf{H} \\ \equiv & \qquad \text{definition of } \mathsf{H} \text{ and } \mathit{in} \end{array}$$

$$\begin{array}{ll}
& \mathit{fac} \cdot \mathit{zero} \triangledown \mathit{succ} = \underline{1} \triangledown \otimes \cdot \mathit{id}_1 + (\mathit{fac} \vartriangle \mathit{id}) \\
\equiv & \quad \text{equations (2.25) and (2.24)} \\
& (\mathit{fac} \cdot \mathit{zero}) \triangledown (\mathit{fac} \cdot \mathit{succ}) = \underline{1} \triangledown (\otimes \cdot \mathit{fac} \vartriangle \mathit{id}) \\
\equiv & \quad \text{definition of } \mathit{fac} \text{ and } \otimes \\
& \mathit{true}\ .
\end{array}$$

Each function defined on the carrier of an initial algebra is a paramorphism:

$$f \quad = \quad [\![f \cdot \mathit{in} \cdot \mathit{exr}\mathsf{F}]\!]\ ,$$

for all functions f. This is an immediate consequence of the Paramorphism Characterisation Theorem. Fusion reads as follows for paramorphisms.

(2.45) Corollary (Parafusion)

$$\phi \cdot (f \times \mathit{id})\mathsf{F} = f \cdot \psi \ \mathit{after}\ ([\![\psi]\!] \vartriangle \mathit{id})\mathsf{F} \quad \equiv \quad f \cdot [\![\psi]\!] = [\![\phi]\!]\ .$$

Mutumorphisms

Various recursive patterns have been investigated after the introduction of paramorphisms, culminating into the 'most general' one satisfied by functions that are called mutumorphisms, see Fokkinga [41]. We briefly review the definition and main theorem of [41]. By definition, f is F-*catamorphic modulo* g if for some ϕ

$$f \cdot \mathit{in} \quad = \quad \phi \cdot (f \vartriangle g)\mathsf{F}\ . \tag{2.46}$$

If g is also F-catamorphic modulo f, we call functions f and g *mutumorphisms*, since their essential property is that they are defined mutually recursive by induction on the structure of the data of the data type. The following theorem shows that tupling mutumorphisms yields a catamorphism.

(2.47) Theorem (Mutumorphisms) *Suppose f and g are catamorphic modulo each other: $f \cdot \mathit{in} = \phi \cdot (f \vartriangle g)\mathsf{F}$, and $g \cdot \mathit{in} = \psi \cdot (g \vartriangle f)\mathsf{F}$. Then $f \vartriangle g$ is an F-catamorphism:*

$$\begin{array}{lcl}
f \vartriangle g & = & (\![\phi \vartriangle (\psi \cdot \mathit{swap}\mathsf{F})]\!) \\
f & = & \mathit{exl} \cdot (\![\phi \vartriangle (\psi \cdot \mathit{swap}\mathsf{F})]\!) \\
g & = & \mathit{exr} \cdot (\![\phi \vartriangle (\psi \cdot \mathit{swap}\mathsf{F})]\!)\ .
\end{array}$$

Proof The last two equalities are immediate consequences of the first equality and the characterisation of split, equation (2.33). The first equality is proved with an easy calculation, using Catamorphism Characterisation in the first step.

$$
\begin{aligned}
& f \vartriangle g = (\!|\phi \vartriangle (\psi \cdot \mathit{swap}\mathsf{F})|\!) \\
\equiv \\
& f \vartriangle g \cdot \mathit{in} = \phi \vartriangle (\psi \cdot \mathit{swap}\mathsf{F}) \cdot (f \vartriangle g)\mathsf{F} \\
\equiv \\
& (f \cdot \mathit{in} = \phi \cdot (f \vartriangle g)\mathsf{F}) \;\wedge\; (g \cdot \mathit{in} = \psi \cdot \mathit{swap}\mathsf{F} \cdot (f \vartriangle g)\mathsf{F}) \\
\equiv \\
& (f \cdot \mathit{in} = \phi \cdot (f \vartriangle g)\mathsf{F}) \;\wedge\; (g \cdot \mathit{in} = \psi \cdot (g \vartriangle f)\mathsf{F}) \\
\equiv \\
& \mathit{true}\ .
\end{aligned}
$$

□

This theorem is often used as follows. Suppose we are given a function f, and we want to construct a catamorphism for f. Often, it is difficult or impossible to express f as a catamorphism, but it is easy to show that f is catamorphic modulo some function g. In this case, we try to show that function g is catamorphic modulo f in order to obtain a catamorphism for the tuple of functions $f \vartriangle g$.

Note that a catamorphism f is catamorphic modulo any function g, since

$$f \cdot \mathit{in} = \phi \cdot f\mathsf{F} \;\equiv\; f \cdot \mathit{in} = \phi \cdot \mathit{exl}\mathsf{F} \cdot (f \vartriangle g)\mathsf{F}\ . \tag{2.48}$$

2.5 Map functors

Many functional programming languages provide the user with a higher-order function called map defined on the data type *list*. This section defines the class of so-called map functors.

The data type *snoc-list*

A data type may be parameterised with another data type. For example, we have the data type list over *nat*, and also the data type list over *boolean*. Given an arbitrary type A, we define the data type *snoc-list* over A. Let functor K_A be defined by

$$\mathsf{K}_A \;=\; \underline{1} + (\mathsf{I} \times \underline{A})\ . \tag{2.49}$$

Define $(A\star, \Box \triangledown \prec\!\!\!\!\!\dashv) = \mu(\mathsf{K}_A)$, where $\Box : 1 \to A\star$ and $\prec\!\!\!\!\!\dashv : A\star \times A \to A\star$. Function $\Box$ constructs the empty list, and we define $\Box\, o = [\,]$, and operator $\prec\!\!\!\!\!\dashv$ appends an element to a list. For example, $[\,]\prec\!\!\!\!\!\dashv 1 \prec\!\!\!\!\!\dashv 8 \prec\!\!\!\!\!\dashv 2$, which is written $[1, 8, 2]$, is an element of $\mathit{nat}\star$. The lists in the carrier of this algebra are constructed by appending elements at the right end, and therefore this data type is called *snoc-list*, in contrast with the data type *cons-list* (defined in Section 2.7), the carrier of which contains lists that are constructed by prepending elements at the left end. For $f : A \to B$ there is a catamorphism f*-map*, denoted $f\star : A\star \to B\star$, which applies the function f to each element in the list. We define $\star$ on functions such that it is a functor, that is, $\star$ satisfies $id_A\star = id_{A\star}$, and $(f \cdot g)\star = f\star \cdot g\star$. This definition of $\star$ is given after the following general discussion on map functors.

Map functors in general

Let L_A be a functor that depends functorially on A, that is, L_A can be written as the section $(\dagger\underline{A})$ for some binary functor $\dagger$. For example, functor K_A defined above depends functorially upon A since it can be defined as $(\ddagger\underline{A})$, where functor $\ddagger$ is defined by $x \ddagger y = \underline{\mathit{1}} + x \times y$. We say that the data type defined as an initial object in the category of $(\dagger\underline{A})$-algebras is *parameterised with* type A. For every data type that is parameterised with another data type we can define a map-functor like $\star$ on the data type *snoc-list*. The map-functor corresponding to functor $(\dagger\underline{A})$ is denoted by $*$; it is defined on objects as the carrier from the initial object of the category of $(\dagger\underline{A})$-algebras, that is, $(A*, in_{(\dagger\underline{A})}) = \mu(\dagger\underline{A})$. $*$ is defined on morphisms in such a way that $*$ becomes a functor. Since for $f : A \to B$ we want $f* : A* \to B*$ to be a catamorphism, the candidate for $f*$ is a catamorphism $(\!|\phi|\!)_{(\dagger\underline{A})}$, where $\phi : B* \dagger A \to B*$. Since $in_{(\dagger\underline{B})} : B* \dagger B \to B*$ and $\dagger$ is a functor, we have $in_{(\dagger\underline{B})} \cdot id_{B*} \dagger f : B* \dagger A \to B*$. We define $f*$ by

$$f* \;=\; (\!|in_{(\dagger\underline{B})} \cdot id_{B*} \dagger f|\!)_{(\dagger\underline{A})} \;. \tag{2.50}$$

For example, if we instantiate the functor $(\dagger\underline{A})$ in the definition of map with the functor K_A for the data type *snoc-list*, we obtain $f\star = (\!|\Box \triangledown \dashv\!\!< \cdot\, id_{\mathit{1}} + (id_{B\star} \times f)|\!)_{\mathsf{K}_A}$, or, more conventionally,

$$\begin{aligned} f\star \cdot \Box &= \Box \\ f\star \cdot \dashv\!\!< &= \dashv\!\!< \cdot f\star \times f \;. \end{aligned}$$

It is easily verified that $id_A* = id_{A*}$. *Map-distributivity*,

$$(f \cdot g)* \;=\; f* \cdot g* \;, \tag{2.51}$$

is an easy consequence of the following *factorisation* corollary, see Verwer [134], which itself is a direct application of Fusion. A catamorphism defined on an initial data type that is parameterised with another data type can be factored into a catamorphism and a map. For a class of data types we have that all catamorphisms defined on these data types can be split into a special kind of catamorphism, called reduction, and a map. Examples of such data types are given in the following section.

(2.52) Corollary (Factorisation)

$$(\!|\phi \cdot id \dagger f|\!) \;=\; (\!|\phi|\!) \cdot f* \;.$$

The proof of this corollary is omitted; it is a direct application of Fusion. Map-distributivity is proved as follows.

$$\begin{array}{ll}
 & (f \cdot g)* \\
= & \quad \text{definition of map (2.50)} \\
 & (\!|in \cdot id_{B*} \dagger (f \cdot g)|\!) \\
= & \quad \dagger \text{ is a functor} \\
 & (\!|in \cdot id_{B*} \dagger f \cdot id_{B*} \dagger g|\!) \\
= & \quad \text{Factorisation} \\
 & (\!|in \cdot id_{B*} \dagger f|\!) \cdot g* \\
= & \quad \text{definition of map (2.50)} \\
 & f* \cdot g* \;.
\end{array}$$

Another property of map is that it 'almost' distributes, so to speak, over split.

(2.53) Law

$$f* \vartriangle g* \quad = \quad exl* \vartriangle exr* \cdot (f \vartriangle g)* \;.$$

Proof

$$\begin{array}{ll}
 & exl* \vartriangle exr* \cdot (f \vartriangle g)* \\
= & \quad (2.13) \\
 & (exl* \cdot (f \vartriangle g)*) \vartriangle (exr* \cdot (f \vartriangle g)*) \\
= & \quad \text{map-distributivity, laws (2.16) and (2.17)} \\
 & f* \vartriangle g* \;.
\end{array}$$

□

2.6 The Boom-hierarchy of data types

Most of the problems for which we want to construct algorithms require the processing of binary trees, lists, bags or sets. To be able to apply the calculational properties given in the previous section, we want to describe the four data types *binary tree*, *list*, *bag*, and *set* as initial objects in a category of functor-algebras. For reasons that will become clear later these four data types form a nice hierarchy. Meertens [95] attributes this observation to H.J. Boom, and therefore this hierarchy is called the Boom-hierarchy.

Consider the functor J_A defined by

$$\mathsf{J}_A \;=\; \underline{1} + \underline{A} + \mathsf{II}\ .$$

An initial object in the category of J_A-algebras is denoted by the algebra $(B,\, \mathit{empty} \triangledown \mathit{single} \triangledown \mathit{join})$. If *empty* were the unit of operator *join*, this algebra would be the usual algebra of binary trees. To obtain the data type binary tree as an initial algebra, we want to be able to impose a law on an algebra. Imposing a law on an algebra means identifying equivalent elements and to consider the quotient algebras.

Transformer and law

Fokkinga [44] introduces a way to describe laws for an algebra without having to introduce a signature. This is a next step in developing algebra theory without using signatures, the first step being the definition of an algebra by means of a functor. We briefly repeat restricted versions of Fokkinga's definitions and theorems.

A *transformer* of type $(A\mathsf{F} \to A) \to (A\mathsf{G} \to A)$ is a mapping T that maps each F-algebra $\phi : A\mathsf{F} \to A$ to a G-algebra $T\,\phi : A\mathsf{G} \to A$ such that for all A, B, $f : A \to B$, $\psi : B\mathsf{F} \to B$

$$f \cdot \phi = \psi \cdot f\mathsf{F} \quad\Rightarrow\quad f \cdot T\,\phi = T\,\psi \cdot f\mathsf{G}\ . \tag{2.54}$$

A *law* is a pair of transformers (T, T') of the same type. For a law $E = (T, T')$ we say that E holds for algebra ϕ if $T\,\phi = T'\,\phi$. An initial object in the category of F-algebras satisfying law E exists provided functors F and G (from the type of transformers T and T'), 'preserve epis'. We do not explain this condition. It is satisfied for the functors that appear in the laws we introduce.

The data type *binary tree*

Let $(C, \epsilon \triangledown f \triangledown \oplus)$ be a J_A-algebra, with $\epsilon : \mathit{1} \to C$, $f : A \to C$, and $\oplus : C \times C \to C$. For the data type *binary tree*, the first element in the Boom-hierarchy, we take an initial object in the category of J_A-algebras satisfying the following law. Let e be the element constructed by function ϵ, $e = \epsilon\, o$. We want e to be the unit of $\oplus$, so for all x in C

$$e \oplus x = x = x \oplus e\ . \tag{2.55}$$

This law is the conjunction of two laws: $e \oplus x = x$ and $x = x \oplus e$. We give the transformers corresponding to the first law. Let $T,\ T' : (D\mathsf{J}_A \to D) \to (D \to D)$ be defined by

$$\begin{array}{lcl} T\,(\epsilon \triangledown f \triangledown \oplus) & = & ((\epsilon\, o)\oplus) \\ T'\,(\epsilon \triangledown f \triangledown \oplus) & = & \mathit{id}\ . \end{array}$$

We verify implication (2.54) for T; T' satisfies implication (2.54) trivially. For all X, Y, $f : X \to Y$, $\phi : X\mathsf{J}_A \to X$ and $\psi : Y\mathsf{J}_A \to Y$ we have to show

$$f \cdot \phi = \psi \cdot f\mathsf{J}_A \quad \Rightarrow \quad f \cdot T\,\phi = T\,\psi \cdot f \ . \tag{2.56}$$

Suppose $\phi = u \triangledown g \triangledown \otimes$ and $\psi = i \triangledown h \triangledown \odot$. It follows from the premise of implication (2.56) that $f \cdot u = i$, and $f \cdot \otimes = \odot \cdot f \times f$. The conclusion of implication (2.56) is calculated as follows.

$$\begin{array}{ll}
 & f \cdot T\,\phi \\
= & \quad \text{definition of } T \\
 & f \cdot ((u\ o)\otimes) \\
= & \quad \text{premise} \\
 & ((f\ u\ o)\odot) \cdot f \\
= & \quad \text{premise} \\
 & ((i\ o)\odot) \cdot f \\
= & \quad \text{definition of } T \\
 & T\,\psi \cdot f \ .
\end{array}$$

The data type *binary tree* —an initial object in the category of J_A-algebras satisfying law (T, T') and law (S, S'), where law (S, S') corresponds to: for all x, $x \oplus e = x$— is denoted by $(A\beta, \diamond \triangledown \alpha \triangledown \bowtie)$.

The data type *join-list*

The second element in the Boom-hierarchy, the data type *join-list*, is an initial object in the category of J_A-algebras $(C, \epsilon \triangledown f \triangledown \oplus)$ satisfying the laws (S, S'), $(T,\, T')$, and the law expressing that operator $\oplus$ is associative: for all x,y and z in C

$$(x \oplus y) \oplus z \quad = \quad x \oplus (y \oplus z) \ .$$

We do not give the transformers corresponding to this law, they can be found in Fokkinga [44]. The data type join-list is denoted by $(A*, \square \triangledown \tau \triangledown +\!\!+)$. The list with consecutive elements 1, 2 and 8, formally $(\tau\,1) +\!\!+ (\tau\,2) +\!\!+ (\tau\,8)$, is written as $[1, 2, 8]$. This list is an element of *nat*$*$. The function $\square$ constructs the empty list, and we define $\square\ o = [\,]$. Note that $\square$ and $[\,]$ are overloaded: they respectively denote the empty list constructor and the empty list on both the data type *join-list* and the data type *snoc-list* (and they will be given the same meaning on the data type *cons-list* later).

The data type *bag*

The data type *bag*, also known as multiset, is the third element in the Boom-hierarchy. A bag is a set with possibly multiple occurrences of equal elements, or, equivalently, lists with no order imposed on the elements. *Bag* is an initial object in the category of J_A-algebras $(C, e \triangledown f \triangledown \oplus)$ that satisfy the laws (S, S'), (T, T'), the law expressing that operator $\oplus$ is associative, and the law expressing that operator $\oplus$ is commutative: for all x and y in C

$$x \oplus y \quad = \quad y \oplus x \ .$$

The data type *bag* is denoted by $(A\varpi, \langle\,\rangle \triangledown \rho \triangledown \uplus)$. A bag with elements 1, 2 and 2 is written as $\langle 1, 2, 2\rangle$, $\langle 2, 1, 2\rangle$, or $\langle 2, 2, 1\rangle$.

The data type *set*

Finally, the last element in the Boom-hierarchy is the data type *set*. *Set* is an initial object in the category of J_A-algebras $(C, e \triangledown f \triangledown \oplus)$ that satisfy the laws (S, S'), (T, T'), the laws expressing that operator $\oplus$ is associative and commutative, and the law expressing that operator $\oplus$ is idempotent: for all x in C

$$x \oplus x \quad = \quad x \ .$$

The data type *set* is denoted by $(A\wr, \{\,\} \triangledown \sigma \triangledown \cup)$. The set with elements 1 and 2 is written as $\{1, 2\}$, or $\{2, 1\}$.

Binary structures

The data types *binary tree*, *join-list*, *bag*, and *set* are called binary structures. The laws satisfied by the binary operator of the algebras in the category of J_A-algebras determine the different data types. If ϵ constructs the unit of operator $\oplus$, and one of the following statements holds:

- $\oplus$ satisfies no laws
- $\oplus$ is associative
- $\oplus$ is associative and commutative
- $\oplus$ is associative, commutative, and idempotent,

we say that 'Laws($\oplus$,ϵ)' holds. We state here the definitions and theorems that apply in the generic case to binary structures. We use 'Laws($\oplus$,ϵ)' to refer to the varying laws of the binary operator of binary structures. For the moment, the algebra $(A*, \square \triangledown \tau \triangledown +\!\!\!+)$ denotes an arbitrary binary structure.

Catamorphisms on binary structures

According to the Catamorphism Characterisation Theorem we have

$$h = (\!|\phi|\!)_{\mathsf{J}_A} \quad \equiv \quad h \cdot \square \triangledown \tau \triangledown +\!\!+ = \phi \cdot h\mathsf{J}_A \ ,$$

where, since ϕ is a J_A-algebra, $\phi = \epsilon \triangledown f \triangledown \oplus$ where operator $\oplus$ and function ϵ satisfy Laws($\oplus$,ϵ). So h is the unique function satisfying

$$\begin{aligned} h \cdot \square &= \epsilon \\ h \cdot \tau &= f \\ h \cdot +\!\!+ &= \oplus \cdot h\Vert \ . \end{aligned}$$

Since it is more common to talk about elements than to talk about the construction of elements (compare 'function $\square$ constructs the unit of operator $+\!\!+$' with '$[\,]$ is the unit of operator $+\!\!+$'), replace function ϵ in the definition of catamorphism by the element e it constructs. Function h is now the unique solution of

$$\begin{aligned} h\,[\,] &= e \\ h \cdot \tau &= f \\ h \cdot +\!\!+ &= \oplus \cdot h\Vert \ . \end{aligned}$$

Map and reduction on binary structures

A catamorphism defined on a binary structure can be written as, or factored into, the composition of a reduction and a map. The map-functor induced by functor J_A is denoted by $*$. Definition (2.50) of map may be rephrased for a binary structure as follows.

$$\begin{aligned} f{*}\,[\,] &= [\,] \\ f{*} \cdot \tau &= \tau \cdot f \\ f{*} \cdot +\!\!+ &= +\!\!+ \cdot f{*}\Vert \ . \end{aligned}$$

For example, $f{*}\,[a,b,c] = [f\ a, f\ b, f\ c]$. The idea is to replace each occurrence of constructor τ by function $\tau{\cdot}f$. The value of applying the *reduction operator* $/$ to an operator $\oplus$ that satisfies Laws($\oplus$,e) for some value e, and a binary structure can be obtained by placing $\oplus$ in between the elements of the binary structure, so, if $\oplus : A \times A \to A$, then $\oplus/ : A{*} \to A$ is the catamorphism $(\!|e \triangledown id \triangledown \oplus|\!)_{\mathsf{J}_A}$, or, equivalently

$$\begin{aligned} \oplus/\,[\,] &= e \\ \oplus/ \cdot \tau &= id \\ \oplus/ \cdot +\!\!+ &= \oplus \cdot \oplus/\Vert \ . \end{aligned}$$

For example, $\oplus/\,[a,b,c] = a \oplus b \oplus c$. The idea is to replace each occurrence of constructor $+\!\!+$ by operator $\oplus$. We instantiate some results described in Section 2.4 for data types in general for binary structures. The following corollary, called the Homomorphism Lemma by Meertens [95], expresses the fact that a catamorphism on a binary structure can be written as the composition of a reduction and a map. It is an instantiation of Factorisation, Corollary 2.52.

(2.57) Corollary (Factorisation of binary structure Catamorphisms)

$$(\![e \triangledown f \triangledown \oplus]\!) = \oplus/ \cdot f* \ .$$

From now on we use the reduction-map notation for a catamorphism defined on a binary structure. An example of a catamorphism on the data type *join-list* is the coercion function *bagify* : $A* \to A\varpi$, which turns a list into a bag, defined by

$$\mathit{bagify} \quad = \quad \uplus/ \cdot \rho* \ . \tag{2.58}$$

Tupling catamorphisms on binary structures

According to the Tupling Catamorphisms Corollary, Corollary 2.38, tupling catamorphisms on binary structures yields a catamorphism on binary structures.

$$(\oplus/ \cdot f*) \vartriangle (\otimes/ \cdot g*) \quad = \quad \odot/ \cdot (f \vartriangle g)* \ , \tag{2.59}$$

where operator $\odot$ is defined by

$$(x, y) \odot (u, v) \quad = \quad (x \oplus u, y \otimes v) \ . \tag{2.60}$$

Fusion on binary structures

The instantiation of Fusion, Corollary 2.41, on a binary structure using the alternative notation for a catamorphism provided by the Factorisation of binary structure Catamorphisms Corollary, reads as follows. First, we rephrase the condition of the corollary.

(2.61) Definition (($\oplus, \otimes$)-fusability) *Let $\oplus : A \times A \to A$ be an operator that satisfies Laws($\oplus$,e) for some value e, $\otimes : B \times B \to B$ be an operator that satisfies Laws($\otimes$,u) for some value u, and let $f : A \to B$ be a function. Then, f is $(\oplus, \otimes)$-fusable after g if for all x, y in the image of g*

$$\begin{array}{lcl} f\, e & = & u \\ f\,(x \oplus y) & = & (f\, x) \otimes (f\, y) \ . \end{array}$$

If no function g (after which f is $(\oplus,\otimes)$-fusable) is specified, assume $g = id$. Function f is $(\oplus,\otimes)$-fusable if and only if it is a J-homomorphism from $(A,\, e \triangledown h \triangledown \oplus)$ to $(B,\, u \triangledown (f\cdot h) \triangledown \otimes)$. For example, each function $f{*}$ is $(+\!\!\!+, +\!\!\!+)$-fusable, and each function $\oplus/$ is $(+\!\!\!+, \oplus)$-fusable. The following result is a corollary of Fusion, Corollary 2.41.

(2.62) Corollary (Fusion on binary structures) *A function f is $(\oplus,\otimes)$-fusable after $\oplus/ \cdot g{*}$ if and only if*

$$f \cdot \oplus/ \cdot g{*} \quad = \quad \otimes/ \cdot (f \cdot g){*}\ .$$

Two properties that follow from Fusion on *bag* are

$$f{*} \cdot \mathit{bagify} \quad = \quad \mathit{bagify} \cdot f{*} \tag{2.63}$$

$$\oplus/ \cdot \mathit{bagify} \quad = \quad \oplus/\ . \tag{2.64}$$

Note that the map and reduction in the left-hand sides of the above equations are defined on the data type *bag*, whereas their right-hand side occurrences are defined on the data type *join-list*. The following corollary of Fusion on binary structures is often used to split a catamorphism on a binary structure over *snoc-list* into the composition of a function and a catamorphism on the binary structure.

(2.65) Corollary *Suppose f satisfies $f \cdot (\mathbin{-\!\!<} a) = (\oplus a) \cdot f$, where operator $\oplus$ is such that section $(\oplus a)$ is $(\otimes,\otimes)$-fusable after $\otimes/ \cdot f{*}$. Then*

$$\otimes/ \cdot f{*} \cdot (\mathbin{-\!\!<} a){*} \quad = \quad (\oplus a) \cdot \otimes/ \cdot f{*}\ .$$

Proof

$$\begin{aligned}
& \otimes/ \cdot f{*} \cdot (\mathbin{-\!\!<} a){*} \\
= \quad & \quad \text{map-distributivity, assumption} \\
& \otimes/ \cdot ((\oplus a) \cdot f){*} \\
= \quad & \quad \text{map-distributivity, Fusion on binary structures} \\
& (\oplus a) \cdot \otimes/ \cdot f{*}\ .
\end{aligned}$$

□

Filter

A widely used catamorphism on binary structures is the *filter operator* $\triangleleft$, which takes a predicate (i.e. a function with *bool* as its target type) and a binary structure, and retains only the elements satisfying the predicate in the binary structure. Define for predicate $p : A \to \mathit{bool}$ function $p\S : A \to A*$ by

$$p\S\, a \;=\; \begin{cases} \tau\, a & \text{if } p\, a \\ [\,] & \text{otherwise} \end{cases}, \tag{2.66}$$

then $p\triangleleft : A* \to A*$ is defined by

$$p\triangleleft \;=\; +\!\!+/ \cdot p\S* \,. \tag{2.67}$$

For example, $\mathit{odd}\triangleleft[1,2,8,6,5] = [1,5]$. An expression of the form $p?_\otimes : A \to A$, called a *guard* (see Fokkinga [40]), is defined by

$$p?_\otimes \;=\; \otimes/ \cdot p\S \,. \tag{2.68}$$

Note that $p?_\otimes\, \nu_\otimes = \nu_\otimes$ for all predicates p. We have

$$\begin{array}{ll}
 & \otimes/ \cdot p\triangleleft \\
= & \quad \text{definition of filter (2.67)} \\
 & \otimes/ \cdot +\!\!+/ \cdot p\S* \\
= & \quad \text{Fusion, } \otimes/ \text{ is } (+\!\!+, \otimes)\text{-fusable} \\
 & \otimes/ \cdot \otimes/* \cdot p\S* \\
= & \quad \text{map-distributivity} \\
 & \otimes/ \cdot (\otimes/ \cdot p\S)* \\
= & \quad \text{definition of guard} \\
 & \otimes/ \cdot p?_\otimes* \,.
\end{array}$$

The equation derived,

$$\otimes/ \cdot p\triangleleft \;=\; \otimes/ \cdot p?_\otimes* \,, \tag{2.69}$$

shows that a filter followed by a reduction is a catamorphism. This equation is applied frequently in the sequel. The proof of the following implication is easy and omitted.

$$f\,\nu_\oplus = \nu_\otimes \;\Rightarrow\; \otimes/ \cdot f* \cdot p?_\oplus* = \otimes/ \cdot f* \cdot p\triangleleft \,. \tag{2.70}$$

Predicate $\mathit{all}\, p$ is defined by

$$\mathit{all}\, p \;=\; \wedge/ \cdot p* \,, \tag{2.71}$$

so $(\mathit{all}\ \mathit{odd})\triangleleft[[1],[1,2],[3,5]] \;=\; [[1],[3,5]]$. Define the boolean operator $\wedge$ on predicates $p : A \to \mathit{bool}$ and $q : A \to \mathit{bool}$ by

$$(p \;\wedge\; q)\,a \;=\; p\,a \;\wedge\; q\,a\ .$$

Operator $\wedge$ is 'lifted' by means of this definition. For all predicates p and q we have

$$p\triangleleft \cdot q\triangleleft \;=\; (p \wedge q)\triangleleft\ . \tag{2.72}$$

For all predicates p and functions f we have the *map-filter swap*

$$p\triangleleft \cdot f* \;=\; f* \cdot (p \cdot f)\triangleleft\ . \tag{2.73}$$

Fusion on binary structures revisited

We make a further remark on Fusion on binary structures. Consider the expression $f \cdot \oplus/ \cdot g* \cdot h$, where h is a function with target $A*$. To apply Fusion on binary structures to obtain $\otimes/ \cdot f* \cdot g* \cdot h$, it suffices to show that f is $(\oplus,\otimes)$-fusable after $\oplus/ \cdot g* \cdot (\in h\,z)\triangleleft$ for all binary structures z, provided $g\,\nu_\oplus = \nu_\oplus$, so when we have to prove that

$$f\,(x \oplus y) = f\,x \otimes f\,y\ ,$$

we may assume that

$$\begin{aligned} x &= \oplus/\ g*\,(\in h\,z)\triangleleft v \\ y &= \oplus/\ g*\,(\in h\,z)\triangleleft w\ , \end{aligned}$$

for some binary structures v and w. In many applications $\oplus$ is a selector and g is a guard, in which case we know that both x and y are elements of $h\,z$, for some z, and hence x and y are of a shape produced by function h. For many applications Fusion on binary structures is applied with this weaker applicability condition. Suppose $h\,\nu_\oplus$ is defined to be the binary structure $[\nu_\oplus]$. Then

$$\begin{aligned} & h\,\nu_\oplus \\ = &\quad \text{definition of } h \\ & [\nu_\oplus] \\ = &\quad \text{property of guard} \\ & [(\in h\,z)?_\oplus\,\nu_\oplus] \\ = &\quad \text{definition of map} \\ & (\in h\,z)?_\oplus*\,h\,\nu_\oplus\ , \end{aligned}$$

for all z. Observe that $h\,z = (\in h\,z)?_\oplus*\,h\,z$. An immediate consequence is

$$h \cdot (=z)?_\oplus \;=\; (\in h\,z)?_\oplus* \cdot h \cdot (=z)?_\oplus\ . \tag{2.74}$$

The following calculation shows that f needs only be $(\oplus,\otimes)$-fusable after $\oplus/ \cdot g* \cdot (\in h\,z)\triangleleft$ for all binary structures z.

$$
\begin{array}{ll}
 & (f \cdot \oplus/ \cdot g* \cdot h)\, z \\
= & \quad \text{definition of guard} \\
 & (f \cdot \oplus/ \cdot g* \cdot h \cdot (= z)?_\oplus)\, z \\
= & \quad \text{equation (2.74)} \\
 & (f \cdot \oplus/ \cdot g* \cdot (\in h\, z)?_\oplus * \cdot h \cdot (= z)?_\oplus)\, z \\
= & \quad \text{map-distributivity} \\
 & (f \cdot \oplus/ \cdot (g \cdot (\in h\, z)?_\oplus)* \cdot h \cdot (= z)?_\oplus)\, z \\
= & \quad \text{Fusion} \\
 & (\otimes/ \cdot f* \cdot (g \cdot (\in h\, z)?_\oplus)* \cdot h \cdot (= z)?_\oplus)\, z \\
= & \quad \text{map-distributivity, first two steps in reverse order} \\
 & (\otimes/ \cdot f* \cdot g* \cdot h)\, z\ ,
\end{array}
$$

provided f is $(\oplus, \otimes)$-fusable after $\oplus/ \cdot (g \cdot (\in h\, z)?_\oplus)*$. Since

$$
\begin{array}{ll}
 & \oplus/ \cdot (g \cdot (\in h\, z)?_\oplus)* \\
= & \quad \text{map-distributivity} \\
 & \oplus/ \cdot g* \cdot (\in h\, z)?_\oplus * \\
= & \quad \text{equation (2.70), assume } g\, \nu_\oplus = \nu_\oplus \\
 & \oplus/ \cdot g* \cdot (\in h\, z)\triangleleft\ ,
\end{array}
$$

this condition amounts to f is $(\oplus, \otimes)$-fusable after $\oplus/ \cdot g* \cdot (\in h\, z)\triangleleft$. Note that if g is a guard $p?_\oplus$ then $g\, \nu_\oplus = \nu_\oplus$ holds. Thus we are able to use information about the context in our derivations. The above calculation is a translation to a functional setting of a similar calculation given by Malcolm [91] in a relational setting.

2.7 The data types snoc-list and cons-list

The data type *join-list* in the Boom-hierarchy is one way to 'implement' the intuitive idea we have of lists. In this section we discuss two other ways to represent lists as a data type. We discuss the data types *snoc-list* and *cons-list*. A cons-list ('cons' is an abbreviation of 'construct') is a list which is constructed from right to left: new elements are added to the left. A snoc-list is constructed just the other way around (which explains its name: 'snoc' is the reverse of 'cons'): from left to right. We state a large number of facts about functions defined on *snoc-list* and *cons-list*. Many of these facts are given without proof.

The data type *snoc-list* has been defined in Section 2.5 as an initial object in the category of K_A-algebras, where the functor K_A is defined in equation (2.49) by

$$\mathsf{K}_A \;=\; \underline{1} + (\mathsf{I} \times \underline{A})\ .$$

Define $(A\star, \Box \triangledown \mathbin{-\!\!<}) = \mu(\mathsf{K}_A)$, where $\Box : \mathit{1} \rightarrow A\star$ and $\mathbin{-\!\!<} : A\star \times A \rightarrow A\star$.

Left-reductions

Most of the algorithms on lists we derive in the second part of this thesis are catamorphisms on the data type *snoc-list*. Therefore, we give a name to catamorphisms on *snoc-list*; a catamorphism on *snoc-list* is called a *left-reduction*. We also introduce a special notation for left-reductions. Define

$$\oplus\!\not\rightarrow\!\epsilon \;\; = \;\; (\![\epsilon \triangledown \oplus]\!)_{\mathsf{K}_A} \; .$$

A left-reduction $\oplus\!\not\rightarrow\!\epsilon : A\star \rightarrow B$ is determined by an operator $\oplus : B \times A \rightarrow B$ and a function $\epsilon : \mathit{1} \rightarrow B$. Function ϵ yields an element $e : B$. Left-reduction $\oplus\!\not\rightarrow\!\epsilon$ evaluates list $\Box\, o$ as follows: $(\oplus\!\not\rightarrow\!\epsilon)\,\Box\, o = \epsilon\, o = e$, so $(\oplus\!\not\rightarrow\!\epsilon)\,[\,] = e$. Since we want to keep the evaluation rules as simple as possible, we replace function ϵ in the definition of a left-reduction by the element e it constructs. From now on, a left-reduction $\oplus\!\not\rightarrow\!e$ is uniquely determined by an operator $\oplus : B \times A \rightarrow B$ and a value $e : B$, which is called the *seed* of the left-reduction, such that

$$\begin{array}{lcl} (\oplus\!\not\rightarrow\!e)\,[\,] & = & e \\ (\oplus\!\not\rightarrow\!e)\,(x \mathbin{-\!\!<} a) & = & ((\oplus\!\not\rightarrow\!e)\,x) \oplus a \; . \end{array} \tag{2.75}$$

Note that in the evaluation of a left-reduction $\oplus\!\not\rightarrow\!e$ we may assume that all left-hand arguments of operator $\oplus$ are of the form $(\oplus\!\not\rightarrow\!e)\,x$ for some list x. Instantiating Identity Catamorphism, Corollary 2.37, on the data type *snoc-list*, it follows that $\mathbin{-\!\!<}\!\not\rightarrow\![\,]$ is the identity left-reduction. If operator $\oplus : A \times A \rightarrow A$ has a unit e, then, by definition, $\oplus/ : A\star \rightarrow A$ is the left-reduction $\oplus\!\not\rightarrow\!e$.

On-line algorithms

Left-reductions appear very often in the description of algorithms; there is a one-one correspondence between left-reductions and on-line algorithms. Let h be a recursively computable function defined on lists, and let ϕ be a program (on a random access machine) that computes the value of h on an argument. Following Galil [53] program ϕ is called *on line* if it processes one element of the argument at a time, and if the value of h on the processed elements is available before the next element from the argument is read. We omit a formal definition of the notion on line. So given an argument x, value $h\,x$ is computed by means of a series of successive approximations $h\,y$ where the y's are initial parts of x. It follows that (the structure of) an on-line program is a left-reduction. A program is called *real time* if it is on line and if in addition there exists a constant c such that the number of steps required to compute $h\,(x \mathbin{-\!\!<} a)$ given the value of $h\,x$ is bounded by c. It follows that a real-time program is a left-reduction the operator of which can be evaluated in constant time.

Fusion on *snoc-list*

Fusion, Corollary 2.41, is instantiated as follows on the data type *snoc-list*.

$$f \cdot \phi = \psi \cdot f\mathsf{K}_A \text{ after } (\!|\phi|\!)_{\mathsf{K}_A}\mathsf{K}_A \quad \equiv \quad f \cdot (\!|\phi|\!)_{\mathsf{K}_A} = (\!|\psi|\!)_{\mathsf{K}_A} \ .$$

Since ϕ and ψ are K_A-algebras, $\phi = \epsilon \triangledown \oplus$, and $\psi = \xi \triangledown \otimes$ for some functions ϵ, and ξ, and some operators $\oplus$, and $\otimes$. We have

$$\begin{array}{ll} & f \cdot \epsilon \triangledown \oplus = \xi \triangledown \otimes \cdot f\mathsf{K}_A \\ = & \qquad \text{equation (2.25), definition of } \mathsf{K}_A \\ & (f \cdot \epsilon) \triangledown (f \cdot \oplus) = \xi \triangledown \otimes \cdot id_1 + (f \times id_A) \\ = & \qquad \text{equations (2.24) and (2.31)} \\ & (f \cdot \epsilon = \xi) \ \wedge \ (f \cdot \oplus = \otimes \cdot f \times id_A) \ . \end{array}$$

To satisfy the first condition define ξ to be equal to $f \cdot \epsilon$. Supplying the arguments in the second condition we get the condition $f\ (x \oplus a) = (f\ x) \otimes a$ for the operators $\oplus$ and $\otimes$. Translating the above corollary of the Fusion Corollary we obtain

(2.76) Corollary (Fusion on *snoc-list***)** *Let $\oplus : A \times C \to A$, $\otimes : B \times C \to B$. Then*

$$f \cdot \oplus\not\rightarrow e \quad = \quad \otimes\not\rightarrow(f\ e)\ ,$$

if and only if $f : A \to B$ satisfies $f\ (x \oplus a) = (f\ x) \otimes a$ for all a, and for all x in the image of $\oplus\not\rightarrow e$.

Note that it follows from the conditions of this corollary that f is a K-homomorphism from $(A, e \triangledown \oplus)$ to $(B, (f\ e) \triangledown \otimes)$.

Concatenation on *snoc-list*

Operator $+\!\!\!\!\!+\!\!<$: $A\star \times A\star \to A\star$ concatenates two snoc-lists; if *snoc-list* and *join-list* are identified, then $x +\!\!\!\!\!+\!\!< y = x +\!\!\!\!\!+ y$. Operator $+\!\!\!\!\!+\!\!<$ is defined by

$$x +\!\!\!\!\!+\!\!< y \quad = \quad (+\!\!<\not\rightarrow x)\ y\ . \tag{2.77}$$

For example, $([\,] +\!\!< 1 +\!\!< 2) +\!\!\!\!\!+\!\!< ([\,] +\!\!< 1) = [\,] +\!\!< 1 +\!\!< 2 +\!\!< 1$. The empty list $[\,]$ is the unit of $+\!\!\!\!\!+\!\!<$ (this is easily proved), and operator $+\!\!\!\!\!+\!\!<$ is associative, that is, for all snoc-lists x, y and z

$$x +\!\!\!\!\!+\!\!< (y +\!\!\!\!\!+\!\!< z) \quad = \quad (x +\!\!\!\!\!+\!\!< y) +\!\!\!\!\!+\!\!< z\ ,$$

or, equivalently, $(x+\!\!\!\!\!+\!\!<) \cdot (y+\!\!\!\!\!+\!\!<) = ((x +\!\!\!\!\!+\!\!< y)+\!\!\!\!\!+\!\!<)$. Associativity of $+\!\!\!\!\!+\!\!<$ is proved by means of Fusion on *snoc-list*.

$$\begin{array}{ll}
 & (x {+\!\!\!+\!\!<}) \cdot (y {+\!\!\!+\!\!<}) = ((x {+\!\!\!+\!\!<} y) {+\!\!\!+\!\!<}) \\
\equiv & \qquad \text{definition of } {+\!\!\!+\!\!<} \\
 & (x {+\!\!\!+\!\!<}) \cdot ({-\!\!<} \not\rightarrow y) = {-\!\!<} \not\rightarrow (x {+\!\!\!+\!\!<} y) \\
\Leftarrow & \qquad \text{Fusion on } \textit{snoc-list} \\
 & (x {+\!\!\!+\!\!<}) \cdot {-\!\!<} = {-\!\!<} \cdot (x {+\!\!\!+\!\!<}) \times \mathit{id} \\
\Leftarrow & \qquad \text{Characterisation of left-reductions} \\
 & (x {+\!\!\!+\!\!<}) = {-\!\!<} \not\rightarrow x \\
\Leftarrow & \qquad \text{definition of } {+\!\!\!+\!\!<} \\
 & \mathit{true}\ .
\end{array}$$

Using operator ${+\!\!\!+\!\!<}$, function *rev*, which reverses a snoc-list, is defined by

$$\mathit{rev} \quad = \quad \oplus \not\rightarrow [\,]\ ,$$

where operator $\oplus$ is defined by $x \oplus a = [a] {+\!\!\!+\!\!<} x$.

Relating the data types *snoc-list* and *join-list*

The data types *snoc-list* and *join-list* are related in the following sense. By definition, $(A*, [\,] \triangledown \tau \triangledown +\!\!\!+)$ is an initial object in the category of J_A-algebras $(B, e \triangledown f \triangledown \oplus)$ where e is the unit of operator $\oplus$, and operator $\oplus$ is associative, and $(A\star, [\,] \triangledown {-\!\!<})$ is an initial algebra in the category of K_A-algebras.

(2.78) Theorem *$(A\star, [\,] \triangledown ([\,]{-\!\!<}) \triangledown {+\!\!\!+\!\!<})$ is an initial object in the category of J_A-algebras for which the binary operator has a unit and is associative, and $(A*, [\,] \triangledown (+\!\!\!+ \cdot \mathit{id} \times \tau))$ is an initial K_A-algebra.*

Proof We first prove that $(A\star, [\,] \triangledown ([\,]{-\!\!<}) \triangledown {+\!\!\!+\!\!<})$ is an initial J_A-algebra. Since operator ${+\!\!\!+\!\!<}$ is associative with unit $[\,]$, it follows that $(A\star, [\,] \triangledown ([\,]{-\!\!<}) \triangledown {+\!\!\!+\!\!<})$ is a J_A-algebra. Hence it suffices to prove that for every J_A-algebra $(B, e \triangledown f \triangledown \oplus)$, where operator $\oplus$ is associative and has unit e, there exists a unique homomorphism from $(A\star, [\,] \triangledown ([\,]{-\!\!<}) \triangledown {+\!\!\!+\!\!<})$ to $(B, e \triangledown f \triangledown \oplus)$. Consider the left-reduction $\otimes \not\rightarrow e$, where operator $\otimes$ is defined by

$$x \otimes y \quad = \quad x \oplus f\ y\ .$$

Left-reduction $\otimes \not\rightarrow e : A\star \rightarrow B$ is a J_A-homomorphism from $(A\star, [\,] \triangledown ([\,]{-\!\!<}) \triangledown {+\!\!\!+\!\!<})$ to $(B, e \triangledown f \triangledown \oplus)$, since

$$\begin{array}{ll}
 & \otimes \not\rightarrow e \cdot [\,] \triangledown ([\,]{-\!\!<}) \triangledown {+\!\!\!+\!\!<} \\
= & \qquad \text{equation (2.25)}
\end{array}$$

$$
\begin{array}{ll}
 & (\otimes\!\not\to\! e \cdot [\,]) \triangledown (\otimes\!\not\to\! e \cdot ([\,]\mathbin{+\!\!\!<})) \triangledown (\otimes\!\not\to\! e \cdot \mathbin{+\!\!\!+\!\!\!<}) \\
= & \quad \text{definition of left-reduction} \\
 & e \triangledown (e\otimes) \triangledown (\otimes\!\not\to\! e \cdot \mathbin{+\!\!\!+\!\!\!<}) \\
= & \quad \text{claim} \\
 & e \triangledown f \triangledown (\oplus \cdot (\otimes\!\not\to\! e) \times (\otimes\!\not\to\! e)) \\
= & \quad \text{equation (2.24), definition of functor } \mathsf{J}_A \\
 & e \triangledown f \triangledown \oplus \cdot (\otimes\!\not\to\! e)\mathsf{J}_A \ .
\end{array}
$$

The claim in this calculation is discharged as follows. First, $e \otimes a = e \oplus f\ a = f\ a$, and second, we have to prove that

$$\otimes\!\not\to\! e \cdot \mathbin{+\!\!\!+\!\!\!<} \quad = \quad \oplus \cdot (\otimes\!\not\to\! e) \times (\otimes\!\not\to\! e) \ ,$$

or, supplying one argument,

$$\otimes\!\not\to\! e \cdot (x\mathbin{+\!\!\!+\!\!\!<}) \quad = \quad ((\otimes\!\not\to\! e)\, x \oplus) \cdot \otimes\!\not\to\! e \ . \qquad (2.79)$$

For the left-hand side of this equality we have, applying Fusion on *snoc-list*

$$\otimes\!\not\to\! e \cdot (x\mathbin{+\!\!\!+\!\!\!<}) \quad = \quad \otimes\!\not\to\!(\otimes\!\not\to\! e)\, x \ .$$

For the right-hand side of equality (2.79) we have the following. Since we want to apply Fusion on *snoc-list*, we verify the condition of Fusion.

$$
\begin{array}{ll}
 & (\otimes\!\not\to\! e)\, x \oplus (z \otimes a) \\
= & \quad \text{definition of } \otimes \\
 & (\otimes\!\not\to\! e)\, x \oplus (z \oplus f\ a) \\
= & \quad \text{operator } \oplus \text{ is associative} \\
 & ((\otimes\!\not\to\! e)\, x \oplus z) \oplus f\ a \\
= & \quad \text{definition of } \otimes \\
 & ((\otimes\!\not\to\! e)\, x \oplus z) \otimes a \ ,
\end{array}
$$

and hence we obtain for the right-hand side of equation (2.79)

$$((\otimes\!\not\to\! e)\, x\oplus) \cdot \otimes\!\not\to\! e \quad = \quad \otimes\!\not\to\!(\otimes\!\not\to\! e)\, x \ ,$$

which proves that equality (2.79) holds. Since for each algebra $(B, e \triangledown f \triangledown \oplus)$ left-reduction $\otimes\!\not\to\! e$ is the unique homomorphism from $(A\star, [\,] \triangledown ([\,]\mathbin{+\!\!\!<}) \triangledown \mathbin{+\!\!\!+\!\!\!<})$ to $(B, e \triangledown f \triangledown \oplus)$ it follows that $(A\star, [\,] \triangledown ([\,]\mathbin{+\!\!\!<}) \triangledown \mathbin{+\!\!\!+\!\!\!<})$ is an initial J_A-algebra.

For the proof of the fact that $(A*, [\,] \triangledown (+\!\!\!+ \cdot id \times \tau))$ is an initial K_A-algebra we see no way to avoid using induction. For every K_A-algebra $(B, e \triangledown \oplus)$ it is required to give a unique homomorphism from $(A*, [\,] \triangledown (+\!\!+ \cdot id \times \tau))$ to $(B, e \triangledown \oplus)$. We shall define a function g such that

$$g \cdot [\,] \triangledown (+\!\!+ \cdot id \times \tau) \quad = \quad e \triangledown \oplus \cdot g\mathsf{K}_A \ , \tag{2.80}$$

and we show that if another function satisfies the equation then it is equal to g. Elaborating equation (2.80), we find the following conditions on g.

$$\begin{array}{ll}
 & g \cdot [\,] \triangledown (+\!\!+ \cdot id \times \tau) = e \triangledown \oplus \cdot g\mathsf{K}_A \\
\equiv & \quad \text{equation (2.25), definition of functor } \mathsf{K}_A\text{, equation (2.24)} \\
 & (g \cdot [\,]) \triangledown (g \cdot +\!\!+ \cdot id \times \tau) = e \triangledown (\oplus \cdot g \times id) \\
\equiv & \quad \text{equation (2.31)} \\
 & (g \cdot [\,] = e) \ \wedge \ (g \cdot +\!\!+ \cdot id \times \tau = \oplus \cdot g \times id) \ .
\end{array}$$

Define $g : A* \to B$ by induction on the length of lists. For the base case, define $g \cdot [\,] = e$. Suppose $g\, y = z$, and define $g\,(y +\!\!+ [a])$ by

$$g\,(y +\!\!+ [a]) \quad = \quad z \oplus a \ .$$

We verify the two conditions function g has to satisfy in order to be a K_A-homomorphism. First, $g \cdot [\,] = e$ is satisfied by definition of g. The second condition is equivalent to the inductive step in the definition of function g. This shows that g is a K_A-homomorphism. Furthermore, we can prove by induction that if h is a K_A-homomorphism from $(A*, [\,] \triangledown (+\!\!+ \cdot id \times \tau))$ to $(B, e \triangledown \oplus)$, then $h = g$, and hence it follows that $(A*, [\,] \triangledown (+\!\!+ \cdot id \times \tau))$ is an initial K_A-algebra. □

Fokkinga [47] describes a general procedure to 'view' one initial algebra as another initial algebra, and applies the procedure to show that the data type *cons-list* can be viewed as the data type *snoc-list*. Using the above theorem it can be shown that the data type *snoc-list* can be viewed as the data type *join-list* and vice versa.

Mutumorphisms on *snoc-list*

Besides catamorphisms on *snoc-list*, we will often use mutumorphisms on *snoc-list*. Function f is catamorphic modulo g on the data type *snoc-list* if for some operator $\oplus$

$$f\,(x \twoheadleftarrow a) \quad = \quad (f\,x, g\,x) \oplus a \ .$$

Instantiating Theorem 2.47 on the data type *snoc-list* we obtain

(2.81) Corollary *Suppose f and g are catamorphic modulo each other on the data type* snoc-list*:*

$$\begin{array}{lcl}
f\,(x \twoheadleftarrow a) & = & (f\,x, g\,x) \oplus a \\
g\,(x \twoheadleftarrow a) & = & (g\,x, f\,x) \otimes a \ .
\end{array}$$

Then

$$f \vartriangle g \quad = \quad \odot\!\not\rightarrow\!(e, u)\ ,$$

where $e = f\,[\,]$, $u = g\,[\,]$, *and operator* $\odot$ *is defined by*

$$(x, y) \odot a \quad = \quad ((x, y) \oplus a, (y, x) \otimes a)\ .$$

Predicates on *snoc-list*

Consider a predicate defined on the data type *snoc-list*, i.e., a function $p : A\star \rightarrow bool$. When can $p?_{\otimes} : A\star \rightarrow A\star$ be written as a left-reduction? This question is important when we try to apply Corollary 2.65 to an expression in which function f is a guard. This will occur frequently. According to Catamorphism Characterisation, Theorem 2.35, instantiated to the data type *snoc-list*, we have that $p?_{\otimes}$ is a left-reduction $\ominus\!\not\rightarrow\! e$ if and only if

$$\begin{array}{lcl} p?_{\otimes}\,[\,] & = & e \\ p?_{\otimes} \cdot \mathbin{-\!\!\!<} & = & \ominus \cdot p?_{\otimes} \times id\ . \end{array}$$

To satisfy the second condition we impose a condition upon predicate p and define operator $\ominus$. Suppose predicate p is such that $p\,[\,]$ holds, and such that for all snoc-lists x and elements a

$$p\,(x \mathbin{+\!\!\!+\!\!\!<} [a]) \quad \Rightarrow \quad p\,x\ . \tag{2.82}$$

We call a predicate p satisfying implication (2.82) and for which $p\,[\,]$ holds *prefix-closed*, a term introduced by Bird [16]. Now define operator $\ominus$ as follows.

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\!<} a & \text{if } x \neq \nu_{\otimes} \ \wedge\ p\,(x \mathbin{-\!\!\!<} a) \\ \nu_{\otimes} & \text{otherwise}\ . \end{cases} \tag{2.83}$$

For prefix-closed predicate p and operator $\ominus$ as defined in (2.83) we have $p?_{\otimes} \cdot \mathbin{-\!\!\!<} = \ominus \cdot p?_{\otimes} \times id$.

(2.84) Theorem *Suppose p is a prefix-closed predicate. Then*

$$p?_{\otimes} \quad = \quad \ominus\!\not\rightarrow\![\,]\ ,$$

where operator $\ominus$ *is defined by*

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\!<} a & \textit{if } x \neq \nu_{\otimes} \ \wedge\ p\,(x \mathbin{-\!\!\!<} a) \\ \nu_{\otimes} & \textit{otherwise}\ . \end{cases}$$

An example of a prefix-closed predicate is the predicate *asc* defined by

$$\begin{array}{lcl} asc\,[\,] & = & true \\ asc\,(x \mathbin{-\!\!\!<} a) & = & asc\,x \;\wedge\; lt\,x \leq a\;, \end{array} \tag{2.85}$$

where *lt* is a function which returns the last element of a nonempty snoc-list. For the purpose of the above definition we assume that $lt\,[\,] = -\infty$. We will encounter many more prefix-closed predicates in the sequel. An important property of a prefix-closed predicate p is that there exists a function δ, called a *derivative* of p, that satisfies

$$p\,(x \mathbin{-\!\!\!<} a) \;\equiv\; p\,x \;\wedge\; \delta_x\,a\;. \tag{2.86}$$

For example, predicate *asc* has derivative δ defined by $\delta_x\,a = lt\,x \leq a$. Given a prefix-closed predicate p, a definition of function δ satisfying (2.86) is $\delta_x\,a = p\,(x \mathbin{-\!\!\!<} a)$.

A predicate $p : A\star \rightarrow$ *bool* is *suffix-closed* if $p\,[\,]$ holds and if for all snoc-lists x and elements a

$$p\,([a] \mathbin{+\!\!\!\!<} x) \quad\Rightarrow\quad p\,x\;. \tag{2.87}$$

Again, the predicate *asc* serves as an example here: *asc* is suffix-closed. A predicate $p : A\star \rightarrow$ *bool* is *segment-closed* if it is both prefix-closed and suffix-closed, so *asc* is segment-closed. Another example of a segment-closed predicate is the predicate *all* p defined in equation (2.71).

A predicate is *robust* if it satisfies for all lists x, y and z with $y \neq [\,]$

$$p\,(x \mathbin{+\!\!\!\!<} y) \;\wedge\; p\,(y \mathbin{+\!\!\!\!<} z) \quad\Rightarrow\quad p\,(x \mathbin{+\!\!\!\!<} y \mathbin{+\!\!\!\!<} z)\;. \tag{2.88}$$

An example of a robust predicate is the predicate *asc* defined in equation (2.85).

The data type *cons-list*

The data type *cons-list* is defined as an initial object in the category of M_A-algebras, where the functor M_A is defined by

$$\mathsf{M}_A \;=\; \underline{1} + (\underline{A} \times \mathsf{I})\;. \tag{2.89}$$

Define $(A\star, \square \triangledown \mathbin{>\!\!\!-}) = \mu(\mathsf{M}_A)$, where $\square : 1 \rightarrow A\star$ and $\mathbin{>\!\!\!-} : A \times A\star \rightarrow A\star$. Function $\square$ constructs the empty list, and operator $\mathbin{>\!\!\!-}$ prepends an element to a list. Note that functor $\star$ is overloaded. A catamorphism on the data type cons-list is called a *right-reduction*. Define

$$\oplus\!\not\leftarrow\! e \;=\; (\!|e \triangledown \oplus|\!)_{\mathsf{M}_A}\;.$$

There exist initial J_A-algebras and K_A-algebras with as carrier the carrier of the data type *cons-list* $A\star$. Conversely, there exist initial M_A-algebras with as carrier the carrier from the data type *snoc-list* or the carrier of the data type *join-list*.

2.8 Conclusions

This chapter presents a framework for program construction. Calculation, or more specifically, equational reasoning, plays an important role in this framework. A program is viewed as a function on some data type. Initial algebras turn out to be a formalisation of the intuitive notion of data type. An initial algebra is equipped with several laws that are useful tools in the derivation of equalities of functions defined on the initial algebra. This chapter first describes the general case in which laws for arbitrary initial algebras are considered, and then it instantiates some of these laws for some particular data types such as the data types in the Boom-hierarchy, and the data type *snoc-list*. Finally, it describes properties specific to (functions defined on) the data types *join-list* and *snoc-list*.

Chapter 3

Auxiliary functions and operators

This chapter presents several auxiliary functions and operators, defined on various data types, which will appear frequently in Part II. Section 3.1 describes well-known functions defined on the data type *snoc-list*, and it gives some equations satisfied by these functions. The operators cross, zip, and merge, together with a set of results, are given in Section 3.2. Section 3.3 discusses minimum and maximum operators, which are also called selectors. Section 3.4 gives some conclusions.

3.1 Functions on *snoc-list*

We define a number of functions on the data type *snoc-list*.

Head and tail

The function $hd : A\star \rightarrow A$ (an abbreviation for head) returns the first element of a nonempty list. It is not defined on the empty list. Alternatively, we may add a fictitious element to the type A representing $hd\,[\,]$. Function hd satisfies

$$hd\,([a] \mathbin{+\!\!\!+\!\!<} x) \quad = \quad a\ . \tag{3.1}$$

Function hd may be defined as a right-reduction (on the data type *cons-list*), or a left-reduction (on the data type *snoc-list*), or a catamorphism on *join-list*. Define $hd = exl \not\rightarrow (hd\,[\,])$. Function $tl : A\star \rightarrow A\star$ (for tail) returns all but the first element of a list, so

$$tl\,([a] \mathbin{+\!\!\!+\!\!<} x) \quad = \quad x\ . \tag{3.2}$$

$tl\,[\,]$ is defined to be a fictitious element, which is added to the data type $A\star$. The definition of function tl as a left-reduction is omitted. From the equations for hd and tl we immediately find that on the domain of nonempty lists

$$\mathbin{+\!\!\!+\!\!\!<} \cdot (\tau \cdot hd) \vartriangle tl \;=\; id\ .$$

Functions *hd* and *tl* are natural transformations, and they satisfy the laws

$$
\begin{aligned}
f \cdot hd &= hd \cdot f\star && (3.3)\\
f\star \cdot tl &= tl \cdot f\star\ , && (3.4)
\end{aligned}
$$

provided $f\ hd\,[\,] = hd\,[\,]$, and $f\star\ tl\,[\,] = tl\,[\,]$.

Last and init

The dual versions of *hd* and *tl* are the functions $lt : A\star \to A$ (for last) and $it : A\star \to A\star$ (for init). Values $lt\,[\,]$ and $it\,[\,]$ are fictitious elements. Functions *lt* and *it* satisfy

$$
\begin{aligned}
lt\,(x \mathbin{+\!\!\!+\!\!\!<} [a]) &= a && (3.5)\\
it\,(x \mathbin{+\!\!\!+\!\!\!<} [a]) &= x\ . && (3.6)
\end{aligned}
$$

Functions *lt* and *it* satisfy laws similar to the laws satisfied by *hd* and *tl*; we mention the equivalents of equations (3.3) and (3.4).

$$
\begin{aligned}
f \cdot lt &= lt \cdot f\star && (3.7)\\
f\star \cdot it &= it \cdot f\star\ , && (3.8)
\end{aligned}
$$

provided $f\ lt\,[\,] = lt\,[\,]$, and $f\star\ it\,[\,] = it\,[\,]$.

An alternative definition of functions *lt* and *it* is obtained by means of function *out* on *snoc-list*. Function *out* satisfies

$$
\begin{aligned}
out\,[\,] &= inl\ o\\
out\,(x \mathbin{-\!\!\!<} a) &= inr\,(x, a)\ .
\end{aligned}
$$

Let $\alpha : 1 \to A$ be a function that constructs a fictitious element of type A, and let $\beta : 1 \to A\star$ be a function that constructs a fictitious element of type $A\star$. Then

$$
\begin{aligned}
\alpha \triangledown exr \cdot out &: A\star \to A\\
\beta \triangledown exl \cdot out &: A\star \to A\star\ ,
\end{aligned}
$$

respectively return the last element of a snoc-list, and all but the last element of a snoc-list. Functions *hd* and *tl* can be defined similarly on the data type *cons-list*.

Generalising head etc. to *snoc-list* over *snoc-list*

Functions *git*, *glt*, *gtl*, and *ghd*, defined on the data type *snoc-list* over base type *snoc-list*, are generalisations of functions *it*, *lt*, *tl*, and *hd*. Function $glt : A\star\star \rightarrow A$ returns the last element of the last nonempty list of a snoc-list of snoc-lists, that is,

$$glt\,x \quad = \quad lt \,{+\!\!\!+\!\!\!<}/\, x \; .$$

Function *glt* is not defined on the empty list or on a list consisting solely of empty lists. For a nonempty list of lists in which at least one list contains an element it satisfies the following equation.

$$glt\,x \quad = \quad \begin{cases} lt\;lt\,x & \text{if } \#\,lt\,x \geq 1 \\ glt\;it\,x & \text{otherwise} \;\; . \end{cases}$$

Function $git : A\star\star \rightarrow A\star\star$ returns all but the last element of the last nonempty list of a snoc-list of snoc-lists. Again, function *git* is not defined on the empty list or on a list consisting solely of empty lists. For a nonempty list of lists in which at least one list contains an element it satisfies the following equation.

$$git\,x \quad = \quad \begin{cases} it\,x \,{+\!\!<}\; it\;lt\,x & \text{if } \#\,lt\,x > 1 \\ it\,x & \text{if } \#\,lt\,x = 1 \\ git\;it\,x & \text{otherwise} \;\; . \end{cases} \tag{3.9}$$

Functions $ghd : A\star\star \rightarrow A$ which returns the first element of the first nonempty snoc-list of a snoc-list of snoc-lists, and function $gtl : A\star\star \rightarrow A\star\star$ which returns all but the first element of the first nonempty snoc-list of a snoc-list of snoc-lists, are defined similarly.

Length

The function $\# : A\star \rightarrow nat$ returns the length of a list. It is defined by

$$\# \quad = \quad ((1+) \cdot exl) \not\rightarrow 0 \; . \tag{3.10}$$

Take and drop

The operator *take* (from the front), $\rightharpoonup : nat \times A\star \rightarrow A\star$, returns, given a natural n and a list x, the first n elements of x if $\#\,x \geq n$, and it returns x itself if $\#\,x < n$. Function $n \rightharpoonup$ is defined as follows.

$$\begin{array}{lcl} 0 \rightharpoonup x & = & [\,] \\ (n{+}1) \rightharpoonup x & = & \begin{cases} [\,] & \text{if } x = [\,] \\ [hd\,x] \,{+\!\!\!+\!\!\!<}\, (n \rightharpoonup tl\,x) & \text{otherwise} \;\; . \end{cases} \end{array} \tag{3.11}$$

The operator which given a natural number n and a list x takes the last n elements of x is denoted by $\leftharpoonup$: $\mathit{nat} \times A\star \rightarrow A\star$, and is defined similar to operator $\rightharpoonup$. Furthermore, there exist operators $\hookrightarrow$: $\mathit{nat} \times A\star \rightarrow A\star$ and $\hookleftarrow$: $\mathit{nat} \times A\star \rightarrow A\star$ that satisfy

$$\begin{aligned} (n \rightharpoonup x) \mathbin{+\!\!+} (n \hookrightarrow x) &= x && (3.12)\\ (n \hookleftarrow x) \mathbin{+\!\!+} (n \leftharpoonup x) &= x\ . && (3.13) \end{aligned}$$

Operator $\hookrightarrow$, called *drop* in Bird and Wadler [22], drops, given a natural number n and a list x, the first n elements of x. It is defined by

$$\begin{aligned} 0 \hookrightarrow x &= x \\ (n+1) \hookrightarrow x &= \begin{cases} [\,] & \text{if } x = [\,] \\ n \hookrightarrow \mathit{tl}\, x & \text{otherwise}\ . \end{cases} && (3.14) \end{aligned}$$

Operator $\hookleftarrow$ is defined similarly. Let operator $\rightleftharpoons$ be any of the four operators $\rightharpoonup$, $\leftharpoonup$, $\hookrightarrow$, or $\hookleftarrow$. Operator $\rightleftharpoons$ satisfies for all functions f and all numbers n

$$f\star \cdot (n \rightleftharpoons) = (n \rightleftharpoons) \cdot f\star\ . \qquad (3.15)$$

Shift

Let $\vdash$ be a function, called *shift right*, that given a natural number c and a pair of lists (x, y) shifts the last c elements of x to y, and let $\dashv$ be the converse of operator $\vdash$. Functions $c\vdash$ and $c\dashv$ are defined by

$$\begin{aligned} c \vdash (x, y) &= (c \hookleftarrow x, (c \leftharpoonup x) \mathbin{+\!\!+} y) && (3.16)\\ c \dashv (x, y) &= (x \mathbin{+\!\!+} (c \rightharpoonup y), c \hookrightarrow y)\ . && (3.17) \end{aligned}$$

The shift functions satisfy the following properties.

$$\begin{aligned} (c{+}d)\vdash &= (c\vdash) \cdot (d\vdash) && (3.18)\\ (c{+}d)\dashv &= (c\dashv) \cdot (d\dashv)\ . && (3.19) \end{aligned}$$

Their definitions can be extended to integers if we define for $c \geq 0$

$$\begin{aligned} -c\vdash &= c \dashv && (3.20)\\ -c\dashv &= c \vdash\ . && (3.21) \end{aligned}$$

3.2 Cross, zip, and merge

Operators cross, zip and merge are used frequently in our calculations.

Cross

Cross, denoted by $\chi : A\wr \times B\wr \to (A \times B)\wr$, takes two sets, and pairs each element of the first set with each element of the second set. The result of cross is a set of these pairs. For example, $\{1,2\} \chi \{3,4,5\} = \{(1,3),(2,3),(1,4),(2,4),(1,5),(2,5)\}$. Operator cross is defined as follows.

$$\begin{array}{rcl} x \chi y &=& \cup/ \, (x\mathsf{x}){*}\, y \\ x \mathsf{x}\, b &=& (\odot b){*}\, x \\ a \odot b &=& (a,b) \; . \end{array} \tag{3.22}$$

Another definition of operator cross can be found in Bird [17]. Often, operator cross is subscripted with a binary operator, by which we mean the following.

$$\chi_{\oplus} \;=\; \oplus{*} \cdot \chi \; . \tag{3.23}$$

It follows that $\chi = \chi_{\odot}$. If, given an operator $\oplus$, we replace the definition of x in (3.22) by

$$x \mathsf{x}\, b \;=\; (\oplus b){*}\, x \; ,$$

equations (3.22) define $\chi_{\oplus}$. Value $\{\nu_{\oplus}\}$ is a left- and right-unit of operator $\chi_{\oplus}$. For unit $\nu_{\oplus}$ of operator $\oplus$ we prove that $\{\nu_{\oplus}\}$ is a right-unit of operator $\chi_{\oplus}$.

$$\begin{array}{ll} & x \chi_{\oplus} \{\nu_{\oplus}\} \\ = & \quad \text{definition of cross} \\ & \cup/ \, (x\mathsf{x}){*}\, \{\nu_{\oplus}\} \\ = & \quad \text{definition of map and reduction} \\ & x \mathsf{x}\, \nu_{\oplus} \\ = & \quad \text{definition (3.24) of } \mathsf{x} \\ & (\oplus \nu_{\oplus}){*}\, x \\ = & \quad \text{definition of } \nu_{\oplus} \text{ and map} \\ & x \; . \end{array}$$

$\chi_{\oplus}$ is associative if $\oplus$ is associative

If operator $\oplus$ is associative, then so is operator $\chi_{\oplus}$, that is, for all x, y, and z

$$(x \chi_{\oplus} y) \chi_{\oplus} z \;=\; x \chi_{\oplus} (y \chi_{\oplus} z) \; . \tag{3.24}$$

This equation is proved using Leibniz' rule and the extensionality axiom. First, rewrite both sides to an application of a catamorphism. For the left-hand side of equation (3.24) we have

$$\begin{array}{ll}
 & (x \mathbin{\mathsf{X}}_{\oplus} y) \mathbin{\mathsf{X}}_{\oplus} z \\
= & \qquad \text{definition of cross (twice)} \\
 & \cup/\,((\cup/\,(x\mathsf{x}){*}\,y)\mathsf{x}){*}\,z\ ,
\end{array}$$

and for the right-hand side of equation (3.24) we have

$$\begin{array}{ll}
 & x \mathbin{\mathsf{X}}_{\oplus} (y \mathbin{\mathsf{X}}_{\oplus} z) \\
= & \qquad \text{definition of cross (twice)} \\
 & \cup/(x\mathsf{x}){*}\,\cup/\,(y\mathsf{x}){*}\,z \\
= & \qquad \text{Fusion on } \textit{set}\text{, map-distributivity} \\
 & \cup/\,(\cup/\cdot(x\mathsf{x}){*}\cdot(y\mathsf{x})){*}\,z\ .
\end{array}$$

Applying Leibniz' rule and the extensionality axiom, the remaining proof obligation is

$$\cup/\cdot(x\mathsf{x}){*}\cdot(y\mathsf{x}) = ((\cup/\,(x\mathsf{x}){*}\,y)\mathsf{x})\ . \tag{3.25}$$

This equation is proved using the extensionality axiom. For the left-hand side of equation (3.25) we have

$$\begin{array}{ll}
 & \cup/\,(x\mathsf{x}){*}\,(y \mathbin{\mathsf{x}} a) \\
= & \qquad \text{definition (3.24) of } \mathsf{x} \\
 & \cup/\,(x\mathsf{x}){*}\,(\oplus a){*}\,y \\
= & \qquad \text{map-distributivity} \\
 & \cup/\,((x\mathsf{x})\cdot(\oplus a)){*}\,y\ ,
\end{array}$$

and for the right-hand side of equation (3.25) we have

$$\begin{array}{ll}
 & (\cup/\,(x\mathsf{x}){*}\,y) \mathbin{\mathsf{x}} a \\
= & \qquad \text{definition (3.24) of } \mathsf{x} \\
 & (\oplus a){*}\,\cup/\,(x\mathsf{x}){*}\,y \\
= & \qquad \text{Fusion on } \textit{set}\text{, map-distributivity} \\
 & \cup/\,((\oplus a){*}\cdot(x\mathsf{x})){*}\,y\ ,
\end{array}$$

which reduces the proof obligation to showing that

$$(x\mathsf{x})\cdot(\oplus a) \quad = \quad (\oplus a){*}\cdot(x\mathsf{x})\ . \tag{3.26}$$

Equation (3.26) is proved by means of the extensionality axiom.

$$\begin{array}{ll}
 & (\oplus a){*}\,(x \mathbin{\mathsf{x}} b) \\
= & \qquad \text{definition (3.24) of } \mathsf{x}
\end{array}$$

$$\begin{array}{ll}
 & (\oplus a){*}\,(\oplus b){*}\,x \\
= & \quad \text{map-distributivity} \\
 & ((\oplus a)\cdot(\oplus b)){*}\,x \\
= & \quad \oplus \text{ is associative} \\
 & (\oplus(b\oplus a)){*}\,x \\
= & \quad \text{definition of } \mathsf{x} \\
 & x\,\mathsf{x}\,(b\oplus a)\ .
\end{array}$$

This calculation proves equation (3.24). Without proof we mention that if operator $\oplus$ is commutative, then so is $\chi_\oplus$. It is not the case that if operator $\oplus$ is idempotent, then so is $\chi_\oplus$.

Calculational properties of cross

Cross satisfies a number of useful calculational properties. The first Cross Law is an easy application of map-distributivity.

(3.27) Law *For all functions* $f : C \to D$ *and operators* $\oplus : A \times B \to C$

$$f{*}\cdot\chi_\oplus \;=\; \chi_{f\cdot\oplus}\ .$$

The second law is more difficult to prove.

(3.28) Law *For all functions* f *and* g

$$\chi_{f\times g} \;=\; \chi\cdot f{*}\times g{*}\ .$$

Proof

$$\begin{array}{ll}
 & x\,\chi_{f\times g}\,y \\
= & \quad \text{definition of cross} \\
 & (f\times g){*}\cup/\,(x\mathsf{x}){*}\,y \\
= & \quad \text{Fusion on } \textit{set}\text{, map-distributivity} \\
 & \cup/\,((f\times g){*}\cdot(x\mathsf{x})){*}\,y \\
= & \quad \text{equation (3.29) below} \\
 & \cup/\,((f{*}\,x)\mathsf{x}){*}\,g{*}\,y \\
= & \quad \text{definition of cross} \\
 & f{*}\,x\;\chi\;g{*}\,y\ ,
\end{array}$$

and the law follows by extensionality. The remaining proof obligation is

$$(f \times g){*} \cdot (x\mathsf{x}) \;=\; ((f{*}\,x)\mathsf{x}) \cdot g\ . \tag{3.29}$$

Using the following equation, which is easily proved, equation (3.29) follows by extensionality.

$$f \times g \cdot (\textcircled{,}a) \;=\; (\textcircled{,}g\ a) \cdot f\ . \tag{3.30}$$

We calculate as follows.

$$\begin{array}{cl}
 & (f \times g){*}\,(x \,\mathsf{x}\, a) \\
= & \quad \text{definition (3.22) of } \mathsf{x}\text{, map-distributivity} \\
 & (f \times g \cdot (\textcircled{,}a)){*}\,x \\
= & \quad \text{equation (3.30), map-distributivity} \\
 & (\textcircled{,}g\ a){*}\,f{*}\,x \\
= & \quad \text{definition of } \mathsf{x} \\
 & (f{*}\,x) \,\mathsf{x}\, (g\ a)\ .
\end{array}$$

□

From laws 3.27 and 3.28 it follows that $\mathsf{X}_{\textcircled{,}}$ is a natural transformation, since for all functions f and g

$$(f \times g){*} \cdot \mathsf{X}_{\textcircled{,}} \;=\; \mathsf{X}_{\textcircled{,}} \cdot f{*} \times g{*}\ .$$

The third result for cross gives sufficient conditions for distributing a reduction over cross.

(3.31) Theorem *Suppose operators*

$$\begin{array}{lcl}
\otimes & : & C \times C \to C \\
\oplus & : & A \times A \to A \\
\ominus & : & B \times B \to B \\
\odot & : & A \times B \to C\ ,
\end{array}$$

are such that for all b, section $(\odot b)$ is $(\oplus, \otimes)$-fusable after $\oplus/$, and for all a, section $(a\odot)$ is $(\ominus, \otimes)$-fusable after $\ominus/$. Then

$$\otimes/ \cdot \mathsf{X}_{\odot} \;=\; \odot \cdot \oplus/ \times \ominus/\ .$$

Proof Using Fusion it follows from $(\odot b)$ is $(\oplus, \otimes)$-fusable after $\oplus/$ that $(\odot b) \cdot \oplus/ = \otimes/ \cdot (\odot b){*}$, and from $(a\odot)$ is $(\ominus, \otimes)$-fusable after $\ominus/$ that $(a\odot) \cdot \ominus/ = \otimes/ \cdot (a\odot){*}$. We now calculate

$$
\begin{array}{ll}
 & \otimes/\,(x \mathbin{\mathsf{X}_{\odot}} y) \\
= & \qquad \text{definition of cross} \\
 & \otimes/\,\cup/\,(x\mathsf{x}) * y \\
= & \qquad \text{Fusion on } \mathit{set}\text{, map-distributivity} \\
 & \otimes/\,(\otimes/\cdot(x\mathsf{x})) * y \\
= & \qquad \text{subdevelopment below} \\
 & \otimes/\,((\oplus/\,x)\odot) * y \\
= & \qquad \text{Fusion on } \mathit{set} \\
 & (\oplus/\,x) \odot (\ominus/\,y)\ ,
\end{array}
$$

which proves this law. The remaining proof obligation is the equality $\otimes/\cdot(x\mathsf{x}) = ((\oplus/\,x)\odot)$.

$$
\begin{array}{ll}
 & \otimes/\,(x \mathbin{\mathsf{x}} b) \\
= & \qquad \text{definition (3.24) of } \mathsf{x} \\
 & \otimes/\,(\odot b) * x \\
= & \qquad \text{Fusion on } \mathit{set} \\
 & (\odot b) \oplus/\,x \\
= & \qquad \text{definition of section} \\
 & (\oplus/\,x) \odot b\ .
\end{array}
$$

□

The laws for cross given by Bird [17] are all easy consequences of the laws for cross given here.

Zip

The operator zip takes two lists of equal length and 'zips' these lists to a list of pairs. If the arguments of zip do not have equal length, zip returns a fictitious element. We give without proof a set of laws and theorems for zip that correspond to the laws given for cross. The definition of zip on the data type *join-list* is a bit awkward, see Bird [17], and therefore we restrict the definition of zip to the data type *snoc-list*. Zip is denoted by $\Upsilon : A\star \times B\star \longrightarrow (A \times B)\star$, and defined by

$$
\begin{array}{lcl}
[\,]\ \Upsilon\ [\,] & = & [\,] \\
(x \mathbin{-\!\!<} a)\ \Upsilon\ (y \mathbin{-\!\!<} b) & = & (x\ \Upsilon\ y) \mathbin{-\!\!<} (a, b)\ .
\end{array}
\qquad (3.32)
$$

Operator zip may be subscripted with a binary operator, by which we mean the following.

$$\Upsilon_{\oplus} \quad = \quad \oplus\star \cdot \Upsilon \, . \tag{3.33}$$

It follows that $\Upsilon = \Upsilon_{\textcircled{,}}$. If operator $\oplus$ is associative, then so is $\Upsilon_{\oplus}$; if operator $\oplus$ is commutative, then so is $\Upsilon_{\oplus}$; and, in contrast with cross, if operator $\oplus$ is idempotent, then so is $\Upsilon_{\oplus}$.

Calculational properties of zip

Like cross, zip satisfies a number of useful calculational properties. The first law corresponds to law 3.27 for cross.

(3.34) Law *For all functions $f : C \rightarrow D$ and operators $\oplus : A \times B \rightarrow C$*

$$f\star \cdot \Upsilon_{\oplus} \quad = \quad \Upsilon_{f \cdot \oplus} \, .$$

The second law, corresponding to law 3.28 for cross, reads as follows.

(3.35) Law *For all functions f and g*

$$\Upsilon_{f \times g} \quad = \quad \Upsilon \cdot f\star \times g\star \, .$$

The third result is the equivalent of Theorem 3.31 for cross.

(3.36) Theorem *Suppose operators*

$$\begin{array}{lcl} \otimes & : & C \times C \rightarrow C \\ \oplus & : & A \times A \rightarrow A \\ \ominus & : & B \times B \rightarrow B \\ \odot & : & A \times B \rightarrow C \, , \end{array}$$

satisfy

$$\begin{array}{lcl} \nu_{\otimes} & = & \nu_{\oplus} \odot \nu_{\ominus} \\ (a \odot b) \otimes (c \odot d) & = & (a \oplus c) \odot (b \ominus d) \, . \end{array}$$

Then

$$\otimes\!\not\rightarrow\!\nu_{\otimes} \cdot \Upsilon_{\odot} \quad = \quad \odot \cdot \oplus\!\not\rightarrow\!\nu_{\oplus} \times \ominus\!\not\rightarrow\!\nu_{\ominus} \, .$$

In case operators $\otimes$, $\oplus$, and $\ominus$ are equal, the second requirement of this law reads: for all a, b, c, and d, $(a \odot b) \otimes (c \odot d) = (a \otimes c) \odot (b \otimes d)$. Operators $\odot$ and $\otimes$ satisfying this requirement are said to *abide with each other*. The notion of *abide* has been introduced by Bird [17], and appears frequently in other situations, see Fokkinga [47].

Operator zip and the maps $exr\star$ and $exl\star$ are related as follows. First, we have on the domain of equal-length lists

$$exl\star \cdot \Upsilon \;=\; exl \tag{3.37}$$
$$exr\star \cdot \Upsilon \;=\; exr\ . \tag{3.38}$$

Second, we have the following law.

(3.39) Law

$$\Upsilon \cdot exl\star \vartriangle exr\star \;=\; id\ .$$

Proof We show that $\Upsilon \cdot exl\star \vartriangle exr\star$ is equal to the left-reduction $\mathbin{-\!\!<}\!\not\rightarrow[\,]$ using the Unique Extension Property. First, $\Upsilon\,(exl\star \vartriangle exr\star)\,[\,] = [\,]$. Applying $\Upsilon \cdot exl\star \vartriangle exr\star$ to $x \mathbin{-\!\!<} a$ gives

$$\begin{array}{ll}
 & \Upsilon\,(exl\star \vartriangle exr\star)\,(x \mathbin{-\!\!<} a)\\
= & \quad \text{definition of } \vartriangle\\
 & exl\star\,(x \mathbin{-\!\!<} a)\ \Upsilon\ exr\star\,(x \mathbin{-\!\!<} a)\\
= & \quad \text{definition of map}\\
 & (exl\star\,x \mathbin{-\!\!<} exl\ a)\ \Upsilon\ (exr\star\,x \mathbin{-\!\!<} exr\ a)\\
= & \quad \text{definition of zip}\\
 & (exl\star\,x\ \Upsilon\ exr\star\,x) \mathbin{-\!\!<} (exl\ a, exr\ a)\\
= & \quad \text{definition of } exl \text{ and } exr\\
 & (exl\star\,x\ \Upsilon\ exr\star\,x) \mathbin{-\!\!<} a\ .
\end{array}$$

Applying the Unique Extension Property we obtain $\Upsilon \cdot exl\star \vartriangle exr\star = \mathbin{-\!\!<}\!\not\rightarrow[\,] = id$. □

An immediate consequence of this law is

$$\Upsilon \cdot f\star \vartriangle g\star \;=\; (f \vartriangle g)\star\ , \tag{3.40}$$

since

$$\begin{array}{ll}
 & \Upsilon \cdot f\star \vartriangle g\star \\
= & \qquad \text{Law 2.53} \\
 & \Upsilon \cdot \mathit{exl}\star \vartriangle \mathit{exr}\star \cdot (f \vartriangle g)\star \\
= & \qquad \text{Law 3.39} \\
 & (f \vartriangle g)\star \;.
\end{array}$$

Let operator $\rightleftharpoons$ be any of the four operators $\rightharpoonup$, $\leftharpoonup$, $\hookrightarrow$, or $\hookleftarrow$. Operator $\rightleftharpoons$ and operator zip satisfy

$$(n \rightleftharpoons) \cdot \Upsilon \;=\; \Upsilon \cdot (n \rightleftharpoons) \times (n \rightleftharpoons) \;. \tag{3.41}$$

Merge

Operator merge, denoted by $⩕ : A\star \times A\star \to A\star\wr$ returns all ways in which two lists can be merged. It is characterised by

$$\begin{array}{lcl}
x ⩕ [\,] & = & \{x\} \\
[\,] ⩕ x & = & \{x\} \\
(x \dashv\!< a) ⩕ (y \dashv\!< b) & = & (\dashv\!< a)\ast\,(x ⩕ (y \dashv\!< b)) \cup (\dashv\!< b)\ast\,((x \dashv\!< a) ⩕ y) \;.
\end{array} \tag{3.42}$$

Operator merge is commutative.

3.3 Selectors

A specification is often of the form 'find the mimsiest borogove that is slithy'. Borogove is some noun, and mimsy and slithy are some adjectives, the meanings of which are not important for this discussion. Moreover, two things can be compared for mimsiness. Suppose the function *borogoves* returns the set of all borogoves. Slithy borogoves are then returned by the composition

$$\mathit{slithy}\triangleleft \cdot \mathit{borogoves} \;.$$

The mimsiest slithy borogove is obtained by reducing this set with some associative, commutative, and idempotent binary operator $\oplus$, which, given two slithy borogoves, selects the mimsier of the two. So the following composition of functions returns the mimsiest slithy borogove.

$$\oplus/ \cdot \mathit{slithy}\triangleleft \cdot \mathit{borogoves} \;.$$

Operator $\oplus$ is called a selector. De Moor [102] gives a formal method for constructing, given a preorder, functions of the form $\oplus/$, where $\oplus$ is a selector. We give the definition of selectors for some specific preorders. Let $\leq$ be a transitive relation that satisfies for all x: $x \leq x$, and for all elements x and y, either $x \leq y$ or $y \leq x$ holds, so $\leq$ is a total preorder. Then $\oplus$ is defined by

$$x \oplus y \quad = \quad \begin{cases} x & \text{if } x \leq y \\ y & \text{if } y \leq x \ . \end{cases} \tag{3.43}$$

This is not a proper definition unless $(A, \leq)$ is a total partial order, that is, if for all elements x and y in A, $x \leq y$ and $y \leq x$ imply $x = y$. Since $(nat, \leq)$ is a total partial order, the operator $\uparrow$: $nat \times nat \rightarrow nat$ defined by

$$x \uparrow y \quad = \quad \begin{cases} x & \text{if } x \leq y \\ y & \text{if } y \leq x \ , \end{cases} \tag{3.44}$$

is a well-defined selector. A selector that will be used frequently is the selector $\uparrow_{\#}$: $A\star \times A\star \rightarrow A\star$ defined by

$$x \uparrow_{\#} y \quad = \quad \begin{cases} x & \text{if } y <_{\#} x \\ y & \text{if } x <_{\#} y \\ x \odot y & \text{otherwise} \ , \end{cases} \tag{3.45}$$

for some selector $\odot$ which is used to break the tie if $x =_{\#} y$. Sometimes the precise definition of selector $\odot$ is unimportant, for example when we can infer from the context that whenever $x \uparrow_{\#} y$ occurs, x and y have different lengths or are equal. For example, when x, $y \in inits\ z$ for some z, where function *inits* returns all initial segments of a list, e.g.

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

then either $x = y$ or $x \neq_{\#} y$. A formal definition of *inits* is given in Chapter 5. However, in other cases the definition of $\uparrow_{\#}$ has to be completed. In general, if selector $\oplus$ is defined by means of a total preorder $\leq$, the relation $\leq$ has to be refined or completed to a total partial order on elements x and y with $x \neq y$ for which both $x \leq y$ and $y \leq x$ hold. Often this refinement is important, and gives rise to ingenious solutions to problems, see the nub theory developed by De Moor and Bird [103]. For example, the total preorder $\leq_{\#}$ can be refined to a total partial order by taking the lexicographically least or greatest of x and y when $x =_{\#} y$. The lexicographical order $\leq_L$ is defined by

$$x \leq_L [\,] \quad \equiv \quad x = [\,] \ ,$$

and for nonempty y

$$x \leq_L y \quad \equiv \quad x \in inits\, y \ \vee \ x <_{hd} y \ \vee \ (x =_{hd} y \ \wedge \ tl\, x \leq_L tl\, y) \ . \tag{3.46}$$

Let $f : A \to B$, where B is a totally ordered type. The selector $\uparrow_f\colon A \times A \to A$, of which $\uparrow_\#$ is an example, is defined by

$$x \uparrow_f y \quad = \quad \begin{cases} x & \text{if } y <_f x \\ y & \text{if } x <_f y \\ x \odot y & \text{otherwise} \end{cases} \quad , \tag{3.47}$$

for some selector $\odot$. The selector $\downarrow_f$ is defined similarly: replace $<$ by $>$ in the above definition. We very often reduce a set with a selector. For function $\oplus/$ we have the following result.

(3.48) Theorem *Let $\oplus$ be a selector and let x and y be finite sets such that $\forall b \in y : \exists a \in x : a \oplus b = a$. Then*

$$\oplus/\,(x \cup y) \quad = \quad \oplus/\,x \ .$$

Proof We have

$$\begin{array}{ll} & \oplus/\,(x \cup y) \\ = & \qquad \text{definition of reduction} \\ & (\oplus/\,x) \oplus (\oplus/\,y) \\ = & \qquad \text{discussion below} \\ & \oplus/\,x \ . \end{array}$$

The last step in this calculation is justified as follows. If $y = \{\,\}$, then $\oplus/\,y = \nu_\oplus$, and the equality holds by definition of $\nu_\oplus$. If $y \neq \{\,\}$, then $\oplus/\,y \in y$ since $\oplus$ is a selector, and so there exists an $a \in x$ such that $a \oplus (\oplus/\,y) = a$. For all $b \in x$ we have $\{b\} \cup x = x$, and so $\oplus/\,x \oplus b = \oplus/\,x$. It follows that

$$\begin{array}{ll} & (\oplus/\,x) \oplus (\oplus/\,y) \\ = & \qquad \oplus/\,x = (\oplus/\,x) \oplus a \\ & ((\oplus/\,x) \oplus a) \oplus (\oplus/\,y) \\ = & \qquad \oplus \text{ is associative, definition of } a \\ & (\oplus/\,x) \oplus a \\ = & \qquad (\oplus/\,x) \oplus a = \oplus/\,x \\ & \oplus/\,x \ . \end{array}$$

□

3.4 Conclusions

The building blocks of many expressions that are used in the calculations in Part II are the constructors of data types given in Chapter 2 and a small set of auxiliary functions and operators. A step in a calculation is often an application of a result for catamorphisms described in Chapter 2, or a result described in this section pertaining to one of the auxiliary functions and operators. We have now developed enough theory to calculate some non-trivial algorithms.

Part II

Specialised theories

Chapter 4

Generator fusion theorems

This chapter derives generator fusion theorems, and it applies these theorems to several example programming problems. Fusion, Corollary 2.41, gives the condition that has to be satisfied in order to fuse the composition of a function with a catamorphism into a catamorphism. Similarly, a generator fusion theorem gives sufficient conditions that have to be satisfied in order to fuse the composition of a catamorphism and a generator into a catamorphism (possibly followed by a projection function) that often can be implemented as an efficient program.

The function *borogoves*, which returns the set of all borogoves, is an example of a *generator*: it generates all borogoves. Given an element of a data type as argument, a generator is usually a polymorphic function that generates a set of elements that are related in a specific manner to the argument. An example of such a generator is the generator *perms* which, given a bag, returns the set of all permutations of the bag. A permutation of a bag x is a list y that contains the same elements as x. For example, list $[3, 2, 2, 4]$ is a permutation of bag $\langle 2, 2, 3, 4\rangle$. The essential property of function *perms* is captured in the following set-comprehension.

$$\mathit{perms}\, x \quad = \quad \{y \mid \mathit{bagify}\, y = x\} \ .$$

There exist generators that generate their elements independent of an argument, e.g. the generator that generates the set $\{2, 3, 5, 7, 11\}$, or the generator that generates all perfect numbers. A generator may be any function, but very often it is a catamorphism or the composition of a projection function with a catamorphism (a paramorphism or a mutumorphism).

The specification of the programming problems considered in this chapter, and in fact in most of the chapters to follow, is of the following form. Let L and M be data types, let $\mathit{borogoves} : L \to M\wr$ be a generator, and let $\oplus/ \cdot f* : M\wr \to A$ be a catamorphism on *set*. The generic specification reads

$$\oplus/ \cdot f* \cdot \mathit{borogoves} \ . \tag{4.1}$$

An example of a specification of this form is a specification of *sorting*. Function *sorting* takes a bag of natural numbers, and returns the list containing the elements of the bag in ascending order.

$$sorting \;=\; \Updownarrow / \cdot asc\triangleleft \cdot perms \;,$$

where operator $\Updownarrow$ is an unspecified selector. Recall that the composition of a reduction with a filter is a catamorphism, see (2.69). Many other examples matching template (4.1) can be found in the following sections and chapters.

Most of the specifications we consider can be trivially implemented. The program obtained is often very inefficient. Since there may be $n!$ different permutations of a bag with n elements, the program corresponding to the specification of *sorting* given above requires worse than exponential time to sort a bag. It is obvious that we want to find a more efficient program for sorting a bag. A possibility to obtain a more efficient program for the generic specification (4.1) is to fuse catamorphism $\oplus/ \cdot f*$ with generator *borogoves* thereby avoiding the enumeration of all possible solutions. A catamorphism $\oplus/ \cdot f*$ can be fused with generator *borogoves* provided some conditions are satisfied. The specific generator fusion theorems derived in this chapter list sufficient conditions a catamorphism has to satisfy in order to fuse it with the specific generator. The derivation of most generator fusion theorems has the following simple structure. If specification (4.1) is not a function composed with a catamorphism, for example in case the generator is the composition of a projection function with a catamorphism, rewrite it into such a form. Apply Fusion to the resulting composition, and elaborate the condition of Fusion to obtain a condition of the specific generator fusion theorem.

The history of generator fusion theorems is short. The generator fusion theorems as we describe them emanated from Horner's rule, see Bird [17]. Horner's rule gives the conditions that have to be satisfied in order to fuse the composition of a catamorphism with the generator *tails*, which returns all tails from a list, into a catamorphism. Fusion theorems for other generators are given in Jeuring [71, 72]. By definition, an optimisation problem is a problem specified as the composition of a catamorphism of which the operator of the reduction is a selector, and a generator. Dynamic programming is a strategy for solving optimisation problems. Many solutions to the programming problems we consider are therefore dynamic programming algorithms. De Moor and Bird [102, 105], present a theory for calculating dynamic programming algorithms for optimisation problems. They derive a generic theorem, which they call the Main Theorem, that gives a solution for optimisation problems of which the generator is the power transpose of a relational catamorphism. Functions *perms*, *subs*, and *parts* defined in the subsequent sections can all be defined as the power transpose of a relational catamorphism. It follows that some of the *perms*-, *subs*-, and *parts*-Fusion Theorems derived in the subsequent sections are instantiations of De Moor and Bird's Main Theorem when applied to optimisation problems. For problems other than optimisation problems, for generators that are difficult to express as the power transpose of a relational catamorphism, and for generators that can be recursively characterised in essentially different ways, we resort to generator fusion theorems. Smith [126] describes a theory for deriving generate-and-test algorithms. For a particular specification of the form described in equation (4.1), so with operator $\oplus$ and function f replaced by instantiations, the theory described in [126] can be used to construct an algorithm for the given problem. Smith does not derive the conditions we give in the generator fusion theorems. For each new problem the same sequence of steps that we use to derive a particular generator fusion theorem has to be repeated.

This chapter is organised as follows. The first section defines the generator *subs*, which returns all subsequences of a list, it derives the corresponding *subs*-Fusion Theorem, and it derives an algorithm for the Zero-One Knapsack problem. The second section defines generator *perms*, it derives the *perms*-Fusion Theorem, and using this theorem it derives merge-sort. The third section gives four recursive characterisations of generator *parts*, which returns all partitions of a list. For the first three of these it derives a fusion theorem, and for the fourth it gives a first step towards the derivation of a fusion theorem. The fourth section derives a fusion theorem for a generator defined on the data type *binary labelled tree* or *moo tree*. Finally, the fifth and last section gives some conclusions.

4.1 Subsequence problems

In this section we give the first example of the development of a theory, and we show how to use this theory in the derivation of an algorithm. It exemplifies the method we apply in the derivation of algorithms. The problems we consider in this section are problems on subsequences. A subsequence of a list is obtained by leaving some elements of the list out. For example, $[1,2,1]$ and $[4,3]$ are subsequences from $[4,2,1,3,2,1]$. The function *subs* returns the set of all subsequences of a list. It is recursively defined as a left-reduction below. Examples of problems on subsequences are the longest upsequence problem, see Gries [59], which requires to find the longest ascending subsequence of a list, and the longest common subsequence problem, see Aho, Hopcroft and Ullman [3], which requires to find, given two lists, the longest common subsequence of the lists. We can specify a class of subsequence problems as the composition of a catamorphism on *set* with function *subs*. The composition of a catamorphism on *set* with function *subs* is a left-reduction provided the constituents of the catamorphism on *set* satisfy some conditions prescribed by Fusion on *snoc-list*, Theorem 2.76.

This section has the following structure. First, it gives a definition as a left-reduction of function *subs*. Then it gives a generic specification for a class of problems involving subsequences. For this specification it derives a theorem in which the conditions are listed that have to be satisfied in order to obtain a usually efficient algorithm. Finally, using this theorem, it derives an algorithm for the Zero-One Knapsack problem.

Derivations of algorithms for other problems on subsequences can be found in Meertens [94] (a derivation of the algorithm of Hunt and Szymanski [68] for finding the longest common subsequence of two strings), in De Moor and Bird [103] (a derivation of the well-known algorithm for finding the longest upsequence, see for example Gries [59]), and Bird [20] (an algorithm for removing duplicates from a list).

The function *subs*

Function *subs* is recursively defined by

$$\begin{array}{lcl} subs & : & A\star \rightarrow A\star\wr \\ subs\,[\,] & = & \{[\,]\} \\ subs\,(x \mathbin{-\!\!\!<} a) & = & (subs\,x) \cup ((\mathbin{-\!\!\!<} a)* \,subs\,x)\ . \end{array} \qquad (4.2)$$

It follows that *subs* is a left-reduction.

$$subs \;=\; \oplus \not\rightarrow \{[\,]\}\ , \qquad (4.3)$$

where $\oplus$ is defined by $x \oplus a = x \cup ((\mathbin{-\!\!\!<} a) * x)$.

subs-problems

The generic specification of the problems involving subsequences we consider is

$$\otimes/ \cdot g* \cdot subs\ , \qquad (4.4)$$

where operator $\otimes$ and function g are arbitrary. Note that by equation (2.69) this class of subsequence problems includes all problems of the form $\otimes/ \cdot p\triangleleft \cdot subs$ for arbitrary predicate p. The number of subsequences is exponential in the length of the list, and therefore the straightforward implementation of our specification for some given operator $\otimes$ and function g requires exponential time. The specification is of the form to which Fusion on *snoc-list*, Corollary 2.76, can be applied. To apply this theorem, conditions will have to be imposed upon $\otimes$ and g. The result of the application of the Fusion on *snoc-list* Theorem is a left-reduction $\odot\not\rightarrow u$. If operator $\odot$ requires constant time for its evaluation, then this left-reduction $\odot\not\rightarrow u$ requires linear time when implemented.

Deriving a left-reduction for a *subs*-problem

Since a left-reduction can be implemented as an efficient program, provided its operator can be evaluated efficiently, we aim at finding conditions on operator $\otimes$ and function g such that the subsequence problem specified in (4.4) equals a left-reduction. Since function *subs* is a left-reduction $\oplus\not\rightarrow\{[\,]\}$, see (4.3), we apply Fusion on *snoc-list*, Corollary 2.76, to specification (4.4). We have

$$\otimes/ \cdot g* \cdot subs \;=\; \odot\not\rightarrow(\otimes/\, g*\,\{[\,]\})\ ,$$

provided operator $\odot$ satisfies $\otimes/\, g*\,(x \oplus a) = (\otimes/\, g*\, x) \odot a$ for x in the image of *subs*. For $\otimes/\, g*\,\{[\,]\}$ we have

$$\begin{array}{ll}
 & \otimes/\, g*\,\{[\,]\} \\
= & \quad \text{definition of map} \\
 & \otimes/\,\{g\,[\,]\} \\
= & \quad \text{definition of reduction} \\
 & g\,[\,]\ .
\end{array}$$

The definition of an operator $\odot$ satisfying $\otimes/\, g*\,(x \oplus a) = (\otimes/\, g*\, x) \odot a$ for x in the image of *subs* is synthesised as follows.

$$\begin{array}{ll}
 & \otimes/\, g*\,(x \oplus a) \\
= & \quad \text{definition of } \oplus \\
 & \otimes/\, g*\,(x \cup ((\mathbin{+\!\!\!<} a)*\, x)) \\
= & \quad \text{definition of catamorphism on } \textit{set} \\
 & (\otimes/\, g*\, x) \otimes (\otimes/\, g*\,(\mathbin{+\!\!\!<} a)*\, x)\ .
\end{array}$$

The left-hand argument of operator $\otimes$ is of the required form. It remains to express the right-hand argument of operator $\otimes$, the expression $\otimes/\, g*\,(\mathbin{+\!\!\!<} a)*\, x$, in terms of $\otimes/\, g*\, x$ and a. We apply Corollary 2.65 to $\otimes/ \cdot g* \cdot (\mathbin{+\!\!\!<} a)*$. Suppose that g is a left-reduction $\oslash\!\not\rightarrow\! i$ such that section $(\oslash a)$ is $(\otimes, \otimes)$-fusable after $\otimes/ \cdot g* \cdot (\in \textit{subs}\, z)\triangleleft$ for all z. Then Corollary 2.65 gives

$$\otimes/\, g*\,(\mathbin{+\!\!\!<} a)*\, x \quad = \quad (\otimes/\, g*\, x) \oslash a\ ,$$

and it follows that operator $\odot$ may be defined by

$$x \odot a \quad = \quad x \otimes (x \oslash a)\ .$$

Observe that $g\,[\,] = (\oslash\!\not\rightarrow\! i)\,[\,] = i$. The above derivation proves the following result.

(4.5) Theorem (*subs*-Fusion) *Let g be a left-reduction $\oslash\!\not\rightarrow\! i$ such that section $(\oslash a)$ is $(\otimes, \otimes)$-fusable after $\otimes/ \cdot g* \cdot (\in \textit{subs}\, z)\triangleleft$ for all z. Then*

$$\otimes/ \cdot g* \cdot \textit{subs} \quad = \quad \odot\!\not\rightarrow\! i\ ,$$

where the operator $\odot$ is defined by

$$x \odot a \quad = \quad x \otimes (x \oslash a)\ .$$

The condition section $(\oslash a)$ is $(\otimes, \otimes)$-fusable after $\otimes/ \cdot g{*} \cdot (\in \mathit{subs}\; z)\triangleleft$ amounts to showing that

$$\begin{array}{lcl} \nu_\otimes \oslash a & = & \nu_\otimes \\ (s \otimes t) \oslash a & = & (s \oslash a) \otimes (t \oslash a)\ , \end{array}$$

for elements $s = \otimes/\, g{*}\, v$, $t = \otimes/\, g{*}\, w$ such that $v, w \subseteq \mathit{subs}\; z$ for an arbitrary list z. This reduction of the proof obligation will turn out to be important in the sequel. Some proofs fundamentally depend on this observation.

Longest-p subsequence problems

We can distinguish several subclasses of subsequence problems. One of the important subclasses is the class specified by

$$\uparrow_\# / \cdot p\triangleleft \cdot \mathit{subs}\ , \tag{4.6}$$

where p is an arbitrary predicate. An element of the class of problems specified in equation (4.6) is called a *longest-p subsequence problem*. We give two examples of longest-p subsequence problems. The longest upsequence problem, see Gries [59], requires to find the longest ascending subsequence of a list. It is specified by

$$\uparrow_\# / \cdot \mathit{asc}\triangleleft \cdot \mathit{subs}\ , \tag{4.7}$$

where the predicate asc determines whether a list is ascending; see for its definition equation (2.85). As a second example we have the longest common substring problem. Let P be a list. It is required to find the longest common subsequence of P and the argument. The specification of this problem reads as follows

$$\uparrow_\# / \cdot (\in \mathit{subs}\; P)\triangleleft \cdot \mathit{subs}\ . \tag{4.8}$$

These two specifications serve as examples of longest-p subsequence problems, we will not derive algorithms for these problems.

Applying the subs-Fusion Theorem to longest-p subsequence problems

For the subclass of subsequence problems that is specified in equation (4.6) we use the *subs*-Fusion Theorem to determine sufficient properties predicate p should satisfy in order to obtain a left-reduction. By definition of guard we have

$$\uparrow_\# / \cdot p\triangleleft \;=\; \uparrow_\# / \cdot p?_{\uparrow_\#}{*}\ ,$$

Let $\omega = \nu_{\uparrow_\#}$. To apply the *subs*-Fusion Theorem we have to find conditions on predicate p, and we have to complete the definition of $\uparrow_\#$ such that

$$p?_{\uparrow_\#} \;=\; \ominus\!\not\rightarrow\! i\ , \tag{4.9}$$

for some operator $\ominus$ and some value i, such that

$$\text{section } (\ominus a) \text{ is } (\uparrow_\#, \uparrow_\#)\text{-fusable after } \uparrow_\#/ \cdot p\triangleleft \cdot (\in \mathit{subs}\, z)\triangleleft \,, \tag{4.10}$$

for all lists z. According to Theorem 2.84, condition (4.9) is satisfied if predicate p is prefix-closed. Operator $\ominus$ is then defined by

$$x \ominus a \;=\; \begin{cases} x \mathbin{-\!\!<} a & \text{if } x \neq \omega \wedge p\,(x \mathbin{-\!\!<} a) \\ \omega & \text{otherwise} \;. \end{cases} \tag{4.11}$$

For the second condition (4.10) of the *subs*-Fusion Theorem we have by definition of $\ominus$ that $\omega \ominus a = \omega$, so for the proof of the fact that $(\ominus a)$ is $(\uparrow_\#, \uparrow_\#)$-fusable it suffices to show that

$$(x \uparrow_\# y) \ominus a \;=\; (x \ominus a) \uparrow_\# (y \ominus a) \;, \tag{4.12}$$

for x and y elements of the form $\uparrow_\#/\, p\triangleleft v$, $\uparrow_\#/\, p\triangleleft w$, where $v, w \subseteq \mathit{subs}\, z$ for some z. This kind of distributivity property appears in almost all derivations we give. The equation does not hold in general: take x, y and a such that $y >_\# x$, $\neg p\,(y \mathbin{-\!\!<} a)$, and $p\,(x \mathbin{-\!\!<} a)$. We refine the definition of selector $\uparrow_\#$, and we impose sufficient conditions on p such that the distributivity property holds. If at least one of x and y equals ω, then equality (4.12) trivially holds. If both x and y are different from ω, then, since $x = \uparrow_\#/\, p\triangleleft v$ and $y = \uparrow_\#/\, p\triangleleft w$, $p\,x$ and $p\,y$ hold. Assume $x \uparrow_\# y = x$, the other case being symmetrical. Then $(x \uparrow_\# y) \ominus a = x \ominus a$. We prove that $(x \ominus a) \uparrow_\# (y \ominus a) = x \ominus a$, provided the definition of selector $\uparrow_\#$ is completed for the case $x =_\# y$, and predicate p satisfies a specific property.

$$\begin{array}{ll}
 & (x \ominus a) \uparrow_\# (y \ominus a) = x \ominus a \\
\Leftarrow & \quad \textbf{assume } (x \uparrow_\# y) \mathbin{-\!\!<} a = (x \mathbin{-\!\!<} a) \uparrow_\# (y \mathbin{-\!\!<} a), \text{ definition of } \ominus \\
 & y \ominus a \neq \omega \;\Rightarrow\; x \ominus a \neq \omega \\
= & \quad x, y \neq \omega, \text{ definition } \ominus \\
 & p\,(y \mathbin{-\!\!<} a) \;\Rightarrow\; p\,(x \mathbin{-\!\!<} a) \\
= & \quad \textbf{assume } (p\,x \;\wedge\; p\,y) \;\Rightarrow\; (p\,(x \mathbin{-\!\!<} a) \;\equiv\; p\,(y \mathbin{-\!\!<} a)) \\
 & \mathit{true} \;.
\end{array}$$

So if

$$(x \uparrow_\# y) \mathbin{-\!\!<} a \quad = \quad (x \mathbin{-\!\!<} a) \uparrow_\# (y \mathbin{-\!\!<} a) \tag{4.13}$$

$$(p\,x \;\wedge\; p\,y) \quad \Rightarrow \quad (p\,(x \mathbin{-\!\!<} a) \;\equiv\; p\,(y \mathbin{-\!\!<} a)) \;, \tag{4.14}$$

then section $(\ominus a)$ is $(\uparrow_\#, \uparrow_\#)$-fusable after $\uparrow_\#/ \cdot p\triangleleft \cdot (\in \mathit{subs}\, z)\triangleleft$ for all lists z.

We discuss the two conditions (4.13) and (4.14). For condition (4.13) we have the following. Recall the definition of lexicographic order $\leq_L$ (3.46): $x \leq_L [\,] \;\equiv\; x = [\,]$, and if $y \neq [\,]$

$$x \leq_L y \;\;\equiv\;\; x \in \mathit{inits}\, y \;\vee\; x <_{hd} y \;\vee\; (x =_{hd} y \;\wedge\; \mathit{tl}\, x \leq_L \mathit{tl}\, y) \;.$$

Ordering $\leq_L$ is anti-symmetric. If $x =_{\#} y$, then we define $\uparrow_{\#}$ by

$$x \uparrow_{\#} y \quad = \quad \begin{cases} x & \text{if } x \leq_L y \\ y & \text{if } y \leq_L x \ . \end{cases}$$

So if the arguments of $\uparrow_{\#}$ are of equal length, $\uparrow_{\#}$ returns the lexicographically lesser of the two. For this selector $\uparrow_{\#}$, condition (4.13) is satisfied. A predicate satisfying condition (4.14) is called *extension-equivalent.*

We have proved the following corollary of the *subs*-Fusion Theorem.

(4.15) Corollary *Let p be a prefix-closed and extension-equivalent predicate, and suppose $\uparrow_{\#}$ takes the lexicographically lesser of two equal-length arguments. Then*

$$\uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{subs} \quad = \quad \odot\!\not\rightarrow\![\,] \ ,$$

where operator $\odot$ is defined by

$$x \odot a \quad = \quad x \uparrow_{\#} (x \ominus a) \ ,$$

where operator $\ominus$ is defined in equality (4.11).

Extension-equivalence is a strong condition on predicates. Both *asc* and $\in\ \mathit{subs}\, P$, the predicates appearing in examples (4.7) and (4.8), are not extension-equivalent. An example of an extension-equivalent predicate is the predicate $(= C)$ for some constant C.

Example: the Zero-One Knapsack problem

A criminal is sentenced to twenty years of solitary imprisonment. She is allowed to take some books from her library with her. These books should fit on the shelf, of some given width W, in her cell. The criminal wants to make a selection of the books in the library that fits on this shelf and that will please her most. This problem is known as the Zero-One Knapsack problem or the Zero-One Integer Programming Problem, see for example Garey and Johnson [55]. It is an NP-complete problem. We derive an $O(n \times W)$ algorithm solving the problem, where n is the number of books in the library. Note that $W = 2^{2\log W}$, and $2\log W$ is the length of input W, and it follows that the straightforward implementation of the algorithm requires time exponential in the length of input W.

The library is represented by a list of books, where a book is a pair of natural numbers, modelling respectively the expected pleasure obtained from it, and its width. The selection of books that will fit on the shelf and pleases most is a subsequence of the given list, the sum of the second components of which does not exceed W, and the sum of the first components of which is maximal. Hence the problem is specified by

$$\uparrow_{\mathit{pleasure}}/ \cdot ((\leq\ W) \cdot \mathit{width})\triangleleft \cdot \mathit{subs} \ , \tag{4.16}$$

where both *width* and *pleasure* are left-reductions defined by

$$\begin{aligned} \mathit{width} &= (+ \cdot id \times exr) \not\rightarrow 0 & (4.17)\\ \mathit{pleasure} &= (+ \cdot id \times exl) \not\rightarrow 0\ . & (4.18) \end{aligned}$$

Abbreviate $\uparrow_{pleasure}$ to $\uparrow$.

The derivation of an efficient algorithm for this problem is structured as follows. First, we show that the components of the specification do not meet the conditions of the *subs*-Fusion Theorem. Then we give an extended specification. The extended specification is rewritten to a form to which the *subs*-Fusion Theorem can be applied (the composition of a catamorphism on *set* with the function *subs*), and finally, we show that the *subs*-Fusion Theorem is applicable.

We rewrite the specification into a form to which we can apply *subs*-Fusion, Theorem 4.5. By definition of filter and guard we have, see equation (2.69),

$$\uparrow/ \cdot ((\leq W) \cdot \mathit{width}) \triangleleft \cdot \mathit{subs} \;=\; \uparrow/ \cdot ((\leq W) \cdot \mathit{width})?_{\uparrow} * \cdot \mathit{subs}\ ,$$

Let $\omega = \nu_{\uparrow}$. We try to apply the *subs*-Fusion Theorem to the right-hand side of this equality. Since $(\leq W) \cdot \mathit{width}$ is prefix-closed, Theorem 2.84 gives a left-reduction $\ominus \not\rightarrow [\,]$ for $((\leq W) \cdot \mathit{width})?_{\uparrow}$, where operator $\ominus$ is defined by

$$x \ominus a \;=\; \begin{cases} x \dashv\!\!< a & \text{if } x \neq \omega \;\wedge\; \mathit{width}\,(x \dashv\!\!< a) \leq W \\ \omega & \text{otherwise}\ . \end{cases}$$

The second condition of the *subs*-Fusion Theorem requires $(\ominus a)$ to be $(\uparrow, \uparrow)$-fusable. However, if we take x, y, and a such that $y >_{pleasure} x$, $\mathit{width}\,(y \dashv\!\!< a) > W$, and $\mathit{width}\,(x \dashv\!\!< a) \leq W$ we find that

$$(x \uparrow y) \ominus a \neq (x \ominus a) \uparrow (y \ominus a)\ ,$$

and hence that $(\ominus a)$ is not $(\uparrow, \uparrow)$-fusable.

Since the *subs*-Fusion Theorem is not directly applicable, we have to think of an extension of the problem. It is here that we have to be inventive. The generalisation step we make is an application of the 'tupling strategy'.

Note that $\mathit{width}\,(x \dashv\!\!< a) \leq W$ is equivalent to $\mathit{width}\, x \leq W - exr\, a$. Suppose we return, instead of the most pleasant subsequence with width not exceeding W, the most pleasant subsequences with width equal to i for all i with $0 \leq i \leq W$. The most pleasant subsequence with width not exceeding W can be found easily given these $W + 1$ subsequences. The new specification of the problem for which we want to find an efficient algorithm reads as follows.

$$(\uparrow/ \cdot q_0 \triangleleft) \vartriangle \ldots \vartriangle (\uparrow/ \cdot q_W \triangleleft) \cdot \mathit{subs}\ , \qquad (4.19)$$

where the predicate q_i with $0 \leq i \leq W$ is an abbreviation for the predicate $(= i) \cdot \mathit{width}$. We want to specify this new problem as the composition of a catamorphism with the function *subs*, so that we can apply the *subs*-Fusion Theorem. Applying instantiation (2.59) of the Tupling catamorphisms Corollary, Corollary 2.38, repeatedly we obtain a catamorphism for the tuple of catamorphisms $(\uparrow/ \cdot q_0\triangleleft) \vartriangle \ldots \vartriangle (\uparrow/ \cdot q_W\triangleleft)$, that is,

$$(\uparrow/ \cdot q_0\triangleleft) \vartriangle \ldots \vartriangle (\uparrow/ \cdot q_W\triangleleft) \quad = \quad \underset{W}{\uparrow}/ \cdot (q_0?_{\uparrow} \vartriangle \ldots \vartriangle q_W?_{\uparrow})* \ ,$$

where operator $\underset{W}{\uparrow}$ is defined on pairs of $W{+}1$-tuples by

$$(a_0, \ldots, a_W) \underset{W}{\uparrow} (b_0, \ldots, b_W) \quad = \quad (a_0 \uparrow b_0, \ldots, a_W \uparrow b_W) \ .$$

Informally speaking, $\underset{W}{\uparrow}$ is the zip with operator $\uparrow$ on $W{+}1$-tuples. Operator $\underset{W}{\uparrow}$ satisfies for all i with $0 \leq i \leq W$ the equation $\pi_i\,(a \underset{W}{\uparrow} b) = (\pi_i\, a) \uparrow (\pi_i\, b)$. Note furthermore that the unit of $\underset{W}{\uparrow}$ is the $W{+}1$-tuple $(\omega, \ldots, \omega)$. It follows that for all i with $0 \leq i \leq W$, function π_i is $(\underset{W}{\uparrow}, \uparrow)$-fusable. Operator $\underset{W}{\uparrow}$ is not a selector, since $x \underset{W}{\uparrow} y$ is not necessarily either x or y, but this is no problem if we apply the *subs*-Fusion Theorem. Operator $\underset{W}{\uparrow}$ is associative, commutative, and idempotent since $\uparrow$ is associative, commutative, and idempotent. To apply the *subs*-Fusion Theorem, In the sequel we use the following abbreviation.

$$qq \quad = \quad q_0?_{\uparrow} \vartriangle \ldots \vartriangle q_W?_{\uparrow} \ .$$

From the above we find that specification (4.19) is equal to

$$\underset{W}{\uparrow}/ \cdot qq* \cdot subs \ .$$

We apply the *subs*-Fusion Theorem. We prove that function qq is a left-reduction. We have $q_0?_{\uparrow}\,[\,] = [\,]$, and for all i with $1 \leq i \leq W$ we have $q_i?_{\uparrow}\,[\,] = \omega$. For the nonempty case we have for all i with $0 \leq i \leq W$

$$q_i?_{\uparrow}\,(x \mathbin{+\!\!\!<} a) \quad = \quad \begin{cases} (q_{i-(exr\ a)}?_{\uparrow}\, x) \mathbin{+\!\!\!<} a & \text{if } q_{i-(exr\ a)}?_{\uparrow}\, x \neq \omega \\ \omega & \text{otherwise} \ , \end{cases}$$

where it has been supposed that for all x, and for all $i < 0$, $\neg q_i\, x$. It follows that for all i with $0 \leq i \leq W$, function $q_i?_{\uparrow}$ is catamorphic modulo the functions $q_j?_{\uparrow}$, with $0 \leq j \leq W$.

Define for i with $0 \leq i \leq W$ operator $\ominus_i$ by

$$x \ominus_i a \quad = \quad \begin{cases} (\pi_{i-(exr\ a)}\, x) \mathbin{+\!\!\!<} a & \text{if } \pi_{i-(exr\ a)}\, x \neq \omega \\ \omega & \text{otherwise} \ . \end{cases}$$

Applying Theorem 2.47 repeatedly we obtain

$$qq \quad = \quad \ominus\!\not\rightarrow([\,], \omega, \ldots, \omega) \ ,$$

where operator $\ominus$ is defined by

$$x \ominus a \quad = \quad (x \ominus_0 a, \ldots, x \ominus_W a) \ . \tag{4.20}$$

It follows that the first condition of the *subs*-Fusion Theorem is satisfied. The second condition requires $(\ominus a)$ to be $(\underset{W}{\uparrow}, \underset{W}{\uparrow})$-fusable on the image of $\underset{W}{\uparrow}/ \cdot qq* \cdot (\in subs\ z)\triangleleft$ for all lists z.

It is easy to verify that $(\omega,\ldots,\omega)\ominus a=(\omega,\ldots,\omega)$. The remaining proof obligation is

$$(x \underset{W}{\updownarrow} y)\ominus a \quad = \quad (x\ominus a)\underset{W}{\updownarrow}(y\ominus a)\ ,$$

for x and y in the image of $\underset{W}{\updownarrow}/\cdot qq\!*\cdot(\in \mathit{subs}\ z)\triangleleft$ for some list z. This equality follows immediately if we can prove for all i with $0\le i\le W$ that

$$\pi_i\,((x \underset{W}{\updownarrow} y)\ominus a) \quad = \quad \pi_i\,((x\ominus a)\underset{W}{\updownarrow}(y\ominus a))\ . \tag{4.21}$$

We prove equation (4.21).

$$\begin{array}{ll}
 & \pi_i\,((x \underset{W}{\updownarrow} y)\ominus a)\\
= & \qquad \text{definition of } \ominus \text{ (4.20)}\\
 & (x \underset{W}{\updownarrow} y)\ominus_i a\\
= & \qquad (\ominus_i a) \text{ is } (\underset{W}{\updownarrow},\uparrow)\text{-fusable, proved below}\\
 & (x\ominus_i a)\uparrow(y\ominus_i a)\\
= & \qquad \text{definition of } \ominus \text{ (4.20)}\\
 & \pi_i\,(x\ominus a)\uparrow\pi_i\,(y\ominus a)\\
= & \qquad \pi_i \text{ is } (\underset{W}{\updownarrow},\uparrow)\text{-fusable}\\
 & \pi_i\,((x\ominus a)\underset{W}{\updownarrow}(y\ominus a))\ .
\end{array}$$

It follows that $(\ominus a)$ is $(\underset{W}{\updownarrow},\underset{W}{\updownarrow})$-fusable after $\underset{W}{\updownarrow}/\cdot qq\!*\cdot(\in \mathit{subs}\ z)\triangleleft$ for all lists z if $(\ominus_i a)$ is $(\underset{W}{\updownarrow},\uparrow)$-fusable after $\underset{W}{\updownarrow}/\cdot qq\!*\cdot(\in \mathit{subs}\ z)\triangleleft$ for all lists z. We prove that $(\ominus_i a)$ is $(\underset{W}{\updownarrow},\uparrow)$-fusable after $\underset{W}{\updownarrow}/\cdot qq\!*\cdot(\in \mathit{subs}\ z)\triangleleft$ for all lists z.

By definition of $\ominus_i$, the equality $(\omega,\ldots,\omega)\ominus_i a=\omega$ holds. Hence it suffices to show that

$$(x \underset{W}{\updownarrow} y)\ominus_i a \quad = \quad (x\ominus_i a)\uparrow(y\ominus_i a)\ . \tag{4.22}$$

for x and y in the image of $\underset{W}{\updownarrow}/\cdot qq\!*\cdot(\in \mathit{subs}\ z)\triangleleft$ for some list z. Consider the definition of operator $\ominus_i$. If at least one of $\pi_{i-(\mathit{exr}\ a)}\,x$ and $\pi_{i-(\mathit{exr}\ a)}\,y$ equals ω, say $\pi_{i-(\mathit{exr}\ a)}\,y=\omega$, we have $y\ominus_i a=\omega$, and hence $(x\ominus_i a)\uparrow(y\ominus_i a)=x\ominus_i a$. The value of the left-hand side of equation (4.22) depends on the value of $\pi_{i-(\mathit{exr}\ a)}\,(x \underset{W}{\updownarrow} y)$. Calculate as follows.

$$\begin{array}{ll}
 & \pi_{i-(\mathit{exr}\ a)}\,(x \underset{W}{\updownarrow} y)\\
= & \qquad \pi_j \text{ is } (\underset{W}{\updownarrow},\uparrow)\text{-fusable}\\
 & \pi_{i-(\mathit{exr}\ a)}\,x\uparrow\pi_{i-(\mathit{exr}\ a)}\,y\\
= & \qquad \text{case assumption}\\
 & \pi_{i-(\mathit{exr}\ a)}\,x\ .
\end{array}$$

If $\pi_{i-(\mathit{exr}\ a)}\,x=\omega$ too, then the left-hand side of equation (4.22) equals ω, and so does $x\ominus_i a$, the right-hand side of equation (4.22). If $\pi_{i-(\mathit{exr}\ a)}\,x\neq\omega$, then the left-hand side of equation (4.22) equals $\pi_{i-(exr\ a)}\ x \mathbin{+\!\!\!\prec} a$, and so does $x \ominus_i a$, the right-hand side of equation (4.22). In case both $\pi_{i-(exr\ a)}\ x$ and $\pi_{i-(exr\ a)}\ y$ are unequal to ω we reason as follows. Note that for all x and y in the image of $\underset{\mathrm{W}}{\uparrow}/ \cdot qq* \cdot (\in subs\ z)\triangleleft$ for some list z such that $\pi_j\ x \neq \omega \neq \pi_j\ y$ the following equation holds.

$$(\pi_j\ x \uparrow \pi_j\ y) \mathbin{+\!\!\!\prec} a \quad = \quad (\pi_j\ x \mathbin{+\!\!\!\prec} a) \uparrow (\pi_j\ y \mathbin{+\!\!\!\prec} a)\ , \tag{4.23}$$

provided for all lists v and w and all elements c

$$v =_{pleasure} w \quad \Rightarrow \quad (v \uparrow w) \mathbin{+\!\!\!\prec} c = (v \mathbin{+\!\!\!\prec} c) \uparrow (w \mathbin{+\!\!\!\prec} c)\ .$$

This implication holds if we define operator $\uparrow$ as follows.

$$x \uparrow y \quad = \quad \begin{cases} x & \text{if } x >_{pleasure} y \ \vee\ (x =_{pleasure} y \ \wedge\ exr{*}\,x \leq_L exr{*}\,y) \\ y & \text{otherwise}\ . \end{cases}$$

Using equation (4.23) we calculate as follows.

$$\begin{array}{ll}
 & (x \underset{\mathrm{W}}{\uparrow} y) \ominus_i a \\
= & \qquad \text{case assumption} \\
 & \pi_{i-(exr\ a)}\ (x \underset{\mathrm{W}}{\uparrow} y) \mathbin{+\!\!\!\prec} a \\
= & \qquad \pi_i \text{ is } (\underset{\mathrm{W}}{\uparrow}, \uparrow)\text{-fusable} \\
 & (\pi_{i-(exr\ a)}\ x \uparrow \pi_{i-(exr\ a)}\ y) \mathbin{+\!\!\!\prec} a \\
= & \qquad \text{equation (4.23)} \\
 & (\pi_{i-(exr\ a)}\ x \mathbin{+\!\!\!\prec} a) \uparrow (\pi_{i-(exr\ a)}\ y \mathbin{+\!\!\!\prec} a) \\
= & \qquad \text{definition of } \ominus_i\text{, case assumption} \\
 & (x \ominus_i a) \uparrow (y \ominus_i a)\ .
\end{array}$$

This concludes the proof of equation (4.22).

Since all the requirements of the *subs*-Fusion Theorem are satisfied, we have that

$$\underset{\mathrm{W}}{\uparrow}/ \cdot qq* \cdot subs \quad = \quad \odot\!\not\to([\,], \omega, \ldots, \omega)\ ,$$

where

$$x \odot a \quad = \quad x \underset{\mathrm{W}}{\uparrow} (x \ominus a)\ .$$

When the left-reduction $\odot\!\not\to([\,], \omega, \ldots, \omega)$ is implemented, we obtain a $O(n \times W)$ algorithm. On a list of length n, operator $\odot$ is evaluated n times and each evaluation requires time W. Variants of the algorithm presented here are described in Schrijver [119]. A derivation in the Bird-Meertens calculus of an algorithm for the same problem without the restriction that volumes are natural numbers, using backtracking and branch-and-bound, is presented in Fokkinga [45].

4.2 Permutation problems

Permutation problems are problems that are specified by means of function *perms* which returns the set of all permutations of a bag. A permutation of a bag is obtained by imposing an order on its elements, so $[4,2,3,1]$ is an element of $perms\,\langle 1,2,3,4\rangle$. Probably the most well-known permutation problem is sorting, which requires to find an ascending permutation of a bag. Another well-known example of a permutation problem is the Travelling Salesman Problem: given a bag of cities, determine the shortest route that visits all cities, that is, determine an order on the cities such that the sum of the distances between consecutive cities is minimal. The class of permutation problems we discuss is specified as the composition of a catamorphism on *set* with function *perms*. The composition of a catamorphism on *set* with function *perms* is a catamorphism on *bag* provided the constituents of the catamorphism on *set* satisfy some conditions prescribed by the Fusion on *bag* Theorem.

This section has the following structure. First, it defines function *perms* as a catamorphism on *bag*. Then it specifies a class of permutation problems, and for this specification it derives the so-called *perms*-Fusion Theorem in which the conditions that have to be satisfied in order to obtain a usually efficient program for a problem involving permutations are listed. Finally, using this theorem, it derives merge-sort.

The function *perms*

The function *perms* returns the set of all permutations of a bag. Function *perms* can also be defined on the data type list, but defining it on *bag* seems to be more natural. It is defined elegantly as the functional inverse of the function *bagify* defined in (2.58) which turns a join-list into a bag:

$$perms\,x \quad = \quad \{y \mid bagify\,y = x\}\ . \tag{4.24}$$

Function *perms* is recursively characterised as a catamorphism on *bag* by

$$\begin{array}{lcl} perms & : & A\varpi \to A{*}\wr \\ perms\,\langle\,\rangle & = & \{[\,]\} \\ perms\,\rho a & = & \{[a]\} \\ perms\,(x \uplus y) & = & \cup/\,(perms\,x \between_{\wedge\!\!\!\wedge}\, perms\,y)\ . \end{array} \tag{4.25}$$

The operator merge $\wedge\!\!\!\wedge$ is defined in equation (3.42). It follows that *perms* is a bag-catamorphism

$$perms \quad = \quad \oplus/\cdot(\sigma\cdot\tau){*}\ , \tag{4.26}$$

where operator $\oplus$ is defined by $x \oplus y = \cup/\,(x \between_{\wedge\!\!\!\wedge}\, y)$. For this catamorphism on *bag* to be well-defined, operator $\oplus$ should be associative and commutative. The proofs of these facts are left to the reader.

perms-problems

The class of permutation problems we consider is specified by

$$\otimes/ \cdot g* \cdot \mathit{perms}\ , \tag{4.27}$$

where operator $\otimes$ and function g are arbitrary. Given a bag with n elements, there are $n!$ permutations of this bag (if all elements are different; if there are equal elements the number of permutations decreases), and hence the straightforward implementation of this specification with $\otimes$ and g instantiated to some operator and function is a very inefficient program.

Deriving a catamorphism on *bag* for a *perms*-problem

Since a catamorphism on *bag* can be implemented as an efficient program, provided its components can be evaluated efficiently, we aim at finding conditions on operator $\otimes$ and function g such that the *perms*-problem specified in equation (4.27) equals a catamorphism on *bag*. Since *perms* is a catamorphism on *bag*, see (4.26), we can apply Fusion on *bag*, the version on the data type *bag* of Corollary 2.62, to the specification. We have

$$\otimes/ \cdot g* \cdot \mathit{perms} \quad = \quad \oslash/ \cdot (g \cdot \tau)*\ ,$$

provided $\otimes/ \cdot g*$ is $(\oplus, \oslash)$-fusable after *perms*, for some operator $\oslash$. This condition is elaborated as follows.

$$\begin{array}{ll}
& \otimes/\ g*\,(x \oplus y) \\
= & \qquad \text{definition } \oplus \\
& \otimes/\ g* \cup/\ (x\ \mathsf{X}_{⩕}\ y) \\
= & \qquad \text{Set Fusion} \\
& \otimes/\ (\otimes/ \cdot g*)*\,(x\ \mathsf{X}_{⩕}\ y) \\
= & \qquad \text{Cross Law 3.27} \\
& \otimes/\ (x\ \mathsf{X}_{\otimes/\cdot g*\cdot ⩕}\ y)\ .
\end{array}$$

To proceed with the calculation we assume that there exists an operator $\oslash$ such that

$$\otimes/ \cdot g* \cdot ⩕ \quad = \quad \oslash \cdot g \times g\ ,$$

and such that for all a and b, section $(\oslash a)$ is $(\otimes, \otimes)$-fusable after $\otimes/ \cdot g* \cdot (\in \mathit{perms}\ z)\triangleleft$ for all bags z, and section $(b\oslash)$ is $(\otimes, \otimes)$-fusable after $\otimes/ \cdot g* \cdot (\in \mathit{perms}\ w)\triangleleft$ for all bags w. Then

$$\begin{array}{ll}
& \otimes/\ (x\ \mathsf{X}_{\otimes/\cdot g*\cdot ⩕}\ y) \\
= & \qquad \text{assumption}
\end{array}$$

$$\begin{array}{ll} & \otimes/\,(x \;\mathbin{X}_{\oslash\cdot g\times g}\; y) \\ = & \quad \text{Cross Law 3.28} \\ & \otimes/\,(g{*}\,x \;\mathbin{X}_{\oslash}\; g{*}\,y) \\ = & \quad \text{Cross Law 3.31, } \textbf{assume} \text{ sections } (\oslash a) \text{ and } (b\oslash) \text{ are } (\otimes,\otimes)\text{-fusable} \\ & (\otimes/\,g{*}\,x) \oslash (\otimes/\,g{*}\,y)\ . \end{array}$$

This concludes the derivation of a catamorphism on *bag* for the specification (4.27). We have proved the *perms*-Fusion Theorem.

(4.28) Theorem (*perms*-Fusion) *Let $\otimes/\cdot g*$ be a catamorphism on* set *such that $\otimes/\cdot g*\cdot \mathbin{\wedge\!\!\wedge} = \oslash\cdot g\times g$ for some operator $\oslash$ such that for all a and b, sections $(\oslash a)$ and $(b\oslash)$ are $(\otimes,\otimes)$-fusable after $\otimes/\cdot g*\cdot(\in perms\, z)\triangleleft$ for all bags z. Then*

$$\otimes/\cdot g*\cdot perms \quad=\quad \oslash/\cdot(g\cdot\tau)*\ .$$

Example: sorting

We want to derive an algorithm for sorting a bag. This problem is specified by

$$sorting \quad=\quad \Updownarrow/\cdot sorted\triangleleft\cdot perms\ , \tag{4.29}$$

where the predicate *sorted* determines whether a list is sorted; on the type list of natural numbers it might be defined as the predicate *asc*, see (2.85), and operator $\Updownarrow$ is an unspecified selector.

To apply the *perms*-Fusion Theorem to the sorting problem we have to exhibit an operator $\oslash$ such that for all a and b, sections $(\oslash a)$ and $(b\oslash)$ are $(\Updownarrow,\Updownarrow)$-fusable after $\Updownarrow/\cdot sorted\triangleleft\cdot$ $(\in perms\, z)\triangleleft$ for all bags z, and such that

$$\Updownarrow/\cdot sorted\triangleleft\cdot \mathbin{\wedge\!\!\wedge} \quad=\quad \oslash\cdot sorted?_{\Updownarrow}\times sorted?_{\Updownarrow}\ . \tag{4.30}$$

Intuitively it is clear what operator $\oslash$ should do: operator $\oslash$ takes two lists, which are either equal to $\nu_{\Updownarrow}$ or are sorted, and it returns $\nu_{\Updownarrow}$ if one of its arguments is equal to $\nu_{\Updownarrow}$, and else it returns the sorted merge of the lists. Define operator $\oslash$ by $[\,]\oslash x = x\oslash[\,] = x$, and

$$(x \mathbin{-\!\!<} a)\oslash(y \mathbin{-\!\!<} b) \quad=\quad \begin{cases} (x\oslash(y \mathbin{-\!\!<} b)) \mathbin{-\!\!<} a & \text{if } a\ge b \\ ((x \mathbin{-\!\!<} a)\oslash y) \mathbin{-\!\!<} b & \text{otherwise}\ , \end{cases} \tag{4.31}$$

and let $\nu_{\Updownarrow}$ be a zero of operator $\oslash$. This operator $\oslash$ satisfies the requirements imposed by the application of the *perms*-Fusion Theorem. Let $x = \Updownarrow/\ sorted\triangleleft v$ and $y = \Updownarrow/\ sorted\triangleleft w$, where $v, w \subseteq perms\, z$ for some bag z. Since the sorted list among the lists of $perms\, z$ is unique (here we assume that $\leq$ is a partial order), $x = \nu_{\Updownarrow}$ or $y = \nu_{\Updownarrow}$ or $x = y$. It follows immediately that

$$\begin{array}{rcl} (x \Updownarrow y) \oslash a & = & (x \oslash a) \Updownarrow (y \oslash a) \\ b \oslash (x \Updownarrow y) & = & (b \oslash x) \Updownarrow (b \oslash y) \; , \end{array}$$

so sections $(\oslash a)$ and $(b \oslash)$ are $(\Updownarrow, \Updownarrow)$-fusable after $\Updownarrow/\, \mathit{sorted} \triangleleft \cdot (\in \mathit{perms}\; z) \triangleleft$ for all bags z, for all a and b, indeed. The last proof obligation, equation (4.30), is verified as follows. If at least one of the lists x and y is not sorted, then $(\mathit{sorted?}_{\Updownarrow}\, x) \oslash (\mathit{sorted?}_{\Updownarrow}\, y) = \nu_{\Updownarrow}$. Furthermore, by definition of $\mathbin{/\!\!\wedge\!\!\backslash}$, if at least one of the lists x and y is not sorted, then all elements of $x \mathbin{/\!\!\wedge\!\!\backslash} y$ are not sorted, and hence, $\Updownarrow/\, \mathit{sorted} \triangleleft (x \mathbin{/\!\!\wedge\!\!\backslash} y) = \nu_{\Updownarrow}$. If both lists x and y are sorted, then $(\mathit{sorted?}_{\Updownarrow}\, x) \oslash (\mathit{sorted?}_{\Updownarrow}\, y) = \Updownarrow/\, \mathit{sorted} \triangleleft (x \mathbin{/\!\!\wedge\!\!\backslash} y)$.

Observe furthermore that $\mathit{sorted?}_{\Updownarrow} \cdot \tau = \tau$. Applying the *perms*-Fusion Theorem we obtain

$$\mathit{sorting} \;=\; \oslash/ \cdot \tau* \; .$$

This algorithm is also known as merge-sort, and was mentioned as early as 1945 by John von Neumann, see Knuth [83].

Other sorting algorithms can be derived by varying the way of enumerating permutations in (i.e. the recursive characterisation of) function *perms*. For example, if *perms* is characterised as a left-reduction, like function *subs* in Section 4.1, applying a similar derivation yields insertion-sort.

Sorting algorithms are often used to exemplify programming methodologies. Darlington [34] gives transformational developments of several sorting algorithms using the fold-unfold technique. The transformation rules applied there are much more low-level than the ones we apply, and therefore the derivations tend to get much longer. Furthermore, no theory (such as the *perms*-Fusion Theorem) is developed. Using a general technique for deriving divide-and-conquer algorithms, Smith [124] reports a derivation of a sorting algorithm. Again, the abstract formulation of the applicability conditions we give in the *perms*-Fusion Theorem is not present there. Finally, using the deductive synthesis framework developed by Manna and Waldinger, Traugott [131] derives a number of sorting algorithms.

4.3 Partition problems

Partition problems are problems that are specified by means of function *parts* computing all partitions of a list. A partition of a list x is a list of lists which equals x when glued together. For example, $[[3,1],[2,4]]$ is a partition of the list $[3,1,2,4]$. Function *parts* returns the set of all partitions of a list. Examples of partition problems can be found in many branches of Computing Science: data compression by means of textual substitution is a partition problem, text-formatting is a partition problem, and many operations research problems, such as the equipment replacement problem, are partition problems. The number of partitions of a list is exponential in the length of the list, and therefore the straightforward implementation of an instantiation of the class of partition problems specified by

$$pp \quad = \quad \otimes/ \cdot g{*} \cdot \mathit{parts} \ , \tag{4.32}$$

is an exponential-time program. This section is divided in five subsections. The first subsection presents a number of recursive characterisations of function *parts* on both the data type *join-list* and the data type *snoc-list*. The second subsection states a *parts*-Fusion Theorem on the data type *snoc-list*. The derivation of this theorem is similar to the derivation of the *subs*-Fusion Theorem and the *perms*-Fusion Theorem. Furthermore, the second subsection derives a corollary of the *parts*-Fusion Theorem that may be of help to find solutions for problems of the form: find the shortest partition of which all elements satisfy predicate p of a list. It illustrates this result with the derivation of an algorithm for the problem of finding the smallest square circumscribing a text. The third subsection derives another *parts*-Fusion Theorem on the data type *snoc-list*. The fourth subsection derives a *parts*-Fusion Theorem on the data type *join-list* and gives a corollary of this theorem. Finally, the fifth subsection discusses the derivation of another *parts*-Fusion Theorem on *join-list*.

Derivations of other problems involving partitions can be found in Bird [15, 18, 19] (in the first two papers theory for partitions is developed by means of which algorithms for text-formatting and run-length encoding are derived, and in the third paper the function *parts* plays an important role in the specification of a problem), Fokkinga [42] (derivations of three optimal partition algorithms, namely, Bird's Leery Theorem and Greedy Theorem, and of Van der Woude's Gluttonous Theorem), De Moor and Bird [104] (derivations of dynamic programming algorithms), and Van der Woude [139] (derivation of the Gluttonous Theorem).

4.3.1 Characterisations of *parts*

Function *parts* returns the set of all partitions of a list. It is defined by means of the following set-comprehension.

$$\mathit{parts}\, x \quad = \quad \{y \mid {+\!\!+}/\, y = x \ \wedge \ \mathit{all}\,(\neq [\,])\, y\} \ . \tag{4.33}$$

Function *parts* can be recursively characterised in various ways. In this thesis we will see four different characterisations of *parts*. The characteristics of the specific problem at hand guide the choice of characterisation of *parts*.

The function *parts* on *snoc-list*

We want to characterise function *parts* as a left-reduction. In general, constructing a left-reduction for some function f amounts to trying to express $f\,(x \mathbin{-\!\!\!<} a)$ in terms of $f\,x$. The following characterisation of function *parts* is obtained if we try to express $parts\,(x \mathbin{-\!\!\!<} a)$ in terms of $parts\,x$ using definition (4.33) of *parts*.

$$\begin{array}{lcl} parts & : & A\star \to A\star\star\wr \\ parts\,[\,] & = & \{[\,]\} \\ parts\,(x \mathbin{-\!\!\!<} a) & = & \cup/\,(\Psi\,a)\ast parts\,x\ , \end{array} \tag{4.34}$$

where the binary operator Ψ is defined by

$$\begin{array}{lcl} \Psi & : & A\star\star \times A \to A\star\star\wr \\ [\,]\Psi a & = & \{[[a]]\} \\ (zs \mathbin{-\!\!\!<} z)\Psi a & = & \{zs \mathbin{-\!\!\!<} (z \mathbin{-\!\!\!<} a), (zs \mathbin{-\!\!\!<} z) \mathbin{-\!\!\!<} [a]\}\ . \end{array}$$

It follows that *parts* is a left-reduction $\oplus\not\rightarrow\{[\,]\}$ where operator $\oplus$ is defined by $x \oplus a = \cup/\,(\Psi\,a)\ast x$. A second characterisation of function *parts* on *snoc-list* is obtained by expressing $parts\,(x \mathbin{-\!\!\!<} a)$ in terms of $parts\,x$ in another, more complicated, way.

$$\begin{array}{lcl} parts & : & A\star \to A\star\star\wr \\ parts\,[\,] & = & \{[\,]\} \\ parts\,(x \mathbin{-\!\!\!<} a) & = & \cup/\,(\Phi[a])\ast parts\,x\ , \end{array} \tag{4.35}$$

where the binary operator Φ is defined recursively by

$$\begin{array}{lcl} \Phi & : & A\star\star \times A\star \to A\star\star\wr \\ [\,]\Phi x & = & \{[x]\} \\ (zs \mathbin{-\!\!\!<} z)\Phi x & = & \{(zs \mathbin{-\!\!\!<} z) \mathbin{-\!\!\!<} x\} \cup (zs\Phi(z \mathbin{+\!\!\!+\!\!\!<} x))\ . \end{array}$$

It follows that *parts* is a left-reduction $\oplus\not\rightarrow\{[\,]\}$ where operator $\oplus$ is defined by $x \oplus a = \cup/\,(\Phi[a])\ast x$. This second characterisation of function *parts* as a left-reduction will turn out to be very useful in the derivation of algorithms for partition problems.

The function *parts* on *join-list*

Function *parts* is characterised as a catamorphism on *join-list* as follows.

$$\begin{array}{lcl} parts & : & A\ast \to A\ast\ast\wr \\ parts\,[\,] & = & \{[\,]\} \\ parts\,[a] & = & \{[[a]]\}\ =\ [\,]\Psi a \\ parts\,(x +\!\!\!+\, y) & = & (parts\,x) \oplus (parts\,y)\ , \end{array} \tag{4.36}$$

where operator $\oplus$ is defined by

$$x \oplus y \;=\; \cup/\,(x \between_{\ominus} y) \tag{4.37}$$

where operator $\ominus$ is defined by $[\,] \ominus x = x \ominus [\,] = \{x\}$, and

$$x \ominus y \;=\; \{x +\!\!+\, y, it\, x +\!\!+\, [lt\, x +\!\!+\, hd\, y] +\!\!+\, tl\, y\}\ . \tag{4.38}$$

In order to show that function *parts* $= \oplus/ \cdot ([\,]\Psi)*$ is well-defined as a catamorphism on *join-list*, it has to be shown that operator $\oplus$ is associative, and that $\{[\,]\}$ is a unit of $\oplus$. The proofs of these facts (the first one of which is rather long) are omitted.

4.3.2 *parts*-Fusion on *snoc-list* I

This subsection derives a *parts*-Fusion Theorem on the data type *snoc-list*, and it derives two corollaries of this theorem one of which is applied to the problem of finding the smallest square in which a text fits.

Deriving a left-reduction for a *parts*-problem

Since the straightforward implementation of the specification *pp* (4.32) is a very inefficient program, and since a left-reduction can be implemented as an efficient program provided its components can be evaluated efficiently, we aim at deriving a left-reduction for *pp*. We impose conditions on operator $\otimes$ and function g such that *pp* is a left-reduction. If function *parts* is the left-reduction $\oplus\not\rightarrow\{[\,]\}$ characterised in equations (4.34), then, applying Fusion on *snoc-list*, Corollary 2.76, we obtain the following equation for specification (4.32).

$$pp \;=\; \oslash\not\rightarrow(g\,[\,])\ , \tag{4.39}$$

for some operator $\oslash$, provided $\otimes/ \cdot g* \cdot (\oplus a) = (\oslash a) \cdot \otimes/ \cdot g*$ after *parts*, where $\oplus$ is the operator defined by $x \oplus a \;=\; \cup/\,(\Psi a)*\,x$. In order to find sufficient conditions for this equality to hold, we calculate as follows.

$$\begin{array}{cl}
 & \otimes/ \cdot g* \cdot (\oplus a) \\
= & \quad \text{definition of } \oplus \\
 & \otimes/ \cdot g* \cdot \cup/ \cdot (\Psi a)* \\
= & \quad \text{Fusion on } \textit{set}\text{, map-distributivity} \\
 & \otimes/ \cdot (\otimes/ \cdot g* \cdot (\Psi a))* \\
= & \quad \textbf{assumption } \otimes/ \cdot g* \cdot (\Psi a) = (\oslash a) \cdot g \\
 & \otimes/ \cdot ((\oslash a) \cdot g)* \\
= & \quad \text{map-distributivity}
\end{array}$$

$$\begin{array}{ll} & \otimes/ \cdot (\oslash a)* \cdot g* \\ = & \quad \textbf{assumption } (\oslash a) \text{ is } (\otimes,\otimes)\text{-fusable, Fusion on } \mathit{set} \\ & (\oslash a) \cdot \otimes/ \cdot g* \ . \end{array}$$

The *parts*-Fusion Theorem on *snoc-list* reads as follows.

(4.40) Theorem (*parts*-Fusion on *snoc-list* I) *Let $\otimes/ \cdot g*$ be a catamorphism on* set *such that $\otimes/ \cdot g* \cdot (\Psi a) = (\oslash a) \cdot g$ for some operator $\oslash$ such that section $(\oslash a)$ is $(\otimes,\otimes)$-fusable after $\otimes/ \cdot g* \cdot (\in \mathit{parts}\, z)\triangleleft$ for all lists z. Then*

$$pp \;=\; \oslash\!\not\to\!(g\,[\,])\ .$$

A subclass of partition problems

A subclass of the partition problems described by (4.32) is obtained by instantiating operator $\otimes$ with selector $\downarrow_f$ and function g with function $(\mathit{all}\, p)?_{\downarrow_f}$. We define, applying equation (2.69),

$$\mathit{fap} \;=\; \downarrow_f/ \cdot (\mathit{all}\, p)\triangleleft \cdot \mathit{parts}\ . \tag{4.41}$$

Many interesting partition problems, among which the problems mentioned at the beginning of this section, are of this form. This specific choice for $\otimes$ and g induces a form of the *parts*-Fusion Theorem that is closer to implementation, provided predicate p is prefix-closed. After the derivation of this special form, we specialise further by choosing f to be the length function # (and $\downarrow_{\#}$ a suitable deterministic selector) and by requiring p to be segment-closed.

The first step in writing *fap* as a left-reduction is the application of the *parts*-Fusion Theorem. We have to construct an operator $\oslash$ such that

$$\downarrow_f/ \cdot (\mathit{all}\, p)\triangleleft \cdot (\Psi a) \;=\; (\oslash a) \cdot (\mathit{all}\, p)?_{\downarrow_f} \text{ after } \mathit{parts}\ , \tag{4.42}$$

such that $(\oslash a)$ is $(\downarrow_f, \downarrow_f)$-fusable after $\downarrow_f/ \cdot (\mathit{all}\, p)\triangleleft \cdot (\in \mathit{parts}\, z)\triangleleft$ for all lists z. We let the second requirement be a premise, and we concentrate on the first requirement (4.42).

The first requirement for operator $\oslash$, equation (4.42), gives us immediately a definition for $(\oslash a)$ on arguments of the form $(\mathit{all}\, p)?_{\downarrow_f}\, ys$, but it leaves freedom for the other arguments. So

$$zs \oslash a \;=\; \begin{cases} \downarrow_f/\,(\mathit{all}\, p)\triangleleft(ys \Psi a) & \text{if } zs = (\mathit{all}\, p)?_{\downarrow_f}\, ys \\ \text{anything} & \text{otherwise}\ . \end{cases}$$

Since

$$(all\,p)?_{\downarrow_f}\ ys \quad = \quad \begin{cases} \nu_{\downarrow_f} & \text{if } \neg(all\,p)\ ys \\ ys & \text{otherwise} \end{cases},$$

it remains to give a definition of $\nu_{\downarrow_f} \oslash a$ such that equation (4.42) is satisfied. Assume $\neg(all\,p)\ ys$, so $ys = zs +\!\!\!< z$, and $\neg(all\,p)\ zs \ \vee\ \neg p\ z$. Then

$$\begin{array}{ll}
 & \downarrow_f /\ (all\,p)\triangleleft\,(ys\,\Psi\,a) \\
= & \quad \text{definition of } \Psi \\
 & \downarrow_f /\ (all\,p)\triangleleft\,\{zs +\!\!\!< (z +\!\!\!< a), ys +\!\!\!< [a]\} \\
= & \quad \neg(all\,p)\ ys \ \Rightarrow\ \neg(all\,p)\ (ys +\!\!\!< [a]) \\
 & \downarrow_f /\ (all\,p)\triangleleft\,\{zs +\!\!\!< (z +\!\!\!< a)\} \\
= & \quad \textbf{assume } p \text{ prefix-closed} : \neg p\ z \ \Rightarrow\ \neg p\ (z +\!\!\!< a) \\
 & \nu_{\downarrow_f}\ .
\end{array}$$

In the last step we are 'forced' to assume predicate p to be prefix-closed, for without this assumption we cannot get rid off the occurrences of zs and z. Writing ξ for $\nu_{\downarrow_f}$, and adding the case analysis present in the definition of operator Ψ, we arrive at the following, rather useless, corollary of the *parts*-Fusion on *snoc-list* I Theorem.

(4.43) Corollary *Suppose predicate p is prefix-closed, and suppose that section $(\oslash a)$ is $(\downarrow_f, \downarrow_f)$-fusable after $\downarrow_f / \cdot (all\,p)\triangleleft \cdot (\in parts\ z)\triangleleft$ for all lists z, where operator $\oslash$ is defined by*

$$\begin{array}{lcl}
\xi \oslash a & = & \xi \\
[\,] \oslash a & = & (all\,p)?_{\downarrow_f}\ [[a]] \\
(zs +\!\!\!< z) \oslash a & = & \begin{cases} \downarrow_f /\ (all\,p)\triangleleft((zs +\!\!\!< z)\Psi a) & \textit{if } all\,p\ (zs +\!\!\!< z) \\ \textit{anything} & \textit{otherwise} \end{cases}.
\end{array}$$

Then

$$fap \ = \ \oslash\!\not\to\![\,]\ .$$

Further specialisation

For a number of applications the selector $\downarrow_f$ is the selector $\downarrow_\#$. Consider the subclass of partition problems of *fap* where the selector $\downarrow_f$ is the selector $\downarrow_\#$

$$sap \ = \ \downarrow_\# / \cdot (all\,p)\triangleleft \cdot parts\ . \qquad (4.44)$$

We determine conditions on p and we complete the definition of operator $\downarrow_\#$ such that the conditions of the above corollary of the *parts*-Fusion on *snoc-list* I Theorem are satisfied, and hence such that

$$sap \;=\; \oslash\not\to[\,]\,, \tag{4.45}$$

for some operator $\oslash$.

Assume that predicate p is prefix-closed and holds for all singletons. The assumption that predicate p holds for all singletons is not really necessary, but it is convenient in the following development. Since $x <_\# y \;\Rightarrow\; x \downarrow_\# y = x$, we can rewrite the definition of operator $\oslash$ as in Corollary 4.43 by

$$\begin{array}{lcl}
\xi \oslash a & = & \xi \\
[\,] \oslash a & = & [[a]] \\
(zs \mathbin{-\!\!\!<} z) \oslash a & = & \begin{cases} zs \mathbin{-\!\!\!<} (z \mathbin{-\!\!\!<} a) & \text{if } \mathit{all}\, p\,(zs \mathbin{-\!\!\!<} z) \;\wedge\; p\,(z \mathbin{-\!\!\!<} a) \\ (zs \mathbin{-\!\!\!<} z) \mathbin{-\!\!\!<} [a] & \text{if } \mathit{all}\, p\,(zs \mathbin{-\!\!\!<} z) \;\wedge\; \neg p\,(z \mathbin{-\!\!\!<} a) \\ \text{anything} & \text{if } \neg \mathit{all}\, p\,(zs \mathbin{-\!\!\!<} z)\,. \end{cases}
\end{array}$$

The requirement that section $(\oslash a)$ be $(\downarrow_\#, \downarrow_\#)$-fusable after $\downarrow_\#/ \cdot (\mathit{all}\, p)\triangleleft \cdot (\in \mathit{parts}\, z)\triangleleft$ for all lists z only requires knowledge of the definition of $(\oslash a)$ on arguments of the form $\downarrow_\#/\,(\mathit{all}\, p)\triangleleft v$, so we are free to leave $(\oslash a)$ undefined on other arguments. We have to show that

$$\xi \oslash a \;=\; \xi \tag{4.46}$$

$$(x \downarrow_\# y) \oslash a \;=\; (x \oslash a) \downarrow_\# (y \oslash a) \tag{4.47}$$

for $x = \downarrow_\#/\,(\mathit{all}\, p)\triangleleft v$ and $y = \downarrow_\#/\,(\mathit{all}\, p)\triangleleft w$ where v and$w$ are sets of partitions of some z. In trying to prove equation (4.47) (equation (4.46) holds by definition of operator $\oslash$) we refine the selector $\downarrow_\#$, for which at the moment we only know that $x <_\# y \;\Rightarrow\; x \downarrow_\# y = x$, suitably.

The derivation of a definition of $\downarrow_\#$

First, we try to prove equation (4.47) in case $x <_\# y$.

$$\begin{array}{ll}
& x <_\# y \\
\Rightarrow & \quad \text{definition of } \oslash\text{, assumption on the form of } x \text{ and } y \\
& x \oslash a <_\# y \oslash a \;\vee\; (x \oslash a =_\# y \oslash a \;\wedge\; \neg p\,(\mathit{lt}\, x \mathbin{-\!\!\!<} a) \;\wedge\; p\,(\mathit{lt}\, y \mathbin{-\!\!\!<} a)) \\
\Rightarrow & \quad \neg p\,(\mathit{lt}\, x \mathbin{-\!\!\!<} a) \;\Rightarrow\; \mathit{lt}\,(x \oslash a) = [a] \\
& x \oslash a <_\# y \oslash a \;\vee\; (x \oslash a =_\# y \oslash a \;\wedge\; \mathit{lt}\,(x \oslash a) <_\# \mathit{lt}\,(y \oslash a)) \\
\Rightarrow & \quad \textbf{assume} \text{ refinement of } \downarrow_\# \text{ below} \\
& (x \oslash a) \downarrow_\# (y \oslash a) = x \oslash a\,,
\end{array}$$

The last step is motivated by the wish to prove equation (4.47). Operator $\downarrow_\#$ is defined as follows on equal-length arguments x and y.

$$x \downarrow_\# y \quad = \quad \begin{cases} x & \text{if } \mathit{lt}\,x <_\# \mathit{lt}\,y \\ y & \text{if } \mathit{lt}\,y <_\# \mathit{lt}\,x\ . \end{cases} \tag{4.48}$$

The last case for which selector $\downarrow_\#$ remains to be defined is $x =_\# y$ and $x =_{\#\cdot \mathit{lt}} y$. The definition of selector $\downarrow_\#$ will be completed in the following derivation. We prove equation (4.47) in case $x =_\# y$ and $\mathit{lt}\,x <_\# \mathit{lt}\,y$.

$$\begin{array}{ll} & x =_\# y \ \wedge \ \mathit{lt}\,x <_\# \mathit{lt}\,y \\ \Rightarrow & \qquad \text{definition of } \oslash\text{, assumption on the form of } x \text{ and } y \\ & (x \oslash a \leq_\# y \oslash a \ \wedge \ (p\,(\mathit{lt}\,y \mathbin{-\!\!\!<} a) \ \Rightarrow \ p\,(\mathit{lt}\,x \mathbin{-\!\!\!<} a))) \ \vee \\ & (x \oslash a >_\# y \oslash a \ \wedge \ p\,(\mathit{lt}\,y \mathbin{-\!\!\!<} a) \ \wedge \ \neg p\,(\mathit{lt}\,x \mathbin{-\!\!\!<} a))\ . \end{array}$$

The second disjunct in the last expression is undesirable, so we want the following implication to hold.

$$\mathit{lt}\,x <_\# \mathit{lt}\,y \quad \Rightarrow \quad (p\,(\mathit{lt}\,y \mathbin{-\!\!\!<} a) \Rightarrow p\,(\mathit{lt}\,x \mathbin{-\!\!\!<} a))\ . \tag{4.49}$$

Since x and y are partitions of z it follows from $\mathit{lt}\,x <_\# \mathit{lt}\,y$ that $\mathit{lt}\,x$ is a proper suffix of $\mathit{lt}\,y$. So implication (4.49) holds provided p is suffix-closed. Since it has been assumed that predicate p is prefix-closed, it follows that predicate p is segment-closed. So

$$\begin{array}{ll} & x =_\# y \ \wedge \ \mathit{lt}\,x <_\# \mathit{lt}\,y \\ \Rightarrow & \qquad \text{assumptions on } p\text{, } x\text{, and } y \\ & (x \oslash a <_\# y \oslash a \ \wedge \ p\,(\mathit{lt}\,x \mathbin{-\!\!\!<} a) \ \wedge \ \neg p\,(\mathit{lt}\,y \mathbin{-\!\!\!<} a)) \ \vee \\ & (x \oslash a =_\# y \oslash a \ \wedge \ x \oslash a <_{\#\cdot \mathit{lt}} y \oslash a \ \wedge \ p\,(\mathit{lt}\,x \mathbin{-\!\!\!<} a) \ \wedge \ p\,(\mathit{lt}\,y \mathbin{-\!\!\!<} a)) \ \vee \\ & (x \oslash a =_\# y \oslash a \ \wedge \ \mathit{lt}\,(x \oslash a) = \mathit{lt}\,(y \oslash a) = [a] \ \wedge \\ & x \oslash a <_{\#\cdot \mathit{lt}\cdot \mathit{it}} y \oslash a \ \wedge \ \neg p\,(\mathit{lt}\,x \mathbin{-\!\!\!<} a) \ \wedge \ \neg p\,(\mathit{lt}\,y \mathbin{-\!\!\!<} a))\ . \end{array}$$

The first two disjuncts of the last expression in the above calculation immediately give the desired result $(x \oslash a) \downarrow_\# (y \oslash a) = x \oslash a$. For the third disjunct we have to refine $\downarrow_\#$ again. In case $x =_\# y$ and $x =_{\#\cdot \mathit{lt}} y$ define operator $\downarrow_\#$ by

$$x \downarrow_\# y \quad = \quad \begin{cases} x & \text{if } \mathit{it}\,x \downarrow_\# \mathit{it}\,y = \mathit{it}\,x \\ y & \text{if } \mathit{it}\,x \downarrow_\# \mathit{it}\,y = \mathit{it}\,y\ . \end{cases}$$

The gives a complete definition of operator $\downarrow_\#$.

$$x \downarrow_\# y \quad = \quad \begin{cases} x & \text{if } x <_\# y \ \vee \ (x =_\# y \ \wedge \ \mathit{rev}\,\#{*}\,x \leq_L \mathit{rev}\,\#{*}\,y) \\ y & \text{otherwise}\ . \end{cases} \tag{4.50}$$

Finally, we prove equation (4.47) in the remaining case.

$$
\begin{array}{ll}
 & x \downarrow_{\#} y = x \;\wedge\; x =_{\#} y \;\wedge\; x =_{\# \cdot lt} y \\
\Rightarrow & \qquad \text{definition of } \oslash\text{, assumptions} \\
 & (\neg p\,(lt\, x \mathbin{-\!\!<} a) \;\wedge\; it\,(x \oslash a) \downarrow_{\#} it\,(y \oslash a) = it\,(x \oslash a) \;\wedge \\
 & lt\,(x \oslash a) = lt\,(y \oslash a) = [a]) \;\vee \\
 & (p\,(lt\, x \mathbin{-\!\!<} a) \;\wedge\; lt\,(x \oslash a) = lt\,(y \oslash a) = lt\, x \mathbin{-\!\!<} a \;\wedge\; x \downarrow_{\#} y = x \;\wedge \\
 & it\, x = it\,(x \oslash a) \;\wedge\; it\, y = it\,(y \oslash a)) \\
= & \qquad \text{definition of } \downarrow_{\#} \\
 & (x \oslash a) \downarrow_{\#} (y \oslash a) = x \oslash a\ .
\end{array}
$$

This results in a useful corollary of Corollary 4.43.

(4.51) Corollary *Suppose predicate p is segment-closed and holds for all singletons. Define selector $\downarrow_{\#}$ on equal-length arguments by equation (4.50). Then*

$$sap \;=\; \oslash \not\rightarrow [\,]\ ,$$

where operator $\oslash$ is defined by

$$
\begin{array}{lcl}
\xi \oslash a & = & \xi \\
[\,] \oslash a & = & [[a]] \\
(zs \mathbin{-\!\!<} z) \oslash a & = & \left\{ \begin{array}{ll} zs \mathbin{-\!\!<} (z \mathbin{-\!\!<} a) & \textit{if } all\, p\,(zs \mathbin{-\!\!<} z) \;\wedge\; p\,(z \mathbin{-\!\!<} a) \\ (zs \mathbin{-\!\!<} z) \mathbin{-\!\!<} [a] & \textit{if } all\, p\,(zs \mathbin{-\!\!<} z) \;\wedge\; \neg p\,(z \mathbin{-\!\!<} a) \\ \textit{anything} & \textit{if } \neg all\, p\,(zs \mathbin{-\!\!<} z)\ . \end{array} \right.
\end{array}
$$

Since the predicate *asc* is segment-closed and holds for all singletons, this corollary gives a left-reduction that can be implemented as a linear-time program for the problem

$$\downarrow_{\#}/ \cdot (all\ asc) \triangleleft \cdot parts\ . \tag{4.52}$$

In the following example we give a solution to a somewhat more complex problem.

Example: the smallest square in which a text fits

Consider the following problem. A piece of text has to be broken into lines in such a way that the area of the smallest square in which the lines fit is minimal. We suppose that a piece of text is a list of words, and that words are encoded as natural numbers which denote the length of the words. This problem *tis* (for 'text in square') can be specified as follows.

$$tis \;=\; \downarrow_{size}/ \cdot parts\ , \tag{4.53}$$

where

$$\begin{array}{lll} \mathit{size}\, x &=& \mathit{height}\, x \uparrow \mathit{breadth}\, x \qquad (4.54)\\ \mathit{height} &=& \# \qquad (4.55)\\ \mathit{breadth} &=& \uparrow\!\not\to\! 0 \cdot \mathit{sum}\star\ , \qquad (4.56) \end{array}$$

where function *sum* is a function returning the sum of a snoc-list. It is the left-reduction $+\!\not\to\!0$. Function $\downarrow_{size}$ is underspecified; when $x =_{size} y$ the result of $x \downarrow_{size} y$ is not specified. Define selector $\downarrow_{size}$ in case $x =_{size} y$ by

$$x <_{height} y \ \vee\ (x =_{height} y \ \wedge\ \mathit{sum}\star\, x \le_L \mathit{sum}\star\, y) \quad\Rightarrow\quad x \downarrow_{size} y = x\ .$$

This adjusted operator $\downarrow_{size}$ is a total operator if x and y are partitions of the same list z.

The derivation of an efficient algorithm for this problem consists of the following steps. First, it is shown that the *parts*-Fusion on *snoc-list* I Theorem is not applicable to *tis*. Then we give an extended specification, and we show that Corollary 4.51 is applicable to the components of the extended specification.

To apply the *parts*-Fusion on *snoc-list* I Theorem we have to verify the applicability conditions of this theorem. If section $(\oslash a)$ is defined as $\downarrow_{size}/ \cdot (\Psi a)$ the first condition is satisfied. For the second condition we have to show that

$$(x \downarrow_{size} y) \oslash a \quad=\quad (x \oslash a) \downarrow_{size} (y \oslash a)\ , \qquad (4.57)$$

for elements $x = \downarrow_{size}/\, v$ and $y = \downarrow_{size}/\, w$, where $v, w \subseteq \mathit{parts}\, z$ for some list z. This equality does not hold. A counterexample is established as follows. Let $x = [[3,2]]$ and $y = [[3],[2]]$ be partitions of $z = [3,2]$. Obviously, $y <_{size} x$, so $(x \downarrow_{size} y) \oslash 7 = y \oslash 7$. But it is easy to verify that $(x \oslash 7) \downarrow_{size} (y \oslash 7) = x \oslash 7$. Since $x \oslash 7 \neq y \oslash 7$, this establishes a counterexample to equation (4.57).

Since the *parts*-Fusion on *snoc-list* I Theorem is not applicable to problem *tis*, we have to think of an extension of our problem.

Note that the size of a minimally sized partition of a list is at most the maximum of the length of the list and the maximum of the list, and at least the maximum of the list. To be precise, if function m returns the maximum of a snoc-list of natural numbers, $m = \uparrow\!\not\to\!0$, then, since the partition of x into singletons has size $\#\, x \uparrow m\, x$, we have

$$m\, x \quad\le\quad \mathit{size} \downarrow_{size}/\, \mathit{parts}\, x \quad\le\quad \#\, x \uparrow m\, x\ . \qquad (4.58)$$

Note furthermore that if $m\, x > \#\, x$, then the partition of x into singletons has size $m\, x$, and this partition is one of the minimally sized ones, and it follows that the problem is solved. Consider the case $m\, x \le \#\, x$.

Problem *tis* is extended as follows. Define for i in between $m\, x$ and $\#\, x$ inclusive

$$s_i \quad=\quad \downarrow_{height}/ \cdot ((\le i) \cdot \mathit{breadth}) \triangleleft \cdot \mathit{parts}\ . \qquad (4.59)$$

By the range of i, the partition of x into singletons satisfies $(\le i) \cdot \mathit{breadth}$, and hence $s_i\, x \neq \nu_{\downarrow_{height}}$ for all i with $m\, x \le i \le \#\, x$. Abbreviate $\downarrow_{size}/\, \mathit{parts}\, x$ to z. We have

$$
\begin{array}{ll}
 & s_{\mathit{size}\ z}\ x \\
= & \quad \text{definition of } s_i \\
 & \downarrow_{\mathit{height}}/\ ((\leq \mathit{size}\ z) \cdot \mathit{breadth}) \triangleleft \mathit{parts}\ x \\
= & \quad z \text{ is an element of } ((\leq \mathit{size}\ z) \cdot \mathit{breadth}) \triangleleft \mathit{parts}\ x \\
 & \downarrow_{\mathit{height}}/\ (\{z\} \cup ((\leq \mathit{size}\ z) \cdot \mathit{breadth}) \triangleleft \mathit{parts}\ x) \\
= & \quad \text{Theorem 3.48, its applicability condition holds} \\
 & \downarrow_{\mathit{height}}/\ \{z\} \\
= & \quad \text{definition of reduction} \\
 & z\ .
\end{array}
$$

It follows from equation (4.58) and this calculation that

$$\downarrow_{\mathit{size}}/\ \mathit{parts}\ x \quad =_{\mathit{size}} \quad \downarrow_{\mathit{size}}/\ \{s_{m\,x}\ x, \ldots, s_{\#\,x}\ x\}\ .$$

Instead of *tis*, the problem we consider now is the efficient computation of the tuple of functions $s_{m\,x}, \ldots, s_{\#\,x}$. Here we make use of the underspecification of operator $\downarrow_{\mathit{size}}$: we want to find the size of the smallest square in which the given text, broken into lines, fits, and we are not interested in exactly how the text is placed in the square. If the placement of the text in the square is important operator $\downarrow_{\mathit{size}}$ can be refined using a valuation function v, for example by

$$x \downarrow_{\mathit{size}} y \quad = \quad \begin{cases} x & \text{if } x <_{\mathit{size}} y \\ y & \text{if } y <_{\mathit{size}} x \\ x \downarrow_v y & \text{otherwise}\ . \end{cases}$$

With this refined definition of operator $\downarrow_{\mathit{size}}$ we obtain an entirely different problem.

The remaining task is to find efficient algorithms for functions s_i for all i with $m\,x \leq i \leq \#\,x$. We want to apply Corollary 4.51 to each of the functions s_i. First refine operator $\downarrow_{\mathit{height}}$ to the operator $\downarrow_{\#}$ defined in equation (4.50). Note that the predicate $(\leq\ i) \cdot \mathit{breadth}$ is equivalent to the predicate $\mathit{all}\,((\leq\ i) \cdot \mathit{sum})$. We verify the conditions of Corollary 4.51. Since $i \geq m\,x$, $(\leq\ i) \cdot \mathit{sum}$ holds for singletons. Furthermore, because the elements of the list to which s_i is applied are natural numbers, the predicate $(\leq\ i) \cdot \mathit{sum}$ is segment-closed. It follows that the conditions of Corollary 4.51 are satisfied, and hence that we obtain a left-reduction for each function s_i with $i : m\,x \leq i \leq \#\,x$.

$$s_i \quad = \quad \oslash \not\rightarrow [\,]\ , \tag{4.60}$$

where operator $\oslash$ is defined by

$$
\begin{array}{lcl}
\nu_{\downarrow_{\#}} \oslash a & = & \nu_{\downarrow_{\#}} \\
[\,] \oslash a & = & [[a]] \\
(zs \dashv\!\!< z) \oslash a & = & \begin{cases} zs \dashv\!\!< (z \dashv\!\!< a) & \text{if } \mathit{breadth}\,(zs \dashv\!\!< z) \leq i\ \wedge\ \mathit{sum}\,(z \dashv\!\!< a) \leq i \\ (zs \dashv\!\!< z) \dashv\!\!< [a] & \text{if } \mathit{breadth}\,(zs \dashv\!\!< z) \leq i\ \wedge\ \mathit{sum}\,(z \dashv\!\!< a) > i \\ \text{anything} & \text{if } \mathit{breadth}\,(zs \dashv\!\!< z) > i \end{cases}
\end{array}
$$

If n is the length of the argument, $O(n)$ left-reductions (one for each i in between the maximum and the length of the list), each of which can be evaluated in linear time, are evaluated. Hence the algorithm for $\downarrow_{size}/ \cdot \mathit{parts}$ we have obtained requires time $O(n^2)$ for its evaluation. This algorithm is not asymptotically optimal; there exists an $O(n \log n)$ algorithm.

4.3.3 *parts*-Fusion on *snoc-list* II

This subsection derives a second *parts*-Fusion on *snoc-list* Theorem, and it derives a corollary of this theorem. In the derivation of this theorem we use the second characterisation of *parts* on *snoc-list*, see (4.35). This subsection shows that the form of a generator fusion theorem may differ for different recursive characterisations of the generator, and that it is not reasonable to speak of 'the' generator fusion pertaining to a generator. The theorem derived in this section only applies to problems in the class *sap* of partition problems.

Deriving another left-reduction for a *parts*-problem

The second characterisation of *parts* on *snoc-list* induces an alternative *parts*-Fusion Theorem on *snoc-list*. Here we derive a fusion theorem for the subclass *sap* of partition problems. We aim at deriving a left-reduction for function *sap*. In the derivation of *parts*-Fusion on *snoc-list* I, Theorem 4.40, an important design decision was taken in the third step

$$\begin{array}{ll} & \otimes/ \, (\otimes/ \cdot g* \cdot (\Psi a))* \, x \\ = & \quad \textbf{assume } \otimes/ \cdot g* \cdot (\Psi a) = (\oslash a) \cdot g \\ & \otimes/ \, ((\oslash a) \cdot g)* \, x \; . \end{array}$$

The assumption amounts to 'simplifying' the action that is performed on each element in the set of partitions x. In the subsequent derivation we do not make such an assumption, but instead we select an 'interesting' subset of x, namely the elements of x which satisfy predicate $\mathit{all}\, p$, and we make an assumption on that subset in order to obtain a left-reduction for *sap*. Apply Fusion on *snoc-list* to obtain

$$\mathit{sap} \;=\; \oslash \not\rightarrow [\,] \; ,$$

provided

$$\downarrow_{\#}/ \, (\mathit{all}\, p) \triangleleft (x \oplus a) \;=\; \downarrow_{\#}/ \, (\mathit{all}\, p) \triangleleft x \oslash a \; ,$$

for x in the image of *parts*, and for operator $\oplus$ defined by $x \oplus a = \cup/ \, (\Phi[a])* \, x$. We have

$$\begin{array}{ll} & \downarrow_{\#}/ \, (\mathit{all}\, p) \triangleleft (x \oplus a) \\ = & \quad \text{definition of } \oplus \end{array}$$

$$\begin{array}{ll}
& \downarrow_{\#}/\,(\mathit{all}\,p)\triangleleft\cup/\,(\Phi[a])*\,x \\
= & \quad \text{Fusion on } \mathit{set} \\
& \downarrow_{\#}/\,(\downarrow_{\#}/\cdot(\mathit{all}\,p)\triangleleft\cdot(\Phi[a]))*\,x \\
= & \quad \text{define } x \odot y = \downarrow_{\#}/\,(\mathit{all}\,p)\triangleleft(x\Phi y) \\
& \downarrow_{\#}/\,(\odot[a])*\,x \\
= & \quad (\mathit{all}\,p \vee \neg\mathit{all}\,p) = \underline{\mathit{true}},\ \underline{\mathit{true}}\triangleleft = \mathit{id} \\
& \downarrow_{\#}/\,(\odot[a])*\,(\mathit{all}\,p \vee \neg\mathit{all}\,p)\triangleleft x \\
= & \quad (p \vee q)\triangleleft x = p\triangleleft x \cup q\triangleleft x \\
& \downarrow_{\#}/\,(\odot[a])*\,((\mathit{all}\,p)\triangleleft x \cup (\neg\mathit{all}\,p)\triangleleft x) \\
= & \quad \text{definition of map} \\
& \downarrow_{\#}/\,((\odot[a])*\,(\mathit{all}\,p)\triangleleft x \cup (\odot[a])*\,(\neg\mathit{all}\,p)\triangleleft x) \\
= & \quad \text{Theorem 3.48, } \textbf{assume} \text{ its applicability condition holds} \\
& \downarrow_{\#}/\,(\odot[a])*\,(\mathit{all}\,p)\triangleleft x \\
= & \quad \text{Fusion on } \mathit{set}\text{, } \textbf{assume } (\odot[a]) \text{ is } (\downarrow_{\#},\downarrow_{\#})\text{-fusable} \\
& (\odot[a])\,\downarrow_{\#}/\,(\mathit{all}\,p)\triangleleft x\ .
\end{array}$$

The condition referred to in the step where Theorem 3.48 is applied is

$$\forall b \in (\odot[a])*\,(\neg\mathit{all}\,p)\triangleleft x : \exists c \in (\odot[a])*\,(\mathit{all}\,p)\triangleleft x : b \downarrow_{\#} c = c\ . \tag{4.61}$$

This calculation proves the following theorem.

(4.62) Theorem (*parts*-Fusion on *snoc-list* II) *Suppose section* $(\odot[a])$ *is* $(\downarrow_{\#},\downarrow_{\#})$*-fusable after* $\downarrow_{\#}/\cdot(\mathit{all}\,p)\triangleleft\cdot(\in \mathit{parts}\,z)\triangleleft$ *for all* z*, where operator* $\odot$ *is defined by*

$$x \odot y = \downarrow_{\#}/\,(\mathit{all}\,p)\triangleleft(x\Phi y)\ .$$

Furthermore, suppose that for x *in the image of* parts *equation (4.61) holds. Then*

$$\mathit{sap} \;=\; \oslash\!\not\rightarrow\![\,]\ ,$$

where operator $\oslash$ *is defined by* $x \oslash a = x \odot [a]$.

As before, we elaborate the conditions of the theorem to derive properties of predicate p that guarantee the validity of the conditions of the theorem.

Elaborating the conditions of the *parts*-Fusion on *snoc-list* II Theorem

We start with the second condition, equation (4.61), of the theorem. We have to show that for x in the image of *parts*, say $x = \mathit{parts}\ v$,

$$\forall b \in (\odot[a])*\,(\neg\mathit{all}\,p)\triangleleft x : \exists c \in (\odot[a])*\,(\mathit{all}\,p)\triangleleft x : b \downarrow_{\#} c = c\ .$$

If the set $(\odot[a])*(\neg\mathit{all}\,p)\triangleleft x$ is empty, then the above statement trivially holds. Suppose the set $(\odot[a])*(\neg\mathit{all}\,p)\triangleleft x$ is nonempty, and let b be an arbitrary element of the set $(\odot[a])*(\neg\mathit{all}\,p)\triangleleft x$. By definition of operator $\odot$, either $b = \nu_{\downarrow_\#}$, or $b = ds \dashv\!\!< (d \dashv\!\!< a)$, where ds is a (possibly empty) list of lists and d is a (possibly empty) list, and $\mathit{all}\,p\,b$ holds. We have to exhibit an element $c \in (\odot[a])*(\mathit{all}\,p)\triangleleft x$ such that

$$b \downarrow_\# c \;=\; c\ .$$

We give a candidate for c. If we assume that predicate p holds for singletons, then the set $(\odot[a])*(\mathit{all}\,p)\triangleleft x$ is nonempty, since the partition of v (recall $x = \mathit{parts}\,v$ for some v) into singletons satisfies $\mathit{all}\,p$. This is our first assumption on predicate p. Distinguish now the two cases $b = \nu_{\downarrow_\#}$, and $b \neq \nu_{\downarrow_\#}$. If $b = \nu_{\downarrow_\#}$, then trivially for any $c \in (\odot[a])*(\mathit{all}\,p)\triangleleft x$ we have $b \downarrow_\# c = c$. If $b \neq \nu_{\downarrow_\#}$, then $b = ds \dashv\!\!< (d \dashv\!\!< a)$, and $\mathit{all}\,p\,b$ holds. Hence $\mathit{all}\,p\,ds$ holds. Since predicate p holds for singletons, $\mathit{all}\,p\,(\tau*\,d)$ holds. It follows that $\mathit{all}\,p\,(ds +\!\!\!+\!\!< \tau*\,d)$ holds, and hence that $ds +\!\!\!+\!\!< \tau*\,d$ is an element of $(\mathit{all}\,p)\triangleleft x$. A candidate for c is therefore

$$c \;=\; (ds +\!\!\!+\!\!< \tau*\,d) \odot [a]\ .$$

We have

$$\begin{array}{ll}
 & b \downarrow_\# c = c \\
\Leftarrow & \qquad \text{definition of } c \\
 & b \in (\mathit{all}\,p)\triangleleft((ds +\!\!\!+\!\!< \tau*\,d)\Phi[a]) \\
\Leftarrow & \qquad \mathit{all}\,p\,b \text{ holds, definition of } b \\
 & ds \dashv\!\!< (d \dashv\!\!< a) \in (ds +\!\!\!+\!\!< \tau*\,d)\Phi[a] \\
= & \qquad \text{definition of } \Phi \\
 & \mathit{true}\ .
\end{array}$$

It follows that the second condition of the *parts*-Fusion on *snoc-list* II Theorem holds if we assume that predicate p holds for singletons.

The first condition of the *parts*-Fusion on *snoc-list* II Theorem requires section $(\odot[a])$ to be $(\downarrow_\#, \downarrow_\#)$-fusable after $\downarrow_\#/\,(\mathit{all}\,p)\triangleleft(\in \mathit{parts}\,z)\triangleleft$ for all lists z, i.e.,

$$\nu_{\downarrow_\#} \odot [a] \;=\; \nu_{\downarrow_\#} \tag{4.63}$$
$$(x \downarrow_\# y) \odot [a] \;=\; (x \odot [a]) \downarrow_\# (y \odot [a])\ , \tag{4.64}$$

for x and y in the image of $\downarrow_\#/\,(\mathit{all}\,p)\triangleleft(\in \mathit{parts}\,z)\triangleleft$ for some list z. The first equality is satisfied by definition of operator $\odot$ if we assume that

$$\nu_{\downarrow_\#} \odot x \;=\; \nu_{\downarrow_\#}\ ,$$

for all x. Let $x \downarrow_\# y = x$. To prove (4.64) we have to show that

$$(x \odot [a]) \downarrow_\# (y \odot [a]) \;=\; x \odot [a]\ . \tag{4.65}$$

Instead of this equation, we prove a generalisation of it. Let x, y, v, and w be such that $x \mathbin{-\!\!\!<} v$ and $y \mathbin{-\!\!\!<} w$ are both elements of $\mathit{parts}\; z$ for some z, and suppose that $\mathit{all}\; p\; x$, $\mathit{all}\; p\; y$, $p\; v$, and $p\; w$ hold. Then

$$x \downarrow_{\#} y = x \quad \Rightarrow \quad (x \odot v) \downarrow_{\#} (y \odot w) = (x \odot v)\ , \tag{4.66}$$

for a suitable refinement of selector $\downarrow_{\#}$. Note that equation (4.65) follows from this implication. We prove equation (4.66) by induction on the length of list x. If $x = \nu_{\downarrow_{\#}}$, then $y = \nu_{\downarrow_{\#}}$, and if $y = \nu_{\downarrow_{\#}}$, then equation (4.66) trivially holds. For the base case $x = [\,]$ we have $x \odot v = [v] = [z]$. Since $\#(y \odot w) \geq 1$, and $\#(y \odot w) = 1 \;\Rightarrow\; y \odot w = [z]$, it follows that implication (4.66) holds if $x = [\,]$. For the induction step we first give a more manageable expression for operator $\odot$.

Note that

$$us \mathbin{-\!\!\!<} u \in ts\Phi t \quad \Rightarrow \quad t \text{ is a suffix of } u\ .$$

Hence, if p is suffix-closed,

$$\neg p\; t \quad \Rightarrow \quad (\mathit{all}\; p) \triangleleft (ts\Phi t) = \{\,\}\ ,$$

and so, if $\mathit{all}\; p\; (ts \mathbin{-\!\!\!<} t)$ and $p\; s$ hold, then

$$\begin{array}{cl}
 & (ts \mathbin{-\!\!\!<} t) \odot s \\
= & \qquad \text{definition of operator } \odot \\
 & \downarrow_{\#}/\, (\mathit{all}\; p) \triangleleft ((ts \mathbin{-\!\!\!<} t)\Phi s) \\
= & \qquad \text{definition of } \Phi \text{, filter, and reduction} \\
 & \downarrow_{\#}/\, (\mathit{all}\; p) \triangleleft \{(ts \mathbin{-\!\!\!<} t) \mathbin{-\!\!\!<} s\} \downarrow_{\#} \downarrow_{\#}/\, (\mathit{all}\; p) \triangleleft (ts\Phi(t \mathbin{+\!\!\!<} s)) \\
= & \qquad p \text{ is suffix-closed, } \mathit{all}\; p\; (ts \mathbin{-\!\!\!<} t) \text{ and } p\; s \text{ hold} \\
 & \left\{ \begin{array}{ll} (ts \mathbin{-\!\!\!<} t) \mathbin{-\!\!\!<} s & \text{if } \neg p\; (t \mathbin{+\!\!\!<} s) \\ ts \odot (t \mathbin{+\!\!\!<} s) & \text{otherwise}\ . \end{array} \right.
\end{array}$$

Hence, if predicate p is suffix-closed (the second assumption on predicate p) and if $p\; s$ and $\mathit{all}\; p(ts \mathbin{-\!\!\!<} t)$ hold, then operator $\odot$ satisfies

$$(ts \mathbin{-\!\!\!<} t) \odot s \quad = \quad \left\{ \begin{array}{ll} (ts \mathbin{-\!\!\!<} t) \mathbin{-\!\!\!<} s & \text{if } \neg p\; (t \mathbin{+\!\!\!<} s) \\ ts \odot (t \mathbin{+\!\!\!<} s) & \text{otherwise}\ . \end{array} \right.$$

We proceed with the induction step in the proof of equation (4.66). Distinguish the four cases

$$\begin{array}{lcl}
p\; (\mathit{lt}\; x \mathbin{+\!\!\!<} v) & \wedge & p\; (\mathit{lt}\; y \mathbin{+\!\!\!<} w) \\
\neg p\; (\mathit{lt}\; x \mathbin{+\!\!\!<} v) & \wedge & p\; (\mathit{lt}\; y \mathbin{+\!\!\!<} w) \\
p\; (\mathit{lt}\; x \mathbin{+\!\!\!<} v) & \wedge & \neg p\; (\mathit{lt}\; y \mathbin{+\!\!\!<} w) \\
\neg p\; (\mathit{lt}\; x \mathbin{+\!\!\!<} v) & \wedge & \neg p\; (\mathit{lt}\; y \mathbin{+\!\!\!<} w)\ .
\end{array}$$

Since $lt\,x \mathbin{+\!\!\!+\!\!\!<} v$ and $lt\,y \mathbin{+\!\!\!+\!\!\!<} w$ are both tails of some list z, the second and third case can be discarded altogether if we assume in addition that predicate p is robust. For the definition of a robust predicate, see equation (2.88). This is our third (and last) assumption on predicate p.

If $\neg p\,(lt\,x \mathbin{+\!\!\!+\!\!\!<} v) \ \wedge\ \neg p\,(lt\,y \mathbin{+\!\!\!+\!\!\!<} w)$, then by definition of operator $\odot$

$$(x \odot v) \downarrow_{\#} (y \odot w) \quad = \quad (x \mathbin{-\!\!\!<} v) \downarrow_{\#} (y \mathbin{-\!\!\!<} w)$$

If $x <_{\#} y$, then trivially

$$(x \mathbin{-\!\!\!<} v) \downarrow_{\#} (y \mathbin{-\!\!\!<} w) \quad = \quad x \mathbin{-\!\!\!<} v\ . \tag{4.67}$$

If $x =_{\#} y$ we refine operator $\downarrow_{\#}$ such that equation (4.67) holds. If $x =_{\#} y$ define selector $\downarrow_{\#}$ by

$$\#\star x \leq_L \#\star y \quad \Rightarrow \quad x \downarrow_{\#} y = x\ .$$

Note that for x and y in the specific domain considered here

$$\#\star x \leq_L \#\star y \ \wedge\ \#\star y \leq_L \#\star x \quad \Rightarrow \quad x = y\ .$$

Equation (4.67) follows immediately for the case $x =_{\#} y$.

In the remaining case, $p\,(lt\,x \mathbin{+\!\!\!+\!\!\!<} v) \ \wedge\ p\,(lt\,y \mathbin{+\!\!\!+\!\!\!<} w)$, we obtain by definition of operator $\odot$

$$(x \odot v) \downarrow_{\#} (y \odot w) \quad = \quad (it\,x \odot (lt\,x \mathbin{+\!\!\!+\!\!\!<} v)) \downarrow_{\#} (it\,y \odot (lt\,y \mathbin{+\!\!\!+\!\!\!<} w))\ .$$

If $x <_{\#} y$, then $it\,x <_{\#} it\,y$, and we can apply the induction hypothesis to obtain

$$\begin{array}{cl}
 & (it\,x \odot (lt\,x \mathbin{+\!\!\!+\!\!\!<} v)) \downarrow_{\#} (it\,y \odot (lt\,y \mathbin{+\!\!\!+\!\!\!<} w)) \\
= & \qquad \text{induction hypothesis, } it\,x <_{\#} it\,y \\
 & it\,x \odot (lt\,x \mathbin{+\!\!\!+\!\!\!<} v) \\
= & \qquad \text{definition of } \odot\text{, case assumption} \\
 & x \odot v\ .
\end{array}$$

If $x =_{\#} y$, then, by definition of selector $\downarrow_{\#}$, $\#\star x \leq_L \#\star y$. It follows that $\#\star it\,x \leq_L \#\star it\,y$, and hence, again applying the induction hypothesis, we obtain

$$(it\,x \odot (lt\,x \mathbin{+\!\!\!+\!\!\!<} v)) \downarrow_{\#} (it\,y \odot (lt\,y \mathbin{+\!\!\!+\!\!\!<} w)) \quad = \quad x \odot v\ .$$

This concludes the proof of equality (4.66). All conditions of the *parts*-Fusion on *snoc-list* II Theorem are satisfied and we have obtained the following corollary of this theorem.

(4.68) Corollary *Suppose predicate p is suffix-closed and robust and holds for singletons, and suppose that operator $\downarrow_{\#}$ is defined such that*

$$x <_{\#} y \ \vee\ (x =_{\#} y \ \wedge\ \#\star x \leq_L \#\star y) \quad \Rightarrow \quad x \downarrow_{\#} y = x\ . \tag{4.69}$$

Then

$$sap \;=\; \oslash\!\nrightarrow\![\,]\;,$$

where operator $\oslash$ *is defined by* $x \oslash a = x \odot [a]$, *and operator* $\odot$ *is defined by*

$$\begin{array}{lcl} \nu_{\downarrow_\#} \odot s & = & \nu_{\downarrow_\#} \\ [\,] \odot s & = & [s] \\ (ts \mathrel{-\!\!\prec} t) \odot s & = & \begin{cases} (ts \mathrel{-\!\!\prec} t) \mathrel{-\!\!\prec} y & \textit{if}\ \neg p\,(t \mathrel{+\!\!\!\prec} s) \\ ts \odot (t \mathrel{+\!\!\!\prec} s) & \textit{otherwise}\ , \end{cases} \end{array}$$

for arguments $ts \mathrel{-\!\!\prec} t$ *and* s *satisfying* $all\ p\,(ts \mathrel{-\!\!\prec} t)$ *and* $p\ s$. *For other arguments operator* $\odot$ *is defined by* $\downarrow_\#/\,(all\ p) \triangleleft ((ts \mathrel{-\!\!\prec} t)\Phi s)$

4.3.4 *parts*-Fusion on *join-list* I

This section derives a *parts*-Fusion Theorem together with a corollary on the data type *join-list*. This section uses the definition of function *parts* as a catamorphism, see (4.36).

Deriving a catamorphism on *join-list* for a *parts*-problem

We want to derive an algorithm that can be implemented as an efficient program for function *pp*. Since function *parts* is a catamorphism, and since a catamorphism can be implemented as an efficient program provided its components can be evaluated efficiently, we apply Fusion on *join-list*, Corollary 2.62, to specification (4.32). We obtain

$$pp \;=\; \oslash/ \cdot (\otimes/ \cdot g* \cdot ([\,]\Psi))*\;,$$

for some operator $\oslash$, provided $\otimes/\ g*\,(x \oplus y) = \otimes/\ g*\,x \oslash \otimes/\ g*\,y$ for x and y in the image of *parts*, where operator $\oplus$ is defined in equation (4.37). For the composition of functions $\otimes/ \cdot g* \cdot ([\,]\Psi)$ we have

$$\begin{array}{cl} & \otimes/\ g*\,([\,]\Psi a) \\ = & \quad \text{definition of } \Psi \\ & \otimes/\ g*\,\{[[a]]\} \\ = & \quad \text{definition of map and reduction on } \textit{set} \\ & g\,[[a]]\;, \end{array}$$

so $\otimes/ \cdot g* \cdot ([\,]\Psi) = g \cdot \tau \cdot \tau$. For the applicability condition of Fusion on *join-list* calculate as follows.

$$
\begin{array}{ll}
 & \otimes/\ g{*}\,(x \oplus y) \\
= & \quad \text{definition of operators } \oplus \text{ (4.37)} \\
 & \otimes/\ g{*} \cup/\ (x \mathbin{\mathsf{X}}_{\ominus}\, y) \\
= & \quad \text{Fusion on } \textit{set} \\
 & \otimes/\ (\otimes/ \cdot g{*}){*}\,(x \mathbin{\mathsf{X}}_{\ominus}\, y) \\
= & \quad \text{Cross Law 3.27} \\
 & \otimes/\ (x \mathbin{\mathsf{X}}_{\otimes/\cdot g*\cdot\ominus}\, y) \\
= & \quad \text{equation (4.70) below} \\
 & \otimes/\ (x \mathbin{\mathsf{X}}_{\oslash\cdot g\times g}\, y) \\
= & \quad \text{Cross Law 3.28} \\
 & \otimes/\ (g{*}\,x \mathbin{\mathsf{X}}_{\oslash}\, g{*}\,y) \\
= & \quad \text{Cross Law 3.31, } \textbf{assume} \text{ sections } (\oslash a) \text{ and } (b\oslash) \text{ are } (\otimes,\otimes)\text{-fusable} \\
 & \otimes/\ g{*}\,x \oslash \otimes/\ g{*}\,y\ .
\end{array}
$$

Equation (4.70) applied in this calculation reads as follows.

$$\otimes/ \cdot g{*} \cdot \ominus \;=\; \oslash \cdot g \times g \tag{4.70}$$

We have derived the following *parts*-Fusion Theorem on *join-list*.

(4.71) Theorem (*parts*-Fusion on *join-list*) *Let $\otimes/ \cdot g{*}$ be a catamorphism on* set *such that $\otimes/ \cdot g{*} \cdot \ominus = \oslash \cdot g \times g$ for some operator $\oslash$ such that sections $(\oslash a)$ and $(b\oslash)$ are $(\otimes,\otimes)$-fusable after $\otimes/ \cdot g{*} \cdot (\in \mathit{parts}\ z){\triangleleft}$ for all lists z. Then*

$$pp \;=\; \oslash/ \cdot (g \cdot \tau \cdot \tau){*}\ .$$

A subclass of partition problems

Again, we consider the subclass *sap*, see equation (4.44), of partition problems. We determine conditions on predicate p such that the *parts*-Fusion on *join-list* Theorem can be applied to this problem to obtain a catamorphism on *join-list* for it. Apply the *parts*-Fusion on *join list* Theorem to this specification. It is required to find an operator $\oslash$ satisfying (applying equation (2.69))

$$\downarrow_{\#}/ \cdot (\mathit{all}\ p){\triangleleft} \cdot \ominus \;=\; \oslash \cdot (\mathit{all}\ p)?_{\downarrow_{\#}} \times (\mathit{all}\ p)?_{\downarrow_{\#}}\ , \tag{4.72}$$

such that sections $(\oslash a)$ and $(b\oslash)$ are $(\downarrow_{\#},\downarrow_{\#})$-fusable after $\downarrow_{\#}/ \cdot (\mathit{all}\ p){\triangleleft} \cdot (\in \mathit{parts}\ z){\triangleleft}$ for all lists z.

The first requirement for operator $\oslash$, equation (4.72), gives us immediately a definition of operator $\oslash$ on arguments of the form $(\mathit{all}\ p)?_{\downarrow_\#}\ \mathit{xs}$, $(\mathit{all}\ p)?_{\downarrow_\#}\ \mathit{ys}$, and it leaves freedom for the other arguments. So

$$\mathit{ts} \oslash \mathit{us} \quad = \quad \begin{cases} \downarrow_\#/\ (\mathit{all}\ p)\triangleleft(\mathit{xs} \ominus \mathit{ys}) & \text{if } \mathit{ts} = (\mathit{all}\ p)?_{\downarrow_\#}\ \mathit{xs} \ \wedge\ \mathit{us} = (\mathit{all}\ p)?_{\downarrow_\#}\ \mathit{ys} \\ \text{anything} & \text{otherwise}\ . \end{cases}$$

Since

$$(\mathit{all}\ p)?_{\downarrow_\#}\ \mathit{zs} \quad = \quad \begin{cases} \xi & \text{if } \neg(\mathit{all}\ p)\ \mathit{zs} \\ \mathit{zs} & \text{otherwise}\ , \end{cases}$$

where $\xi = \nu_{\downarrow_\#}$ it remains to give a definition of $\xi \oslash x$ and $x \oslash \xi$ such that equation (4.72) is satisfied. Assume $\neg(\mathit{all}\ p)\ \mathit{xs}$. Then

$$\begin{array}{cl}
 & \downarrow_\#/\ (\mathit{all}\ p)\triangleleft(\mathit{xs} \ominus \mathit{ys}) \\
= & \qquad \text{definition of } \ominus \\
 & \downarrow_\#/\ (\mathit{all}\ p)\triangleleft\{\mathit{xs} +\!\!+\ \mathit{ys}, \mathit{it}\ \mathit{xs} +\!\!+ [\mathit{lt}\ \mathit{xs} +\!\!+ \mathit{hd}\ \mathit{ys}] +\!\!+ \mathit{tl}\ \mathit{ys}\} \\
= & \qquad \neg(\mathit{all}\ p)\ \mathit{xs} \ \Rightarrow\ \neg(\mathit{all}\ p)\ (\mathit{xs} +\!\!+ \mathit{ys}) \\
 & \downarrow_\#/\ (\mathit{all}\ p)\triangleleft\{\mathit{it}\ \mathit{xs} +\!\!+ [\mathit{lt}\ \mathit{xs} +\!\!+ \mathit{hd}\ \mathit{ys}] +\!\!+ \mathit{tl}\ \mathit{ys}\} \\
= & \qquad \textbf{assume} \text{ predicate } p \text{ is prefix-closed, } \neg(\mathit{all}\ p)\ \mathit{xs} \\
 & \xi\ .
\end{array}$$

In the last step we are 'forced' to assume predicate p to be prefix-closed, for without this assumption we cannot get rid off the occurrences of xs and ys. Similarly, if predicate p is suffix-closed and $\neg(\mathit{all}\ p)\ \mathit{ys}$, then

$$\downarrow_\#/\ (\mathit{all}\ p)\triangleleft(\mathit{xs} \ominus \mathit{ys}) \quad = \quad \xi\ .$$

It follows that if predicate p is segment-closed, then operator $\oslash$ defined by

$$\mathit{ts} \oslash \mathit{us} \quad = \quad \begin{cases} \xi & \text{if } \mathit{ts} = \xi \ \vee\ \mathit{us} = \xi \\ \downarrow_\#/\ (\mathit{all}\ p)\triangleleft(\mathit{ts} \ominus \mathit{us}) & \text{if } \mathit{all}\ p\ \mathit{ts} \ \wedge\ \mathit{all}\ p\ \mathit{us} \\ \text{anything} & \text{otherwise}\ , \end{cases}$$

satisfies equation (4.72).

For the second requirement of the *parts*-Fusion on *join-list* Theorem we have to show that sections $(\oslash a)$ and $(b\oslash)$ are $(\downarrow_\#, \downarrow_\#)$-fusable after $\downarrow_\#/\cdot(\mathit{all}\ p)\triangleleft\cdot(\in \mathit{parts}\ z)\triangleleft$ for all lists z. Since $\xi \oslash x = x \oslash \xi = \xi$, it remains to show that for x and y elements in the image of $\downarrow_\#/\cdot(\mathit{all}\ p)\triangleleft\cdot(\in \mathit{parts}\ z)\triangleleft$ for some list z,

$$(x \downarrow_\# y) \oslash v \quad = \quad (x \oslash v) \downarrow_\# (y \oslash v) \qquad (4.73)$$

$$v \oslash (x \downarrow_\# y) \quad = \quad (v \oslash x) \downarrow_\# (v \oslash y)\ . \qquad (4.74)$$

For the proof of these equalities we have to refine the definition of operator $\downarrow_\#$ suitably.

The derivation of a definition of $\downarrow_{\#}$

Let $x \downarrow_{\#} y = x$. We want to show that

$$(x \oslash v) \downarrow_{\#} (y \oslash v) \;=\; (x \oslash v) \qquad (4.75)$$
$$(v \oslash x) \downarrow_{\#} (v \oslash y) \;=\; (v \oslash x)\ . \qquad (4.76)$$

Since $x \downarrow_{\#} y = x$, either $x <_{\#} y$ or $x =_{\#} y$ holds.

$$\begin{array}{ll} & x <_{\#} y \\ \Rightarrow & \quad \text{definition of } \oslash \\ & x \oslash v <_{\#} y \oslash v \ \vee\ (x \oslash v =_{\#} y \oslash v \ \wedge\ \neg p\,(\mathit{lt}\,x +\!\!+ \mathit{hd}\,v) \ \wedge\ p\,(\mathit{lt}\,y +\!\!+ \mathit{hd}\,v)) \\ \Rightarrow & \quad p \text{ is suffix-closed, } \textbf{assume } p \text{ is robust, so } p\,(\mathit{lt}\,x +\!\!+ \mathit{hd}\,v) \equiv p\,(\mathit{lt}\,y +\!\!+ \mathit{hd}\,v) \\ & x \oslash v <_{\#} y \oslash v\ , \end{array}$$

for if predicate p is suffix-closed and robust we have for $x,\ y \in \mathit{parts}\,z$ for some z that $p\,(\mathit{lt}\,y +\!\!+ \mathit{hd}\,v) \ \Rightarrow\ p\,(\mathit{lt}\,x +\!\!+ \mathit{hd}\,v)$. Similarly, if predicate p is prefix-closed and robust, then

$$x <_{\#} y \ \Rightarrow\ v \oslash x <_{\#} v \oslash y\ .$$

In the remaining case we have

$$\begin{array}{ll} & x =_{\#} y \\ \Rightarrow & \quad \text{definition of } \oslash \\ & (x \oslash v <_{\#} y \oslash v \ \wedge\ p\,(\mathit{lt}\,x +\!\!+ \mathit{hd}\,v) \ \wedge\ \neg p\,(\mathit{lt}\,y +\!\!+ \mathit{hd}\,v)) \ \vee \\ & (x \oslash v =_{\#} y \oslash v \ \wedge\ p\,(\mathit{lt}\,x +\!\!+ \mathit{hd}\,v) \ \equiv\ p\,(\mathit{lt}\,y +\!\!+ \mathit{hd}\,v)) \\ \Rightarrow & \quad p \text{ is segment-closed and robust} \\ & x \oslash v =_{\#} y \oslash v\ , \end{array}$$

and similarly

$$x =_{\#} y \ \Rightarrow\ v \oslash x =_{\#} v \oslash y\ .$$

It follows that for equal-length arguments different definitions of operator $\downarrow_{\#}$ can be chosen in order to ensure that equations (4.75) and (4.76) hold. Two possibilities are the following. Suppose $x =_{\#} y$. Define operator $\downarrow_{\#}$ by

$$x \downarrow_{\#} y \;=\; \begin{cases} x & \text{if } \#{*}\,y \leq_L \#{*}\,x \\ y & \text{otherwise}\ . \end{cases} \qquad (4.77)$$

Note that if $x,\ y \in \mathit{parts}\,z$ for some z, then

$$\#{*}\,y \leq_L \#{*}\,x \ \wedge\ \#{*}\,x \leq_L \#{*}\,y \ \Rightarrow\ x = y\ .$$

Clearly, if $x =_{\#} y$, and $\# * y \leq_L \# * x$, then

$$\begin{array}{lll} \# * (y \oslash v) & \leq_L & \# * (x \oslash v) \\ \# * (v \oslash y) & \leq_L & \# * (v \oslash x) \; . \end{array}$$

Another refinement of selector $\downarrow_{\#}$ which ensures that equations (4.75) and (4.76) hold is

$$x \downarrow_{\#} y \quad = \quad \begin{cases} x & \text{if } \mathit{rev}\,\# * x \leq_L \mathit{rev}\,\# * y \\ y & \text{otherwise} \; . \end{cases} \tag{4.78}$$

In conclusion,

(4.79) Corollary *Suppose predicate p is segment-closed and robust. Define selector $\downarrow_{\#}$ on equal-length arguments by equation (4.77). Then*

$$\downarrow_{\#}/ \cdot (\mathit{all}\, p) \triangleleft \cdot \mathit{parts} \quad = \quad \oslash/ \cdot ((\mathit{all}\, p)?_{\downarrow_{\#}} \cdot \tau \cdot \tau) * \; .$$

where operator $\oslash$ is defined by

$$ts \oslash us \quad = \quad \begin{cases} \xi & \mathit{if}\ ts = \xi \ \vee \ us = \xi \\ \downarrow_{\#}/\, (\mathit{all}\, p) \triangleleft (ts \ominus us) & \mathit{if\ all}\ p\ ts \ \wedge \ \mathit{all}\ p\ us \\ \mathit{anything} & \mathit{otherwise} \; , \end{cases}$$

4.3.5 *parts*-Fusion on *join-list* II

In this section we try to derive another *parts*-Fusion on *join-list* Theorem. We do not succeed, but the definitions and results given here will reappear later in Chapter 7, which derives incremental algorithms for partition problems.

A fourth characterisation of *parts* is the following. Consider the following set-comprehension. For nonempty z

$$\mathit{parts}\ z \quad = \quad \{u +\!\!+ [v] +\!\!+ w \mid z = +\!\!+/\, u +\!\!+ v +\!\!+ +\!\!+/\, w, \mathit{all}\, (\neq [\,])\, (u +\!\!+ [v] +\!\!+ w)\} \; .$$

Function *splits2*

A straightforward translation of the above set-comprehension expression into a recursive expression is in terms of function *splits2*. Given a list, function *splits2* generates the set of all pairs of lists which when concatenated are equal to the argument. For example,

$$\mathit{splits2}\,[1,3,2] \quad = \quad \{([\,],[1,3,2]),([1],[3,2]),([1,3],[2]),([1,3,2],[\,])\} \; .$$

Function $\mathit{splits2} : A* \to (A* \times A*)\wr$ is characterised on *join-list* by

$$\begin{array}{lcl} \mathit{splits2}\,[\,] & = & \{([\,],[\,])\} \\ \mathit{splits2}\,[a] & = & \{([\,],[a]),([a],[\,])\} \\ \mathit{splits2}\,(x \mathbin{+\!\!+} y) & = & (\mathit{id} \times (\mathbin{+\!\!+} y))*\,\mathit{splits2}\,x \cup ((x\mathbin{+\!\!+}) \times \mathit{id})*\,\mathit{splits2}\,y\ . \end{array}$$

To define $\mathit{splits2}$ as a catamorphism, we replace x in $((x\mathbin{+\!\!+}) \times \mathit{id})*$ by $\mathit{exl} \uparrow_{\#\cdot \mathit{exl}}/\ \mathit{splits2}\ x$ and we replace y in $(\mathit{id} \times (\mathbin{+\!\!+} y))*$ by $\mathit{exr} \uparrow_{\#\cdot \mathit{exr}}/\ \mathit{splits2}\ y$, and we obtain

$$\mathit{splits2} \ = \ \oplus/ \cdot r*\ ,$$

where function r is defined by $r\ a = \{([\,],[a]),([a],[\,])\}$, and operator $\oplus$ is defined by

$$x \oplus y \ = \ (\mathit{id} \times (\mathbin{+\!\!+} \mathit{exr} \uparrow_{\#\cdot \mathit{exr}}/\ y))*\,x \cup ((\mathit{exl} \uparrow_{\#\cdot \mathit{exl}}/\ x\mathbin{+\!\!+}) \times \mathit{id})*\,y\ .$$

This rather complicated definition is avoided if function $\mathit{splits2}$ returns a join-list instead of a set, $\mathit{splits2} : A* \to (A* \times A*)*$. Function $\mathit{splits2}$ is the following catamorphism on *join-list*.

$$\mathit{splits2} \ = \ \oplus/ \cdot r*\ , \tag{4.80}$$

where function r is defined by $r\ a = [([\,],[a]),([a],[\,])]$, and operator $\oplus$ is defined by

$$x \oplus y \ = \ (\mathit{id} \times (\mathbin{+\!\!+} \mathit{exr}\ \mathit{hd}\ y))*\ \mathit{it}\ x \mathbin{+\!\!+} ((\mathit{exl}\ \mathit{lt}\ x\mathbin{+\!\!+}) \times \mathit{id})*\ y\ . \tag{4.81}$$

Alternatively, operator $\oplus$ may be defined by

$$x \oplus y \ = \ (\mathit{id} \times (\mathbin{+\!\!+} \mathit{exr}\ \mathit{hd}\ y))*\ x \mathbin{+\!\!+} ((\mathit{exl}\ \mathit{lt}\ x\mathbin{+\!\!+}) \times \mathit{id})*\ \mathit{tl}\ y\ . \tag{4.82}$$

We list some properties of function $\mathit{splits2}$. Function $\mathit{splits2}$ is injective, two left-inverses of $\mathit{splits2}$ are given in the following equations.

$$\mathit{exl} \cdot \mathit{lt} \cdot \mathit{splits2} \ = \ \mathit{id} \tag{4.83}$$

$$\mathit{exr} \cdot \mathit{hd} \cdot \mathit{splits2} \ = \ \mathit{id}\ . \tag{4.84}$$

Other left-inverses are

$$(\mathit{hd} \cdot \mathit{exr})* \cdot \mathit{it} \cdot \mathit{splits2} \ = \ \mathit{id} \tag{4.85}$$

$$(\mathit{lt} \cdot \mathit{exl})* \cdot \mathit{tl} \cdot \mathit{splits2} \ = \ \mathit{id}\ . \tag{4.86}$$

Furthermore,

$$\mathit{splits2}\ x \ = \ (\mathit{id} \times \mathit{it})*\ \mathit{it}\ \mathit{splits2}\ (x \mathbin{+\!\!+} [a]) \tag{4.87}$$

$$\mathit{splits2}\ x \ = \ (\mathit{tl} \times \mathit{id})*\ \mathit{tl}\ \mathit{splits2}\ ([a] \mathbin{+\!\!+} x)\ . \tag{4.88}$$

Function *parts* again

Function *parts* is defined in terms of function *splits2* as follows. Again, we have $\mathit{parts}\,[\,] = \{[\,]\}$ and $\mathit{parts}\,[a] = \{[[a]]\}$. On the concatenation of two join-lists function *parts* is defined by

$$\begin{aligned} \mathit{parts}\,(x \mathbin{+\!\!+} y) &= \cup/\,((\mathit{parts} \times \mathit{id})\ast\, \mathit{splits2}\; x \mathbin{\chi_{\odot}} (\mathit{id} \times \mathit{parts})\ast\, \mathit{splits2}\; y) && (4.89)\\ (x,y) \odot (u,v) &= x \mathbin{\chi_{\ominus_{y \mathbin{+\!\!+} u}}} v && (4.90)\\ a \ominus_c b &= a \mathbin{+\!\!+} [c] \mathbin{+\!\!+} b\ . && (4.91) \end{aligned}$$

To obtain a catamorphism for function *parts* thus characterised proceed by replacing x and y in equation (4.89) by $\mathbin{+\!\!+}/\,s$ and $\mathbin{+\!\!+}/\,t$, where s is an arbitrary element from $\mathit{parts}\,x$ and t is an arbitrary element from $\mathit{parts}\,y$, respectively. This definition as a catamorphism is unsatisfactory in a sense, since it throws away the information obtained from $\mathit{parts}\,x$ and $\mathit{parts}\,y$, and computes $\mathit{parts}\,(x \mathbin{+\!\!+} y)$ from scratch. For this reason we do not give the catamorphism. From equation (4.89) we guess that it might be possible to construct a catamorphism for function

$$(\mathit{parts} \times \mathit{id})\ast \vartriangle (\mathit{id} \times \mathit{parts})\ast \cdot \mathit{splits2}\ , \qquad (4.92)$$

but we will not dwell upon deriving it.

Deriving a catamorphism on *join-list* for a *parts*-problem

This section discusses the derivation of a *parts*-Fusion Theorem on *join-list* using the characterisation of function *parts* in terms of function *splits2*, see (4.89).

We want to construct a catamorphism on *join-list* for specification (4.32). Since *parts* as characterised in equation (4.89) is not a catamorphism, we cannot apply Fusion on *join-list* to expression (4.32). Instead, we apply Characterisation of Catamorphisms on *join-list* in our derivation. For the empty list we have

$$\otimes/\; g\ast\, \mathit{parts}\,[\,] \;=\; g\,[\,]\ ,$$

for singleton lists we have

$$\otimes/\; g\ast\, \mathit{parts}\,[a] \;=\; g\,[[a]]\ ,$$

and in case the argument is of the form $x \mathbin{+\!\!+} y$ we calculate as follows.

$$\begin{aligned} & \otimes/\; g\ast\, \mathit{parts}\,(x \mathbin{+\!\!+} y) \\ = & \quad \text{equation (4.89)} \\ & \otimes/\; g\ast \cup/\,((\mathit{parts} \times \mathit{id})\ast\, \mathit{splits2}\; x \mathbin{\chi_{\odot}} (\mathit{id} \times \mathit{parts})\ast\, \mathit{splits2}\; y) \\ = & \quad \text{Fusion on } \mathit{set} \end{aligned}$$

$$
\begin{array}{ll}
& \otimes/\ (\otimes/\cdot g*)*\ ((\mathit{parts}\times \mathit{id})*\ \mathit{splits2}\ x \between_{\odot} (\mathit{id}\times \mathit{parts})*\ \mathit{splits2}\ y) \\
= & \quad \text{Cross Law 3.27} \\
& \otimes/\ ((\mathit{parts}\times \mathit{id})*\ \mathit{splits2}\ x \between_{\otimes/\cdot g*\cdot\odot} (\mathit{id}\times \mathit{parts})*\ \mathit{splits2}\ y) \\
= & \quad \text{equation (4.93) below} \\
& \otimes/\ ((\mathit{parts}\times \mathit{id})*\ \mathit{splits2}\ x \between_{\oslash\cdot((\otimes/\cdot g*)\times id)\times(id\times(\otimes/\cdot g*))} (\mathit{id}\times \mathit{parts})*\ \mathit{splits2}\ y) \\
= & \quad \text{Cross Law 3.28, map distributivity, equation (2.9)} \\
& \otimes/\ (((\otimes/\cdot g*\cdot \mathit{parts})\times \mathit{id})*\ \mathit{splits2}\ x \between_{\oslash} (\mathit{id}\times (\otimes/\cdot g*\cdot \mathit{parts}))*\ \mathit{splits2}\ y)\ .
\end{array}
$$

So, assuming that there exist an operator $\oslash$ such that

$$\otimes/\cdot g*\cdot\odot \quad = \quad \oslash\cdot((\otimes/\cdot g*)\times id)\times(id\times(\otimes/\cdot g*))\ , \tag{4.93}$$

we have derived the following equation

$$\mathit{pp}\,(x +\!\!+\, y) \quad = \quad \otimes/\ ((\mathit{pp}\times \mathit{id})*\ \mathit{splits2}\ x \between_{\oslash} (\mathit{id}\times \mathit{pp})*\ \mathit{splits2}\ y)\ . \tag{4.94}$$

We have not been able to express $\mathit{pp}\,(x +\!\!+\, y)$ in terms of $\mathit{pp}\ x$ and $\mathit{pp}\ y$, and hence we have not been able to express *pp* as a catamorphism. From equation (4.94) we guess that it might be possible to construct a catamorphism fo function

$$(\mathit{pp}\times \mathit{id})* \vartriangle (\mathit{id}\times \mathit{pp})*\cdot \mathit{splits2}\ , \tag{4.95}$$

(note the similarity with equation (4.92)), but that is another story that will be told in another chapter.

4.4 Treecut problems

This section is the last section on the derivation of a generator fusion in the chapter on generator fusion theorems. In the previous sections, the data type *list* has been the primary data type. This section presents the data type *binary labelled tree* or *moo tree*, and it develops some theory for the derivation of algorithms defined on this data type. The main purpose of presenting this theory is to show that the calculational methods applied on different data types are very similar.

Within the Bird-Meertens calculus different studies on the data type *tree* have been written, see Gibbons [56, 57, 58], Meertens [96, 97], Bird [17], and Runciman [118]. Here we define the data type *binary labelled tree*, and we give the fusion theorem pertaining to this data type. Furthermore, we define two generators *treecuts* and *substrees*, which respectively correspond to the generators *segs*, which returns all segments or substrings of a list, see Chapter 5, and *subs* defined on the data type *list*. For generator *treecuts* we derive a *treecuts*-Fusion Theorem, and we apply this theorem to an example.

4.4.1 The data type moo tree

There are various definitions of the data type *tree*; for an (incomplete but extensive) overview see Gibbons [57]. This section defines the data type *binary labelled tree* or *moo tree*, and it gives the Fusion Corollary pertaining to this data type.

The data type *moo tree*, with elements of type A in the leaves and elements of type B in the internal nodes, is an initial algebra in the category of $\mathsf{T}_{A,B}$-algebras, where functor $\mathsf{T}_{A,B}$ is defined by

$$\mathsf{T}_{A,B} \quad = \quad \underline{A} + \mathsf{I} \times \underline{B} \times \mathsf{I} \; .$$

Define $((A,B)\beta, \perp \triangledown \otriangle) = \mu(\mathsf{T}_{A,B})$. Informally, if $a \in A$, then $\perp a$ is a moo tree, and if x and y are moo trees, and $b \in B$, then $x \otriangle_b y$ is a moo tree. A catamorphism $\llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}}$ on the data type *moo tree* satisfies the following equations.

$$\begin{array}{lcl} \llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}} \perp a & = & g\, a \\ \llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}} (x \otriangle_b y) & = & \llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}} x \oplus_b \llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}} y \; . \end{array}$$

An example of a catamorphism is the function $\#$, which returns the sum of the number of leaves and nodes of a moo tree, $\# = \llparenthesis \underline{1} \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}}$, where operator $\oplus$ is a ternary operator defined by $x \oplus_b y = x + 1 + y$. Fusion on *moo tree* is the following instantiation of Fusion, Corollary 2.41.

(4.96) Corollary (Fusion on *moo tree***)** *Function f satisfies*

$$f\,(x \oplus_b y) \quad = \quad f\,x \otimes_b f\,y \; ,$$

for x and y in the image of $\llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}}$ if and only if

$$f \cdot \llparenthesis g \triangledown \oplus \rrparenthesis_{\mathsf{T}_{A,B}} \quad = \quad \llparenthesis (f \cdot g) \triangledown \otimes \rrparenthesis_{\mathsf{T}_{A,B}} \; .$$

4.4.2 Treecuts and substrees

This section defines a number of generators on the data type *moo tree*. Some of these generators intuitively correspond to generators defined on the data type *list*. For example, the generators called *treecuts* and *substrees* defined below respectively correspond to the generators *segs*, see Section 5.1 and *subs*, see Section 4.1, defined on the data type *list*.

The function *treecuts*

Consider the data type *moo tree* where the elements at the leaves and in the nodes are of the same type. If the types of the elements at the leaves and in the internal nodes coincide, chopping off branches may result in proper trees with leaves which were previously internal nodes. For example, in the tree $(\perp 3 \oslash_2 \perp 4) \oslash_1 \perp 5$, the tree $\perp 2 \oslash_1 \perp 5$ is a proper moo tree, and we call such a tree a *treecut*. Note that, except for the root, all nodes in a treecut have the same parent as they have in the argument tree. Hoffmann and O'Donnell [64] call such a tree a subtree.

Function *treecuts* returns the set of all treecuts of an argument. Function *treecuts* is defined by means of an auxiliary function *chops* as follows.

$$\begin{array}{lcl} \mathit{treecuts}\ \perp a & = & \{\perp a\} \\ \mathit{treecuts}\ (x \oslash_a y) & = & \mathit{treecuts}\ x \cup \mathit{chops}\ (x \oslash_a y) \cup \mathit{treecuts}\ y\ . \end{array} \tag{4.97}$$

Function *chops* returns the set of all elements that are obtained by chopping off branches of the argument. Function *chops* is defined by

$$\begin{array}{lcl} \mathit{chops}\ \perp a & = & \{\perp a\} \\ \mathit{chops}\ (x \oslash_a y) & = & (\mathit{chops}\ x \times_{\oslash_a} \mathit{chops}\ y) \cup \{\perp a\}\ . \end{array} \tag{4.98}$$

It follows that function *chops* is a *moo tree* catamorphism, i.e.,

$$\mathit{chops} \quad = \quad \llparenthesis(\sigma \cdot \perp) \triangledown \ominus\rrparenthesis_{\mathsf{T}_{A,A}}\ ,$$

where operator $\ominus$ is defined by

$$x \ominus_a y \quad = \quad (x \times_{\oslash_a} y) \cup \{\perp a\}\ .$$

Function *treecuts* is a catamorphism, but the definition as a catamorphism is rather awkward, and a more elegant definition is obtained as follows. We want to express the tuple of functions *treecuts* $\vartriangle$ *chops* as a catamorphism. For that purpose we apply Mutumorphisms on *moo tree*, the instantiation of Theorem 2.47 on the data type *moo tree*, which reads as follows.

(4.99) Corollary *Suppose f and g are $\mathsf{T}_{A,B}$-catamorphic modulo each other:*

$$\begin{array}{lcl} f\,(x \oslash_b y) & = & (f\,x, g\,x) \oplus_b (f\,y, g\,y) \\ g\,(x \oslash_b y) & = & (g\,x, f\,x) \ominus_b (g\,y, f\,y)\ . \end{array}$$

Then

$$f \vartriangle g \quad = \quad \llparenthesis(f \vartriangle g \cdot \perp) \triangledown \odot\rrparenthesis_{\mathsf{T}_{A,B}}\ ,$$

where operator $\odot$ is defined by

$$(x, s) \odot_b (y, t) \quad = \quad ((x, s) \oplus_b (y, t), (s, x) \ominus_b (t, y))\ .$$

From equations (4.97) and (4.98) it follows that functions *treecuts* and *chops* are catamorphic modulo each other. Applying Mutumorphisms on *moo tree* we obtain the following catamorphism on *moo tree* for the tuple of functions *treecuts* $\vartriangle$ *chops*.

$$\mathit{treecuts} \vartriangle \mathit{chops} \;=\; (\!|((\sigma\cdot\,_\perp) \vartriangle (\sigma\cdot\,_\perp)) \triangledown \oplus|\!)_{\mathsf{T}_{A,A}}\;, \tag{4.100}$$

where operator $\oplus$ is defined by

$$\begin{aligned}(x,v)\oplus_a(y,w) \;&=\; (x\cup z\cup y, z)\\ &\qquad \textbf{where } z = (v \;{\chi}_{\,\oslash_a}\; w)\cup\{_\perp\, a\}\;.\end{aligned} \tag{4.101}$$

The number of treecuts of a tree is exponential in the number of nodes.

Another way to define function *treecuts* is by means of the two functions *bucs* and *chops*. Function *chops* has been defined above. Function *bucs* returns the set of all 'bottom-up' components of a tree. The following equations define function *bucs*.

$$\begin{aligned}\mathit{bucs}\,_\perp\, a \;&=\; \{_\perp\, a\}\\ \mathit{bucs}\,(x \oslash_a y) \;&=\; \mathit{bucs}\,x \cup \{x \oslash_a y\} \cup \mathit{bucs}\,y\;.\end{aligned}$$

Recall Paramorphism Characterisation, Theorem 2.44. In the case of the data type *moo-tree*, this theorem states that function h is a paramorphism $[\![f \triangledown \oplus]\!]_{\mathsf{T}_{A,A}}$ if and only if it satisfies

$$\begin{aligned}h\,_\perp\, a \;&=\; f\,a\\ h\,(x\oslash_a y) \;&=\; (h\,x, x)\oplus_a(h\,y, y)\;.\end{aligned}$$

It follows that function *bucs* is a paramorphism defined on the data type *moo tree*, i.e., we have

$$\mathit{bucs} \;=\; [\![(\sigma\cdot\,_\perp)\triangledown\oplus]\!]_{\mathsf{T}_{A,A}}\;,$$

where operator $\oplus$ is an operator that takes five arguments and is defined by

$$(x,v)\oplus_a(y,w) \;=\; x\cup(v\oslash_a w)\cup y\;.$$

Using functions *bucs* and *chops*, function *treecuts* is defined by

$$\mathit{treecuts} \;=\; \cup/\cdot \mathit{chops}{*}\cdot \mathit{bucs}\;.$$

It is easy to show that function *treecuts* thus defined satisfies the characterising equations (4.97). This definition is very similar to definition (5.8) of function *segs*.

The function *substrees*

A substree is the equivalent of the notion of subsequence defined on lists. For example, the tree $\perp 3 \oslash_1 \perp 5$ is a substree of the tree $(\perp 3 \oslash_2 \perp 4) \oslash_1 \perp 5$. In contrast with the notion treecut, a node in a substree of a tree need not have the same parent as in the argument tree. Function *substrees* returns the set of all substrees of a moo tree. It is defined by

$$\begin{array}{lcl} \mathit{substrees} \perp a & = & \{\perp a\} \\ \mathit{substrees}\,(x \oslash_a y) & = & (\mathit{substrees}\ x \chi_{\oslash_a} \mathit{substrees}\ y) \cup \{\perp a\} \cup \\ & & \mathit{substrees}\ x \cup \mathit{substrees}\ y \end{array}$$

It follows that function *substrees* is a catamorphism.

$$\mathit{substrees} \quad = \quad (\!|(\sigma \cdot \perp) \triangledown \oplus|\!)_{\mathsf{T}_{A,A}} \ ,$$

where operator $\oplus$ is defined by

$$x \oplus_a y \quad = \quad (x \chi_{\oslash_a} y) \cup \{\perp a\} \ .$$

4.4.3 *treecuts*-Fusion

This section derives a fusion theorem for generator *treecuts*.

treecuts **problems**

The class of treecut problems we consider is specified by

$$\otimes/ \cdot g* \cdot \mathit{treecuts} \ , \tag{4.102}$$

where operator $\otimes$ and function g are arbitrary. Since the number of treecuts returned by function *treecuts* is exponential in the number of nodes in the argument, the straightforward implementation of this specification with $\otimes$ and g instantiated to some operator and function is a very inefficient program. An example of a treecut problem is a specification of a version of the pattern-matching problem on *moo tree*. Given a pattern P, it is required to find occurrences of P in a given tree, or, if there are no such occurrences, the largest element of $\mathit{bucs}\ P$ occurring in the tree.

$$\uparrow_{\#}/ \cdot (\in \mathit{bucs}\ P)\triangleleft \cdot \mathit{treecuts} \ .$$

There is a large body of literature on this and very similar problems, see for example Hoffmann and O'Donnell [64] and Cai, Paige, and Tarjan [28].

Deriving a catamorphism on *moo tree* for a *treecuts* problem

Since the straightforward implementation of specification (4.102) with $\otimes$ and g instantiated to some operator and function is a very inefficient program, and since a catamorphism on *moo tree* can be implemented as an efficient program, provided its components can be evaluated efficiently, we try to construct a catamorphism on *moo tree* for the specification. Since *treecuts* has not been defined as a catamorphism we cannot proceed as we did in the previous sections. Using equation (2.14) we derive for arbitrary function h

$$\begin{array}{ll} & \otimes/ \cdot g* \cdot \mathit{treecuts} \\ = & \quad \text{equation (2.16)} \\ & \otimes/ \cdot g* \cdot \mathit{exl} \cdot \mathit{treecuts} \vartriangle \mathit{chops} \\ = & \quad \text{equation (2.14), } h \text{ is an arbitrary function} \\ & \mathit{exl} \cdot (\otimes/ \cdot g*) \times h \cdot \mathit{treecuts} \vartriangle \mathit{chops}\ . \end{array}$$

For h we can choose any function that suits us, and in the derivation below a natural candidate will emerge. The tuple of functions *treecuts* $\vartriangle$ *chops* is a catamorphism on *moo tree*, so we can apply Fusion on *moo tree* to the expression $(\otimes/ \cdot g*) \times h \cdot \mathit{treecuts} \vartriangle \mathit{chops}$.

$$\begin{array}{ll} & (\otimes/ \cdot g*) \times h \cdot \mathit{treecuts} \vartriangle \mathit{chops} \\ = & \quad \text{equation (4.100)} \\ & (\otimes/ \cdot g*) \times h \cdot (\!|((\sigma \cdot \perp) \vartriangle (\sigma \cdot \perp)) \triangledown \oplus|\!) \\ = & \quad \text{Fusion on } \mathit{moo\ tree} \\ & (\!|((\otimes/ \cdot g*) \times h \cdot (\sigma \cdot \perp) \vartriangle (\sigma \cdot \perp)) \triangledown \odot|\!)\ , \end{array}$$

provided operator $\odot$ satisfies

$$j\,((x,v) \oplus_a (y,w)) \;=\; j\,(x,v) \odot_a j\,(y,w)\ , \tag{4.103}$$

for (x, v) and (y, w) in the image of *treecuts* $\vartriangle$ *chops*, where function j is the following abbreviation.

$$j \;=\; (\otimes/ \cdot g*) \times h\ . \tag{4.104}$$

For the first component of the junc in the derived catamorphism we have

$$\begin{array}{ll} & (\otimes/ \cdot g*) \times h \cdot (\sigma \cdot \perp) \vartriangle (\sigma \cdot \perp) \\ = & \quad \text{equation (2.12)} \\ & (\otimes/ \cdot g* \cdot \sigma \cdot \perp) \vartriangle (h \cdot \sigma \cdot \perp) \\ = & \quad \text{definition of map and reduction on } \mathit{set} \\ & (g \cdot \perp) \vartriangle (h \cdot \sigma \cdot \perp)\ . \end{array}$$

If we suppose that g is a *moo tree* catamorphism $g = (\![k \triangledown \oslash]\!)$, then the left-hand side of the above split equals k. The assumption that g is a *moo tree* catamorphism will turn out to be useful in the subsequent derivation. An operator $\odot$ satisfying equation (4.103) is synthesised as follows.

$$
\begin{array}{ll}
 & j\ ((x, v) \oplus_a (y, w)) \\
= & \quad \text{definition of } \oplus \text{ (4.101), } z = (v \mathbin{\mathsf{X}}_{\Delta\!\!\!\!\oslash_a} w) \cup \{\perp a\} \\
 & j\ (x \cup z \cup y, z) \\
= & \quad \text{definition of } j \text{ (4.104)} \\
 & (\otimes/\ g{*}\ (x \cup z \cup y), h\ z) \\
= & \quad \text{definition of catamorphism on } \textit{set} \\
 & (\otimes/\ g{*}\ x \otimes \otimes/\ g{*}\ z \otimes \otimes/\ g{*}\ y, h\ z)\ .
\end{array}
$$

So we have found

$$
\begin{array}{rcl}
j\ ((x, v) \oplus_a (y, w)) & = & (p \otimes \otimes/\ g{*}\ z \otimes r, h\ z) \\
 & & \textbf{where } (p, q) = j\ (x, v) \\
 & & \qquad\quad (r, s) = j\ (y, w) \\
 & & \qquad\quad z = (v \mathbin{\mathsf{X}}_{\Delta\!\!\!\!\oslash_a} w) \cup \{\perp a\}\ .
\end{array}
$$

It follows that it remains to express $\otimes/\ g{*}\ z$ and $h\ z$, where $z = (v \mathbin{\mathsf{X}}_{\Delta\!\!\!\!\oslash_a} w) \cup \{\perp a\}$, in terms of $j\ (x, v)$, a, and $j\ (y, w)$. For $\otimes/\ g{*}\ z$ we have

$$
\begin{array}{ll}
 & \otimes/\ g{*}\ ((v \mathbin{\mathsf{X}}_{\Delta\!\!\!\!\oslash_a} w) \cup \{\perp a\}) \\
= & \quad \text{definition of catamorphism on } \textit{set} \\
 & \otimes/\ g{*}\ (v \mathbin{\mathsf{X}}_{\Delta\!\!\!\!\oslash_a} w) \otimes g \perp a \\
= & \quad \text{Cross Law 3.27} \\
 & \otimes/\ (v \mathbin{\mathsf{X}}_{g \cdot \Delta\!\!\!\!\oslash_a} w) \otimes g \perp a \\
= & \quad \textbf{assume } g = (\![k \triangledown \oslash]\!) \\
 & \otimes/\ (v \mathbin{\mathsf{X}}_{\oslash_a \cdot g \times g} w) \otimes k\ a \\
= & \quad \text{Cross Law 3.28} \\
 & \otimes/\ (g{*}\ v \mathbin{\mathsf{X}}_{\oslash_a} g{*}\ w) \otimes k\ a \\
= & \quad \text{Cross Law 3.31, } \textbf{assume } (\oslash_a b) \text{ and } (c \oslash_a) \text{ are } (\otimes, \otimes)\text{-fusable} \\
 & (\otimes/\ g{*}\ v \oslash_a \otimes/\ g{*}\ w) \otimes k\ a\ .
\end{array}
$$

In this calculation we have assumed that function g is a *moo tree* catamorphism $g = (\![k \triangledown \oslash]\!)$ such that sections $(\oslash_a b)$ and $(c \oslash_a)$ are $(\otimes, \otimes)$-fusable for all b and c after $\otimes/ \cdot g{*} \cdot (\in \textit{chops}\ z)\triangleleft$ for all trees z. If these conditions are satisfied, then $\otimes/\ g{*}\ z$, where

$z = (v \between_{\bigcirc\!\!\!\!\wedge} w) \cup \{\perp a\}$, can be expressed in terms of $j\,(x, v)$, a, and $j\,(y, w)$. Since $h\,z$ should also be expressed in terms of $j\,(x, v)$, a, and $j\,(y, w)$, a natural choice for function h is

$$h \quad = \quad \otimes/ \cdot g* \; .$$

Substituting $\otimes/ \cdot g*$ for h in the above calculations yields the following result.

(4.105) Theorem (*treecuts*-Fusion) *Suppose $g = (\![k \triangledown \oslash]\!)$ such that sections $(\oslash_a b)$ and $(c \oslash_a)$ are $(\otimes, \otimes)$-fusable for all b and c after $\otimes/ \cdot g* \cdot (\in \mathit{chops}\ z)\triangleleft$ for all trees z. Then*

$$\otimes/ \cdot g* \cdot \mathit{treecuts} \quad = \quad \mathit{exl} \cdot (\![(k \vartriangle k) \triangledown \odot]\!) \; ,$$

where operator $\odot$ is defined by

$$(x, v) \odot_a (y, w) \quad = \quad (x \otimes (v \oslash_a w) \otimes k\ a \otimes y, (v \oslash_a w) \otimes k\ a) \; .$$

This theorem could be called Horner's rule on *moo-tree*, compare with Theorem 5.21.

Example: the maximum treecut sum problem

The *moo tree* catamorphism $(\![id \triangledown +]\!)_{\top_{I,I}}$, where $+$ is the ternary operator defined by $x +_a y = x + a + y$, and I is the type of integers, returns the sum of the elements of a moo tree with integers at the nodes and leaves. The maximum treecut sum problem is specified by

$$\mathit{mts} \quad = \quad \uparrow/ \cdot (\![id \triangledown +]\!)* \cdot \mathit{treecuts} \; .$$

Since $(\![id \triangledown +]\!)$ is a *moo tree* catamorphism such that sections $(+_a b)$ and $(c +_a)$ are $(\uparrow, \uparrow)$-fusable for all b and c, we can apply *treecuts*-Fusion to obtain

$$\mathit{mts} \quad = \quad \mathit{exl} \cdot (\![(id \vartriangle id) \triangledown \odot]\!) \; ,$$

where operator $\odot$ is defined by

$$(x, v) \odot_a (y, w) \quad = \quad (x \uparrow (v +_a w) \uparrow a \uparrow y, (v +_a w) \uparrow a) \; .$$

The algorithm derived for *mts* can be implemented as a linear-time program.

More examples of derivations of algorithms for *treecuts* problems and *substrees* problems can be found in Jeuring [71, 73].

4.5 Conclusions

A generator fusion theorem lists sufficient conditions a catamorphism has to satisfy in order to fuse the catamorphism with the generator. A generator may be any function, but usually it is a catamorphism or a catamorphism followed by a projection function. The derivation of a generator fusion theorem for a given generator is straightforward, but does occasionally require inventive steps, and therefore cannot be done automatically. The derivation is guided by the wish to construct a catamorphism for the composition of a catamorphism and the generator. The algorithms derived by means of a generator fusion theorem include dynamic programming algorithms and divide-and-conquer algorithms.

This chapter derives at least ten generator fusion theorems and corollaries of generator fusion theorems. For most new generators it is now straightforward to derive a generator fusion theorem, given this set of examples. However, it is desirable to have theorems that given a generator return a generator fusion theorem. Such a theorem is given by De Moor and Bird in [105]. This theorem gives just one fusion theorem for each power transpose of a relational catamorphisms. Since there exist different *parts*-Fusion Theorems for function *parts* defined on *snoc-list* other theorems which can be specialised to a generator fusion theorem for a given generator should be formulated. Another subject for future research is the further development of the different generator specific theories.

Chapter 5

Segment problems

A segment of a list x is a list consisting of a number of consecutive elements from x. Formally, list y is a segment of list v if and only if there exist lists x and z such that $v = x \mathbin{+\!\!\!+} y \mathbin{+\!\!\!+} z$. In the literature, a segment is also called a subword, a factor, or a substring. Problems on segments have been studied widely; the amount of literature on just the pattern-matching problem, which requires to find occurrences of a given pattern (segments equal to the pattern) in a list, is enormous. A large number of (solutions to) segment problems can be found in the book 'Combinatorial algorithms on words' edited by Apostolico and Galil [5].

Since there is such a large number of segment problems, it is worthwhile to try to catalogue and classify the problems. Zantema [142], Van den Eijnde [38], and De Moor and Swierstra [129] deal with some subclasses of segment problems. In this section we build upon results from Bird, Gibbons, and Jones [21], developing yet another part of segment theory. We derive a number of theorems and corollaries that can be used to construct efficient algorithms for a number of segment problems such as the pattern-matching problem, the palindrome problem, which requires to find the longest palindromic segment of a list, and the problem of finding the lexicographically least rotation (rotations will be defined in terms of segments) of a list. We focus on longest-p segment problems: problems which require to find a longest segment satisfying predicate p of a list. Shortest-p segment problems are discussed by Jeuring and Meertens [79]. Although all algorithms we derive can be implemented as linear-time programs, there exist segment problems for which no linear-time solution exists, see Zantema [142].

This chapter consists of eight sections. The first section gives several functions generating the segments of a list. The second section derives a *segs*-Fusion Theorem, which is used in the third and fourth section to derive two corollaries, respectively the Sliding Tails Theorem and the Hopping Tails Theorem. These three sections also contain several worked out examples. The fifth section contains the derivation of algorithms for some more involved and well-known problems, such as the pattern-matching problem. The sixth section derives the Hopping Splits Theorem, the theorem that corresponds to the Hopping Tails Theorem when contexts of segments are taken into account. The seventh section derives an algorithm for finding the longest palindromic segment of a list, and the eighth section gives some conclusions.

5.1 Segments

A segment of a list is a list consisting of a number of consecutive elements from the list. Formally, list y is a segment of list v if and only if there exist lists x and z such that $v = x ⧺ y ⧺ z$. In the literature, a segment is also called a subword, a factor, or a substring.

There are many ways to define the function *segs* that returns all segments of a list as a set. Using set-comprehension we have

$$\mathit{segs}\; v \;=\; \{y \mid \exists x, z : x ⧺ y ⧺ z = v\}\ .$$

For example $\mathit{segs}\,[1,2,3] = \{[\,],[1],[2],[3],[1,2],[2,3],[1,2,3]\}$. Note that an element y of $\mathit{segs}\,v$ can occur in several different positions within v. If the context of a segment is important we use the function *splits3* which returns the set containing all partitions of the argument into three parts.

$$\mathit{splits3}\; v \;=\; \{(x,y,z) \mid x ⧺ y ⧺ z = v\}\ .$$

It follows that $\mathit{segs} = \pi_1 * \cdot\, \mathit{splits3}$. In fact, we define a little hierarchy of functions that partition a list.

$$\begin{aligned} \mathit{splits2}\; v \;&=\; \{(x,y) \mid x ⧺ y = v\} \\ \mathit{splits1}\; v \;&=\; \{x \mid x = v\}\ . \end{aligned} \tag{5.1}$$

The recursive characterisation of *splits3* that will be given later in this section uses the functions *splits2* and *splits1*. For that purpose, the elements of *splits2* v and *splits1* v have to be triples. This is done as follows.

$$\begin{aligned} \mathit{3splits2}\; v \;&=\; \{(x,y,[\,]) \mid x ⧺ y = v\} \\ \mathit{3splits1}\; v \;&=\; \{(x,[\,],[\,]) \mid x = v\} \end{aligned}$$

It follows that $\mathit{3splits2} = (\propto_2 [\,])* \cdot \mathit{splits2}$, and $\mathit{3splits1} = (\propto_2 [\,])* \cdot (\propto_1 [\,])* \cdot \mathit{splits1}$. Continuing in this vein, $\mathit{3splits3} = \mathit{splits3}$.

We discuss recursive characterisations of *segs* and *3splitsi*, with $i : 1 \leq i \leq 3$. Since $\uparrow_{\#}/\cdot \mathit{segs} = \mathit{id}$, function *segs* is injective, and hence it is a catamorphism, see Corollary 2.36. For the domain of the catamorphism there are three choices: *segs* can be defined on *join-list*, on *cons-list*, or on *snoc-list*. The choice of definition determines the form of the final algorithm that will be constructed for a problem specified by means of function *segs*, that is, if *segs* is defined on *join-list*, the algorithm that will be constructed will be defined on *join-list*, etc. Conversely, if we desire the final algorithm to be for example a left-reduction, we define function *segs* on the data type *snoc-list*.

The function *segs* on *join-list*

Function *segs* satisfies the following equations on the data type *join-list*.

$$\begin{array}{lcl}
segs & : & A* \longrightarrow A*\wr \\
segs\,[\,] & = & \{[\,]\} \\
segs\,[a] & = & \{[\,],[a]\} \\
segs\,(x +\!\!+\, y) & = & segs\,x \cup (tails\,x \between_{+\!+} inits\,y) \cup segs\,y \ ,
\end{array} \qquad (5.2)$$

where the functions *tails* and *inits* return respectively, given a list, all tail segments of this list as a set and all initial segments of this list as a set. Using set comprehension, *tails* and *inits* are defined by

$$\begin{array}{lcl}
tails\,x & = & \{z \mid \exists y : y +\!\!+\, z = x\} \\
inits\,x & = & \{z \mid \exists y : z +\!\!+\, y = x\} \ .
\end{array} \qquad (5.3)$$

So $tails\,[1,2,3] = \{[\,],[3],[2,3],[1,2,3]\}$, and $inits\,[1,2,3] = \{[\,],[1],[1,2],[1,2,3]\}$. Using characterisation (5.1) of function *splits2* we find that

$$tails = exr* \cdot splits2 \qquad (5.4)$$

$$inits = exl* \cdot splits2 \ . \qquad (5.5)$$

To obtain a *join-list* catamorphism or paramorphism for *segs* function *segs* is tupled with the functions *tails* and *inits*. Define function *sti* by

$$sti = segs \vartriangle tails \vartriangle inits \ .$$

Then by equation (2.16)

$$segs = \pi_0 \cdot sti \ . \qquad (5.6)$$

Recursive characterisations of *tails* and *inits* on the data type *join-list* read as follows. For *tails* we have

$$\begin{array}{lcl}
tails & : & A* \longrightarrow A*\wr \\
tails\,[\,] & = & \{[\,]\} \\
tails\,[a] & = & \{[\,],[a]\} \\
tails\,(x +\!\!+\, y) & = & (+\!\!+\,y)*\,tails\,x \cup tails\,y \ ,
\end{array} \qquad (5.7)$$

and for *inits* we have

$$\begin{array}{lcl} \mathit{inits} & : & A* \to A*\wr \\ \mathit{inits}\,[\,] & = & \{[\,]\} \\ \mathit{inits}\,[a] & = & \{[\,],[a]\} \\ \mathit{inits}\,(x +\!\!\!+\, y) & = & \mathit{inits}\,x \cup (x+\!\!\!+)*\,\mathit{inits}\,y\ . \end{array}$$

Recall Paramorphism Characterisation, Theorem 2.44. In the case of the data type *join-list*, this theorem states that function h is a paramorphism $[\![e \triangledown f \triangledown \oplus]\!]$ if and only if it satisfies

$$\begin{array}{lcl} h\,[\,] & = & e \\ h\,[a] & = & f\ a \\ h\,(x +\!\!\!+\, y) & = & (h\,x, x) \oplus (h\,y, y)\ . \end{array}$$

It follows that function *tails* and *inits* are paramorphisms, i.e.,

$$\mathit{tails} \;=\; [\![\{[\,]\} \triangledown f \triangledown \oplus]\!]\ ,$$

where function f is defined by $f\ a = \{[\,],[a]\}$, and operator $\oplus$ is defined by

$$(x,y) \oplus (u,v) \;=\; (+\!\!\!+v)*\,x \cup u\ ,$$

and

$$\mathit{inits} \;=\; [\![\{[\,]\} \triangledown f \triangledown \ominus]\!]\ ,$$

where operator $\ominus$ is defined by

$$(x,y) \ominus (u,v) \;=\; x \cup (y+\!\!\!+)*\,u\ .$$

For function *sti* we have

$$\mathit{sti} \;=\; [\![(\{[\,]\},\{[\,]\},\{[\,]\}) \triangledown (f \vartriangle f \vartriangle f) \triangledown \otimes]\!]\ ,$$

where operator $\otimes$ is defined by

$$(x,y,z,s) \otimes (u,v,w,t) \;=\; (x \cup (y \mathbin{\chi_{+\!\!\!+}} w) \cup u, (y,s) \oplus (v,t), (z,s) \ominus (w,t))\ .$$

Smith [125] gives the same characterisation of *segs* in another setting. After some experimentations we decided to always include the empty list in the result of functions *tails* and *inits*. Had we decided never to include the empty list in the result of functions *tails* and *inits* we would have obtained a marginally different and slightly more disagreeable definition.

Probably the most well-known characterisation of *segs* is given by Bird [16]. Function *segs* satisfies

$$\mathit{segs} \;=\; \cup/ \cdot \mathit{tails}* \cdot \mathit{inits}\ . \tag{5.8}$$

Switching the roles of *inits* and *tails* gives a similar characterisation of *segs*:

$$segs \;=\; \cup/ \cdot inits* \cdot tails\ . \tag{5.9}$$

Bird uses characterisation (5.8) in [16] to derive left-reductions for problems involving segments. For the derivation of left-reductions for problems involving segments the following characterisation of *segs* is more suitable: the derivations tend to be shorter and a bit more straightforward. In some cases, however, Bird's characterisation is preferable.

The function *segs* on *snoc-list*

For the derivation of on-line algorithms for problems involving segments we define *segs* as a left-reduction. If we apply Bird's characterisation of the function *segs* to the list $x \mathbin{+\!\!+} [a]$ we obtain

$$segs\,(x \mathbin{+\!\!+} [a]) \;=\; segs\,x \cup tails\,(x \mathbin{+\!\!+} [a])\ .$$

So, two characterising equations for *segs* on the data type *snoc-list* are

$$\begin{array}{lcl} segs & : & A\star \rightarrow A\star\wr \\ segs\,[\,] & = & \{[\,]\} \\ segs\,(x \mathbin{-\!\!<} a) & = & segs\,x \cup tails\,(x \mathbin{-\!\!<} a)\ . \end{array} \tag{5.10}$$

We define function *tails* as a left-reduction. Its characterising equations are

$$\begin{array}{lcl} tails & : & A\star \rightarrow A\star\wr \\ tails\,[\,] & = & \{[\,]\} \\ tails\,(x \mathbin{-\!\!<} a) & = & (\mathbin{-\!\!<} a)*\,tails\,x \cup \{[\,]\}\ . \end{array} \tag{5.11}$$

If operator $\odot$ is defined by

$$x \odot a \;=\; (\mathbin{-\!\!<} a)*\,x \cup \{[\,]\}\ ,$$

then function *tails* is defined by

$$tails \;=\; \odot\!\not\to\!\{[\,]\}\ .$$

If we replace list x occurring in $tails\,(x \mathbin{-\!\!<} a)$ in the characterising equations for *segs* by $\uparrow_{\#}/\ segs\,x$ we can extract a definition as a left-reduction for *segs* from the two equations. Because this definition is rather awkward it is omitted. The characterising equations suggest an alternative definition. From equations (5.10) and (5.11) it follows that functions *segs* and *tails* are mutumorphisms, and applying Theorem 2.47 we obtain

$$segs \vartriangle tails \;=\; \otimes\!\not\to\!(\{[\,]\},\{[\,]\})\ ,$$

where the operator $\otimes$ is defined by

$$(x,y) \otimes a \;=\; (x \cup (y \odot a),\, y \odot a)\ .$$

Abbreviate the tuple of functions $segs \vartriangle tails$ to st,

$$st \;=\; segs \vartriangle tails\ .$$

Then by equation (2.16)

$$segs \;=\; exl \cdot st\ . \tag{5.12}$$

Segments with contexts: the *3splitsi*-functions

If the context of a segment is important, we use the function *3splits3*, which splits a list into three contiguous segments. We want to characterise *3splits3* as a (composition of a projection function with a) left-reduction. We have

$$\begin{array}{lcl}
\mathit{3splits3} & : & A\star \rightarrow (A\star \times A\star \times A\star)\wr \\
\mathit{3splits3}\,[\,] & = & \{([\,],[\,],[\,])\} \\
\mathit{3splits3}\,(x \mathbin{-\!\!\!<} a) & = & (\mathbin{-\!\!\!<}_3 a)* \mathit{3splits3}\ x \cup (\mathbin{-\!\!\!<}_2 a)* \mathit{3splits2}\ x \cup (\mathbin{-\!\!\!<}_1 a)* \mathit{3splits1}\ x\ ,
\end{array}$$

where sections $(\mathbin{-\!\!\!<}_3 a)$, $(\mathbin{-\!\!\!<}_2 a)$, and $(\mathbin{-\!\!\!<}_1 a)$ are respectively defined by

$$\begin{array}{lcl}
(\mathbin{-\!\!\!<}_3 a) & = & id \times id \times (\mathbin{-\!\!\!<} a) \\
(\mathbin{-\!\!\!<}_2 a) & = & id \times (\mathbin{-\!\!\!<} a) \times id \\
(\mathbin{-\!\!\!<}_1 a) & = & (\mathbin{-\!\!\!<} a) \times id \times id\ .
\end{array}$$

The last equation in the characterisation of *3splits3* shows that the three functions *3splits3*, *3splits2*, and *3splits1* are mutumorphisms. Let *sss* be defined by

$$sss \;=\; \mathit{3splits3} \vartriangle \mathit{3splits2} \vartriangle \mathit{3splits1}\ .$$

Abbreviate $([\,],[\,],[\,])$ to $[\,]^3$. Applying Theorem 2.47 twice and some simplification gives

$$sss \;=\; \oplus\!\not\to\!(\{[\,]^3\},\{[\,]^3\},\{[\,]^3\})\ , \tag{5.13}$$

where operator $\oplus$ is defined by

$$\begin{array}{lcl}
(x_3,x_2,x_1) \oplus a & = & (r_3 \cup r_2 \cup r_1, r_2 \cup r_1, r_1) \\
 & & \quad \textbf{where } r_i = (\mathbin{-\!\!\!<}_i a)* x_i\ ,
\end{array}$$

with $i : 1 \le i \le 3$.

5.2 *segs*-Fusion

The *segs*-Fusion Theorem given in this section is the main means with which solutions for segment problems are constructed. This theorem induces three corollaries, the first of which, called Horner's rule, is given in this section. We devote two separate sections to two other corollaries.

Segment problems

The generic specification of segment problems is given by

$$\oplus/ \cdot f* \cdot \mathit{segs}\ , \tag{5.14}$$

for arbitrary operator $\oplus$ and function f. Given a specific operator $\oplus$ and function f, the specification can be implemented as a functional program. Given a list x of length n, this program requires at least (and almost always more than) time $\Omega(n^2)$ to find the value of $\oplus/ f* \mathit{segs}\, x$, since *segs* returns a set containing $\Omega(n^2)$ segments.

Deriving a left-reduction for a segment problem

Since the straightforward implementation of a segment problem is almost always an inefficient program, we aim at finding conditions on operator $\oplus$ and function f such that the function specified in equation (5.14) equals a left-reduction that can be implemented as an efficient program. Since *segs* has not been defined as a left-reduction we cannot proceed as we did in most of the derivations in the previous chapter. Using equation (2.14) we derive for arbitrary function g

$$\begin{array}{ll} & \oplus/ \cdot f* \cdot \mathit{segs} \\ = & \quad \text{equation (5.12)} \\ & \oplus/ \cdot f* \cdot \mathit{exl} \cdot \mathit{st} \\ = & \quad \text{equation (2.14)} \\ & \mathit{exl} \cdot (\oplus/ \cdot f*) \times g \cdot \mathit{st}\ . \end{array}$$

For g we can choose any function that suits us, and in the subsequent derivation a natural candidate will emerge. Function *st* is a left-reduction, so we can apply Fusion on *snoc-list* to the expression $(\oplus/ \cdot f*) \times g \cdot \mathit{st}$. We have

$$(\oplus/ \cdot f*) \times g \cdot \mathit{st} \quad = \quad \oslash\!\not\to\!(((\oplus/ \cdot f*) \times g)\,(\{[\,]\}, \{[\,]\}))\ ,$$

provided operator $\oslash$ satisfies for all (x, y) in the image of *st*, and for all a

$$((\oplus/ \cdot f*) \times g)\,((x, y) \otimes a) \quad = \quad ((\oplus/ \cdot f*) \times g)\,(x, y) \oslash a\ , \tag{5.15}$$

where

$$\begin{array}{lll} (x, y) \otimes a & = & (x \cup (y \odot a), y \odot a) \\ y \odot a & = & (\prec\!\!\!+ a)* \, y \cup \{[\,]\}\ . \end{array}$$

Note that since (x, y) is in the image of *st*, it follows that y is in the image of *tails*. For the seed of the left-reduction we have

$$
\begin{array}{ll}
 & ((\oplus/\cdot f*)\times g)\,(\{[\,]\},\{[\,]\}) \\
= & \quad \text{definition of } \times \\
 & (\oplus/\,f*\{[\,]\},g\,\{[\,]\}) \\
= & \quad \text{definition of catamorphism} \\
 & (f\,[\,],g\,\{[\,]\})\ .
\end{array}
$$

The definition of an operator $\oslash$ satisfying equation (5.15) is synthesised as follows. Meanwhile we will find a candidate for function g.

$$
\begin{array}{ll}
 & ((\oplus/\cdot f*)\times g)\,((x,y)\otimes a) \\
= & \quad \text{definition of } \otimes \\
 & ((\oplus/\cdot f*)\times g)\,(x\cup(y\odot a),y\odot a) \\
= & \quad \text{definition of } \times \\
 & (\oplus/\,f*\,(x\cup(y\odot a)),g\,(y\odot a)) \\
= & \quad \text{definition of catamorphism} \\
 & (\oplus/\,f*\,x\oplus\oplus/\,f*\,(y\odot a),g\,(y\odot a))\ .
\end{array}
$$

In the last expression of this calculation we can distinguish three subexpressions:

$$
\begin{array}{l}
\oplus/\,f*\,x \\
\oplus/\,f*\,(y\odot a) \\
g\,(y\odot a)\ .
\end{array}
$$

In view of the form of the desired expression the first subexpression $\oplus/\,f*\,x$ need not be developed any further. The subexpressions $\oplus/\,f*\,(y\odot a)$ and $g\,(y\odot a)$ have to be expressed in terms of $\oplus/\,f*\,x$, $g\,y$ and a. For $\oplus/\,f*\,(y\odot a)$ we have

$$
\begin{array}{ll}
 & \oplus/\,f*\,(y\odot a) \\
= & \quad \text{definition of } \odot \\
 & \oplus/\,f*\,((-\!\!\!<a)*\,y\cup\{[\,]\}) \\
= & \quad \text{definition of catamorphism} \\
 & \oplus/\,f*\,(-\!\!\!<a)*\,y\oplus f\,[\,] \\
= & \quad \textbf{assume } \oplus/\cdot f*\cdot(-\!\!\!<a)* = (\ominus a)\cdot\oplus/\cdot f* \\
 & (\ominus a)\,\oplus/\,f*\,y\oplus f\,[\,]\ ,
\end{array}
$$

for some operator $\ominus$. So $\oplus/\,f*\,(y\odot a)$ can be expressed in terms of $\oplus/\,f*\,y$ and a, provided

$$
\oplus/\cdot f*\cdot(-\!\!\!<a)* \quad = \quad (\ominus a)\cdot\oplus/\cdot f*\ ,
$$

after *tails*. It should have been expressed in terms of $\oplus/\, f{*}\, x$, $g\, y$ and a. Since the only expression involving y is $g\, y$, a natural choice for function g is

$$g \;=\; \oplus/ \cdot f{*}\,. \tag{5.16}$$

An additional advantage of this particular choice is that $g\,(y \odot a)$ has been expressed in terms of $g\, y$ and a above. Thus we have found that operator $\oslash$ can be defined by

$$(x,y) \oslash a \;=\; (x \oplus (y ⊜ a) \oplus u, (y ⊜ a) \oplus u)\ , \tag{5.17}$$

where u abbreviates $f\,[\,]$. We summarise the derivation in the following theorem.

(5.18) Theorem (*segs*-Fusion) *Suppose operator* $⊜$ *satisfies* $\oplus/ \cdot f{*} \cdot (\mathbin{-\!\!\prec} a){*} = (⊜ a) \cdot \oplus/ \cdot f{*}$ *after tails. Then*

$$\begin{aligned}
(\oplus/ \cdot f{*})_{\|} \cdot st &= \oslash\!\not\rightarrow\!(u,u)\\
\oplus/ \cdot f{*} \cdot segs &= exl \cdot \oslash\!\not\rightarrow\!(u,u)\\
\oplus/ \cdot f{*} \cdot tails &= exr \cdot \oslash\!\not\rightarrow\!(u,u)\ ,
\end{aligned}$$

where u abbreviates $f\,[\,]$, and operator $\oslash$ is defined by

$$(x,y) \oslash a \;=\; (x \oplus (y ⊜ a) \oplus u, (y ⊜ a) \oplus u)\ .$$

Note that when evaluating the left-reduction $\oslash\!\not\rightarrow\!(u,u)$ we may assume that all left-hand arguments of operator $\oslash$ are of the form $(\oslash\!\not\rightarrow\!(u,u))\, x$ for some list x

A result for tail problems

One of the results of the *segs*-Fusion Theorem is that

$$\oplus/ \cdot f{*} \cdot tails \;=\; exr \cdot \oslash\!\not\rightarrow\!(u,u)\ .$$

Since

$$exr\,((x,y) \oslash a) \;=\; y \ominus a\ ,$$

where operator $\ominus$ is defined by

$$x \ominus a \;=\; (x ⊜ a) \oplus u\ ,$$

we can apply Fusion on *snoc-list* to obtain

$$\oplus/ \cdot f{*} \cdot tails \;=\; \ominus\!\not\rightarrow\! u\ . \tag{5.19}$$

Horner's rule

The equality

$$\oplus/ \cdot (f \cdot (\mathbin{-\!\!\!\!\prec} a))* \quad = \quad (\circledequal a) \cdot \oplus/ \cdot f* \, , \tag{5.20}$$

assumed in the *segs*-Fusion Theorem can be split into separate assumptions on the constituents $\oplus$ and f. There are various ways to obtain equality (5.20); we will show at least three different definitions of an operator $\circledequal$ satisfying equation (5.20). Corollary 2.65 lists two conditions which imply equality (5.20). If f is a left-reduction $\circledequal\!\nrightarrow\! u$ such that for all a, section $(\circledequal a)$ is $(\oplus, \oplus)$-fusable, then equality (5.20) holds. Equality (5.20) is only required to hold on the image of *tails*, so $(\circledequal a)$ needs only be $(\oplus, \oplus)$-fusable after $\oplus/ \cdot f* \cdot (\in \textit{tails}\ z)\triangleleft$ for all lists z. The following theorem is a direct application of Corollary 2.65 and the *segs*-Fusion Theorem.

(5.21) Theorem (Horner's rule) *Let f be a left-reduction $\circledequal\!\nrightarrow\! u$ such that section $(\circledequal a)$ is $(\oplus, \oplus)$-fusable after $\oplus/ \cdot f* \cdot (\in \textit{tails}\ z)\triangleleft$ for all lists z. Then*

$$\begin{aligned}
(\oplus/ \cdot f*)\| \cdot st &= \oslash\!\nrightarrow\!(u, u) \\
\oplus/ \cdot f* \cdot segs &= exl \cdot \oslash\!\nrightarrow\!(u, u) \\
\oplus/ \cdot f* \cdot tails &= exr \cdot \oslash\!\nrightarrow\!(u, u) \, ,
\end{aligned}$$

where operator $\oslash$ is defined by

$$(x, y) \oslash a \quad = \quad (x \oplus (y \circledequal a) \oplus u, (y \circledequal a) \oplus u) \, .$$

This theorem is a reformulation (with a new proof) of a theorem, which Bird [17] calls Horner's rule, in which a left-reduction for the function $\oplus/ \cdot f* \cdot \textit{tails}$ is derived. Horner's rule can be applied to the maximum segment sum problem.

The maximum segment sum problem

The maximum segment sum problem requires finding a sum of a segment with maximum sum. It is specified by

$$mss \quad = \quad \uparrow/ \cdot (+\!\nrightarrow\! 0)* \cdot segs \, . \tag{5.22}$$

Since section $(+a)$ is $(\uparrow, \uparrow)$-fusable for all numbers a, we have

$$mss \quad = \quad exl \cdot \oslash\!\nrightarrow\!(0, 0) \, ,$$

where the operator $\oslash$ is defined by

$$(x, y) \oslash a \quad = \quad (x \uparrow (y + a) \uparrow 0, (y + a) \uparrow 0) \, .$$

When implemented, this algorithm requires time $\Omega(n)$ when applied to a list of length n.

5.3 The Sliding Tails Theorem

This section derives a corollary of the *segs*-Fusion Theorem for a subclass of the class of segment problems.

Longest-p segment problems

Consider the problem of finding a longest ascending segment; it is specified by

$$las \quad = \quad \uparrow_\# / \cdot asc\triangleleft \cdot segs \ , \tag{5.23}$$

where the predicate *asc* is defined in equation (2.85). This problem belongs to the subclass of segment problems specified by

$$\uparrow_\# / \cdot p\triangleleft \cdot segs \ , \tag{5.24}$$

where p is an arbitrary predicate. A problem of this form is called a *longest-p segment problem.*

Deriving a left-reduction for a longest-p segment problem

We want to derive a left-reduction that can be implemented as an efficient program for a given longest-p segment problem. It is possible to apply Horner's rule to transform expression (5.24) to a simple expression involving a left-reduction, but this would impose severe restrictions on predicate p. We will show that specific, but less severe, properties of predicate p guide the construction of an operator ⊜ satisfying the condition of *segs*-Fusion, Theorem 5.18. We start afresh from the *segs*-Fusion Theorem.

$$\uparrow_\# / \cdot p\triangleleft \cdot segs \quad = \quad exl \cdot \oslash\!\!\not\rightarrow\!(u, u) \ , \tag{5.25}$$

where $u = p?_{\uparrow_\#}\,[\,]$, provided

$$\uparrow_\# / \cdot p\triangleleft \cdot (-\!\!<\! a)* \quad = \quad (⊜ a) \cdot \uparrow_\# / \cdot p\triangleleft \ , \tag{5.26}$$

after *tails*. Our goal is to prove equation (5.26), thus proving equation (5.25).

$$\begin{array}{ll}
 & \uparrow_\# / \cdot p\triangleleft \cdot (-\!\!<\! a)* \cdot tails \\
= & \quad \text{map-filter swap (2.73)} \\
 & \uparrow_\# / \cdot (-\!\!<\! a)* \cdot (p \cdot (-\!\!<\! a))\triangleleft \cdot tails \\
= & \quad \textbf{assume } p \text{ is prefix-closed, } (p \wedge q)\triangleleft = p\triangleleft \cdot q\triangleleft \\
 & \uparrow_\# / \cdot (-\!\!<\! a)* \cdot (p \cdot (-\!\!<\! a))\triangleleft \cdot p\triangleleft \cdot tails \\
= & \quad \textbf{assume } p\triangleleft \cdot tails = h \cdot \uparrow_\# / \cdot p\triangleleft \cdot tails \text{ for some } h
\end{array}$$

$$\begin{array}{ll}
 & \uparrow_{\#}/ \cdot (\mathbin{-\!\!<} a)* \cdot (p \cdot (\mathbin{-\!\!<} a))\triangleleft \cdot h \cdot \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{tails} \\
= & \quad \text{definition below} \\
 & (\circledequal a) \cdot \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{tails}\ ,
\end{array}$$

where operator $\circledequal$ is assumed to be defined by

$$x \circledequal a \quad = \quad \uparrow_{\#}/ \, (\mathbin{-\!\!<} a)* \, (p \cdot (\mathbin{-\!\!<} a))\triangleleft \, h \, x\ . \tag{5.27}$$

The two assumptions under which equality (5.26) holds are

- p is prefix-closed;
- there exists a function h such that

$$p\triangleleft \cdot \mathit{tails} \quad = \quad h \cdot \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{tails}\ . \tag{5.28}$$

A function h satisfying equation (5.28) is the function $p\triangleleft \cdot \mathit{tails}$, that is, we have

$$p\triangleleft \cdot \mathit{tails} \quad = \quad p\triangleleft \cdot \mathit{tails} \cdot \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{tails}\ , \tag{5.29}$$

which states that the set of all tails satisfying an arbitrary predicate p equals the set of all tails satisfying p of the longest tail satisfying p. This equation is proved as follows. Let z be $\uparrow_{\#}/ \, p\triangleleft \, \mathit{tails}\, x$, and suppose that $x = y \mathbin{+\!\!\!+\!\!<} z$. Then, using characterisation (5.7) of *tails* (on the data type *snoc-list* instead of the data type *join-list*),

$$\begin{array}{ll}
 & p\triangleleft \, \mathit{tails}\, x \\
= & \quad \text{characterisation (5.7) of } \mathit{tails}\text{, definition of filter} \\
 & (p\triangleleft \, \mathit{tails}\, z) \cup (p\triangleleft \, (\mathbin{+\!\!\!+\!\!<} z)* \, \mathit{tails}\, y) \\
= & \quad \text{definition of } z,\ z \leq_{\#} (\mathbin{+\!\!\!+\!\!<} z)* \, \mathit{tails}\, y \\
 & p\triangleleft \, \mathit{tails}\, z\ .
\end{array}$$

We rewrite the definition of operator $\circledequal$ as follows. First, substituting the definition of h, we obtain

$$\begin{array}{ll}
 & x \circledequal a \\
= & \quad \text{definition of } \circledequal \text{ (5.27)} \\
 & \uparrow_{\#}/ \, (\mathbin{-\!\!<} a)* \, (p \cdot (\mathbin{-\!\!<} a))\triangleleft \, h \, x \\
= & \quad \text{definition of } h \text{ (5.29)} \\
 & \uparrow_{\#}/ \, (\mathbin{-\!\!<} a)* \, (p \cdot (\mathbin{-\!\!<} a))\triangleleft \, p\triangleleft \, \mathit{tails}\, x \\
= & \quad p \text{ is prefix-closed, } p\triangleleft \cdot q\triangleleft = (p \wedge q)\triangleleft \\
 & \uparrow_{\#}/ \, (\mathbin{-\!\!<} a)* \, (p \cdot (\mathbin{-\!\!<} a))\triangleleft \, \mathit{tails}\, x \\
= & \quad \text{map-filter swap (2.73)} \\
 & \uparrow_{\#}/ \, p\triangleleft \, (\mathbin{-\!\!<} a)* \, \mathit{tails}\, x\ .
\end{array}$$

Distinguish the three cases $p\,(x \mathbin{-\!\!\!<} a)$, $\neg p\,(x \mathbin{-\!\!\!<} a) \;\wedge\; x \neq [\,]$, and $\neg p\,(x \mathbin{-\!\!\!<} a) \;\wedge\; x = [\,]$ in the definition of operator $\circledequal$.

- $p\,(x \mathbin{-\!\!\!<} a)$. Then,

$$\begin{array}{cl} & x \circledequal a \\ = & \quad \text{definition of } \circledequal \\ & \uparrow_{\#}/\; p \triangleleft (\mathbin{-\!\!\!<} a) * \mathit{tails}\; x \\ = & \quad \text{case assumption} \\ & x \mathbin{-\!\!\!<} a\ . \end{array}$$

- $\neg p\,(x \mathbin{-\!\!\!<} a)$ and $x \neq [\,]$. Then

$$\begin{array}{cl} & x \circledequal a \\ = & \quad \text{definition of } \circledequal \\ & \uparrow_{\#}/\; p \triangleleft (\mathbin{-\!\!\!<} a) * \mathit{tails}\; x \\ = & \quad \text{case assumption} \\ & \uparrow_{\#}/\; p \triangleleft (\mathbin{-\!\!\!<} a) * \mathit{tails}\; \mathit{tl}\; x \\ = & \quad \text{definition of } \circledequal \\ & (\mathit{tl}\; x) \circledequal a\ . \end{array}$$

- $\neg p\,(x \mathbin{-\!\!\!<} a)$ and $x = [\,]$. Then

$$\begin{array}{cl} & x \circledequal a \\ = & \quad \text{definition of } \circledequal \\ & \uparrow_{\#}/\; p \triangleleft (\mathbin{-\!\!\!<} a) * \mathit{tails}\; [\,] \\ = & \quad \text{case assumption} \\ & \omega\ , \end{array}$$

  where $\omega = \nu_{\uparrow_{\#}}$.

Define operator $\ominus$ by

$$x \ominus a \quad = \quad (x \circledequal a) \uparrow_{\#} [\,]\ .$$

A simple proof by induction on the length of list x shows that operator $\ominus$ satisfies

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\!<} a & \text{if } p\,(x \mathbin{-\!\!\!<} a) \\ (\mathit{tl}\; x) \ominus a & \text{if } \neg p\,(x \mathbin{-\!\!\!<} a) \;\wedge\; x \neq [\,] \\ [\,] & \text{otherwise}\ . \end{cases} \tag{5.30}$$

We obtain the following theorem from the *segs*-Fusion Theorem.

(5.31) Theorem (Sliding Tails) *Let p be a prefix-closed predicate. Then*

$$\begin{array}{rcl}
(\uparrow_{\#}/ \cdot p\triangleleft)\| \cdot st & = & \oslash\not\rightarrow([\,],[\,]) \\
\uparrow_{\#}/ \cdot p\triangleleft \cdot segs & = & exl \cdot \oslash\not\rightarrow([\,],[\,]) \\
\uparrow_{\#}/ \cdot p\triangleleft \cdot tails & = & \ominus\not\rightarrow[\,]\ ,
\end{array}$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \quad = \quad (x \uparrow_{\#} (y \ominus a), y \ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{+\!\!\!<} a & \textit{if } p\,(x \mathbin{+\!\!\!<} a) \\ (tl\,x) \ominus a & \textit{if } \neg p\,(x \mathbin{+\!\!\!<} a) \ \wedge\ x \neq [\,] \\ [\,] & \textit{otherwise}\ . \end{cases}$$

The name of this theorem refers to a visual interpretation of the algorithm $\oslash\not\rightarrow([\,],[\,])$. Given a list, this left-reduction proceeds from left to right, at each point returning for the scanned part of the list the longest segment satisfying p and the longest tail satisfying p. The longest tail satisfying p is either the extension of the longest tail satisfying p at the previous point, or the extension of the tail of that tail, or the extension of the tail of that tail of that tail, etc. Thus the longest tail 'slides' over the argument.

If we take a cons-view on its argument, section $(\ominus a)$ can be defined as a *cons-list* paramorphism. We have

$$(\ominus a) \quad = \quad [\![(f\ a) \triangledown \oplus_a]\!]\ ,$$

where function f is defined by

$$f\ a \quad = \quad \begin{cases} [a] & \text{if } p\,[a] \\ [\,] & \text{otherwise}\ , \end{cases}$$

and operator $\oplus$ is defined by

$$b \oplus_a (y,x) \quad = \quad \begin{cases} b \mathbin{>\!\!\!+} x \mathbin{>\!\!\!+\!\!\!+} [a] & \text{if } p\,(b \mathbin{>\!\!\!+} x \mathbin{>\!\!\!+\!\!\!+} [a]) \\ y & \text{otherwise}\ , \end{cases}$$

where operator $\mathbin{>\!\!\!+\!\!\!+}$ defined on the data type *cons-list* is the 'converse' of operator $\mathbin{+\!\!\!+\!\!\!<}$.

Example: the longest ascending segment problem

The Sliding Tails Theorem can be used to give an algorithm for the problem of finding a longest ascending segment. This problem is specified in (5.23) by

$$las \quad = \quad \uparrow_{\#}/ \cdot asc\triangleleft \cdot segs\ .$$

Since *asc* is prefix-closed, the Sliding Tails Theorem states that

$$las \quad = \quad exl \cdot \oslash\!\not\rightarrow([\,],[\,]) \ ,$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \quad = \quad (x \uparrow_{\#} (y \ominus a), y \ominus a) \ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{+\!\!\!<} a & \text{if } asc\,(x \mathbin{+\!\!\!<} a) \\ (tl\,x) \ominus a & \text{if } \neg asc\,(x \mathbin{+\!\!\!<} a) \ \wedge \ x \neq [\,] \\ [\,] & \text{otherwise} \ . \end{cases}$$

Given a list of length n, a straightforward implementation of this algorithm requires time $\Omega(n^2)$. The number of evaluations of $\oslash$ and $\ominus$ is linear, but each evaluation of $\oslash$ and $\ominus$ requires linear time, since evaluating $\uparrow_{\#}$ requires linear time and determining whether $asc\,(x \mathbin{+\!\!\!<} a)$ holds requires linear time. Suppose that each list is tupled with its length. Then we can replace operator $\uparrow_{\#}$ by $\uparrow_{\pi_1}$, and $\uparrow_{\pi_1}$ can be evaluated in constant time. A technique for replacing $asc\,(x \mathbin{+\!\!\!<} a)$ by an expression that can be implemented as a program that can be evaluated in constant time is described in the following section.

Straightforward implementations of many of the functions obtained from the application of the Sliding Tails Theorem, Theorem 5.31, to longest-p segment problems with prefix-closed predicate p are not linear-time programs. The expensive part of the algorithms corresponds to operator $\ominus$, in which the value $p\,(x \mathbin{+\!\!\!<} a)$ is required. Determining whether or not $p\,(x \mathbin{+\!\!\!<} a)$ holds for predicates often costs time at least linear in the length of $x \mathbin{+\!\!\!<} a$. In the following section the notion of derivative of a prefix-closed predicate is used to obtain more efficient ways to determine the value of $p\,(x \mathbin{+\!\!\!<} a)$.

5.4 The Hopping Tails Theorem

Many of the solutions for segment problems obtained by means of the theory developed until now, the *segs*-Fusion Theorem and the Sliding Tails Theorem, are not optimal.

Consider the case in which the Sliding Tails Theorem is applied to a longest-p segment problem with prefix-closed predicate p. Since $p\,(x \mathbin{+\!\!\!<} a)$ holds only if $p\,x$ holds, it follows that $x \ominus a$ evaluates to $(tl\,x) \ominus a$ as long as $p\,x$ does not hold (and $x \neq [\,]$). So $x \ominus a = y \ominus a$, where y is the longest tail of x satisfying p. Instead of evaluating operator $\ominus$ as often as $\#\,x - \#\,y$ times, it can be concluded immediately (that is, in one step) that $x \ominus a = (lpt\ tl\,x) \ominus a$, where function lpt is defined by

$$lpt \quad = \quad \uparrow_{\#}/ \cdot p\triangleleft \cdot tails \ . \tag{5.32}$$

This observation leads to efficient algorithms for some problems. We prove this informal argument. Recall from the previous section, see equation (5.26), that for prefix-closed predicate p

$$\uparrow_{\#}/ \cdot p\triangleleft \cdot (\mathbin{+\!\!\!<} a)* \cdot \mathit{tails} \quad = \quad (\ominus a) \cdot \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{tails} \ ,$$

where operator $\ominus$ is defined by

$$x \ominus a \quad = \quad \uparrow_{\#}/\, p\triangleleft (\mathbin{+\!\!\!<} a)* \, x \ .$$

We develop an alternative definition of operator $\ominus$. Distinguish again the three cases $p\,(x \mathbin{+\!\!\!<} a)$, $\neg p\,(x \mathbin{+\!\!\!<} a) \ \wedge\ x \neq [\,]$, and $\neg p\,(x \mathbin{+\!\!\!<} a) \ \wedge\ x = [\,]$.

- $p\,(x \mathbin{+\!\!\!<} a)$. This case remains unchanged, so

$$x \ominus a \quad = \quad x \mathbin{+\!\!\!<} a \ .$$

- $\neg p\,(x \mathbin{+\!\!\!<} a)$ and $x \neq [\,]$. In this case we apply equation (5.26) again.

$$\begin{array}{ll} & x \ominus a \\ = & \quad \text{definition of } \ominus \\ & \uparrow_{\#}/\, p\triangleleft (\mathbin{+\!\!\!<} a)* \, \mathit{tails}\ x \\ = & \quad \text{case assumption} \\ & \uparrow_{\#}/\, p\triangleleft (\mathbin{+\!\!\!<} a)* \, \mathit{tails}\ \mathit{tl}\ x \\ = & \quad \text{equation (5.26)} \\ & (\mathit{lpt}\ \mathit{tl}\ x) \ominus a \ . \end{array}$$

- $\neg p\,(x \mathbin{+\!\!\!<} a)$ and $x = [\,]$. This case remains unchanged, so

$$x \ominus a \quad = \quad \omega \ .$$

It follows that operator $\ominus$ may be defined by

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{+\!\!\!<} a & \text{if } p\,(x \mathbin{+\!\!\!<} a) \\ (\mathit{lpt}\ \mathit{tl}\ x) \ominus a & \text{if } \neg p\,(x \mathbin{+\!\!\!<} a) \ \wedge\ x \neq [\,] \\ \omega & \text{otherwise} \ . \end{cases} \tag{5.33}$$

Define operator $\circleddash$ by

$$x \circleddash a \quad = \quad (x \ominus a) \uparrow_{\#} [\,] \ .$$

A simple proof by induction on the length of x shows that operator $\circleddash$ satisfies

$$x \circleddash a \quad = \quad \begin{cases} x \mathbin{+\!\!\!<} a & \text{if } p\,(x \mathbin{+\!\!\!<} a) \\ (\mathit{lpt}\ \mathit{tl}\ x) \circleddash a & \text{if } \neg p\,(x \mathbin{+\!\!\!<} a) \ \wedge\ x \neq [\,] \\ [\,] & \text{otherwise} \ . \end{cases} \tag{5.34}$$

For many predicates p, determining whether or not $p\,(x \mathbin{-\!\!\prec} a)$ holds requires time at least linear in the length of the list $x \mathbin{-\!\!\prec} a$. Since the left-hand argument of operator $\ominus$ always satisfies predicate p, we try to use this information to replace the occurrences of $p\,(x \mathbin{-\!\!\prec} a)$ by an expression that often can be implemented as a more efficient program. The definition of operator $\ominus$ is adjusted as follows.

When evaluating the left-reduction $\oslash\!\not\rightarrow\!(u, u)$ obtained from an application of the *segs*-Fusion Theorem to a longest-p segment problem with prefix-closed predicate p, we may assume that a left-hand argument of operator $\oslash$ is of the form $(\uparrow_{\#}/\ p \triangleleft \mathit{segs}\ x, \uparrow_{\#}/\ p \triangleleft \mathit{tails}\ x)$ for some list x. This implies that, when evaluating the left-reduction $\oslash\!\not\rightarrow\!(u, u)$, a left-hand argument of operator $\ominus$ is a list satisfying p. Recall the definition of derivative (2.86). Derivative δ of prefix-closed predicate p satisfies $p\,(x \mathbin{-\!\!\prec} a) \equiv p\,x \ \wedge\ \delta_x\,a$. Since predicate p is prefix-closed, and since $p\,x$ holds whenever $x \ominus a$ is evaluated, it is sufficient to verify $\delta_x\,a$ instead of $p\,(x \mathbin{-\!\!\prec} a)$. Hence operator $\ominus$ equals

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\prec} a & \text{if } \delta_x\,a \\ (\mathit{lpt}\ \mathit{tl}\ x) \ominus a & \text{if } \neg\delta_x\,a \ \wedge\ x \neq [\,] \\ [\,] & \text{otherwise} \ . \end{cases} \tag{5.35}$$

Since operator $⊜$ defined in equation (5.33) satisfies equation (5.26), the following theorem is another consequence of the *segs*-Fusion Theorem.

(5.36) Theorem (Hopping Tails) *Let p be a prefix-closed predicate. Then*

$$\begin{array}{lcl} (\uparrow_{\#}/ \cdot p\triangleleft)\| \cdot \mathit{st} & = & \oslash\!\not\rightarrow\!([\,], [\,]) \\ \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{segs} & = & \mathit{exl} \cdot \oslash\!\not\rightarrow\!([\,], [\,]) \\ \uparrow_{\#}/ \cdot p\triangleleft \cdot \mathit{tails} & = & \ominus\!\not\rightarrow\![\,] \ , \end{array}$$

where operator $\oslash$ is defined by

$$(x, y) \oslash a \quad = \quad (x \uparrow_{\#} (y \ominus a), y \ominus a) \ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\prec} a & \textit{if } \delta_x\,a \\ (\mathit{lpt}\ \mathit{tl}\ x) \ominus a & \textit{if } \neg\delta_x\,a \ \wedge\ x \neq [\,] \\ [\,] & \textit{otherwise} \ . \end{cases}$$

Again, the name of this theorem refers to a visual interpretation of the algorithm $\oslash\!\not\rightarrow\!([\,], [\,])$. Given a list, this left-reduction proceeds from left to right, at each point returning for the part of the list scanned the longest segment satisfying p and the longest tail satisfying p. The longest tail satisfying p is either the extension of the longest tail satisfying p at the previous point, or the extension of the longest tail satisfying p of the tail of that tail, etc. Thus the longest tail satisfying p 'hops' over the argument.

This theorem is just a first step towards an efficient algorithm for $\uparrow_{\#}/\cdot p\triangleleft\cdot \mathit{segs}$. Let p be some prefix-closed predicate. The straightforward implementation of the algorithm obtained by means of Theorem 5.36 is almost always not a linear-time program. The inefficient part of this program is the part corresponding to operator $\ominus$. Determining whether or not $\delta_x\, a$ holds may be expensive, and the same holds for finding value $\mathit{lpt}\ \mathit{tl}\, x$. So, given a longest-$p$ segment problem with prefix-closed predicate p, after applying Theorem 5.36 these two problems have to be addressed.

Two corollaries of the Hopping Tails Theorem

If a prefix-closed predicate p satisfies some specific additional properties, we can derive useful corollaries of the Hopping Tails Theorem. The result of an application of the corollaries is an algorithm which very often can be implemented as an efficient program.

Suppose predicate p is not only prefix-closed, but also suffix-closed (and hence segment-closed). Then, since $p\, x$ is *true* for all x occurring in the expression $x \ominus a$, and since p is suffix-closed, it follows that $\mathit{lpt}\ \mathit{tl}\, x = \mathit{tl}\, x$. Hence we obtain the following corollary of the Hopping Tails Theorem.

(5.37) Corollary *Let p be a segment-closed predicate. Then*

$$\begin{array}{rcl} (\uparrow_{\#}/\cdot p\triangleleft)\| \cdot \mathit{st} & = & \oslash\!\not\to([\,],[\,]) \\ \uparrow_{\#}/\cdot p\triangleleft\cdot \mathit{segs} & = & \mathit{exl}\cdot\oslash\!\not\to([\,],[\,]) \\ \uparrow_{\#}/\cdot p\triangleleft\cdot \mathit{tails} & = & \ominus\!\not\to[\,]\ , \end{array}$$

where operator $\oslash$ is defined by

$$(x,y)\oslash a \;=\; (x \uparrow_{\#} (y\ominus a), y\ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x\ominus a \;=\; \begin{cases} x \mathbin{-\!\!\!\prec} a & \text{if } \delta_x\, a \\ (\mathit{tl}\, x)\ominus a & \text{if } \neg\delta_x\, a \;\wedge\; x\neq[\,] \\ [\,] & \text{otherwise}\ . \end{cases}$$

Another corollary of the Hopping Tails Theorem is obtained by considering another property of predicates. If p is robust, see (2.88), then implication

$$p\, x \;\wedge \neg p\,(x \mathbin{-\!\!\!\prec} a) \quad\Rightarrow\quad \neg p\,(y \mathbin{-\!\!\!\prec} a) \tag{5.38}$$

holds for all $y\in \mathit{tails}\, x$ with $y\neq[\,]$. It follows that if p is a prefix-closed and robust predicate, then the second case in the definition of operator $\ominus$ obtained from the Hopping Tails Theorem collapses to $[\,]\ominus a$, since $\mathit{lpt}\ \mathit{tl}\, x = [\,]$. Thus, we obtain the following corollary of the Hopping Tails Theorem.

(5.39) Corollary *Let p be a prefix-closed and robust predicate. Then*

$$\begin{array}{lcl} (\uparrow_{\#}/ \cdot p\triangleleft)\| \cdot st & = & \oslash\!\not\rightarrow([\,],[\,]) \\ \uparrow_{\#}/ \cdot p\triangleleft \cdot segs & = & exl \cdot \oslash\!\not\rightarrow([\,],[\,]) \\ \uparrow_{\#}/ \cdot p\triangleleft \cdot tails & = & \ominus\!\not\rightarrow[\,]\ , \end{array}$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \quad = \quad (x \uparrow_{\#} (y \ominus a), y \ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\!<} a & \textit{if } \delta_x\, a \\ [a] & \textit{if } \neg\delta_x\, a \ \wedge\ p\,[a] \\ [\,] & \textit{otherwise}\ . \end{cases}$$

The Sliding Tails Theorem, the Hopping Tails Theorem and the two corollaries Corollary 5.37 and Corollary 5.39 are tools with which efficient algorithms can be derived for many longest-p segment problems. We discuss some of these.

Example: the longest ascending segment problem

Consider the problem of finding a longest ascending segment, discussed in Section 5.3, again. By applying the Sliding Tails Theorem, the algorithm obtained for *las* can be implemented as an $\Omega(n^2)$ program. Observe that predicate *asc* is prefix-closed, with a derivative $\delta_x\, a = (lt\, x) \leq a$, and robust. The conditions of Corollary 5.39 are satisfied, and applying this corollary gives

$$las \quad = \quad exl \cdot \oslash\!\not\rightarrow([\,],[\,])\ ,$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \quad = \quad (x \uparrow_{\#} (y \ominus a), y \ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \quad = \quad \begin{cases} x \mathbin{-\!\!\!<} a & \text{if } lt\, x \leq a \\ [a] & \text{if } lt\, x > a\ , \end{cases}$$

where we suppose that $lt\,[\,] \leq a$ for all a. If each list is tupled with its length and its last element, and operator $\uparrow_{\#}$ is replaced by $\uparrow_{\pi_1}$, we have obtained an algorithm that can be implemented as a linear-time program.

For the following problems, posed in the small programming exercises from M. Rem [115], [116] in the journal Science of Computer Programming, the theory developed yields efficient algorithms.

Example: the longest almost ascending segment problem

Given a list, find a longest almost ascending segment, where a list is called 'almost ascending' if it can be split into two ascending lists, that is, list x is almost ascending if there exist ascending lists y and z such that $x = y +\!\!\!+ z$. The predicate aa determines whether or not a list is almost ascending. Predicate aa is segment-closed, with a derivative $\delta_x\, a = asc\, x \;\vee\; lt\, x \leq a$, but not robust. Applying Corollary 5.37 yields

$$\uparrow_{\#}/ \cdot aa\triangleleft \cdot segs \;=\; exl \cdot \oslash\not\to([\,],[\,])\ ,$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \;=\; (x \uparrow_{\#} (y \ominus a), y \ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \;=\; \begin{cases} x +\!\!\!< a & \text{if } asc\, x \;\vee\; lt\, x \leq a \\ (tl\, x) \ominus a & \text{if } \neg asc\, x \;\wedge\; a < lt\, x \;\wedge\; x \neq [\,] \\ [\,] & \text{otherwise}\ . \end{cases}$$

Since the guard of the last case in the definition of operator $\ominus$ corresponds to $\neg asc\, x \;\wedge\; a < lt\, x \;\wedge\; x = [\,] \;\equiv\; false$, operator $\ominus$ equals

$$x \ominus a \;=\; \begin{cases} x +\!\!\!< a & \text{if } asc\, x \;\vee\; lt\, x \leq a \\ (tl\, x) \ominus a & \text{otherwise}\ . \end{cases}$$

As it stands, this algorithm cannot be implemented as a linear-time program: the occurrences of $asc\, x$ in the definition of operator $\ominus$ require linear time for their evaluation. This indicates that whether or not the left-hand argument of operator $\ominus$ is ascending is important information. The left-hand argument of operator $\ominus$ corresponds to an almost ascending tail of some argument, in fact, we have, applying Corollary 5.37, that

$$\uparrow_{\#}/ \cdot aa\triangleleft \cdot tails \;=\; \ominus\not\to[\,]\ .$$

Abbreviate function $\uparrow_{\#}/ \cdot aa\triangleleft$ to f. Corollary 5.37 gives a left-reduction for $f\| \cdot st$. We replace function $f \cdot tails$ in this specification by $id \vartriangle asc \cdot f \cdot tails$, and we want to derive a left-reduction for this composition. Since function $f \cdot tails$ is a left-reduction $\ominus\not\to[\,]$, we have, applying Fusion on *snoc-list*

$$id \vartriangle asc \cdot f \cdot tails \;=\; \odot\not\to([\,], true)\ ,$$

provided

$$(id \vartriangle asc)\,(x \ominus a) \;=\; (id \vartriangle asc)\, x \odot a\ , \tag{5.40}$$

for all x in the image of $\ominus\not\to[\,]$. Distinguish the two cases in the definition of operator $\ominus$. From this case distinction, and the desired equality (5.40) above we obtain conditions on operator $\odot$, and these conditions suggest a definition of operator $\odot$. Let (x,y) be $(id \vartriangle asc)\, x$.

- If $asc\, x \;\vee\; lt\, x \leq a$, or equivalently $y \;\vee\; lt\, x \leq a$, then

$$\begin{array}{ll} & (id \vartriangle asc)\,(x \ominus a) \\ = & \quad \text{definition of } \ominus\text{, case assumption} \\ & (id \vartriangle asc)\,(x \mathbin{-\!\!\!<} a) \\ = & \quad \text{definition of } \vartriangle \\ & (x \mathbin{-\!\!\!<} a, asc\,x \;\wedge\; lt\,x \leq a)\ , \end{array}$$

so in case $y \;\vee\; lt\,x \leq a$ operator $\odot$ is defined by

$$(x,y) \odot a \;\;=\;\; (x \mathbin{-\!\!\!<} a, y \;\wedge\; lt\,x \leq a)\ .$$

- To obtain a definition of operator $\odot$ in case $\neg asc\,x \;\wedge\; lt\,x > a$, or equivalently $\neg y \;\wedge\; lt\,x > a$, we prove equation (5.40) by induction on the length of list x. For the base case $x = [\,]$, $asc\,x$ holds and the above case in the definition of operator $\odot$ applies. If $x \neq [\,]$ and $y \;\vee\; lt\,x \leq a$, then again the above case in the definition of operator $\odot$ applies. If $x \neq [\,]$ and $\neg y \;\wedge\; lt\,x > a$, we reason as follows.

$$\begin{array}{ll} & (id \vartriangle asc)\,(x \ominus a) \\ = & \quad \text{case assumption, definition of } \ominus \\ & (id \vartriangle asc)\,(tl\,x \ominus a) \\ = & \quad \text{induction hypothesis} \\ & (id \vartriangle asc)\,tl\,x \odot a \\ = & \quad \text{definition of } \odot \text{ below} \\ & (id \vartriangle asc)\,x \odot a\ , \end{array}$$

provided in case $\neg y \;\wedge\; lt\,x > a$ operator $\odot$ is defined by

$$(x,y) \odot a \;\;=\;\; (tl\,x, hd\,x > hd\,tl\,x) \odot a\ .$$

In the last step of the above calculation we express $asc\,tl\,x$ in terms of $asc\,x$ and x itself. Since $\neg asc\,x$ and $aa\,x$ hold, $asc\,tail\,x \equiv hd\,x > hd\,tl\,x$. This concludes the proof of equation (5.40).

For the longest almost ascending segment problem we have obtained the following solution.

$$\uparrow_{\#}/ \cdot aa\triangleleft \cdot segs \;\;=\;\; exl \cdot \oslash\!\not\to\!([\,],([\,],true))\ ,$$

where operator $\oslash$ is defined by

$$(x,(y,z)) \oslash a \;\;=\;\; (x \uparrow_{\#} exl\,((y,z) \odot a), (y,z) \odot a)\ ,$$

and operator $\odot$ is defined by

$$(x,y) \odot a \;\;=\;\; \begin{cases} (x \mathbin{-\!\!\!<} a, y \;\wedge\; lt\,x \leq a) & \text{if } y \;\vee\; lt\,x \leq a \\ (tl\,x, hd\,x > hd\,tl\,x) \odot a & \text{otherwise}\ . \end{cases}$$

Example: the longest smooth segment problem

Given a list, find a longest smooth segment, where a segment is called smooth if the difference between two succeeding elements is at most 1. The predicate *smooth* determines whether or not a predicate is smooth. Predicate *smooth* is prefix-closed, with a derivative $\delta_x\, a = (-1 \leq (\mathit{lt}\, x - a) \leq 1)$, and robust. Applying Corollary 5.39 yields

$$\uparrow_{\#}/ \cdot \mathit{smooth}\triangleleft \cdot \mathit{segs} \;\; = \;\; \mathit{exl} \cdot \oslash\!\not\to([\,],[\,])\ ,$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \;\; = \;\; (x \uparrow_{\#} (y \ominus a), y \ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \;\; = \;\; \begin{cases} x \mathbin{+\!\!\!<} a & \text{if } -1 \leq \mathit{lt}\, x - a \leq 1 \\ [a] & \text{if } \mathit{lt}\, x - a > 1 \;\vee\; \mathit{lt}\, x - a < -1 \\ [\,] & \text{otherwise}\ . \end{cases}$$

This algorithm can be implemented as a linear-time program.

Example: the longest high segment problem

Given a list, find a longest high segment, where a segment is called high if all of its elements are at least equal to its length. This problem is also known as the problem of the largest square under a histogram. The predicate *high*, which determines whether or not a list is high, is defined by

$$\mathit{high}\, x \;\; = \;\; \downarrow/\, x \geq \#\, x\ .$$

Predicate *high* is segment-closed, if we assume that $\downarrow/\,[\,] \geq 0$, but not robust. A derivative of *high* is defined by $\delta_x\, a = (\downarrow/\, x) \downarrow a \geq \#\, x + 1$. Applying Corollary 5.37 we get

$$\uparrow_{\#}/ \cdot \mathit{high}\triangleleft \cdot \mathit{segs} \;\; = \;\; \mathit{exl} \cdot \oslash\!\not\to([\,],[\,])\ ,$$

where operator $\oslash$ is defined by

$$(x,y) \oslash a \;\; = \;\; (x \uparrow_{\#} (y \ominus a), y \ominus a)\ ,$$

and operator $\ominus$ is defined by

$$x \ominus a \;\; = \;\; \begin{cases} x \mathbin{+\!\!\!<} a & \text{if } (\downarrow/\, x) \downarrow a \geq \#\, x + 1 \\ (\mathit{tl}\, x) \ominus a & \text{if } (\downarrow/\, x) \downarrow a < \#\, x + 1 \;\wedge\; x \neq [\,] \\ [\,] & \text{otherwise}\ . \end{cases}$$

The left-reduction $\oslash\!\not\to([\,],[\,])$ cannot be implemented as a linear-time program. The expensive part of this program corresponds to the case distinction in operator $\ominus$. If the second case in the definition of operator $\ominus$ applies, $x \ominus a$ is evaluated to $(tl\, x) \ominus a$, and in the next evaluation of operator $\ominus$, value $\downarrow/\, tl\, x$ is required. Abbreviate $\uparrow_{\#}/ \cdot high\triangleleft$ to f. A first attempt to obtain an algorithm that can be implemented as a linear-time program is to replace function $f \cdot tails$ in the specification $f\Vert \cdot st$ by function $id \vartriangle \downarrow/ \cdot f \cdot tails$. However, it is in general not possible to determine the value of $\downarrow/\, tl\, x$ in constant time from the values $\downarrow/\, x$ and x. We wish to be able to determine value $\downarrow/\, tl\, x$ in constant time on the average. This is accomplished with a technique called 'applying maximal partitions'.

It is possible to determine value $\downarrow/\, tl\, x$ in constant time if for each element in the list the minimum of the remaining part of the list can be obtained in constant time. Consider the function *sarmp* ('shortest all right-minimal partition') defined by

$$\begin{array}{lcl} sarmp & = & \downarrow_{\#}/ \cdot (all\; rm)\triangleleft \cdot parts \\ rm\; y & = & all\,(\geq lt\; y)\; y\;. \end{array}$$

Function *sarmp* satisfies a number of desirable properties. First of all, since predicate *rm* is suffix-closed and robust, and holds for singletons, we can apply the theory on partition problems, in particular the *parts*-Fusion on *snoc-list* II Theorem, to obtain a definition as a left-reduction of function *sarmp*. This is the first property of function *sarmp*.

$$sarmp \;=\; \oslash\!\not\rightarrow\![\,]\;,$$

where operator $\oslash$ is defined by

$$x \oslash a \;=\; x \odot [a]\;,$$

and operator $\odot$ is defined by

$$\begin{array}{lcl} \nu_{\downarrow\#} \odot x & = & \nu_{\downarrow\#} \\ [\,] \odot y & = & [y] \\ (zs \mathbin{-\!\!<} z) \odot y & = & \begin{cases} (zs \mathbin{-\!\!<} z) \mathbin{-\!\!<} y & \text{if } lt\; z < lt\; y \\ zs \odot (z \mathbin{+\!\!\!+\!\!<} y) & \text{otherwise}\;. \end{cases} \end{array}$$

The second property of function *sarmp* reads as follows.

$$\downarrow/ \;=\; lt \cdot hd \cdot sarmp\;, \tag{5.41}$$

that is, the last element of the first element of the shortest all right-minimal partition of a list equals its minimum. We do not give the proof (by induction) of this equality. The third property of function *sarmp* is an immediate consequence of its first property. Recall the definition of function $gtl : A{\star\star} \to A{\star\star}$, see Section 3.1. By definition of operator $\oslash$

$$gtl\,(x \oslash a) \;=\; (gtl\; x) \oslash a\;. \tag{5.42}$$

The proof by induction of this equality is omitted. It follows that

$$sarmp \cdot tl \;=\; gtl \cdot sarmp\;. \tag{5.43}$$

We prove this equality by induction. For a list of length one we have

$$\begin{array}{ll} & \mathit{gtl}\;\mathit{sarmp}\,[a] \\ = & \quad \text{definition of } \mathit{sarmp} \\ & \mathit{gtl}\,[[a]] \\ = & \quad \text{definition of } \mathit{gtl} \\ & [\,] \\ = & \quad \text{definition of } \mathit{tl} \text{ and } \mathit{sarmp} \\ & \mathit{sarmp}\;\mathit{tl}\,[a]\ . \end{array}$$

For the inductive step, suppose for all lists of length at most n equality (5.43) holds. Assume $\#\, x = n$. Then

$$\begin{array}{ll} & \mathit{gtl}\;\mathit{sarmp}\,(x \mathrel{-\!\!\!\!<} a) \\ = & \quad \mathit{sarmp} = \oslash\!\not\to\![\,] \\ & \mathit{gtl}\,(\mathit{sarmp}\,x \oslash a) \\ = & \quad \text{equation (5.42)} \\ & \mathit{gtl}\;\mathit{sarmp}\,x \oslash a \\ = & \quad \text{induction hypothesis} \\ & \mathit{sarmp}\;\mathit{tl}\,x \oslash a \\ = & \quad \mathit{sarmp} = \oslash\!\not\to\![\,] \\ & \mathit{sarmp}\,(\mathit{tl}\,x \mathrel{-\!\!\!\!<} a) \\ = & \quad \text{definition of } \mathit{tl} \\ & \mathit{sarmp}\;\mathit{tl}\,(x \mathrel{-\!\!\!\!<} a)\ , \end{array}$$

which proves the inductive step. Since function *sarmp* offers efficient means to determine the minimum value of the remaining part of a list, we have now enough machinery to determine a longest high segment of a list in time linear in the length of the list. Replace function $f \cdot \mathit{tails}$ in the specification $f_{\Vert} \cdot \mathit{st}$ by the function $\mathit{id} \vartriangle \mathit{sarmp} \cdot f \cdot \mathit{tails}$. From Corollary 5.37 it follows that $f \cdot \mathit{tails}$ is a left-reduction $\ominus\!\not\to\![\,]$. Apply Fusion on *snoc-list* to obtain

$$\mathit{id} \vartriangle \mathit{sarmp} \cdot f \cdot \mathit{tails} \quad = \quad \otimes\!\not\to\!([\,],[\,])\ ,$$

provided operator $\otimes$ satisfies

$$(\mathit{id} \vartriangle \mathit{sarmp})\,(x \ominus a) \quad = \quad (\mathit{id} \vartriangle \mathit{sarmp})\,x \otimes a\ . \tag{5.44}$$

A definition of operator $\otimes$ is obtained by distinguishing the different cases in the definition of operator $\ominus$. Let (x, y) be $(\mathit{id} \vartriangle \mathit{sarmp})\,x$.

- If $(\downarrow\!/\,x) \downarrow a \geq \#\,x + 1$, or equivalently, by equation (5.41), $\mathit{lt}\,\mathit{hd}\,y \downarrow a \geq \#\,x + 1$, then

$$\begin{aligned}
& (id \vartriangle sarmp)\,(x \ominus a) \\
= \quad & \text{definition of } \ominus \text{, case assumption} \\
& (id \vartriangle sarmp)\,(x \mathbin{-\!\!<} a) \\
= \quad & \text{definition of } \vartriangle,\ sarmp = \oslash\!\not\to\![\,] \\
& (x \mathbin{-\!\!<} a,\, sarmp\, x \oslash a) \\
= \quad & \text{definition of } \otimes \text{ above} \\
& (x, y) \otimes a\ ,
\end{aligned}$$

provided operator $\otimes$ is in case $lt\, hd\, y \downarrow a \geq \#\, x + 1$ defined by

$$(x, y) \otimes a \quad = \quad (x \mathbin{-\!\!<} a,\, y \oslash a)\ .$$

- If $(\downarrow\!/\, x) \downarrow a < \#\, x + 1\ \wedge\ x = [\,]$, or equivalently $a < 1\ \wedge\ x = [\,]$, then

$$\begin{aligned}
& (id \vartriangle sarmp)\,(x \ominus a) \\
= \quad & \text{definition of } \ominus \text{, case assumption} \\
& (id \vartriangle sarmp)\,[\,] \\
= \quad & \text{definition of } \vartriangle,\ sarmp = \oslash\!\not\to\![\,] \\
& ([\,], [\,]) \\
= \quad & \text{definition of } \otimes \text{ below} \\
& (x, y) \otimes a\ ,
\end{aligned}$$

provided operator $\otimes$ is in case $a < 1\ \wedge\ x = [\,]$ defined by

$$(x, y) \otimes a \quad = \quad ([\,], [\,])\ .$$

- The last case in the definition of $\ominus$, case $(\downarrow\!/\, x) \downarrow a < \#\, x{+}1 \wedge x \neq [\,]$, or equivalently $lt\, hd\, y \downarrow a < \#\, x + 1\ \wedge\ x \neq [\,]$, induces the following definition of operator $\otimes$. We prove equation (5.44) by induction on the length of x. In case $x = [\,]$, the previously defined cases of operator $\otimes$ show that equation (5.44) holds. If $x \neq [\,]$, we either have $lt\, hd\, y \downarrow a \geq \#\, x + 1$, in which case the above definition of operator $\otimes$ applies, or $lt\, hd\, y \downarrow a < \#\, x{+}1$, in which case a definition of operator $\otimes$ is obtained as follows.

$$\begin{aligned}
& (id \vartriangle sarmp)\,(x \ominus a) \\
= \quad & \text{definition of } \ominus \text{, case assumption} \\
& (id \vartriangle sarmp)\,(tl\, x \ominus a) \\
= \quad & \text{induction hypothesis} \\
& (id \vartriangle sarmp)\, tl\, x \otimes a \\
= \quad & \text{equation (5.43)}
\end{aligned}$$

$$\begin{array}{ll} & (tl\,x, gtl\,y) \otimes a \\ = & \quad \text{definition of } \otimes \text{ below} \\ & (x, y) \otimes a\ , \end{array}$$

provided operator $\otimes$ is in case $lt\,hd\,y \downarrow a < \#\,x + 1\ \wedge\ x \neq [\,]$, defined by

$$(x, y) \otimes a \quad = \quad (tl\,x, gtl\,y)\ .$$

This proves equation (5.44).

We have found the following algorithm for the longest high segment problem.

$$\uparrow_{\#}/ \cdot high\triangleleft \cdot segs \quad = \quad exl \cdot \oplus\!\not\to\!([\,], ([\,], [\,]))\ ,$$

where operator $\oplus$ is defined by

$$(x, (y, z)) \oplus a \quad = \quad (x \uparrow_{\#} exl\,((y, z) \otimes a), (y, z) \otimes a)\ ,$$

and operator $\otimes$ is defined by

$$(y, z) \otimes a \quad = \quad \begin{cases} (y \mathbin{+\!\!\!<} a, z \oslash a) & \text{if } lt\,hd\,z \downarrow a \geq \#\,y + 1 \\ (tl\,y, gtl\,z) \otimes a & \text{if } lt\,hd\,z \downarrow a < \#\,y + 1\ \wedge\ y \neq [\,] \\ ([\,], [\,]) & \text{otherwise}\ , \end{cases}$$

where operator $\oslash$ is the operator of the left-reduction for function *sarmp*. The technique applied to obtain the minimum of the longest high tail is called 'applying maximal partitions' by Zantema [142]. Using this technique, Zantema derives an algorithm for the problem of finding the longest segment of which the difference between the maximum and the minimum does not exceed a constant C: the longest ribbon problem.

Example: the pattern-matching problem

If predicate p of a longest-p segment problem is prefix-closed but not suffix-closed nor robust Corollaries 5.37 and 5.39 do not apply. An example of such a problem is a variant of the pattern-matching problem. Let P be a pattern (a list). It is required to find occurrences of P in a given text, or, if there are no such occurrences, the longest prefix of P occurring in the given text.

$$lpos \quad = \quad \uparrow_{\#}/ \cdot (\in inits\,P)\triangleleft \cdot segs\ , \tag{5.45}$$

where *lpos* abbreviates 'longest prefix of *segs*'. Predicate $\in inits\,P$ is prefix-closed but it is not robust nor suffix-closed. Since

$$x \mathbin{+\!\!\!<} a \in inits\,P \quad \equiv \quad (x \in inits\,P)\ \wedge\ (a = hd\,(\#\,x \hookrightarrow P))\ ,$$

function $\delta_x\, a = (a = hd\,(\#\, x \hookrightarrow P))$ is a derivative of predicate $\in$ *inits* P. Applying the Hopping tails Theorem gives two operators, $\oslash$ and $\ominus$, where operator $\ominus$ is defined by

$$x \ominus a \;=\; \begin{cases} x \mathbin{-\!\!\!<} a & \text{if } a = hd\,(\#\, x \hookrightarrow P) \\ (lpot\; tl\, x) \ominus a & \text{if } a \neq hd\,(\#\, x \hookrightarrow P) \;\wedge\; x \neq [\,] \\ [\,] & \text{otherwise} \;, \end{cases}$$

where function *lpot* is defined by

$$lpot \;=\; \uparrow_{\#}/ \cdot (\in inits\, P) \triangleleft \cdot\, tails\;.$$

The two problems that have to be addressed after applying the Hopping Tails Theorem are giving an efficient way to determine the value of $a = hd\,(\#\, x \hookrightarrow P)$, and finding the value *lpot tl* x efficiently. These topics are treated in the following section. A rather complicated but efficient method to determine the value of *lpot tl* x is presented in Bird, Gibbons, and Jones [21].

5.5 Pattern matching and other applications

The pattern-matching problem can be viewed as a segment problem. Given a pattern P, a list of characters, it is required to find the position of the first occurrence of P in a text x. An occurrence of a pattern P in a text x is an element of *segs* x which is equal to P. Algorithms for pattern matching are used in many places; an obvious and well-known application can be found in texteditors. Lots of algorithms for pattern matching have been devised, see Weiner [136], Knuth, Morris and Pratt [85], Boyer and Moore [25], etc. This section presents, among others, a derivation of a version of the Knuth, Morris and Pratt pattern-matching algorithm. The algorithm is an adaptation of an algorithm for the problem of finding a longest initial part of a list that also occurs as a non-initial segment, the *iani* (abbreviating 'initial and non-initial') problem. The *iani* problem is discussed in Weiner [136]. Surprisingly, this algorithm can also be used in the derivation of an algorithm for finding the lexicographically least circular substring (rotation), see Booth [24]. Derivations of (parts of) pattern-matching algorithms are popular, examples can be found in Dijkstra [35], Bird, Gibbons and Jones [21], Hoogerwoord [65], Van der Woude [138], Boiten [23], Partsch and Stomp [111], Partsch and Völker [112], etc. The idea of solving the pattern-matching problem by means of the *iani* problem is taken from Van der Woude [138], who derives an imperative version of the pattern-matching algorithm using the Dijkstra-style of program derivation.

This section is organised as follows. First we give a derivation of an efficient algorithm for the *iani* problem. Then we explain the connection of this problem with the pattern-matching problem, and we give a version of the Knuth, Morris and Pratt algorithm for pattern matching. We conclude this section with a derivation of an algorithm for finding a lexicographically least rotation of a string by means of the algorithm for the *iani* problem.

5.5.1 The *iani* problem

The *iani* problem is specified as follows. It is required to find a longest non-initial segment that also occurs as an initial segment.

$$iani\,x \;=\; \uparrow_{\#}/\,(\in inits\,x)\triangleleft segs\;tl\,x\ .$$

We want to find an algorithm that can be implemented as an efficient program for this problem. The specification is not a longest-p segment problem as specified in (5.24) because of the simultaneous occurrence of x in the predicate $\in$ *inits* x and in the argument *tl* x of function *segs*. For this reason it is not possible to apply the theory developed in the previous sections. Instead of starting from scratch for this problem, we generalise it in such a way that the theory developed can be applied. Consider the following separation of the argument of function *iani*.

$$iani\,x \;=\; lpos_x\;tl\,x\ , \tag{5.46}$$

where function $lpos_v$ ('longest v-prefix of *segs*') is defined by

$$lpos_v \;=\; \uparrow_{\#}/\cdot(\in inits\,v)\triangleleft\cdot\,segs\ .$$

Function $lpos_v$ is a longest-p segment problem. We return to the *iani* problem after the following discussion on problem $lpos_v$. As noted in the previous subsection, predicate $\in$ *inits* v is prefix-closed, but not robust nor suffix-closed, so neither of the corollaries of the Hopping Tails Theorem is applicable. Apply the Hopping Tails Theorem to obtain

$$lpost_v \;=\; lpos_v \vartriangle lpot_v \;=\; \oslash_v\!\not\rightarrow\!([\,],[\,])\ , \tag{5.47}$$

where function $lpot_v$ is defined by

$$lpot_v \;=\; \uparrow_{\#}/\cdot(\in inits\,v)\triangleleft\cdot\,tails\ ,$$

operator $\oslash_v$ is defined by

$$(x,y)\oslash_v a \;=\; (x\uparrow_{\#}(y\ominus_v a),\,y\ominus_v a)\ , \tag{5.48}$$

and operator $\ominus_v$ is defined by

$$y\ominus_v a \;=\; \begin{cases} y \mathbin{+\!\!\!<} a & \text{if } a = hd\,(\#\,y \hookrightarrow v)\\ (lpot_v\;tl\,y)\ominus_v a & \text{if } a \neq hd\,(\#\,y\hookrightarrow v)\;\wedge\; y\neq[\,]\\ [\,] & \text{otherwise}\ . \end{cases} \tag{5.49}$$

Note that for all q, r, and p we have

$$r\in inits\,p \;\wedge\; \#\,q\leq\#\,r \;\Rightarrow\; lpost_r\,q = lpost_p\,q\ . \tag{5.50}$$

Deriving a left-reduction for $(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id}$

We return to the *iani* problem. Using the left-reduction for function lpost_v given in equation (5.47) we aim at constructing a left-reduction for the tuple of functions $\mathit{iani} \vartriangle \mathit{ianit}$, where function *ianit* is defined by

$$\mathit{ianit}\, x \quad = \quad \uparrow_{\#}/ \, (\in \mathit{inits}\, x) \triangleleft \mathit{tails}\; \mathit{tl}\, x \ ,$$

Note that function *ianit* satisfies

$$\mathit{ianit}\, x \quad = \quad \mathit{lpot}_x\; \mathit{tl}\, x \ . \tag{5.51}$$

We want to find a left-reduction that can be implemented as an efficient program for the tuple of functions $\mathit{iani} \vartriangle \mathit{ianit}$. We define $\mathit{iani}\,[\,]$ and $\mathit{ianit}\,[\,]$ both to be equal to ω, the unit of $\uparrow_{\#}$. This is possible because functions *segs* and *tails* are not defined on $\mathit{tl}\,[\,]$. If $x = [\,]$, then $(\mathit{iani} \vartriangle \mathit{ianit})\,(x \mathbin{+\!\!\!<} a) = ([\,],[\,])$, and for $x \neq [\,]$ we have

$$\begin{array}{ll}
 & (\mathit{iani} \vartriangle \mathit{ianit})\,(x \mathbin{+\!\!\!<} a) \\
= & \qquad \text{equations (5.46) and (5.51)} \\
 & \mathit{lpost}_{x \mathbin{+\!\!\!<} a}\; \mathit{tl}\,(x \mathbin{+\!\!\!<} a) \\
= & \qquad \mathit{tl}\,(x \mathbin{+\!\!\!<} a) <_{\#} x \mathbin{+\!\!\!<} a, \text{ equation (5.50)} \\
 & \mathit{lpost}_x\; \mathit{tl}\,(x \mathbin{+\!\!\!<} a) \\
= & \qquad x \neq [\,] \\
 & \mathit{lpost}_x\,(\mathit{tl}\, x \mathbin{+\!\!\!<} a) \\
= & \qquad \mathit{lpost}_x \text{ is a left-reduction, see equation (5.47)} \\
 & \mathit{lpost}_x\; \mathit{tl}\, x \oslash_x a \\
= & \qquad \text{equations (5.46) and (5.51)} \\
 & (\mathit{iani} \vartriangle \mathit{ianit})\, x \oslash_x a \ .
\end{array}$$

Since the argument x is present in operator $\oslash_x$, this is not a derivation of a left-reduction for the tuple of functions $\mathit{iani} \vartriangle \mathit{ianit}$. However, it is a derivation of a paramorphism for $\mathit{iani} \vartriangle \mathit{ianit}$, so tuple with function *id*. We obtain

$$(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id} \quad = \quad \otimes \not\rightarrow ((\omega,\omega),[\,]) \ , \tag{5.52}$$

where operator $\otimes$ is defined by

$$((x,y),v) \otimes a \quad = \quad \left\{ \begin{array}{ll} ((x,y) \oslash_v a, v \mathbin{+\!\!\!<} a) & \text{if } x,y \neq \omega \\ (([\,],[\,]),[a]) & \text{if } x,y = \omega \ , \end{array} \right.$$

where operator $\oslash$ is defined in equation (5.48). The tuple of functions $\mathit{ianit} \vartriangle \mathit{id}$ is a left-reduction, according to the above derivation, and the Hopping Tails Theorem.

$$\mathit{ianit} \vartriangle \mathit{id} \quad = \quad \ominus \not\rightarrow (\omega,[\,]) \ , \tag{5.53}$$

where operator $⊜$ is defined by

$$(y, v) ⊜ a \quad = \quad \begin{cases} (y \ominus_v a, v \mathbin{-\!\!\!\prec} a) & \text{if } y \neq \omega \\ ([\,], [a]) & \text{if } y = \omega\ , \end{cases}$$

where operator $\ominus$ is defined in equation (5.49).

Suppose left-reduction $\otimes\!\not\to\!((\omega, \omega), [\,])$ is applied to some argument z. Whenever expression $y \ominus_v a$ is evaluated in the computation of $(\otimes\!\not\to\!((\omega, \omega), [\,]))\ z$, the components y and v satisfy $v \in \mathit{inits}\ z$ and $y \in \mathit{inits}\ v$. Hence

$$\begin{array}{ll} & \mathit{lpot}_v\ \mathit{tl}\ y \\ = & \quad \mathit{tl}\ y <_{\#} y, \text{ equation (5.50)} \\ & \mathit{lpot}_y\ \mathit{tl}\ y \\ = & \quad \text{equation (5.51)} \\ & \mathit{ianit}\ y\ , \end{array}$$

and we may replace expression $\mathit{lpot}_v\ \mathit{tl}\ y$ by $\mathit{ianit}\ y$ in definition (5.49) of operator $\ominus$. It remains to construct efficient means with which the values of $a = \mathit{hd}\ (\#\ y \hookrightarrow v)$ and $\mathit{ianit}\ y$ can be determined.

Improving the efficiency of the left-reduction for $(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id}$

The evaluation of operator $\otimes$ of the left-reduction equal to $(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id}$ given above is expensive: operator $\ominus$, by means of which operator $\otimes$ is defined, might require linear time for its evaluation. To improve the efficiency of the left-reduction for $(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id}$ we tuple function ianit with some auxiliary functions.

Consider the definition of operator $\ominus$ given in equation (5.49). Concerning the value of $a = \mathit{hd}\ (\#\ y \hookrightarrow v)$ we have the following. While evaluating $y \ominus_v a$, we would like to have the first element of the list $\#\ y \hookrightarrow v$ available. Since this element should remain available in the recursion, we want to have the pair of lists $(\#\ y \rightharpoonup v, \#\ y \hookrightarrow v) = \#\ y \dashv ([\,], v)$ available, where operator $\dashv$ is defined in equation (3.17). Tuple function ianit with function $f \cdot \mathit{ianit} \vartriangle \mathit{id}$, where function f is defined by

$$f\ (y, v) \quad = \quad \#\ y \dashv ([\,], v)\ .$$

It follows immediately that value $\#\ y$ should also be available, and hence that function ianit should be tupled with function $\# \cdot \mathit{ianit}$ too.

The recursion in the second alternative of definition (5.49) of operator $\ominus$ shows the need for values $\mathit{ianit}\ z$ and $\mathit{hd}\ (\#\ \mathit{ianit}\ z \hookrightarrow z)$ for all $z \in \mathit{inits}\ v$. Since

$$\begin{array}{l} \quad \mathit{ianit}\ z \\ = \\ \quad \mathit{exl}\,(\#\,\mathit{ianit}\ z \dashv ([\,], z)) \\ = \\ \quad \mathit{exl}\,(\#\,\mathit{ianit}\ z \dashv ([\,], v))\ , \end{array}$$

and

$$\begin{array}{l} \quad \mathit{hd}\,(\#\,\mathit{ianit}\ z \hookrightarrow z) \\ = \\ \quad \mathit{hd}\ \mathit{exr}\,(\#\,\mathit{ianit}\ z \dashv ([\,], z)) \\ = \\ \quad \mathit{hd}\ \mathit{exr}\,(\#\,\mathit{ianit}\ z \dashv ([\,], v))\ , \end{array}$$

it suffices to have value $\#\,\mathit{ianit}\,z$ available for all $z \in \mathit{inits}\,v$. This suggests to tuple function *ianit* with (a variant of) function $(\# \cdot \mathit{ianit}){*} \cdot \mathit{inits}$. If we tuple function *ianit* with function $g \cdot \mathit{ianit} \vartriangle \mathit{id}$, where function g is defined by

$$g\,(y, z) \quad = \quad \#\,y \dashv ([\,], (\# \cdot \mathit{ianit})\star\,\mathit{inits}\,z)\ ,$$

where function *inits* returns a snoc-list of initial segments in ascending order of length instead of a set of initial segments, that is,

$$\begin{array}{lcl} \mathit{inits} & : & A\star \to A\star\star \\ \mathit{inits}\,[\,] & = & [[\,]] \\ \mathit{inits}\,(x \dashv\!\!< a) & = & \mathit{inits}\,x \dashv\!\!< (x \dashv\!\!< a)\ , \end{array}$$

then the occurrence of $\mathit{ianit}\,y$ in the definition of operator $\ominus$ may be replaced by expression $\mathit{hd}\ \mathit{exr}\ g\,(y, z) \rightharpoonup y$, or equivalently, by $(\#y - \mathit{hd}\ \mathit{exr}\ g\,(y, z)) \hookleftarrow y$.

An extended specification

We collect the wishes for obtaining an algorithm that can be implemented as an efficient program for $(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id}$ listed above. Replace the tuple of functions $\mathit{ianit} \vartriangle \mathit{id}$ in the tuple of functions $(\mathit{iani} \vartriangle \mathit{ianit}) \vartriangle \mathit{id}$ by the following composition of functions.

$$h \cdot \mathit{ianit} \vartriangle \mathit{id}\ , \tag{5.54}$$

where function h is defined by

$$h \quad = \quad \mathit{exl} \vartriangle (\# \cdot \mathit{exl}) \vartriangle f \vartriangle g\ .$$

To obtain an efficient program for $h \cdot \mathit{ianit} \vartriangle \mathit{id}$ we want to find a left-reduction equal to $h \cdot \mathit{ianit} \vartriangle \mathit{id}$. Since the tuple of functions $\mathit{ianit} \vartriangle \mathit{id}$ is a left-reduction $\ominus\!\not\to\!(\omega, [\,])$, see (5.53), we can apply Fusion on *snoc-list* to specification (5.54) to obtain a left-reduction for the composition of functions. We have

$$h \cdot \mathit{ianit} \vartriangle \mathit{id} \quad = \quad \odot\!\not\rightarrow\!(h\,(\omega, [\,])) \;, \tag{5.55}$$

provided operator $\odot$ satisfies

$$h\,((y,z) \circledequal a) \quad = \quad h\,(y,z) \odot a \;, \tag{5.56}$$

for (y,z) in the image of $\mathit{ianit} \vartriangle \mathit{id}$. The seed $h\,(\omega, [\,])$ of the left-reduction satisfies the following equality.

$$h\,(\omega, [\,]) \quad = \quad (\omega, 0, ([\,],[\,]), ([\,],[0])) \;,$$

if $\#\,\omega$ is defined to be 0. To obtain a definition of operator $\odot$ satisfying equation (5.56), unfold the definition of $\circledequal$, and distinguish the different cases in the definition of operator $\ominus$. Let $(p,q,r,s) = h\,(y,z)$.

- In case $y = \omega$ and $z = [\,]$, or equivalently $p = \omega$, we have

$$\begin{array}{ll} & h\,((y,z) \circledequal a) \\ = & \quad \text{case assumption, definition of } \circledequal \\ & h\,([\,],[a]) \\ = & \quad \text{definition of } \odot \\ & (p,q,r,s) \odot a \;, \end{array}$$

  provided operator $\odot$ is in case $p = \omega$ defined by

$$(p,q,r,s) \odot a \quad = \quad ([\,], 0, ([\,],[a]), ([\,],[0,0])) \;.$$

- In case $a = \mathit{hd}\,(\#\,y \hookrightarrow z)$, or equivalently $a = \mathit{hd}\;\mathit{exr}\;r$, we have

$$\begin{array}{ll} & h\,((y,z) \circledequal a) \\ = & \quad \text{case assumption, definition of } \circledequal \\ & h\,(y \ominus_z a, z \mathbin{-\!\!<} a) \\ = & \quad \text{case assumption, definition of } \ominus \\ & h\,(y \mathbin{-\!\!<} a, z \mathbin{-\!\!<} a) \\ = & \quad \text{definition of } \odot \\ & (p,q,r,s) \odot a \;, \end{array}$$

  provided operator $\odot$ is in case $a = \mathit{hd}\;\mathit{exr}\;r$ defined by

$$(p,q,r,s) \odot a \quad = \quad (p \mathbin{-\!\!<} a, q{+}1, (1 \dashv r) \mathbin{-\!\!<}_2 a, (1 \dashv s) \mathbin{-\!\!<}_2 q{+}1) \;, \tag{5.57}$$

  where operator $\mathbin{-\!\!<}_2$ is defined on pairs instead of triples. The third and fourth component of $(p,q,r,s) \odot a$ are obtained as follows. We have

$$\begin{array}{cl}
 & f\,(y \mathbin{-\!\!<} a, z \mathbin{-\!\!<} a) \\
= & \quad \text{definition of } f \\
 & \#\,(y \mathbin{-\!\!<} a) \dashv ([\,], z \mathbin{-\!\!<} a) \\
= & \quad \text{definition of } \# \text{ and } \mathbin{-\!\!<}_2 \\
 & \#\,y{+}1 \dashv (([\,], z) \mathbin{-\!\!<}_2 a) \\
= & \quad \text{property (3.19)}, \#\,z \geq \#\,y + 1 \\
 & (1 \dashv (\#\,y \dashv ([\,], z))) \mathbin{-\!\!<}_2 a \\
= & \quad \text{definition of } r \\
 & (1 \dashv r) \mathbin{-\!\!<}_2 a \ ,
\end{array}$$

and similarly

$$g\,(y \mathbin{-\!\!<} a, z \mathbin{-\!\!<} a) \;=\; (1 \dashv s) \mathbin{-\!\!<}_2 q{+}1 \ .$$

- If $a \neq \mathit{hd}\ \mathit{exr}\ r$ and $p = [\,]$, we obtain

$$\begin{array}{cl}
 & h\,((y, z) \circeq a) \\
= & \quad \text{case assumption, definition of } \circeq \\
 & h\,(y \ominus_z a, z \mathbin{-\!\!<} a) \\
= & \quad \text{case assumption, definition of } \ominus \\
 & h\,([\,], z \mathbin{-\!\!<} a) \\
= & \quad \text{definition of } \odot \\
 & (p, q, r, s) \odot a \ ,
\end{array}$$

provided operator $\odot$ is in case $a \neq \mathit{hd}\ \mathit{exr}\ r$ and $p = [\,]$ defined by

$$(p, q, r, s) \odot a \;=\; ([\,], 0, r \mathbin{-\!\!<}_2 a, s \mathbin{-\!\!<}_2 0) \ . \tag{5.58}$$

- The remaining case in the definition of operator $\ominus$, $a \neq \mathit{hd}\ \mathit{exr}\ r$ and $p \neq [\,]$, is dealt with as follows. We prove equation (5.56) by induction on the length of list y. The base case $h\,(([\,], z) \circeq a) = h\,([\,], z) \odot a$ follows immediately from the cases (5.57) and (5.58) in the definition of operator $\odot$. If $y \neq [\,]$ and $a = \mathit{hd}\ \mathit{exr}\ r$, then equation (5.56) follows from definition (5.57) of operator $\odot$. If $y \neq [\,]$ and $a \neq \mathit{hd}\ \mathit{exr}\ r$ we reason as follows. From $y \neq [\,]$ it follows that $\mathit{ianit}\ y <_{\#} y$. We have

$$\begin{array}{cl}
 & h\,((y, z) \circeq a) \\
= & \quad \text{case assumption, definition of } \circeq \\
 & h\,(y \ominus_z a, z \mathbin{-\!\!<} a) \\
= & \quad \text{case assumption, definition of } \ominus
\end{array}$$

$$\begin{array}{cl}
 & h\,(\mathit{ianit}\,y \ominus_z a, z \nprec a) \\
= & \quad \text{definition of } \circleddash \\
 & h\,((\mathit{ianit}\,y, z) \circleddash a) \\
= & \quad \text{induction hypothesis, } \mathit{ianit}\,y <_{\#} y \\
 & h\,(\mathit{ianit}\,y, z) \odot a \\
= & \quad \text{definition of } (p,q,r,s),\ t = q - \mathit{hd}\ \mathit{exr}\ s \\
 & (t \hookleftarrow p, q-t, t \vdash r, t \vdash s) \odot a \\
= & \quad \text{definition of } \odot \\
 & (p,q,r,s) \odot a\ ,
\end{array}$$

provided operator $\odot$ is in case $a \neq \mathit{hd}\ \mathit{exr}\ r$ and $p \neq [\,]$ defined by

$$\begin{array}{rcl}
(p,q,r,s) \odot a & = & (t \hookleftarrow p, q-t, t \vdash r, t \vdash s) \odot a \\
 & & \textbf{where } t = q - \mathit{hd}\ \mathit{exr}\ s\ .
\end{array} \tag{5.59}$$

This concludes the proof of equality (5.56).

All cases in the definition of operator $\ominus$ have been considered, and it follows that if operator $\odot$ is defined by

$$\begin{array}{rcl}
(p,q,r,s) \odot a & = & \left\{ \begin{array}{ll}
([\,], 0, ([\,],[a]), ([\,],[0,0])) & \text{if } p = \omega \\
(p \nprec a, q{+}1, (1{\dashv}r){\nprec_2}a, (1{\dashv}s){\nprec_2}q{+}1) & \text{if } a = u \\
([\,], 0, r \nprec_2 a, s \nprec_2 0) & \text{if } a \neq u\ \wedge\ p = [\,] \\
(t \hookleftarrow p, q-t, t \vdash r, t \vdash s) \odot a & \text{otherwise}
\end{array} \right. \\
 & & \textbf{where } t = q - \mathit{hd}\ \mathit{exr}\ s \\
 & & \qquad\quad u = \mathit{hd}\ \mathit{exr}\ r\ ,
\end{array}$$

then equation (5.55) holds. For the tuple of functions $\mathit{iani} \vartriangle (h \cdot \mathit{ianit} \vartriangle \mathit{id})$ we have found the following left-reduction.

$$\mathit{iani} \vartriangle (h \cdot \mathit{ianit} \vartriangle \mathit{id}) \ = \ \otimes\!\not\to\! e\ ,$$

where $e = (\omega, (\omega, [\,], 0, ([\,],[\,]), ([\,],[0])))$ and operator $\otimes$ is defined by

$$(x,y) \otimes a \ = \ (x \uparrow_{\#} \pi_0\,(y \odot a), y \odot a)\ ,$$

where operator $\odot$ is defined above.

Discussion

The algorithm for $\mathit{iani} \vartriangle (h \cdot \mathit{ianit} \vartriangle \mathit{id})$ can be implemented as a linear-time program. Obviously, the number of evaluations of $\otimes$ is linear in the length of the argument, and each evaluation of $\otimes$ requires constant time on the average if operator $\odot$ requires constant time on the average. It remains to show that the total time required for the evaluations of operator $\odot$ is linear in the length of the argument. Note that the first element of the left-hand argument of operator $\odot$ is only expanded in the non-recursive alternatives of operator $\odot$, and each time the recursive alternative of operator $\odot$ is chosen, the first element of the left-hand argument of operator $\odot$ is shrunk, $t \hookleftarrow p$, and the time required for this particular evaluation of $\odot$ is linear in the expansion or shrinking. Let y_i be the length of the first component of the left-hand argument of operator $\odot$ at position i in the argument list. The total time required for the evaluations is

$$\sum_{i=1}^{n} |y_i - y_{i-1}| \quad \leq \quad 2n \; .$$

Compared with the algorithm for the *iani* problem given by Weiner [136], which for a large part consists of a description of a construction of a suffix tree, see also McCreight [93], the algorithm described above is much simpler. However, a suffix tree is very useful for solving some other problems.

5.5.2 Pattern matching

A specification of the pattern-matching problem has been given in Section 5.4 (5.45). Given a pattern P it is required to find occurrences of P in a given text, or, if there are no such occurrences, the longest prefix of P occurring in the given text.

$$\mathit{lpos} \quad = \quad \uparrow_{\#}/ \cdot (\in \mathit{inits}\, P)\triangleleft \cdot \mathit{segs} \; .$$

We want to construct an efficient algorithm for pattern-matching. Since $\in$ *inits* P is prefix-closed with a derivative $\delta_x\, a = (a = \mathit{hd}\,(\#\, x \hookrightarrow P))$ the Hopping Tails Theorem gives

$$\uparrow_{\#}/ \cdot (\in \mathit{inits}\, P)\triangleleft \cdot \mathit{segs} \quad = \quad \mathit{exl} \cdot \oslash\!\not\rightarrow([\,],[\,]) \; ,$$

where operator $\oslash$ is defined by

$$(x, y) \oslash a \quad = \quad (x \uparrow_{\#} (y \ominus a), y \ominus a) \; ,$$

where operator $\ominus$ is defined by

$$y \ominus a \quad = \quad \begin{cases} y \mathbin{+\!\!\!<} a & \text{if } a = \mathit{hd}\,(\#\, y \hookrightarrow P) \\ \mathit{lpot}\; \mathit{tl}\, y \ominus a & \text{if } a \neq \mathit{hd}\,(\#\, y \hookrightarrow P) \;\wedge\; x \neq [\,] \\ [\,] & \text{otherwise} \; . \end{cases}$$

Function *lpot* has been defined in equation (5.48).

Improving the efficiency of the left-reduction for $\mathit{lpos} \vartriangle \mathit{lpot}$

The straightforward implementation of the algorithm for pattern matching obtained by an application of the Hopping Tails Theorem is not a linear-time program. The two remaining problems are: the computation of value $a = \mathit{hd}\,(\#\, y \hookrightarrow P)$ and the computation of $\mathit{lpot}\ \mathit{tl}\ y$. To determine the value of $a = \mathit{hd}\,(\#\, y \hookrightarrow P)$ in constant time apply the same construction as applied in the derivation of an algorithm for the *iani* problem: tuple with the the computation of the pair $(\#\, y \rightharpoonup P, \#\, y \hookrightarrow P) = \#\, y \dashv ([\,], P)$. For the computation of $\mathit{lpot}\ \mathit{tl}\ y$ we cannot apply the same construction as applied in the derivation of an algorithm for the *iani* problem. There we used the fact that if $y = \mathit{ianit}\ v$, then y occurs in $\mathit{inits}\ v$. The corresponding implication if $y = \mathit{lpot}_P\ v$, then y occurs in $\mathit{inits}\ v$ does not hold. However, the implication

$$y = \mathit{lpot}_P\ v \quad \Rightarrow \quad y \in \mathit{inits}\ P \ ,$$

does hold. In the previous subsection we have shown that $\mathit{lpot}_P\ \mathit{tl}\ y = \mathit{ianit}\ y$ for $y \in \mathit{inits}\ P$. If the values $\mathit{ianit}\star\ \mathit{inits}\ P$ are available, we can find the value of $\mathit{ianit}\ y$ in a fashion similar to the method sketched for the *iani* problem.

Values $\mathit{ianit}\star\ \mathit{inits}\ P$ are easily obtained:

$$\mathit{ianit}\star\ \mathit{inits}\ P \quad = \quad (+\!\!\!+ \cdot \pi_3 \cdot \odot\!\not\rightarrow\!(h\,(\omega, [\,])))\ P \ ,$$

where $\odot\!\not\rightarrow\!(h\,(\omega, [\,]))$ is the left-reduction for $h \cdot \mathit{ianit} \vartriangle \mathit{id}$ given in equation (5.55). This application corresponds to the precomputation phase given by Knuth et. al. [85]. Define functions f', g', and h' by

$$\begin{aligned} f'\,y &\;=\; \#\, y \dashv ([\,], P) \\ g'\,y &\;=\; \#\, y \dashv ([\,], \mathit{ianit}\star\ \mathit{inits}\ P) \\ h' &\;=\; \mathit{id} \vartriangle \# \vartriangle f' \vartriangle g' \ , \end{aligned}$$

and replace *lpot* in the specification $\mathit{lpos} \vartriangle \mathit{lpot}$ by $h' \cdot \mathit{lpot}$. We can exactly replay the derivation of a left-reduction for $h \cdot \mathit{ianit} \vartriangle \mathit{id}$ for $h' \cdot \mathit{lpot}$ with the following result.

$$\mathit{lpos} \vartriangle (h' \cdot \mathit{lpot}) \quad = \quad \oslash\!\not\rightarrow\! e \ ,$$

where $e = ([\,], ([\,], 0, ([\,], P), ([\,], \mathit{ianit}\star\ \mathit{inits}\ P)))$, and operator $\oslash$ is defined by

$$(x, y) \oslash a \quad = \quad (x \uparrow_{\#} \pi_0\,(y \odot a), y \odot a) \ ,$$

where operator $\odot$ is defined by

$$(p, q, r, s) \odot a \quad = \quad \begin{cases} ([\,], 0, ([\,], P), ([\,], \mathit{ianit}\star\ \mathit{inits}\ P)) & \text{if } p = \omega \\ (p \mathbin{-\!\!\!\prec} a, q{+}1, 1 \dashv r, 1 \dashv s) & \text{if } a = u \\ ([\,], 0, r, s) & \text{if } a \neq u \ \wedge \ p = [\,] \\ (t \hookleftarrow p, q{-}t, t \vdash r, t \vdash s) \odot a & \text{otherwise} \end{cases}$$

$$\begin{aligned} \textbf{where}\ t &= q - \mathit{hd}\ \mathit{exr}\ s \\ u &= \mathit{hd}\ \mathit{exr}\ r \ , \end{aligned}$$

Discussion

The argument that was used to show that the program for the *iani* problem is linear time can be repeated for the program for pattern matching, the straightforward implementation of left-reduction $\oslash \not\rightarrow e$, to show that it is linear time indeed. For the precomputation phase time $O(p)$, where $p = \#\, P$, is required.

The tabulation used in the algorithm sketched above is simpler than the tabulation used in the algorithm for pattern matching given by Bird et al. [21]. Bird et al. use lazy evaluation to build a table in which the values of $\mathit{ianit}{\star}\ \mathit{inits}\ P$ are stored. Their construction is a bit opaque.

We briefly discuss the problem of pattern matching with a set of patterns. Given a set SP of patterns, this problem is specified by

$$\uparrow_{\#}/ \cdot (\in \cup/\ \mathit{inits}{*}\ SP) \triangleleft \cdot \mathit{segs}\ .$$

Note that predicate $\in \cup/\ \mathit{inits}{*}\ SP$ is prefix-closed, and

$$\delta_x\, a \quad = \quad \exists P \in SP : a = \mathit{hd}\,(\#\, x \hookrightarrow P)$$

is a derivative. An algorithm for this problem is given by Aho and Corasick, see [2]. This algorithm is very similar to the pattern matching algorithm with a single pattern from Knuth et. al. [85], a version of which is derived above. Only the tabulation part, which is used to find value $\mathit{ianit}\, y$ quickly, is different. Maintaining a list of previously computed values of function *ianit* is not sufficient anymore, a (prefix) tree to store the previously computed values of function *ianit* is needed here. We omit a derivation and description of the algorithm.

5.5.3 The lexicographically least rotation

Consider the problem of finding a lexicographically least rotation of a list. This problem, discussed by Booth [24], arises in graph isomorphism algorithms where a canonical form is sought for certain cycles within a graph. Furthermore, it serves as another example of an application of the algorithm for the *iani* problem. The lexicographically least rotation problem is similar to a problem discussed by Shiloach [121] in which two circular substrings are compared for equivalence, although neither problem reduces immediately to the other.

Function $\mathit{rots} : A{\star} \rightarrow A{\star}\wr$, which returns the rotations of a list, is defined in terms of function *segs* by

$$\mathit{rots}\, x \quad = \quad (=_{\#}\, x) \triangleleft \mathit{segs}\,(x \mathbin{+\!\!\!+\!\!<} x)\ . \tag{5.60}$$

A lexicographically least rotation of a list is returned by function $\mathit{llr} : A{\star} \rightarrow A{\star}$, which is specified by

$$\mathit{llr} \quad = \quad \downarrow/ \cdot \mathit{rots}\ , \tag{5.61}$$

where $\downarrow$ denotes the minimum operation with respect to the homogeneous lexicographical ordering, that is, the lexicographical ordering defined on equal-length lists. The straightforward implementation of the specification is an inefficient program, and our aim is to construct an algorithm that can be implemented as a linear-time program.

Expressing *llr* as a tail problem

To obtain an algorithm that can be implemented as an efficient program for function *llr*, we first express *llr* as a problem on *tails*, and then apply results from the previous sections to obtain an efficient algorithm. Calculate as follows.

$$\begin{array}{ll}
 & \downarrow/\ \mathit{rots}\ x \\
= & \quad \text{definition of } \mathit{rots} \\
 & \downarrow/\ (=_{\#}\ x)\triangleleft\, \mathit{segs}\ (x +\!\!\!+\!\!\!< x) \\
= & \quad \text{characterisation (5.9) for } \mathit{segs} \\
 & \downarrow/\ (=_{\#}\ x)\triangleleft \cup/\ \mathit{inits}{*}\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x) \\
= & \quad \text{Fusion on } \mathit{set}\text{, map-distributivity} \\
 & \downarrow/ \cup/\ ((=_{\#}\ x)\triangleleft \cdot \mathit{inits}){*}\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x)\ .
\end{array}$$

Apply now the following equality, the proof of which is trivial and omitted.

$$(=_{\#}\ x)\triangleleft \cdot \mathit{inits} \quad = \quad (\#\, x \rightharpoonup){*} \cdot (\geq_{\#}\ x)\S\ , \tag{5.62}$$

where $p\S\ a = \{a\}$ if $p\ a$ holds, and $p\S\ a = \{\,\}$ otherwise, see (2.66). This equality gives

$$\begin{array}{ll}
 & \downarrow/ \cup/\ ((=_{\#}\ x)\triangleleft \cdot \mathit{inits}){*}\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x)\ . \\
= & \quad \text{equation (5.62)} \\
 & \downarrow/ \cup/\ ((\#\, x \rightharpoonup){*} \cdot (\geq_{\#}\ x)\S){*}\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x) \\
= & \quad \text{map-distributivity, Fusion on } \mathit{set} \\
 & \downarrow/\ (\#\, x \rightharpoonup){*} \cup/\ (\geq_{\#}\ x)\S{*}\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x) \\
= & \quad \text{definition of filter on the data type } \mathit{set}\ (2.67) \\
 & \downarrow/\ (\#\, x \rightharpoonup){*}\ (\geq_{\#}\ x)\triangleleft\, \mathit{tails}\ (x +\!\!\!+\!\!\!< x) \\
= & \quad \text{wish} \\
 & \#\, x \rightharpoonup\ \downarrow/\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x)\ .
\end{array}$$

The wish results from the hope that the composition of $\downarrow/$ with the left-reduction *tails* can be written as an algorithm that can be implemented as an efficient program. To accomplish the wish selector $\downarrow$ has to be extended to denote the minimum operation with respect to a heterogeneous lexicographical ordering $\leq_R$ (since $\downarrow/\ \mathit{tails}\ (x +\!\!\!+\!\!\!< x)$ is not well defined) in such a way that

- section $(\#\, x \rightharpoonup)$ is $(\downarrow, \downarrow)$-fusable after $\downarrow/ \cdot (\geq_{\#}\ x)\triangleleft \cdot (\in \mathit{tails}\, v)\triangleleft$;
- $\downarrow/\,(\geq_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x) = \downarrow/\ \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x)$.

The usual extension of $\downarrow$ to the lexicographical ordering $\leq_L$, see (3.46), as used in dictionaries, does not satisfy the requirements. We shall try to deduce an extension of operator $\downarrow$ from the two requirements.

First, the requirement that $(\#\, x \rightharpoonup)$ be $(\downarrow, \downarrow)$-fusable after $\downarrow/\cdot(\geq_{\#}\ x)\triangleleft\cdot(\in \mathit{tails}\, v)\triangleleft$ amounts to

$$\begin{array}{lcl} \#\, x \rightharpoonup \nu_{\downarrow} & = & \nu_{\downarrow} \\ \#\, x \rightharpoonup (y \downarrow z) & = & (\#\, x \rightharpoonup y) \downarrow (\#\, x \rightharpoonup z)\ , \end{array}$$

where y and z are elements in the image of $\downarrow/ \cdot (\geq_{\#}\ x)\triangleleft \cdot (\in \mathit{tails}\, v)\triangleleft$ for some list v. These requirements are satisfied for any lexicographical ordering $\leq_R$ that satisfies for nonempty lists y and z

$$y \leq_R z \quad \Leftarrow \quad y <_{hd} z\ \vee\ (y =_{hd} z\ \wedge\ \mathit{tl}\, y \leq_R \mathit{tl}\, z)\ ,$$

if $\#\, x \rightharpoonup \nu_{\downarrow}$ is defined by $\#\, x \rightharpoonup \nu_{\downarrow} = \nu_{\downarrow}$.

For the second requirement we have

$$\begin{array}{ll} & \downarrow/\ \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x) \\ = & \qquad \text{split } \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x) \text{ in two parts} \\ & \downarrow/\,((\geq_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x) \cup (<_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x)) \\ = & \qquad \text{Theorem 3.48 and proviso below} \\ & \downarrow/\,(\geq_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x)\ , \end{array}$$

provided

$$\forall b \in (<_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x) : \exists a \in (\geq_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x) : a \downarrow b = a\ .$$

Suppose $\leq_R$ is defined such that for all lists v and w

$$v \mathbin{+\!\!+\!\!<} w \quad \leq_R \quad v\ . \tag{5.63}$$

Then, for all $b \in (<_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x)$, we have $b \mathbin{+\!\!+\!\!<} x \in (\geq_{\#}\ x)\triangleleft\, \mathit{tails}\,(x \mathbin{+\!\!+\!\!<} x)$, and by assumption (5.63) $(b \mathbin{+\!\!+\!\!<} x) \downarrow b = b \mathbin{+\!\!+\!\!<} x$. So lexicographical ordering $\leq_R$ is defined by

$$x \leq_R y \ \equiv\ y \in \mathit{inits}\, x\ \vee\ x <_{hd} y\ \vee\ (x =_{hd} y\ \wedge\ \mathit{tl}\, x \leq_R \mathit{tl}\, y)\ . \tag{5.64}$$

It is left to the reader to check that $\leq_R$ is a linear order on lists that extends the homogeneous lexicographical ordering. An important property of $\leq_R$ is

$$y \leq_R w\ \wedge\ w \notin \mathit{inits}\, y \quad \Rightarrow \quad y \mathbin{-\!\!<} a \leq_R w \mathbin{-\!\!<} a\ . \tag{5.65}$$

Furthermore, we have

$$y \leq_R w \quad \Rightarrow \quad \forall i \in \mathit{inits}\, w : y \leq_R i \ . \tag{5.66}$$

From now on, selector $\downarrow$ is the minimum operation with respect to the lexicographical ordering $\leq_R$. For this selector we have derived

$$\downarrow\!/\ \mathit{rots} \;=\; \#\, x \rightharpoonup \downarrow\!/\ \mathit{tails}\,(x \mathbin{+\!\!\!+} x)\ .$$

Deriving a left-reduction for $\downarrow\!/ \cdot \mathit{tails}$

The value $\downarrow\!/\ \mathit{rots}\, x$ has been expressed in terms of the value $\downarrow\!/\ \mathit{tails}\,(x \mathbin{+\!\!\!+} x)$. We want to construct an algorithm, that can be implemented as a linear-time program, for finding the latter value. We construct a left-reduction for $\downarrow\!/ \cdot \mathit{tails}$. Recall *tails* is a left-reduction $\odot\!\not\rightarrow\!\{[\,]\}$, where operator $\odot$ is defined by $x \odot a = (\mathbin{+\!\!<} a) * x \cup \{[\,]\}$. Suppose there exists an operator $\oslash$ such that $\downarrow\!/\ (x \odot a) = (\downarrow\!/\ x) \oslash a$ for all x in the image of *tails*. Then, applying Fusion on *snoc-list* we obtain

$$\downarrow\!/ \cdot \mathit{tails} \;=\; \oslash\!\not\rightarrow\![\,]\ . \tag{5.67}$$

An operator $\oslash$ satisfying $\downarrow\!/\ (x \odot a) = (\downarrow\!/\ x) \oslash a$ for all x in the image of *tails* (i.e. $x = \mathit{tails}\, z$ for some list z) is derived as follows.

$$\begin{array}{ll}
 & \downarrow\!/\ (\mathit{tails}\, z \odot a) \\
= & \quad \text{definition of } \odot \\
 & \downarrow\!/\ ((\mathbin{+\!\!<} a) * \mathit{tails}\, z \cup \{[\,]\}) \\
= & \quad \text{Theorem 3.48} \\
 & \downarrow\!/\ (\mathbin{+\!\!<} a) * \mathit{tails}\, z\ .
\end{array}$$

Let y be $\downarrow\!/\ \mathit{tails}\, z$, and split the set $\mathit{tails}\, z$ into $\mathit{tails}\, y$ and $(>_\#\ y) \triangleleft \mathit{tails}\, z$. Let v be an arbitrary element in $(>_\#\ y) \triangleleft \mathit{tails}\, z$. Then $v \notin \mathit{inits}\, y$, and by definition of y, $y \leq_R v$. Property (5.65) implies $y \mathbin{+\!\!<} a \leq_R v \mathbin{+\!\!<} a$. Hence, applying Theorem 3.48 we obtain

$$\begin{array}{ll}
 & \downarrow\!/\ (\mathbin{+\!\!<} a) * \mathit{tails}\, z \\
= & \quad \text{split } \mathit{tails}\, z \text{ in two parts} \\
 & \downarrow\!/\ (\mathbin{+\!\!<} a) * (\mathit{tails}\, y \cup (>_\#\ y) \triangleleft \mathit{tails}\, z) \\
= & \quad \text{definition of map} \\
 & \downarrow\!/\ ((\mathbin{+\!\!<} a) * \mathit{tails}\, y \cup (\mathbin{+\!\!<} a) * (>_\#\ y) \triangleleft \mathit{tails}\, z) \\
= & \quad \text{Theorem 3.48} \\
 & \downarrow\!/\ (\mathbin{+\!\!<} a) * \mathit{tails}\, y\ .
\end{array}$$

It follows that operator $\oslash$ can be defined by

$$t \oslash a \;=\; \downarrow\!/\ (\mathbin{+\!\!<} a) * \mathit{tails}\, t\ . \tag{5.68}$$

Improving the efficiency of the left-reduction for $\downarrow/ \cdot \mathit{tails}$, step 1

The operator $\oslash$ of the left-reduction for $\downarrow/ \cdot \mathit{tails}$ given in equation (5.68) requires time quadratic in the length of list t when implemented. We improve the efficiency of the left-reduction for $\downarrow/ \cdot \mathit{tails}$ in three steps.

Keep in mind that the left-hand argument t of operator $\oslash$ may be assumed to be of the form $\downarrow/\, \mathit{tails}\ z$ for some z. If $t = [\,]$, then $t \oslash a = [a]$. If $t \neq [\,]$, then

$$\begin{array}{ll}
 & \downarrow/\, (-\!\!<\! a)* \mathit{tails}\ t \\
= & \qquad \text{split } \mathit{tails}\ t \\
 & \downarrow/\, (-\!\!<\! a)* (\{t\} \cup \mathit{tails}\ \mathit{tl}\ t) \\
= & \qquad \text{definition of map} \\
 & \downarrow/\, (\{t -\!\!<\ a\} \cup (-\!\!<\! a)* \mathit{tails}\ \mathit{tl}\ t)\ .
\end{array}$$

To restrict the number of candidates for the minimum, note that for $w \in \mathit{tails}\ \mathit{tl}\ t \subseteq \mathit{tails}\ z$ we have the following.

$$\begin{array}{ll}
 & t -\!\!<\ a \leq_R w -\!\!<\ a \\
\Leftarrow & \qquad \text{equation (5.65), } t = \downarrow/\, \mathit{tails}\ z\text{, so } t \leq_R w \\
 & w \notin \mathit{inits}\ t \\
\equiv & \qquad w \in \mathit{tails}\ \mathit{tl}\ t \\
 & w \notin (\in \mathit{inits}\ t) \triangleleft \mathit{tails}\ \mathit{tl}\ t \\
\Leftarrow & \qquad \text{let } v = \mathit{ianit}\ t\text{, definition of } \mathit{ianit} \\
 & w >_{\#} v \\
\equiv & \qquad w \leq_{\#} v \equiv w \in \mathit{tails}\ v \\
 & w \notin \mathit{tails}\ v\ .
\end{array}$$

So, if $w \notin \mathit{tails}\ v$, then w is not a candidate for the minimum. Hence Theorem 3.48 gives

$$\begin{array}{ll}
 & \downarrow/\, (\{t -\!\!<\ a\} \cup (-\!\!<\! a)* \mathit{tails}\ \mathit{tl}\ t) \\
= & \qquad \text{split } \mathit{tails}\ \mathit{tl}\ t,\ v = \mathit{ianit}\ t\text{, definition of map} \\
 & \downarrow/\, (\{t -\!\!<\ a\} \cup (-\!\!<\! a)* \mathit{tails}\ v \cup (-\!\!<\! a)* (>_{\#} v) \triangleleft \mathit{tails}\ \mathit{tl}\ t) \\
= & \qquad \text{Theorem 3.48} \\
 & \downarrow/\, (\{t -\!\!<\ a\} \cup (-\!\!<\! a)* \mathit{tails}\ v) \\
= & \qquad \text{definition of reduction} \\
 & (t -\!\!<\ a) \downarrow \downarrow/\, (-\!\!<\! a)* \mathit{tails}\ v \\
= & \qquad \text{definition of operator } \oslash \text{ (5.68)}
\end{array}$$

$$(t \mathbin{-\!\!<} a) \downarrow (v \oslash a) \ .$$

Hence operator $\oslash$ satisfies $[\,] \oslash a = [a]$, and for $t \neq [\,]$,

$$t \oslash a \ = \ (t \mathbin{-\!\!<} a) \downarrow (\mathit{ianit}\ t \oslash a) \ . \tag{5.69}$$

This recursive characterisation of operator $\oslash$ is, when implemented, more efficient than definition (5.68) of operator $\oslash$. However, for the computation of a lexicographically least tail of a list, the implementation of left-reduction $\oslash\!\not\to\![\,]$ still requires more than linear time. We use one last result to improve the performance of operator $\oslash$.

Improving the efficiency of the left-reduction for $\downarrow\!/ \cdot \mathit{tails}$, step 2

An application of the left-reduction $\oslash\!\not\to\![\,]$ for $\downarrow\!/ \cdot \mathit{tails}$ to an argument of length n may require $O(n^2)$ evaluations of operator $\oslash$. The following lemma suggests a definition of operator $\oslash$ such that an application of the left-reduction $\oslash\!\not\to\![\,]$ to an argument of length n requires $O(n)$ evaluations of operator $\oslash$.

(5.70) Lemma *Let $t = \downarrow\!/\ \mathit{tails}\ z$. Then for all v in $\mathit{inits\ it}\ t$ we have*

$$t \mathbin{-\!\!<} a \leq_R v \mathbin{-\!\!<} a \quad \Rightarrow \quad \forall w \in \mathit{tails}\ v : t \mathbin{-\!\!<} a \leq_R w \mathbin{-\!\!<} a \ .$$

In particular

$$t \mathbin{-\!\!<} a \leq_R (\mathit{ianit}\ t) \mathbin{-\!\!<} a \quad \Rightarrow \quad t \oslash a = t \mathbin{-\!\!<} a \ .$$

Proof Let $t = \downarrow\!/\ \mathit{tails}\ z$, $v \in \mathit{inits\ it}\ t$, and w an arbitrary element in $\mathit{tails}\ v$. We have to prove that $t \mathbin{-\!\!<} a \leq_R v \mathbin{-\!\!<} a \ \Rightarrow \ t \mathbin{-\!\!<} a \leq_R w \mathbin{-\!\!<} a$. Since $t \mathbin{-\!\!<} a \leq_R t$ and $\leq_R$ is transitive, it suffices to show that $t \leq_R w \mathbin{-\!\!<} a$. Calculate as follows.

$$\begin{array}{ll}
 & t \mathbin{-\!\!<} a \leq_R v \mathbin{-\!\!<} a \\
\Rightarrow & \qquad v \in \mathit{inits\ it}\ t \ \Rightarrow \ t = (v \mathbin{-\!\!<} b) \mathbin{+\!\!\!+\!\!<} u \text{ for some } b \text{ and } u \\
 & t = (v \mathbin{-\!\!<} b) \mathbin{+\!\!\!+\!\!<} u \ \wedge \ b \leq a \\
\Rightarrow & \qquad w \in \mathit{tails}\ v \\
 & (w \mathbin{-\!\!<} b) \mathbin{+\!\!\!+\!\!<} u \in \mathit{tails}\ t \ \wedge \ b \leq a \\
\Rightarrow & \qquad t = \downarrow\!/\ \mathit{tails}\ z,\ \mathit{tails}\ t \subseteq \mathit{tails}\ z \\
 & t \leq_R (w \mathbin{-\!\!<} b) \mathbin{+\!\!\!+\!\!<} u \ \wedge \ b \leq a \\
\Rightarrow & \qquad (w \mathbin{-\!\!<} b) \mathbin{+\!\!\!+\!\!<} u \leq_R w \mathbin{-\!\!<} a \text{ by definition of } \leq_R \\
 & t \leq_R w \mathbin{-\!\!<} a \ .
\end{array}$$

This proves the first implication of this lemma. For the second implication, suppose that $t \mathbin{-\!\!<} a \leq_R (\mathit{ianit}\ t) \mathbin{-\!\!<} a$ holds.

$$
\begin{array}{cl}
 & t \oslash a \\
= & \quad \text{equation (5.69)} \\
 & (t \mathbin{-\!\!<} a) \downarrow (\mathit{ianit}\ t \oslash a) \\
= & \quad \text{equation (5.68)} \\
 & (t \mathbin{-\!\!<} a) \downarrow (\downarrow/\ (\mathbin{-\!\!<} a) * \mathit{tails}\ \mathit{ianit}\ t) \\
= & \quad \text{first implication of this lemma} \\
 & t \mathbin{-\!\!<} a\ .
\end{array}
$$

□

We have obtained the following definition for operator $\oslash$.

$$
t \oslash a \quad = \quad \begin{cases} [a] & \text{if } t = [\,] \\ t \mathbin{-\!\!<} a & \text{if } t \mathbin{-\!\!<} a \leq_R v \mathbin{-\!\!<} a \\ v \oslash a & \text{if } v \mathbin{-\!\!<} a \leq_R t \mathbin{-\!\!<} a \end{cases} \qquad (5.71)
$$
$$
\textbf{where } v = \mathit{ianit}\ t\ ,
$$

resulting in the following algorithm for finding a lexicographically least rotation.

$$
\mathit{llr}\ x \quad = \quad \#\, x \rightharpoonup (\oslash\!\not\rightarrow\![\,])\,(x \mathbin{-\!\!\!+\!\!<} x)\ ,
$$

where operator $\oslash$ is defined in equation (5.71). This algorithm can be implemented as a linear-time program if v can be determined in time at most $O(\#\, t - \#\, v)$ and if $t \mathbin{-\!\!<} a \leq_R v \mathbin{-\!\!<} a$ can be determined in constant time. We discuss these two conditions.

An extended specification

The straightforward implementation of the left-reduction $\oslash\!\not\rightarrow\![\,]$ derived for $\downarrow/\cdot \mathit{tails}$, where operator $\oslash$ is defined in equation (5.71), is not a linear-time program because of the occurrences of $\mathit{ianit}\ t$ in the definition of the operator $\oslash$. We tuple function $\downarrow/\cdot \mathit{tails}$ with some extra functions in order to obtain an algorithm that can be implemented as a linear-time program. This is the third and last step that improves the efficiency of the left-reduction $\oslash\!\not\rightarrow\![\,]$ for $\downarrow/\cdot \mathit{tails}$.

First, since $v = \mathit{ianit}\ t$, we have by definition of $\leq_R$ that

$$
\begin{array}{ll}
v \mathbin{-\!\!<} a \leq_R t \mathbin{-\!\!<} a & \text{if } a < \mathit{hd}\,(\#\, v \hookrightarrow t) \\
t \mathbin{-\!\!<} a \leq_R v \mathbin{-\!\!<} a & \text{if } \mathit{hd}\,(\#\, v \hookrightarrow t) \leq a\ .
\end{array}
$$

This suggests to tuple function $\downarrow/\cdot \mathit{tails}$ with the function that returns value $\#\, v \dashv ([\,], t)$.

For the computation of value $\mathit{ianit}\, t$ we tuple function $\downarrow/ \cdot \mathit{tails}$ with the function $\mathit{ianit} \cdot \downarrow/ \cdot \mathit{tails}$. To obtain a left-reduction that can be implemented as an efficient program for the resulting tuple of functions, bearing in mind the derivation of a left-reduction for function *ianit*, we tuple function $\downarrow/ \cdot \mathit{tails}$ with function $h \cdot \mathit{ianit} \vartriangle \mathit{id} \cdot \downarrow/ \cdot \mathit{tails}$. Since $\downarrow/ \cdot \mathit{tails}$ is a left-reduction $\oslash\!\not\to\![\,]$, the problem j we consider now is specified by

$$j \cdot \oslash\!\not\to\![\,]\ ,$$

where j is defined by

$$\begin{array}{lcl} j & = & \mathit{id} \vartriangle (h \cdot \mathit{ianit} \vartriangle \mathit{id}) \\ h & = & \mathit{exl} \vartriangle (\# \cdot \mathit{exl}) \vartriangle f \vartriangle g\ . \end{array}$$

Applying Fusion on *snoc-list* we obtain

$$j \cdot \oslash\!\not\to\![\,] \ = \ \ominus\!\not\to\!(j\,[\,])\ , \tag{5.72}$$

provided operator $\ominus$ satisfies

$$j\,(t \oslash a) \ = \ j\,t \ominus a\ . \tag{5.73}$$

The seed of the above left-reduction satisfies

$$j\,[\,] \ = \ ([\,], (\omega, 0, ([\,], [\,]), ([\,], [0])))\ .$$

To obtain an operator $\ominus$ satisfying equation (5.73) distinguish the different cases in the definition of operator $\oslash$. Let $(t, (p, q, r, s))$ be $j\,t$.

- If $t = [\,]$, then $j\,(t \oslash a) = j\,[a]$, so operator $\ominus$ is in case $t = [\,]$ defined by

$$(t, (p, q, r, s)) \ominus a \ = \ ([a], ([\,], 0, ([\,], [a]), ([\,], [0, 0])))\ . \tag{5.74}$$

- Suppose $t \twoheadleftarrow a \leq_R v \twoheadleftarrow a$, where $v = \mathit{ianit}\, t$. This case is equivalent to $\mathit{hd}\ \mathit{exr}\ r \leq a$. Then

$$\begin{array}{cl} & j\,(t \oslash a) \\ = & \quad \text{case assumption, definition of } \oslash \\ & j\,(t \twoheadleftarrow a) \\ = & \quad \text{definition of } j \\ & (\mathit{id} \vartriangle (h \cdot \mathit{ianit} \vartriangle \mathit{id}))\,(t \twoheadleftarrow a) \\ = & \quad h \cdot \mathit{ianit} \vartriangle \mathit{id} = \odot\!\not\to\!(h\,(\omega, [\,])) \\ & (t \twoheadleftarrow a, (h \cdot \mathit{ianit} \vartriangle \mathit{id})\,t \odot a) \\ = & \quad \text{definition of } (p, q, r, s) \\ & (t \twoheadleftarrow a, (p, q, r, s) \odot a) \\ = & \quad \text{definition of } \ominus \\ & (t, (p, q, r, s)) \ominus a\ , \end{array}$$

provided operator $\ominus$ is in case $\mathit{hd}\,\mathit{exr}\,r \leq a$ defined by

$$(t,(p,q,r,s)) \ominus a \quad = \quad (t \mathbin{-\!\!<} a, (p,q,r,s) \odot a)\ . \tag{5.75}$$

- To resolve the case $v \mathbin{-\!\!<} a \leq_R t \mathbin{-\!\!<} a$ or equivalently case $a < \mathit{hd}\,\mathit{exr}\,r$, we define operator $\ominus$ as follows. We prove equation (5.73) by induction on the length of list t. The base case $j\,([\,] \oslash a)$ has been treated in equation (5.74). If $t \neq [\,]$ and $\mathit{hd}\,\mathit{exr}\,r \leq a$, then equation (5.73) follows from definition (5.75) of operator $\ominus$. If $t \neq [\,]$ and $\mathit{hd}\,\mathit{exr}\,r > a$ we reason as follows. From $t \neq [\,]$ it follows that $\mathit{ianit}\,t <_{\#} t$. We have

$$\begin{array}{cl}
 & j\,(t \oslash a) \\
= & \qquad \text{case assumption, definition of } \oslash,\ v = \mathit{ianit}\,t \\
 & j\,(v \oslash a) \\
= & \qquad \text{induction hypothesis, } v <_{\#} t \\
 & j\,v \ominus a \\
= & \qquad \text{definition of } (t,(p,q,r,s)),\ l = \mathit{hd}\,\mathit{exr}\,s \\
 & (p,((q{-}l) \hookrightarrow p, l, l{-}q \dashv r, l{-}q \dashv s)) \ominus a \\
= & \qquad \text{case assumption, definition of } \ominus \\
 & (t,(p,q,r,s)) \ominus a\ ,
\end{array}$$

provided operator $\ominus$ is in case $a < \mathit{hd}\,\mathit{exr}\,r$ defined by

$$\begin{array}{rcl}
(t,(p,q,r,s)) \ominus a & = & (p,(l \hookleftarrow p, q{-}l, l \vdash r, l \vdash s)) \ominus a \\
 & & \textbf{where } l = q - \mathit{hd}\,\mathit{exr}\,s\ .
\end{array}$$

This concludes the proof of equation (5.73).

If operator $\ominus$ is defined by

$$\begin{array}{rcl}
(t,(p,q,r,s)) \ominus a & = & \left\{\begin{array}{ll}
([a],([\,],0,([\,],[a]),([\,],[0,0]))) & \text{if } t = [\,] \\
(t \mathbin{-\!\!<} a,(p,q,r,s) \odot a) & \text{if } \mathit{hd}\,\mathit{exr}\,r \leq a \\
(p,(l \hookleftarrow p, q{-}l, l \vdash r, l \vdash s)) \ominus a & \text{otherwise}
\end{array}\right. \\
 & & \textbf{where } l = q - \mathit{hd}\,\mathit{exr}\,s\ ,
\end{array}$$

then equation (5.72) holds. We have found

$$\mathit{llr}\,x \quad = \quad (\#\,x \rightharpoonup)\,\mathit{exl}\,(\ominus\!\not\to([\,],(\omega,0,([\,],[\,]),([\,],[0]))))\,(x \mathbin{-\!\!\!+\!\!\!<} x)\ .$$

The argument that was used to show that the program for the *iani* problem is linear time can be repeated for the straightforward implementation of the above algorithm to show that it is linear time.

5.6 The Hopping Splits Theorem

When function *segs* is replaced by the function which also returns the context of a segment, function *3splits3*, we can derive theorems corresponding to the *segs*-Fusion Theorem and the Hopping Tails Theorem.

c-slow predicates

A list is a palindrome if it is equal to its reverse, e.g. the words 'madam' and 'dad' are palindromes. The predicate *pal* determines whether or not a list is a palindrome. It is required to find a longest palindromic segment of a list. This problem is specified by

$$lps \quad = \quad \uparrow_{\#}/ \cdot pal\lhd \cdot segs \ . \tag{5.76}$$

None of the theory developed in the previous subsections is applicable to the *lps*-problem, because predicate *pal* is not prefix-closed. A property similar to prefix-closedness satisfied by *pal* is

$$pal\,([a] +\!\!\!+\!\!\!< x +\!\!\!+\!\!\!< [a]) \quad \Rightarrow \quad pal\, x \ .$$

Both the property satisfied by *pal* and the property prefix-closed are captured in the notion c-slow. A predicate is *c-slow* for a constant $c \geq 0$ if for all lists y with $\#\, y = c$, all lists x, and all elements a

$$p\,(y +\!\!\!+\!\!\!< x +\!\!\!+\!\!\!< [a]) \quad \Rightarrow \quad p\, x \ .$$

Equivalently, a predicate is c-slow if for all lists x with $\#\, x \geq c$,

$$p\,(x +\!\!\!< a) \quad \Rightarrow \quad p\,(c \hookrightarrow x) \ . \tag{5.77}$$

A prefix-closed predicate is 0-slow. The predicate *pal* is 1-slow. A predicate can be c-slow for various constants c, consider for example segment-closed predicates. A segment-closed predicate is c-slow for all natural numbers c. For the moment we assume in addition that c-slow predicates hold for all lists of length at most c. Like for prefix-closed predicates, for every c-slow predicate p there exists a *derivative* function ∇ such that for all lists y with $\#\, y = c$, all lists x, and all values a

$$p\,(y +\!\!\!+\!\!\!< x +\!\!\!< a) \quad = \quad p\, x \ \wedge \ a \nabla_x y \ .$$

For example, predicate *pal* has a derivative function ∇ defined by

$$a \nabla_x [b] \quad = \quad (a = b) \ .$$

The *3splits3*-Fusion Theorem

Theorems like the Sliding Tails Theorem and the Hopping Tails Theorem exist for longest-p segment problems where p is a c-slow predicate. In the derivation of these theorems, the context of a segment plays an important role. For that purpose we no longer use function *segs* but instead function *3splits3*. A longest-p segment problem is specified by (remember that function π_1 returns the second component of a triple)

$$\uparrow_{\#\cdot\pi_1}/\cdot(p\cdot\pi_1)\triangleleft\cdot \mathit{3splits3}\ .$$

Let ω be $\nu_{\uparrow_{\#\cdot\pi_1}}$. For this function we can derive theorems corresponding to the Sliding Tails Theorem and the Hopping Tails Theorem. We give the derivation of a variant of the Hopping Tails Theorem called the Hopping Splits Theorem. The derivation of this theorem proceeds along exactly the same lines as its original. First, the derivation of the *segs*-Fusion Theorem presented in subsection 5.2 is repeated for *segs* replaced by *3splits3*. The conditions of the *3splits3*-Fusion Theorem correspond to assumption (5.20). Then we give specific operators that validate the assumption.

The generic specification of a segment problem (5.14) is translated into

$$\oplus/\cdot f*\cdot \mathit{3splits3}\ ,$$

for arbitrary operator $\oplus$ and function f. The derivation that proves the *3splits3*-Fusion Theorem corresponds almost exactly to the derivation of the *segs*-Fusion Theorem.

(5.78) Theorem (*3splits3*-Fusion) *Suppose there exist operators $\ominus_i$ with $i : 1\leq i\leq 3$ such that $\oplus/\cdot f*\cdot(\nprec_i a)* = (\ominus_i a)\cdot\oplus/\cdot f*$ on the image of 3splitsi. Then*

$$\oplus/\cdot f*\cdot \mathit{3splitsi} \quad=\quad \pi_{3-i}\cdot\oslash\!\not\to\!(v,v,v)\ ,$$

where v abbreviates $f\ ([\,],[\,],[\,])$, and operator $\oslash$ is defined by

$$\begin{array}{rcl}(x_3,x_2,x_1)\oslash a &=& (r_3\oplus r_2\oplus r_1, r_2\oplus r_1, r_1)\\ && \textbf{where}\ r_i = x_i\ominus_i a\ .\end{array}$$

Proof Remember that function π_1 returns the *second* component of a triple, whereas operator $\nprec_1$ appends an element to the *first* element of a triple. This theorem is an immediate consequence of Fusion on *snoc-list*. Recall the definition of functor III: $\mathsf{III} = \mathsf{I}\times\mathsf{I}\times\mathsf{I}$. Since function *sss* is a left-reduction $\odot\!\not\to\!(\{[\,]^3\},\{[\,]^3\},\{[\,]^3\})$, where operator $\odot$ is defined by

$$\begin{array}{rcl}(x_3,x_2,x_1)\odot a &=& (r_3\cup r_2\cup r_1, r_2\cup r_1, r_1)\\ && \textbf{where}\ r_i = (\nprec_i a)*\,x_i\ ,\end{array}$$

with $i : 1 \leq i \leq 3$, see (5.13), we have by Fusion on *snoc-list*

$$(\oplus/ \cdot f*)\mathbin{|||} \cdot sss \quad = \quad \oslash\!\not\rightarrow(v, v, v) \ ,$$

provided $v = \oplus/ f* \{[\,]^3\}$, which is equivalent to $v = f[\,]^3$, and

$$(\oplus/ \cdot f*)\mathbin{|||} ((x, y, z) \odot a) \quad = \quad (\oplus/ \cdot f*)\mathbin{|||} (x, y, z) \oslash a \ ,$$

for (x, y, z) in the image of sss. It is easily verified that the definition of operator $\oslash$ given in the theorem satisfies this condition. □

Applying the *3splits3*-Fusion Theorem to a longest-p segment problem

Apply the *3splits3*-Fusion Theorem to a longest-p segment problem. The equations

$$\begin{aligned}\uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \cdot (\text{⤙}_1 a)* \quad &= \quad (\text{⤙}_1 a) \cdot \uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \\ \uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \cdot (\text{⤙}_3 a)* \quad &= \quad (\text{⤙}_3 a) \cdot \uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \ ,\end{aligned}$$

are easily seen to hold, so we may choose $\ominus_1 = \text{⤙}_1$ and $\ominus_3 = \text{⤙}_3$. It remains to give an operator $\ominus_2$ satisfying the equation

$$\uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \cdot (\text{⤙}_2 a)* \quad = \quad (\ominus_2 a) \cdot \uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \ , \tag{5.79}$$

on the image of *3splits2*. The derivation of a definition of operator $\ominus_2$ proceeds along exactly the same lines as presented in the sections on the Sliding Tails Theorem and the Hopping Tails Theorem.

Equalities concerning function *3splits2*

Two important equalities we use in the derivation of a definition of operator $\ominus_2$ correspond to equality $p\triangleleft \cdot tails = p\triangleleft \cdot tails \cdot \uparrow_{\#}/ \cdot p\triangleleft \cdot tails$ proved in Section 5.3, see (5.29). The first of these reads as follows.

Let $\succ\!\!\!+\!\!\!+_1$ be defined by

$$y \succ\!\!\!+\!\!\!+_1 (u, v, w) \quad = \quad (y +\!\!\!+\!\!\!\prec u, v, w) \ ,$$

and let *lrp* be defined by

$$lrp \quad = \quad \uparrow_{\#\cdot\pi_1}/ \cdot (p \cdot \pi_1)\triangleleft \cdot \mathit{3splits2} \ . \tag{5.80}$$

Let $d \leq c$, let (y, z) be $d \dashv ([\,], x)$, and let $(u, v, [\,])$ be $lrp\ z$. Then

$$(p \cdot (c \hookrightarrow) \cdot \pi_1)\triangleleft\, \mathit{3splits2}\ x \quad = \quad (p \cdot (c \hookrightarrow) \cdot \pi_1)\triangleleft (s \succ\!\!\!+\!\!\!+_1)*\, \mathit{3splits2}\ t \ , \tag{5.81}$$

where $(s, t) = c \vdash (y +\!\!\!+\!\!\!\prec u, v)$. This equality is proved as follows.

$$\begin{array}{ll}
& (p \cdot (c \hookrightarrow) \cdot \pi_1) \triangleleft \mathit{3splits2}\ x \\
= & \quad \text{splits the set } \mathit{3splits2}\ x \text{ in two parts} \\
& (p \cdot (c \hookrightarrow) \cdot \pi_1) \triangleleft ((<_{\#}\ s \cdot \pi_0) \triangleleft \mathit{3splits2}\ x \cup (\geq_{\#}\ s \cdot \pi_0) \triangleleft \mathit{3splits2}\ x) \\
= & \quad \text{claim: for } s \text{ as defined above: } \forall a \in (<_{\#}\ s \cdot \pi_0) \triangleleft \mathit{3splits2}\ x : \neg p\,(c \hookrightarrow)\,\pi_1 a \\
& (p \cdot (c \hookrightarrow) \cdot \pi_1) \triangleleft (\geq_{\#}\ s \cdot \pi_0) \triangleleft \mathit{3splits2}\ x \\
= & \quad \text{definition of } \mathit{3splits2} \text{ and } (s, t) \\
& (p \cdot (c \hookrightarrow) \cdot \pi_1) \triangleleft (s \succ\!\!+\!\!+\ _1) * \mathit{3splits2}\ t\ .
\end{array}$$

It remains to prove the statement claimed at the second step of this calculation.

If $s = [\,]$, then $(<_{\#}\ s) \cdot \pi_0 = \underline{\mathit{false}}$, and the statement is true. Assume $s >_{\#} [\,]$. Let a be an element of $(<_{\#}\ s \cdot \pi_0) \triangleleft \mathit{3splits2}\ x$. It follows that $a = (k, l, [\,])$, with $k <_{\#} s$, and hence with $l >_{\#} t$.

$$\begin{array}{ll}
& \neg p\,(c \hookrightarrow)\,\pi_1\ a \\
\equiv & \quad a = (k, l, [\,]) \in \mathit{3splits2}\ x \\
& \neg p\,(c \hookrightarrow l) \\
\Leftarrow & \quad (u, v, [\,]) = \mathit{lrp}\ z \\
& c \hookrightarrow l >_{\#} v\ \ \wedge\ \ c \hookrightarrow l \in \pi_1 * \mathit{3splits2}\ z \\
\Leftarrow & \quad k +\!\!+\!\!\prec l = y +\!\!+\!\!\prec z = x \\
& \#\, l - c > \#\, v\ \ \wedge\ \ \#\,(c \hookrightarrow l) \leq \#\, z \\
\Leftarrow & \quad \#\,(c \hookrightarrow l) \leq \#\,(c \hookrightarrow x) \text{ and } \#\, x - c \leq \#\, x - d = \#\, z \\
& \#\, l - c > \#\, v\ \ \wedge\ \ \#\,(c \hookrightarrow x) \leq \#\, x - c \\
\equiv & \quad \#\, x - c \geq \#\, l - c > 0\ \Rightarrow\ \#\,(c \hookrightarrow x) = \#\, x - c \\
& \#\, l > \#\, v + c \\
\equiv & \quad s \neq [\,]\ \Rightarrow\ \#\, t = \#\, v + c \\
& \#\, l > \#\, t \\
\Leftarrow & \quad k +\!\!+\!\!\prec l = s +\!\!+\!\!\prec t \\
& \#\, k < \#\, s\ .
\end{array}$$

The second equality we need in the subsequent derivation is an immediate consequence of equality (5.81). Let (u, v) be $c \vdash (y, z)$, where $(y, z, [\,])$ is $\mathit{lrp}\ x$. Then

$$(p \cdot (c \hookrightarrow) \cdot \pi_1) \triangleleft \mathit{3splits2}\ x\ \ =\ \ (p \cdot (c \hookrightarrow) \cdot \pi_1) \triangleleft (u \succ\!\!+\!\!+\ _1) * \mathit{3splits2}\ v\ . \qquad (5.82)$$

This equality follows from equality (5.81) if we take $d = 0$.

A first definition of operator $\ominus_2$

The following calculation derives a first definition of operator $\ominus_2$ satisfying equation (5.79), provided predicate p is c-slow with a derivative ∇.

$$
\begin{array}{cl}
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(-\!\!\!<_2 a)*\ \mathit{3splits2}\ x \\
= & \quad \text{map-filter swap (2.73)} \\
 & \uparrow_{\#\cdot\pi_1}/\,(-\!\!\!<_2 a)*\,(p\cdot\pi_1\cdot(-\!\!\!<_2 a))\triangleleft\ \mathit{3splits2}\ x \\
= & \quad p \text{ is } c\text{-slow, equation (5.77), } (p\wedge q)\triangleleft = p\triangleleft\cdot q\triangleleft \\
 & \uparrow_{\#\cdot\pi_1}/\,(-\!\!\!<_2 a)*\,(p\cdot\pi_1\cdot(-\!\!\!<_2 a))\triangleleft(p\cdot(c\hookrightarrow)\cdot\pi_1)\triangleleft\ \mathit{3splits2}\ x \\
= & \quad \text{equation (5.82), } (y,z,[\,]) = \mathit{lrp}\ x,\ (u,v) = c\vdash(y,z) \\
 & \uparrow_{\#\cdot\pi_1}/\,(-\!\!\!<_2 a)*\,(p\cdot\pi_1\cdot(-\!\!\!<_2 a))\triangleleft(p\cdot(c\hookrightarrow)\cdot\pi_1)\triangleleft(u\succ\!\!+\!\!\!+_1)*\ \mathit{3splits2}\ v \\
= & \quad \text{above steps in reverse order} \\
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(-\!\!\!<_2 a)*\,(u\succ\!\!+\!\!\!+_1)*\ \mathit{3splits2}\ v\ .
\end{array}
$$

Since this last expression expresses $\uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(-\!\!\!<_2 a)*\ \mathit{3splits2}\ x$ in terms of a and (u,v), where $(u,v,[\,]) = \mathit{lrp}\ x$, we have obtained the required dependency. Define operator $\ominus_2$ by

$$
\begin{array}{lcl}
(y,z,[\,])\ominus_2 a & = & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(-\!\!\!<_2 a)*\,(u\succ\!\!+\!\!\!+_1)*\ \mathit{3splits2}\ v \\
 & & \textbf{where } (u,v) = c\vdash(y,z)\ .
\end{array}
\tag{5.83}
$$

Now that we have found a definition of an operator $\ominus_2$ satisfying equation (5.79), we can apply the *3splits3*-Fusion Theorem to a longest-p segment problem with c-slow predicate p. However, the result of this application is an algorithm that is not very efficient when implemented, and therefore we try to find another (recursive) definition of operator $\ominus_2$.

A second definition of operator $\ominus_2$

To obtain a recursive definition of operator $\ominus_2$, distinguish the following four cases.

$$
\begin{array}{l}
\#\,y \geq c\ \wedge\ p\,(v -\!\!\!< a) \\
\#\,y \geq c\ \wedge\ \neg p\,(v -\!\!\!< a)\ \wedge\ z \neq [\,] \\
\#\,y < c\ \wedge\ z \neq [\,] \\
(\#\,y < c\ \vee\ \neg p\,(v -\!\!\!< a))\ \wedge\ z = [\,]\ .
\end{array}
$$

- Assume $\#\,y \geq c\ \wedge\ p\,(v -\!\!\!< a)$. Then

$$
\begin{array}{cl}
 & (y,z,[\,])\ominus_2 a \\
= & \quad \text{definition (5.83) of } \ominus_2,\ (u,v) = c\vdash(y,z)
\end{array}
$$

$$\begin{array}{ll}
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,(u\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,\mathit{3splits2}\ v\\
= & \quad\text{case assumption}\\
 & (u, v\mathbin{-\!\!\prec} a, [\,])\ .
\end{array}$$

- If $\#\,y \geq c\ \wedge\ \neg p\,(v\mathbin{-\!\!\prec} a)\ \wedge\ z \neq [\,]$, then $(u,v) = c \vdash (y,z)$ (and hence, since $z \neq [\,]$ we have $v \neq [\,]$) and $(y,z) = c \dashv (u,v)$.

$$\begin{array}{ll}
 & (y,z,[\,])\ominus_2 a\\
= & \quad\text{definition (5.83) of }\ominus_2,\ (u,v) = c\vdash(y,z)\\
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,(u\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,\mathit{3splits2}\ v\\
= & \quad\text{case assumption}\\
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,((u\mathbin{-\!\!\prec}\mathit{hd}\ v)\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,\mathit{3splits2}\ \mathit{tl}\ v\ .
\end{array}$$

At this point in the calculation we could fold, using definition (5.83) of operator $\ominus_2$, the last expression to $(y\mathbin{-\!\!\prec}\mathit{hd}\ z, \mathit{tl}\ z, [\,])\ominus_2 a$. The final result of this derivation would then have been a theorem corresponding to the Sliding Tails Theorem. To obtain a theorem corresponding to the Hopping Tails Theorem, proceed as follows. Let $w = u\mathbin{-\!\!\prec}\mathit{hd}\ v$.

$$\begin{array}{ll}
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,(w\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,\mathit{3splits2}\ \mathit{tl}\ v\\
= & \quad(w\mathbin{\succ\!\!\!+\!\!\!+}_1)*\text{ commutes with }(\mathbin{-\!\!\prec}_2 a)*\text{ and }(p\cdot\pi_1)\triangleleft\\
 & \uparrow_{\#\cdot\pi_1}/\,(w\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,\mathit{3splits2}\ \mathit{tl}\ v\\
= & \quad\text{map-filter swap, predicate }p\text{ is }c\text{-slow}\\
 & \uparrow_{\#\cdot\pi_1}/\,(w\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,(\mathbin{-\!\!\prec}_2 a)*\,(p\cdot\pi_1\cdot(\mathbin{-\!\!\prec}_2 a))\triangleleft(p\cdot(c\hookrightarrow)\cdot\pi_1)\triangleleft\mathit{3splits2}\ \mathit{tl}\ v\\
= & \quad(5.81),\ (s,t,[\,]) = \mathit{lrp}\ \mathit{tl}\ z,\ (q,r) = c\vdash((c\rightharpoonup\mathit{tl}\ v)\mathbin{+\!\!\!+\!\!\prec} s, t),\text{ etc.}\\
 & \uparrow_{\#\cdot\pi_1}/\,((w\mathbin{+\!\!\!+\!\!\prec} q)\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,(\mathbin{-\!\!\prec}_2 a)*\,(p\cdot\pi_1\cdot(\mathbin{-\!\!\prec}_2 a))\triangleleft(p\cdot(c\hookrightarrow)\cdot\pi_1)\triangleleft\mathit{3splits2}\ r\\
= & \quad\text{map-filter swap, etc.}\\
 & \uparrow_{\#\cdot\pi_1}/\,((w\mathbin{+\!\!\!+\!\!\prec} q)\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,\mathit{3splits2}\ r\\
= & \quad((w\mathbin{+\!\!\!+\!\!\prec} q)\mathbin{\succ\!\!\!+\!\!\!+}_1)*\text{ commutes with }(\mathbin{-\!\!\prec}_2 a)*\text{ and }(p\cdot\pi_1)\triangleleft\\
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!\prec}_2 a)*\,((w\mathbin{+\!\!\!+\!\!\prec} q)\mathbin{\succ\!\!\!+\!\!\!+}_1)*\,\mathit{3splits2}\ r\ .
\end{array}$$

By definition of the different components of the above derivation we have

$$\begin{array}{lll}
s\mathbin{+\!\!\!+\!\!\prec} t & = & \mathit{tl}\ z\\
\#\,r & = & \#\,t + c\\
q\mathbin{+\!\!\!+\!\!\prec} r & = & (c\rightharpoonup\mathit{tl}\ v)\mathbin{+\!\!\!+\!\!\prec}\mathit{tl}\ z\ =\ \mathit{tl}\ v\ .
\end{array}$$

It follows that

$$(u\mathbin{-\!\!\prec}\mathit{hd}\ v\mathbin{+\!\!\!+\!\!\prec} q, r)\ =\ c\vdash(y\mathbin{-\!\!\prec}\mathit{hd}\ z\mathbin{+\!\!\!+\!\!\prec} s, t)\ ,$$

and hence we obtain from the last expression of the above calculation and equation (5.83) that

$$
\begin{array}{ll}
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!<}_2 a)*((w \mathbin{+\!\!\!+\!\!<} q)\mathbin{\succ\!\!+\!\!\!+}_1)*\ \mathit{3splits2}\ r \\
= & \\
 & (y \mathbin{-\!\!<} \mathit{hd}\ z \mathbin{+\!\!\!+\!\!<} s, t, [\,])\ominus_2 a\ .
\end{array}
$$

- If $\#\,y < c\ \wedge\ z \neq [\,]$, then

$$
\begin{array}{ll}
 & (y, z, [\,])\ominus_2 a \\
= & \quad \text{definition (5.83) of } \ominus_2,\ (u, v) = c\vdash(y, z) \text{ so } u = [\,]\ \wedge\ v = y \mathbin{+\!\!\!+\!\!<} z \\
 & \uparrow_{\#\cdot\pi_1}/\,(p\cdot\pi_1)\triangleleft(\mathbin{-\!\!<}_2 a)*\ \mathit{3splits2}\ v \\
= & \quad \text{map-filter swap, predicate } p \text{ is } c\text{-slow} \\
 & \uparrow_{\#\cdot\pi_1}/\,(\mathbin{-\!\!<}_2 a)*\,(p\cdot\pi_1\cdot(\mathbin{-\!\!<}_2 a))\triangleleft(p\cdot(c\hookrightarrow)\cdot\pi_1)\triangleleft\ \mathit{3splits2}\ v \\
= & \quad \text{equation (5.81), } (s, t, [\,]) = \mathit{lrp}\ \mathit{tl}\ z,\ (q, r) = c\vdash(y \mathbin{-\!\!<} \mathit{hd}\ z \mathbin{+\!\!\!+\!\!<} s, t) \\
 & \uparrow_{\#\cdot\pi_1}/\,(\mathbin{-\!\!<}_2 a)*\,(p\cdot\pi_1\cdot(\mathbin{-\!\!<}_2 a))\triangleleft(p\cdot(c\hookrightarrow)\cdot\pi_1)\triangleleft(q\mathbin{\succ\!\!+\!\!\!+}_1)*\ \mathit{3splits2}\ r \\
= & \quad (q, r) = c\vdash(y \mathbin{-\!\!<} \mathit{hd}\ z \mathbin{+\!\!\!+\!\!<} s, t),\ \text{equality (5.83)} \\
 & (y \mathbin{-\!\!<} \mathit{hd}\ z \mathbin{+\!\!\!+\!\!<} s, t, [\,])\ominus_2 a\ .
\end{array}
$$

- Finally, if $(\#\,y < c\ \vee\ \neg p\,(v \mathbin{-\!\!<} a))\ \wedge\ z = [\,]$, then, since $p\,x$ holds for all x with $\#\,x \leq c$ for c-slow predicates p,

$$
(y, z, [\,])\ominus_2 a \quad = \quad (c{-}1\vdash(y, z \mathbin{-\!\!<} a))\propto_2 [\,]\ .
$$

We have obtained the following recursive characterisation of operator $\ominus_2$. Note that the second and third case can be treated as one case. We have $(x, y, [\,])\ominus_2 a = ((x, y)\circeq_2 a)\propto_2 [\,]$, where operator $\circeq_2$ is defined by

$$
(x, y)\circeq_2 a \quad = \quad \left\{
\begin{array}{ll}
c\vdash(x, y \mathbin{-\!\!<} a) & \text{if } \#\,x \geq c\ \wedge\ p\,(v \mathbin{-\!\!<} a) \\
(t\dashv(x, y))\circeq_2 a & \text{if } (\#\,x < c\ \vee\ \neg p\,(v \mathbin{-\!\!<} a))\ \wedge\ y \neq [\,] \\
c{-}1\vdash(x, y \mathbin{-\!\!<} a) & \text{otherwise}
\end{array}\right.
$$

$$
\begin{array}{ll}
\textbf{where} & t = \#\,y - \#\,\pi_1\ \mathit{lrp}\ \mathit{tl}\ y \\
 & (u, v) = c\vdash(x, y)\ .
\end{array}
$$

Again, using this definition of an operator $\ominus_2$, which satisfies equation (5.79), we can apply the *3splits3*-Fusion Theorem to a longest-p segment problem with c-slow predicate p. Again however, the result of this application is an algorithm that is not very efficient when implemented, because of the occurrence of $\mathit{lrp}\ \mathit{tl}\ y$ in the where-clause. Therefore, we try to adjust the definition of operator $\ominus_2$.

A third definition of operator $\ominus_2$

Operator $⊜_2$ is adjusted as follows. Observe that at each time operator $⊜_2$ is evaluated, the second component of its first argument satisfies predicate p. This observation invites to use a derivative of predicate p. If ∇ is a derivative of p, then

$$p\,(w \mathbin{+\!\!\!<} y \mathbin{-\!\!\!<} a) \quad\equiv\quad p\,y \;\wedge\; a\nabla_y w\ ,$$

for all lists w of length c. Redefine operator $⊜_2$ by

$$(x,y) ⊜_2 a \quad=\quad \begin{cases} c \vdash (x, y \mathbin{-\!\!\!<} a) & \text{if } \#\,x \geq c \;\wedge\; a\nabla_y w \\ (t \dashv (x,y)) ⊜_2 a & \text{if } (\#\,x < c \;\vee\; \neg(a\nabla_y w)) \;\wedge\; y \neq [\,] \\ c{-}1 \vdash (x, y \mathbin{-\!\!\!<} a) & \text{otherwise} \end{cases}$$
$$\textbf{where } t = \#\,y - \#\,\pi_1\, \mathit{lrp}\ \mathit{tl}\ y \qquad w = c \leftharpoonup x\ .$$

We have obtained the following corollary of the *3splits3*-Fusion Theorem.

(5.84) Theorem (Hopping Splits) *Let p be a c-slow predicate. Then*

$$\uparrow_{\#\cdot\pi_1}/ \cdot (p\cdot\pi_1)\triangleleft \cdot \mathit{3splits3} \quad=\quad \pi_0 \cdot \oslash\!\not\rightarrow (v,v,v)\ ,$$

where $v = [\,]^3$, and operator $\oslash$ is defined by

$$(x_3,x_2,x_1) \oslash a \quad=\quad (r_3 \uparrow_{\#\cdot\pi_1} r_2 \uparrow_{\#\cdot\pi_1} r_1,\ r_2 \uparrow_{\#\cdot\pi_1} r_1,\ r_1)$$
$$\textbf{where } r_i = x_i \ominus_i a\ ,$$

where $\ominus_1 = \mathbin{-\!\!\!<}_1$, $\ominus_3 = \mathbin{-\!\!\!<}_3$, and operator $\ominus_2$ is defined by $(x,y,[\,]) \ominus_2 a = ((x,y) ⊜_2 a) \propto_2 [\,]$, where operator $⊜_2$ is defined by

$$(x,y) ⊜_2 a \quad=\quad \begin{cases} c \vdash (x, y \mathbin{-\!\!\!<} a) & \textit{if } \#\,x \geq c \;\wedge\; a\nabla_y w \\ (t \dashv (x,y)) ⊜_2 a & \textit{if } (\#\,x < c \;\vee\; \neg(a\nabla_y w)) \;\wedge\; y \neq [\,] \\ c{-}1 \vdash (x, y \mathbin{-\!\!\!<} a) & \textit{otherwise} \end{cases}$$
$$\textbf{where } t = \#\,y - \#\,\pi_1\, \mathit{lrp}\ \mathit{tl}\ y \qquad w = c \leftharpoonup x\ .$$

The Hopping Splits Theorem is a generalisation of the Hopping Tails Theorem. The Hopping Tails Theorem is obtained from the Hopping Splits Theorem by setting c to 0, i.e., requiring predicate p to be prefix-closed, and applying function π_1 to both sides of the equality

$$\uparrow_{\#\cdot\pi_1}/ \cdot (p\cdot\pi_1)\triangleleft \cdot \mathit{3splits3} \quad=\quad \pi_0 \cdot \oslash\!\not\rightarrow (v,v,v)\ .$$

A short calculation shows that

$$\pi_1 \cdot {\uparrow_{\#\cdot\pi_1}}/ \cdot (p \cdot \pi_1)\triangleleft \cdot \mathit{3splits3} \quad = \quad {\uparrow_{\#}}/ \cdot p\triangleleft \cdot \mathit{segs} \ ,$$

and a rather long calculation shows that

$$\pi_1 \cdot \pi_0 \cdot \oslash\!\not\rightarrow(v, v, v) \quad = \quad \pi_0 \cdot \otimes\!\not\rightarrow([\,], [\,]) \ ,$$

where operator $\otimes$ is the operator $\oslash$ occurring in the Hopping Tails Theorem. Both calculations are omitted.

Example: the longest roof segment problem

As a first application of the Hopping Splits Theorem we consider the following problem. Predicate *monasc* determines whether or not a list is monotonely ascending, that is,

$$\mathit{monasc}\, y \quad \equiv \quad y = [\,] \ \vee \ y = [a] \ \vee \ (\exists z : y = z \mathbin{-\!\!\!\prec} a \ \wedge \ \mathit{monasc}\, z \ \wedge \ a = \mathit{lt}\, z + 1) \ .$$

A list is a *roof* if it satisfies the predicate *roof*, which is defined by

$$\mathit{roof}\, x \quad \equiv \quad x = [\,] \ \vee \ (\exists y : (x = y \mathbin{+\!\!\!\!\prec} \mathit{rev}\ \mathit{it}\ y) \ \wedge \ \mathit{monasc}\, y) \ .$$

It is required to find the longest roof among the segments of a list:

$${\uparrow_{\#\cdot\pi_1}}/ \cdot (\mathit{roof} \cdot \pi_1)\triangleleft \cdot \mathit{3splits3} \ .$$

Predicate *roof* is 1-slow and a derivative ∇ of *roof* is defined by

$$a \nabla_y [b] \quad \equiv \quad a = b \ \wedge \ a = \mathit{lt}\, y - 1 \ .$$

Note that $\mathit{roof}\, x \ \Rightarrow \ \mathit{pal}\, x$. Since predicate *roof* is 1-slow, apply the Hopping Splits Theorem to obtain the following left-reduction for the longest roof segment problem.

$${\uparrow_{\#\cdot\pi_1}}/ \cdot (\mathit{roof} \cdot \pi_1)\triangleleft \cdot \mathit{3splits3} \quad = \quad \pi_0 \cdot \oslash\!\not\rightarrow(v, v, v) \ ,$$

where $v = [\,]^3$, and operator $\oslash$ is defined by

$$\begin{array}{lcl} (x_3, x_2, x_1) \oslash a & = & (r_3 \uparrow_{\#\cdot\pi_1} r_2 \uparrow_{\#\cdot\pi_1} r_1, r_2 \uparrow_{\#\cdot\pi_1} r_1, r_1) \\ & & \quad \textbf{where } r_i = x_i \ominus_i a \ , \end{array}$$

where $\ominus_1 = \mathbin{-\!\!\!\prec}_1$, $\ominus_3 = \mathbin{-\!\!\!\prec}_3$, and operator $\ominus_2$ is defined by $(x, y, [\,]) \ominus_2 a = ((x, y) \circeq_2 a) \propto_2 [\,]$, where operator $\circeq_2$ is defined by

$$\begin{array}{lcl} (x, y) \circeq_2 a & = & \begin{cases} 1 \vdash (x, y \mathbin{-\!\!\!\prec} a) & \text{if } \# x \geq 1 \ \wedge \ a\nabla_y[\mathit{lt}\, x] \\ (t \dashv (x, y)) \circeq_2 a & \text{if } (\# x < 1 \ \vee \ \neg(a\nabla_y[\mathit{lt}\, x])) \ \wedge \ y \neq [\,] \\ (x, y \mathbin{-\!\!\!\prec} a) & \text{otherwise} \end{cases} \\ & & \textbf{where } t = \# y - \# \pi_1\, \mathit{lrp}\ \mathit{tl}\ y \ , \end{array}$$

where *lrp* is the function defined by

$$lrp \quad = \quad \uparrow_{\#\cdot\pi_1}/ \cdot (\mathit{roof} \cdot \pi_1)\triangleleft \cdot \mathit{3splits2}\ .$$

We simplify the definition of operator $\ominus_2$. Predicate *roof* satisfies

$$\mathit{roof}\ z \quad \Rightarrow \quad (\forall y \in \mathit{tails}\ z : 1 < \#\, y < \#\, z \quad \Rightarrow \quad \neg \mathit{roof}\ y)\ .$$

From this implication it follows that $\#\,\pi_1\ lrp\ tl\ y = 1$ if $tl\ y \neq [\,]$, and hence that

$$(t \dashv (x, y)) \mathbin{⊜}_2 a \quad = \quad (x \mathbin{+\!\!\!<} it\ y, [lt\ y]) \mathbin{⊜}_2 a\ ,$$

if $tl\ y \neq [\,]$. If $tl\ y = [\,]$, then $lrp\ tl\ y = [\,]$, and hence

$$(t \dashv (x, y)) \mathbin{⊜}_2 a \quad = \quad (x \mathbin{+\!\!\!<} y, [a])\ .$$

Redefine operator $⊜_2$ by

$$(x, y) \mathbin{⊜}_2 a \quad = \quad \begin{cases} 1 \vdash (x, y \mathbin{-\!\!<} a) & \text{if } \#\, x \geq 1\ \wedge\ a\nabla_y[lt\ x] \\ (x \mathbin{+\!\!\!<} it\ y, [lt\ y]) \mathbin{⊜}_2 a & \text{if } (\#\, x < 1\ \vee\ \neg(a\nabla_y[lt\ x]))\ \wedge\ tl\ y \neq [\,] \\ (x \mathbin{+\!\!\!<} y, [a]) & \text{if } (\#\, x < 1\ \vee\ \neg(a\nabla_y[lt\ x]))\ \wedge\ tl\ y = [\,] \\ (x, y \mathbin{-\!\!<} a) & \text{otherwise}\ . \end{cases}$$

The resulting algorithm can be implemented as a linear-time program.

A result for longest-p tail problems

If predicate p is c-slow, then function *lrp* defined in (5.80) itself satisfies

$$lrp \quad = \quad \ominus_2 \not\rightarrow [\,]^3\ , \tag{5.85}$$

where operator $\ominus_2$ is defined as in the Hopping Splits Theorem. This follows from the fact that operator $\ominus_2$ satisfies equation (5.79). We have

$$\begin{array}{ll} & \uparrow_{\#\cdot\pi_1}/\ (p \cdot \pi_1)\triangleleft\, \mathit{3splits2}\ (x \mathbin{-\!\!<} a) \\ = & \quad \text{definition of } \mathit{3splits2} \\ & \uparrow_{\#\cdot\pi_1}/\ (p \cdot \pi_1)\triangleleft\, ((\mathbin{-\!\!<}_2 a)*\ \mathit{3splits2}\ x \cup \{(x \mathbin{-\!\!<} a, [\,], [\,])\}) \\ = & \quad \text{definition of filter and reduce} \\ & (\uparrow_{\#\cdot\pi_1}/\ (p \cdot \pi_1)\triangleleft\, (\mathbin{-\!\!<}_2 a)*\ \mathit{3splits2}\ x) \uparrow_{\#\cdot\pi_1} (x \mathbin{-\!\!<} a, [\,], [\,]) \\ = & \quad \text{equation (5.79)} \\ & (\uparrow_{\#\cdot\pi_1}/\ (p \cdot \pi_1)\triangleleft\, \mathit{3splits2}\ x \ominus_2 a) \uparrow_{\#\cdot\pi_1} (x \mathbin{-\!\!<} a, [\,], [\,])\ , \end{array}$$

and we can prove by induction on the length of list y that

$$((x, y, [\,]) \ominus_2 a) \uparrow_{\#\cdot\pi_1} (x \mathbin{-\!\!<} a, [\,], [\,]) \quad = \quad (x, y, [\,]) \ominus_2 a\ .$$

Equality (5.85) follows.

5.7 Finding palindromes

In this section we derive, using the theory developed in the previous section, a linear-time algorithm for finding a longest palindromic segment of a list. This problem is specified in equation (5.76) by

$$lps \quad = \quad \uparrow_{\#}/ \cdot pal\triangleleft \cdot segs \ ,$$

where predicate *pal* is defined by $pal\,x \ \equiv \ x = rev\,x$. An occurrence of a palindrome in a string is called *extendible* if it is preceded and followed by equal characters, otherwise it is called *maximal*, including the case when there is no element preceding or following it. For example, in 'horror' the substring 'rr' is an extendible palindrome, and the substrings 'orro' and 'ror' are maximal palindromes. Formally, we have

$$y\;maxpal\;s \quad \equiv \quad y \in segs\;s \ \wedge \ pal\;y \ \wedge \ (\forall z \ : \ z +\!\!\!+ y +\!\!\!+ rev\;z \in segs\;s \ \Rightarrow \ z = [\,]) \ .$$

A longest palindromic segment of a list is of course a maximal palindrome.

Palindromes have been studied extensively in algorithm and complexity theory. Efficient algorithms for recognising the set of palindromes P, and similar problems have been constructed on several computing models. Cole [32] gives a real-time algorithm for recognising P on an iterative array of finite-state machines. Seiferas [120] shows how to recognise palindromes of even length on a computing model similar to the iterative array. Algorithms for recognising P on several different Turing-machine models are given in Hopcroft and Ullman [66]. A lower bound on the complexity of recognising palindromes on probabilistic Turing machines is derived by Yao [140].

Recognising initial palindromes in a string was the next problem related to palindromes to be addressed in several papers. Manacher [92] gives a linear-time algorithm on the random access machine (RAM) computing model finding the smallest initial palindrome of even length. He also describes how to adjust his algorithm in order to find the smallest initial palindrome of odd length ≥ 3. Manacher's algorithm is on line, that is, it is linear time, but in between reading two symbols from the input string more than constant time may be spent. The algorithm constructed by Manacher is obtained by Galil [51] using several theoretical results on fast simulations. Using their algorithm for pattern matching, Knuth, Morris, and Pratt [85] give an off-line linear-time RAM algorithm for finding the longest initial palindrome of even length. The ideas of Manacher are generalised by Galil and Seiferas [54] and Jeuring [70] to find (the positions of) all maximal palindromes in a string on line on the RAM computing model. Crochemore and Rytter [33] give a parallel version of this algorithm. In Jeuring [78] an algorithm for finding a shortest partition of a list into palindromes is given.

The papers mentioned above contain RAM algorithms for recognising palindromes. Algorithms on Turing machines have been devised too. Fischer and Paterson [39] give a linear-time off-line Turing-machine algorithm for finding all initial palindromes in a string.

Finally, Galil [50] and [52], describes a real-time (that is, on line, but in between reading two symbols from the input string constant time is spent) multitape Turing-machine algorithm finding all initial palindromes in a string. Galil's algorithm improves on Slisenko's work [123] on algorithms for finding palindromes.

The derivation of an algorithm for finding a longest palindromic segment consists of four parts. The first subsection applies the Hopping Splits Theorem to obtain a left-reduction for *lps*. The second subsection shows that maximal palindromes provide sufficient information to determine palindromic tails of a list, which are needed in the left-reduction given in the first subsection. The third subsection derives a left-reduction that returns all maximal palindromes of a list, and the fourth subsection, finally, describes the algorithm we have obtained, and discusses its complexity.

5.7.1 Applying the Hopping Splits Theorem

This section applies the Hopping Splits Theorem to the longest palindromic segment problem.

The specification of the problem of finding a longest palindromic segment has been given in Section 5.6, see (5.76). In terms of *3splits3* this specification reads as follows.

$$lps \;=\; \uparrow_{\#\cdot\pi_1}/ \cdot (pal \cdot \pi_1)\triangleleft \cdot \mathit{3splits3}\ .$$

In order to obtain a semantically equivalent on-line algorithm for this specification we apply the Hopping Splits Theorem. Since the predicate *pal* is 1-slow and has a derivative function $a\nabla_x[b] = (a = b)$, the Hopping Splits Theorem gives

$$lps \;=\; \pi_0 \cdot \oslash\!\not\rightarrow([\,]^3, [\,]^3, [\,]^3)\ ,$$

where operator $\oslash$ is defined by

$$(x_3, x_2, x_1) \oslash a \;=\; (r_3 \uparrow_{\#\cdot\pi_1} r_2 \uparrow_{\#\cdot\pi_1} r_1, r_2 \uparrow_{\#\cdot\pi_1} r_1, r_1) \\ \textbf{where } r_i = x_i \ominus_i a\ ,$$

where $\ominus_1 = \mathbin{-\!\!\!<}_1$, $\ominus_3 = \mathbin{-\!\!\!<}_3$, and operator $\ominus_2$ is defined by $(x, y, [\,]) \ominus_2 a = ((x, y) \circledequal_2 a) \propto_2 [\,]$, where operator $\circledequal_2$ is defined by

$$(x, y) \circledequal_2 a \;=\; \begin{cases} 1 \vdash (x, y \mathbin{-\!\!\!<} a) & \text{if } x \neq [\,] \ \wedge \ a = lt\, x \\ (t \dashv (x, y)) \circledequal_2 a & \text{if } (x = [\,] \ \vee \ a \neq lt\, x) \ \wedge \ y \neq [\,] \\ (x, y \mathbin{-\!\!\!<} a) & \text{otherwise} \end{cases} \\ \textbf{where } t = \#\, y - \#\, \pi_1\, lrp\ tl\ y\ ,$$

where function *lrp* is defined by

$$lrp \;=\; \uparrow_{\#\cdot\pi_1}/ \cdot (pal \cdot \pi_1)\triangleleft \cdot \mathit{3splits2}\ .$$

The straightforward implementation of this solution is a cubic-time program. This is explained as follows. Suppose *lps* is applied to a list of length n. Operator $\oslash$ is evaluated n times. Evaluating the expressions $x_1 \ominus_1 a$ and $x_3 \ominus_3 a$ and the expressions involving $\uparrow_{\#\cdot\pi_1}$ can be done in constant time. The evaluation of $x_2 \ominus_2 a$, however, might require quadratic time. The computation of $\#\,\pi_1\, \mathit{lrp}\ \mathit{tl}\ y$ is expensive, and the following subsections are devoted to finding means with which the value of $\#\,\pi_1\, \mathit{lrp}\ \mathit{tl}\ y$ can be found in constant time on the average.

5.7.2 Determining tail palindromes

In this section we show that the maximal palindromes of a list provide sufficient information to determine the value of $\#\,\pi_1\, \mathit{lrp}\ \mathit{tl}\ y$, where y is a palindrome.

The main idea in the construction of an efficient algorithm for finding a longest palindromic segment of a list is to make use of previously computed palindromes. Until now we only used the fact that predicate *pal* is 1-slow with a derivate $a\nabla_x[b] = (a = b)$. The longest palindromic tail of a palindrome, i.e. *lrp tl y*, the value of which is required in the algorithm given in the previous section, may be obtained from previously computed palindromes. It is to be expected that the previously encountered palindromic segments of y determine, at least partly, the tail palindromes of y. In fact, value *lrp tl y* can be expressed in terms of *llp it y*, where *llp* ('longest left palindrome') is the following function.

$$\mathit{llp} \quad = \quad \uparrow_{\#\cdot\pi_0}/ \cdot (\mathit{pal} \cdot \pi_0)\triangleleft \cdot \mathit{3splits2}\ . \tag{5.86}$$

Value *llp it y* on its turn can be determined form previously computed maximal palindromes. First we show how to express *lrp tl y* in terms of *llp it y*.

Expressing tail palindromes in initial palindromes

The following series of equalities is used to express tail palindromes in initial palindromes. If x is a palindrome, then

$$\mathit{tl}\, x \quad = \quad \mathit{rev}\ \mathit{it}\ x\ . \tag{5.87}$$

Define the function *prev* ('pair reverse'), which returns a reverse of the first two components of a triple of lists, by

$$\mathit{prev}\,(x, y, z) \quad = \quad (\mathit{rev}\, y, \mathit{rev}\, x, z)\ . \tag{5.88}$$

It can be verified that *prev*, *rev* and *3splits2* satisfy the following equation.

$$\mathit{prev}{*} \cdot \mathit{3splits2} \quad = \quad \mathit{3splits2} \cdot \mathit{rev}\ . \tag{5.89}$$

Furthermore, function *prev* is $(\uparrow_{\#\cdot\pi_0}, \uparrow_{\#\cdot\pi_1})$-fusable on the image of $(\in \mathit{3splits2}\ z)?_{\uparrow_{\#\cdot\pi_0}}*$ for all lists z, and it follows that it is $(\uparrow_{\#\cdot\pi_0}, \uparrow_{\#\cdot\pi_1})$-fusable on the image of $\uparrow_{\#\cdot\pi_0}/\cdot(\mathit{pal}\cdot\pi_0)\triangleleft\cdot(\in \mathit{3splits2}\ z)?_{\uparrow_{\#\cdot\pi_0}}*$ for all lists z too. Function *prev* satisfies

$$\pi_1 \cdot \mathit{prev} \quad = \quad \mathit{rev} \cdot \pi_0 \ . \tag{5.90}$$

This series of equalities is used in the following derivation, which proves that $\mathit{lrp}\ \mathit{tl}\ y = \mathit{prev}\ \mathit{llp}\ \mathit{it}\ y$.

$$\begin{aligned}
& \mathit{lrp}\ \mathit{tl}\ y \\
= \quad & \quad y \text{ is a palindrome, equality (5.87)} \\
& \mathit{lrp}\ \mathit{rev}\ \mathit{it}\ y \\
= \quad & \quad \text{definition } \mathit{lrp}\text{, equality (5.89)} \\
& \uparrow_{\#\cdot\pi_1}/\ (\mathit{pal}\cdot\pi_1)\triangleleft \mathit{prev}*\ \mathit{3splits2}\ \mathit{it}\ y \\
= \quad & \quad \text{map-filter swap (2.73)} \\
& \uparrow_{\#\cdot\pi_1}/\ \mathit{prev}*\ (\mathit{pal}\cdot\pi_1\cdot\mathit{prev})\triangleleft \mathit{3splits2}\ \mathit{it}\ y \\
= \quad & \quad \text{equality (5.90), } \mathit{pal}\cdot\mathit{rev} = \mathit{pal}\text{, } \mathit{prev} \text{ is } (\uparrow_{\#\cdot\pi_0}, \uparrow_{\#\cdot\pi_1})\text{-fusable} \\
& \mathit{prev}\ \uparrow_{\#\cdot\pi_0}/\ (\mathit{pal}\cdot\pi_0)\triangleleft \mathit{3splits2}\ \mathit{it}\ y \\
= \quad & \quad \text{definition } \mathit{llp} \\
& \mathit{prev}\ \mathit{llp}\ \mathit{it}\ y \ .
\end{aligned}$$

It follows that

$$\begin{aligned}
& \#\ \pi_1\ \mathit{lrp}\ \mathit{tl}\ y \\
= \quad & \quad \text{above calculation} \\
& \#\ \pi_1\ \mathit{prev}\ \mathit{llp}\ \mathit{it}\ y \\
= \quad & \quad \text{equality (5.90)} \\
& \#\ \mathit{rev}\ \pi_0\ \mathit{llp}\ \mathit{it}\ y \\
= \quad & \quad \#\cdot\mathit{rev} = \# \\
& \#\ \pi_0\ \mathit{llp}\ \mathit{it}\ y \ .
\end{aligned}$$

We return to the computation of value $\#\ \pi_0\ \mathit{llp}\ \mathit{it}\ y$ after the following discussion.

Palindromes and maximal palindromes

There are several ways to denote a palindrome in a list. Using the *3splits3*-view of segments, a palindrome in a list x is a triple (u, v, w) such that $(u, v, w) \in \mathit{3splits3}\ x$ and $\mathit{pal}\ v$. A second, and for our purposes convenient, way to represent a palindrome in a list is by means of its centre within the list and its length. Let x be a list with $\# x = n$. Function *centres* returns the list of centres of x in ascending order.

$$\mathit{centres}\, x \quad = \quad [0, 1/2, 1, 3/2, \ldots, n] \ .$$

By definition, the pair (c, l) is a palindrome in x if there exists a triple of lists $(u, v, w) \in \mathit{3splits3}\, x$ such that $\# v = l$, $\mathit{pal}\, v$, and $\# u + \# v/2 = c$. Furthermore, a maximal palindrome around centre c is the palindrome (c, l) in x such that for no $k > l$, (c, k) is a palindrome in x. Since for all centres c of x, $(c, 0)$ is a palindrome in x, there exists a unique maximal palindrome (c, l) in x for all centres c of x. Let function $\mathit{mp}_x\, c$ return the maximal palindrome in x with centre c

$$\mathit{mp}_x\, c = \uparrow_{exr}/\ \{(c, l) \mid (c, l) \text{ is a palindrome in } x\} \ .$$

Function mp_x satisfies the following properties. Suppose $\mathit{mp}_x\, c = (c, l)$, and $c + l/2 < \# x$. Then for all lists y

$$\mathit{mp}_x\, c \quad = \quad \mathit{mp}_{x +\!\!\!+\!\!\!< y}\, c \ . \tag{5.91}$$

Similarly, if $c - l/2 > 0$, then for all lists y

$$(c, l) = \mathit{mp}_x\, c \quad \equiv \quad (c + \# y, l) = \mathit{mp}_{y +\!\!\!+\!\!\!< x}\, (c + \# y) \ . \tag{5.92}$$

If y is a palindrome and $c > \# y/2$, then

$$(c, l) = \mathit{mp}_y\, c \quad \equiv \quad (\# y - c, l) = \mathit{mp}_y\, (\# y - c) \ . \tag{5.93}$$

The maximal palindromes of a list determine all palindromes of a list.

Expressing initial palindromes in maximal palindromes

Maximal palindromes are used to determine value $\# \pi_0\, \mathit{llp}\ \mathit{it}\, y$, where y is a palindrome. Recall that $(x, y, [\,])$ itself is a palindrome in some list v. The length of the longest initial palindrome in $\mathit{it}\, y$, $\# \pi_0\, \mathit{llp}\ \mathit{it}\, y$, is the length of the palindrome (c, l) in v with the largest centre c satisfying $\# x \le c \le \# x + (\# y - 1)/2$, and $c - l/2 = \# x$. Palindrome (c, l) is surrounded by a maximal palindrome $\mathit{mp}_v\, c$. Denote $\mathit{mp}_v\, c$ by (c, k). By definition of maximal palindrome we have $k \ge l$, and hence $c - k/2 \le \# x$. If we define (d, k) by

$$(d, k) \quad = \quad \mathit{lt}\, (p_{\# x}) \triangleleft \mathit{mp}_v {*}\, [0, 1/2, 1, \ldots, \# x + (\# y - 1)/2] \ ,$$

where predicate p_n is defined by

$$p_n\, (c, l) \quad = \quad c - l/2 \le n \ ,$$

then

$$\# \pi_0\, \mathit{llp}\ \mathit{it}\, y \quad = \quad 2 \times (d - \# x) \ .$$

So given the maximal palindromes with centres up to $\# x + (\# y - 1)/2$, we can determine value $\# \pi_0\, \mathit{llp}\ \mathit{it}\, y$.

Using maximal palindromes for finding palindromes

The left-reduction $\oslash\not\to([\,]^3,[\,]^3,[\,]^3)$ given in the previous subsection is equal to the specification $(\uparrow_{\#\cdot\pi_1}/\cdot(pal\cdot\pi_1)\triangleleft)⫴\cdot sss$. From the above we conclude that we want to have the maximal palindromes with centre smaller than the centre of the second coordinate of the left-reduction available. Replace function lrp in the specification by function $id \vartriangle f\cdot lrp$, where function f is defined by

$$f\,(x,y,[\,]) \;=\; mp_{x+\!\!\!+\!\!\!<y}*[0,1/2,1,3/2,\ldots,\#\,x+(\#\,y-1)/2]\,. \tag{5.94}$$

Suppose

$$id \vartriangle f\cdot lrp \;=\; \odot\not\to e\,, \tag{5.95}$$

for some value e and some operator $\odot$. Then

$$(\uparrow_{\#\cdot\pi_1}/\cdot(pal\cdot\pi_1)\triangleleft)⫴\cdot sss \;=\; \oslash'\not\to([\,]^3,e,[\,]^3)\,,$$

where operator $\oslash'$ is defined by

$$\begin{aligned}(x_3,x_2,x_1)\oslash' a \;&=\; (r_3\uparrow_{\#\cdot\pi_1} exl\; r_2\uparrow_{\#\cdot\pi_1} r_1, r_2, r_1)\\ &\qquad \textbf{where } r_i = x_i\ominus_i' a\,,\end{aligned}$$

where $\ominus_1' = -\!\!<_1$, $\ominus_3' = -\!\!<_3$, and $\ominus_2' = \odot$. It remains to find an operator $\odot$ and a value e such that equation (5.95) is satisfied.

5.7.3 A left-reduction returning maximal palindromes

This subsection derives a left-reduction which returns maximal palindromes of a list.

We want to construct a left-reduction for the function $id\vartriangle f\cdot lrp$, where function f is defined in equation (5.94). Since, by equation (5.85)

$$lrp \;=\; \ominus_2\not\to[\,]^3\,,$$

where operator $\ominus_2$ is defined by $(x,y,[\,])\ominus_2 a = ((x,y)\circleddash_2 a)\propto_2[\,]$, where operator $\circleddash_2$ is defined by

$$\begin{aligned}(x,y)\circleddash_2 a \;&=\; \begin{cases}1\vdash(x,y-\!\!<a) & \text{if } x\neq[\,]\;\wedge\; a=lt\,x\\ (t\dashv(x,y))\circleddash_2 a & \text{if } (x=[\,]\;\vee\; a\neq lt\,x)\;\wedge\; y\neq[\,]\\ (x,y-\!\!<a) & \text{otherwise}\end{cases}\\ &\qquad \textbf{where } t=\#\,y-2\times(d-\#\,x)\\ &\qquad\qquad (d,k)=lt\,(p_{\#\,x})\triangleleft mp_{x+\!\!\!+\!\!\!<y}*[0,1/2,\ldots,\#\,x+(\#\,y-1)/2]\,,\end{aligned}$$

we can apply Fusion on *snoc-list* to obtain

$$id\vartriangle f\cdot lrp \;=\; \odot\not\to([\,]^3,[\,])\,,$$

provided operator $\odot$ satisfies

$$(\mathit{id} \vartriangle f)\,((x, y, [\,]) \ominus_2 a) \;=\; (\mathit{id} \vartriangle f)\,(x, y, [\,]) \odot a \;, \tag{5.96}$$

for $(x, y, [\,])$ in the image of *lrp*. We construct a definition of operator $\odot$ such that this equation is satisfied. Since operator $\ominus_2$ is defined in terms of operator $\circledequal_2$, i.e.,

$$(x, y, [\,]) \ominus_2 a \;=\; ((x, y) \circledequal_2 a) \propto_2 [\,] \;,$$

we distinguish the different cases in the definition of operator $\circledequal_2$. Let $((x, y, [\,]), z)$ be $(\mathit{id} \vartriangle f)\,(x, y, [\,])$.

- If $x \neq [\,] \;\wedge\; a = \mathit{lt}\, x$, then

$$\begin{array}{cl}
& (\mathit{id} \vartriangle f)\,((x, y, [\,]) \ominus_2 a) \\
= & \quad \text{definition of } \ominus_2 \\
& (\mathit{id} \vartriangle f)\,(((x, y) \circledequal_2 a) \propto_2 [\,]) \\
= & \quad \text{definition of } \circledequal_2\text{, case assumption} \\
& (\mathit{id} \vartriangle f)\,(\mathit{it}\, x, [\mathit{lt}\, x] \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a, [\,]) \\
= & \quad \text{definition of } z\text{, see calculation below} \\
& ((\mathit{it}\, x, [\mathit{lt}\, x] \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a, [\,]), z) \\
= & \quad \text{definition of } \odot \text{ given below} \\
& ((x, y, [\,]), z) \odot a \;,
\end{array}$$

  provided operator $\odot$ is in case $x \neq [\,] \;\wedge\; a = \mathit{lt}\, x$ defined by

$$((x, y, [\,]), z) \odot a \;=\; ((\mathit{it}\, x, [\mathit{lt}\, x] \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a, [\,]), z) \;. \tag{5.97}$$

  The second component z of $((x, y, [\,]), z) \odot a$ is obtained as follows.

$$\begin{array}{cl}
& f\,(\mathit{it}\, x, [\mathit{lt}\, x] \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a, [\,]) \\
= & \quad \text{definition of } f \\
& \mathit{mp}_{x \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a} * [0, 1/2, 1, 3/2, \ldots, \#\, \mathit{it}\, x + (\#\, ([\mathit{lt}\, x] \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a) - 1)/2] \\
= & \quad \text{definition of } \# \\
& \mathit{mp}_{x \mathbin{+\!\!\!+\!\!\!<} y \mathbin{+\!\!\!<} a} * [0, 1/2, 1, 3/2, \ldots, \#\, x + (\#\, y - 1)/2] \\
= & \quad \text{equation (5.91), the applicability condition is verified below} \\
& \mathit{mp}_{x \mathbin{+\!\!\!+\!\!\!<} y} * [0, 1/2, 1, 3/2, \ldots, \#\, x + (\#\, y - 1)/2] \\
= & \quad \text{definition of } z \\
& z \;.
\end{array}$$

To apply equation (5.91) in the above calculation we have to show that for all centres c in the list $[0, 1/2, 1, 3/2, \ldots, \# x + (\# y - 1)/2]$, and

$$(c, l) \quad = \quad mp_{x +\!\!\!+\!\!< y}\; c$$

we have

$$c + l/2 \quad < \quad \# x + \# y\ .$$

Since $(x, y, [\,]) = \mathit{lrp}\; z$ for some list z, this equation holds.

- If the third case in the definition of operator $\ominus_2$ holds, i.e., $(x = [\,] \;\vee\; a \neq \mathit{lt}\, x) \;\wedge\; y = [\,]$, then

$$\begin{array}{ll}
 & (\mathit{id} \vartriangle f)\; ((x, y, [\,]) \ominus_2 a) \\
= & \qquad \text{definition of } \ominus_2 \\
 & (\mathit{id} \vartriangle f)\; (((x, y) ⊜_2 a) \propto_2 [\,]) \\
= & \qquad \text{definition of } ⊜_2\text{, case assumption} \\
 & (\mathit{id} \vartriangle f)\; (x, [a], [\,]) \\
= & \qquad \text{definition of } z \text{ and } f\text{, see calculation below} \\
 & ((x, [a], [\,]), z \mathbin{-\!\!<} (\# x, 0)) \\
= & \qquad \text{definition of } \odot \text{ given below} \\
 & ((x, y, [\,]), z) \odot a\ ,
\end{array}$$

provided operator $\odot$ is in case $(x = [\,] \;\vee\; a \neq \mathit{lt}\, x) \;\wedge\; y = [\,]$ defined by

$$((x, y, [\,]), z) \odot a \quad = \quad ((x, [a], [\,]), z \mathbin{-\!\!<} (\# x, 0))\ .$$

The second component of $((x, y, [\,]), z) \odot a$ is obtained as follows. Note that since $(x = [\,] \;\vee\; a \neq \mathit{lt}\, x) \;\wedge\; y = [\,]$ the maximal palindrome around centre $\# x$ has length 0.

$$\begin{array}{ll}
 & f\; (x, [a], [\,]) \\
= & \qquad \text{definition of } f \\
 & mp_{x \mathbin{-\!\!<} a} * [0, 1/2, 1, 3/2, \ldots, \# x + (\# [a] - 1)/2] \\
= & \qquad \text{definition of } \# \\
 & mp_{x \mathbin{-\!\!<} a} * [0, 1/2, 1, 3/2, \ldots, \# x] \\
= & \qquad \text{definition of map} \\
 & mp_{x \mathbin{-\!\!<} a} * [0, 1/2, 1, 3/2, \ldots, \# x - 1/2] \mathbin{-\!\!<} (mp_{x \mathbin{-\!\!<} a}\; \# x) \\
= & \qquad mp_{x \mathbin{-\!\!<} a}\; \# x = (\# x, 0) \\
 & mp_{x \mathbin{-\!\!<} a} * [0, 1/2, 1, 3/2, \ldots, \# x - 1/2] \mathbin{-\!\!<} (\# x, 0) \\
= & \qquad \text{equation (5.91)}
\end{array}$$

$mp_x * [0, 1/2, 1, 3/2, \ldots, \# x - 1/2] \mathbin{-\!\!\!<} (\# x, 0)$
$=$ definition of z
$z \mathbin{-\!\!\!<} (\# x, 0)$.

- Finally, in the remaining case in the definition of operator $\circledequal_2$ we have $(x = [\,] \;\vee\; a \neq lt\, x) \;\wedge\; y \neq [\,]$. We prove equation (5.96) by induction on the length of list y. The base case $(id \vartriangle f)\,((x, [\,], [\,]) \ominus_2 a) = (id \vartriangle f)\,(x, [\,], [\,]) \odot a$ follows immediately from the previous cases in the definition of operator $\odot$. If $y \neq [\,]$ and $x \neq [\,] \;\wedge\; a = lt\, x$ then definition (5.97) of operator $\odot$ applies. If $y \neq [\,]$ and $(x = [\,] \;\vee\; a \neq lt\, x)$ we reason as follows. Let (d, k) be $lt\,(p_{\# x}) \triangleleft z$, and $t = \# y - 2 \times (d - \# x)$.

$(id \vartriangle f)\,((x, y, [\,]) \ominus_2 a)$
$=$ definition of operator $\ominus_2$
$(id \vartriangle f)\,(((x, y) \circledequal_2 a) \propto_2 [\,])$
$=$ definition of $\circledequal_2$, case assumption
$(id \vartriangle f)\,(((t \dashv (x, y)) \circledequal_2 a) \propto_2 [\,])$
$=$ definition of $\ominus_2$
$(id \vartriangle f)\,(((t \dashv (x, y)) \propto_2 [\,]) \ominus_2 a)$
$=$ induction hypothesis
$(id \vartriangle f)\,((t \dashv (x, y)) \propto_2 [\,]) \odot a$
$=$ definition of $\odot$ given below
$((x, y, [\,]), z) \odot a$,

provided operator $\odot$ is in case $(x = [\,] \;\vee\; a \neq lt\, x) \;\wedge\; y \neq [\,]$ defined by

$$((x, y, [\,]), z) \odot a \quad = \quad ((t \dashv (x, y)) \propto_2 [\,], \zeta) \odot a \ ,$$

where ζ is the following list.

ζ
$=$ desired definition of $\odot$
$f\,((t \dashv (x, y)) \propto_2 [\,])$
$=$ definition of f, $b = \# x + \# y + t/2$
$mp_{x \mathbin{+\!\!\!+\!\!\!<} y} * [0, 1/2, 1, 3/2, \ldots, b - 1/2]$
$=$ definition of map, $b - 1/2 > \# x + (\# y - 1)/2$
$mp_{x \mathbin{+\!\!\!+\!\!\!<} y} * [0, 1/2, \ldots, \# x + (\# y - 1)/2] \mathbin{+\!\!\!+\!\!\!<}$
$mp_{x \mathbin{+\!\!\!+\!\!\!<} y} * [\# x + \# y/2, \ldots, b - 1/2]$
$=$ definition of z

$$
\begin{array}{ll}
 & z \mathbin{+\!\!\!+\!\!<} mp_{x+\!\!\!+\!\!<y}* [\# x + \# y/2, \ldots, b - 1/2] \\
= & \quad mp_{x+\!\!\!+\!\!<y} (\# x + \# y/2) = (\# x + \# y/2, \# y) \text{ by the case assumption} \\
 & z \mathbin{-\!\!<} (\# x + \# y/2, \# y) \mathbin{+\!\!\!+\!\!<} mp_{x+\!\!\!+\!\!<y}* [\# x + (\# y + 1)/2, \ldots, b - 1/2] \; .
\end{array}
$$

It remains to determine $mp_{x+\!\!\!+\!\!<y}* [\# x + (\# y + 1)/2, \ldots, b - l/2]$. Let c be a centre with $\# x + \# y/2 < c < b$. We want to determine $mp_{x+\!\!\!+\!\!<y}\, c$. Let l be the length of the maximal palindrome with centre c. By definition of b, $c + l/2 < \# x + \# y$. It follows that

$$
\begin{array}{ll}
 & (c, l) = mp_{x+\!\!\!+\!\!<y}\, c \\
\equiv & \quad c + l/2 < \# x + \# y;\ c > \# x + \# y/2, \text{ equation (5.92)} \\
 & (c - \# x, l) = mp_y\, (c - \# x) \\
\equiv & \quad y \text{ is a palindrome, equation (5.93)} \\
 & (\# y + \# x - c, l) = mp_y\, (\# y + \# x - c) \\
\equiv & \quad \# y + \# x - c - l/2 > 0, \text{ equation (5.92)} \\
 & (\# y + 2\# x - c, l) = mp_{x+\!\!\!+\!\!<y}\, (\# y + 2\# x - c) \; .
\end{array}
$$

Since

$$
\begin{array}{ll}
 & \# x + \# y/2 < c < b \\
\equiv & \quad \text{multiply with } -1 \\
 & -\# x - \# y/2 > -c > -b \\
\equiv & \quad \text{add } 2\# x + \# y \\
 & \# x + \# y/2 > 2\# x + \# y - c > 2\# x + \# y - b \; ,
\end{array}
$$

and since z contains the maximal palindromes with centres upto $\# x + \# y/2$, the length of the maximal palindrome around centre c can be determined from z. Let $m = \# x + \# y/2$. We have

$$mp_{x+\!\!\!+\!\!<y}* [\# x + (\# y + 1)/2, \ldots, b - 1/2] \;\; = \;\; rev\; g_m * (q_{m-t/2}) \triangleleft z \; ,$$

where predicate q_n is defined by

$$q_n\, (c, l) \;\; = \;\; c > n \; ,$$

and function g is defined by

$$g_n\, (c, l) \;\; = \;\; (n - c, l) \; .$$

We have obtained the following definition of operator $\odot$ satisfying (5.96).

$$
((x, y, [\,]), z) \odot a \;\; = \;\; \begin{cases}
(1 \dashv (x, y \mathbin{-\!\!<} a)) \propto_2 [\,], z) & \text{if } x \neq [\,] \;\wedge\; a = lt\, x \\
((t \dashv (x, y)) \propto_2 [\,], \zeta) \odot a & \text{if } (x = [\,] \;\vee\; a \neq lt\, x) \;\wedge\; y \neq [\,] \\
((x, [a], [\,]), z \mathbin{-\!\!<} (\# x, 0)) & \text{otherwise}
\end{cases}
$$

$$\begin{aligned}\textbf{where }\ \zeta &= z \mathbin{-\!\!\!<} (m, \#\, y) \mathbin{+\!\!\!+\!\!\!<}\ \mathit{rev}\ g_{2m}{*}\ q_{m-t/2}{\triangleleft}\, z\\ t &= \#\, y - 2\times(d - \#\, x)\\ (d,k) &= \mathit{lt}\,(p_{\#\, x}){\triangleleft}\, z\\ m &= \#\, x + \#\, y/2\ .\end{aligned}$$

In operational words: in the recursive alternative, z is scanned backwards for the first pair (d, k) such that $d - k/2 \le \#\, x$. The elements (c, l) for which $c - l/2 > \#\, x$ are pasted in reverse order, and with c replaced by $2\#\, x + \#\, y - c$, to z. The maximal palindrome around centre $\#\, x + \#\, y$ is the glue.

5.7.4 The algorithm and its complexity

This section gives the algorithm for finding a longest palindromic segment, and it discusses its complexity.

In the previous subsections we have derived the following left-reduction for *lps*.

$$\mathit{lps} \;=\; \pi_0 \cdot \oslash' \!\not\rightarrow\! ([\,]^3, ([\,]^3, [\,]), [\,]^3)\ ,$$

where operator $\oslash'$ is defined by

$$\begin{aligned}(x_3, x_2, x_1) \oslash' a \;&=\; (r_3 \uparrow_{\#\cdot\pi_1} \mathit{exl}\ r_2 \uparrow_{\#\cdot\pi_1} r_1, r_2, r_1)\\ &\qquad \textbf{where }\ r_i = x_i \ominus'_i a\ ,\end{aligned}$$

where $\ominus'_1 = \mathbin{-\!\!\!<}_1$, $\ominus'_3 = \mathbin{-\!\!\!<}_3$, and $\ominus'_2 = \odot$, where operator $\odot$ is defined by

$$((x, y, [\,]), z) \odot a \;=\; \begin{cases} (1 \dashv (x, y \mathbin{-\!\!\!<} a)) \propto_2 [\,], z) & \text{if } x \ne [\,] \;\wedge\; a = \mathit{lt}\, x\\ ((t {\dashv} (x, y)) {\propto_2} [\,], \zeta) \odot a & \text{if } (x = [\,] \vee a \ne \mathit{lt}\, x) \wedge y \ne [\,]\\ ((x, [a], [\,]), z \mathbin{-\!\!\!<} (\#\, x, 0)) & \text{otherwise}\end{cases}$$

$$\begin{aligned}\textbf{where }\ \zeta &= z \mathbin{-\!\!\!<} (m, \#\, y) \mathbin{+\!\!\!+\!\!\!<}\ \mathit{rev}\ g_{2m}{*}\ q_{m-t/2}{\triangleleft}\, z\\ t &= \#\, y - 2\times(d - \#\, x)\\ (d,k) &= \mathit{lt}\,(p_{\#\, x}){\triangleleft}\, z\\ m &= \#\, x + \#\, y/2\ .\end{aligned}$$

where predicates p_n and q_n are defined by

$$\begin{aligned}p_n\,(c, l) \;&=\; c - l/2 \le n\\ q_n\,(c, l) \;&=\; c > n\ ,\end{aligned}$$

and function g_n is defined by

$$g_n\,(c, l) \;=\; (n - c, l)\ .$$

For the purpose of computing all maximal palindromes, we append a fictitious element to the argument, which forces the centre of the last palindromic tail to exceed the length of the original argument, so that all maximal palindromes occurring in the argument are returned.

This algorithm for finding the longest palindromic segment of a list, together with a list of maximal palindromes of a list, can be implemented as a RAM program which requires time linear in the length of the list to which it is applied. The parts of this implementation that are not entirely straightforward are the parts corresponding to the computation of (d, k) and ζ and in particular the parts corresponding to respectively the expression $\mathit{rev}\, g_m {*}\, q_{m-d} \triangleleft z$ and the expression $\mathit{lt}\,(p_{\#\,x}) \triangleleft z$. In both cases the implementation should scan z from right to left starting at its right end. We give a rather informal argument to explain why this implementation is a linear-time program.

One of the key observations is that the number of maximal palindromes occurring in a list is linear in the length of the list. To be precise, given a list x of length n, there are exactly $2n{+}1$ maximal palindromes in x. This is proved by showing that there are exactly $2n{+}1$ centre positions in a list of length n, and around every position there is exactly one maximal palindrome.

Given a list of length n, the implementation of operator $\oslash'$ is evaluated exactly n times. However, in the definition of operator $\oslash'$ there is an occurrence of operator $\odot$ which is defined recursively. Obviously, the implementations of the first and third clause in the definition of operator $\odot$ require constant time for their evaluation. For the implementation of the second clause in the definition of $\odot$ we have that the total number of steps made in this clause is linear in the number of positions the centre of the longest palindromic tail under consideration has moved. Hence the implementation of operator $\odot$ requires constant time on the average. Since the implementation of operator $\odot$ requires constant time on the average, the implementation of left-reduction $\oslash' \not\rightarrow ([\,]^3, ([\,]^3, [\,]), [\,]^3)$ requires linear time.

5.8 Conclusions

This chapter presents a comprehensive theory for the generator *segs* which returns all segments of a list. The development of this theory is started with a derivation of a *segs*-Fusion Theorem, another example of a generator fusion theorem. Further development of the theory is guided by considering subclasses of segment problems. In this chapter we focus on longest-p segment problems: problems which require finding a longest segment satisfying predicate p of a list. For prefix-closed predicates p we derive the Sliding Tails Theorem and the Hopping Tails Theorem. Both theorems give an algorithm for a longest-p segment problem. If predicate-p satisfies some additional properties, corollaries giving an algorithm that can be implemented as a very efficient program can be derived. Using the theory, we construct algorithms for several segment problems, such as the pattern-matching problem. For the derivation of an algorithm for finding a longest palindromic segment, the theory is extended with a theory for the generator *3splits3*, the generator which returns all partitions of the argument into three parts. Using theorems for *3splits3* problems similar to the theorems for segment problems, an algorithm for finding palindromes that can be implemented as a linear-time program is derived.

A lot of algorithms for segment problems have been derived in the programming methodology described by Dijkstra and Feyen [37]. An important difference with our approach is that they define a list to be a function, and that retrieving an element of a list given an index can be done in constant time. We do not make this assumption a priori. Another difference is that Dijkstra and Feyen develop hardly any theory, so for each longest-p segment problem with prefix-closed predicate p a solution is calculated from scratch.

This chapter presents a small fraction of the theory of segment problems. First of all, it only derives left-reductions for segment problems. For the derivation of catamorphism on *join-list* for segment problems a separate theory has to be developed. Smith [125] gives a derivation of a catamorphism on *join-list* for the maximum segment sum problem. Second, there exist many classes of segment problems which do not fit in the class of longest-p segment problems, for example the class of shortest-p segment problems discussed in Jeuring and Meertens [79]. Finally, we do not give solutions for longest-p segment problems where predicate p is not c-slow. An example of such a problem is the longest low segment problem specified by

$$\uparrow_{\#}/ \cdot \mathit{low}\triangleleft \cdot \mathit{segs}\ ,$$

where predicate *low* is defined by

$$\mathit{low}\, x \quad \equiv \quad \uparrow/\, x \leq \#\, x\ .$$

A derivation of an algorithm for the longest low segment problem can be found in Swierstra and De Moor [129].

Chapter 6

Algorithms on the data type array

The data type *array* has been studied since 1858 (see Cayley [29]) and appears in many places in Mathematics, Physics and Computing Science, see [109, 61]. In this chapter we discuss the data type *n-dimensional array* for arbitrary natural number n. Zero-dimensional arrays are scalars: elements of any data type, like *bool* or *natural number*; one-dimensional arrays or vectors are snoc-lists and have been discussed in Chapter 2. The data type *n-dimensional array* with $n > 1$ is defined in terms of the data type *$(n-1)$-dimensional array*. For each n, the data type *n-dimensional array* is an initial object in a category of functor-algebras of a specific form. The theory pertaining to this initial object, such as properties like Catamorphism Characterisation and Fusion, is used in the derivation of algorithms on the data type *n-dimensional array*. This construction yields a hierarchy of data types, indexed by natural numbers, in which the nth element is expressed in terms of the $(n-1)$th element. A hierarchy of catamorphisms (algorithms) is defined on a hierarchy of data types and it contains a catamorphism (algorithm) for each element in the hierarchy of data types. The definition of the data type *n-dimensional array* we give is close to the intuitive idea we have of arrays, and should be compared with other definitions of the data type. The different definitions of *n-dimensional array* should be judged on conciseness and ease of manipulation of expressions in the type. For different problem domains one definition may be more suitable than another. Four attempts have been made to define the data type *(two-dimensional) array* within the Bird-Meertens calculus, see Banger [11], Bird [17], Jeuring [74, 77], and Malcolm [88]. Other definitions of arrays can be found in Mullin [107], [108], Hains and Mullin [62], and Wise [137].

This chapter is based on [74, 77]. It consists of five sections. The first section introduces hierarchical data types. Section 6.2 defines the hierarchy of data types for n-dimensional arrays, compares it with other definitions of the data type *n-dimensional array*, and defines some auxiliary functions on the data type *n-dimensional array*. Section 6.3 gives a hierarchy of algorithms returning all subarrays (a subarray of an array corresponds to a segment of a list), and derives a hierarchical version of Horner's rule, see Theorem (5.21). This rule is illustrated with the derivation of a hierarchy of algorithms for finding the maximum subarray sum. Furthermore, it briefly discusses the generalisation of longest-p segment problems to arrays. Section 6.4 shows the derivation of a hierarchy of algorithms for pattern matching on n-dimensional arrays, and Section 6.5, finally, gives some conclusions.

6.1 Hierarchical data types

Hierarchical data types and hierarchical catamorphisms are convenient for the derivation and description of hierarchies of algorithms. In this section we give the definitions of hierarchical data types and hierarchical catamorphisms. Most of the results are easy consequences of the theory of data types presented in Chapter 2.

Let $\dagger$ and $\ddagger$ be binary functors, and let $\dagger\underline{A}$ and $\ddagger\underline{B}$ be the induced monofunctors. Define the map-functors M and F by

$$\begin{array}{rcl}(B\mathsf{M}, in_{\ddagger\underline{B}}) & = & \mu(\ddagger\underline{B}) \\ (A\mathsf{F}, in_{\dagger\underline{A}}) & = & \mu(\dagger\underline{A}) \ .\end{array}$$

The hierarchical data type induced by the functor $\dagger\underline{A}$ (on the outer level) and $\ddagger\underline{B}$ (on the inner level) is defined to be $\mu(\dagger\underline{B\mathsf{M}})$. We have

$$(B\mathsf{MF}, in_{\dagger\underline{B\mathsf{M}}}) \ = \ \mu(\dagger\underline{B\mathsf{M}}) \ .$$

The data type $\mu(\dagger\underline{B\mathsf{M}})$ consists of two levels. We will need hierarchical data types of an arbitrary number of levels. Hierarchical data types are defined inductively as follows.

(6.1) Definition (Hierarchical Data Type) *A hierarchical data type of level 0 is any data type like 1,* bool*, etc. A hierarchical data type of level 1 is an initial fixed-point of a functor. Let (B, jn) be a hierarchical data type of level $n-1$. Furthermore, let L be the map-functor induced by a binary functor $\dagger$. Then $(B\mathsf{L}, in_{\dagger\underline{B}}) = \mu(\dagger\underline{B})$ is a hierarchical data type of level n.*

Let L_1, $\mathsf{L}_2, \ldots, \mathsf{L}_n$ be the map-functors induced by the binary functors $\dagger_1, \dagger_2, \ldots, \dagger_n$. Abbreviate the monofunctors $\dagger_n\underline{A\mathsf{L}_1 \ldots \mathsf{L}_{n-1}}$ and $\dagger_{n-1}\underline{A\mathsf{L}_1 \ldots \mathsf{L}_{n-2}}$, which occur frequently in the following definitions and theorems, to respectively $\ddagger_n\underline{A}$ and $\ddagger_{n-1}\underline{A}$. Hierarchical catamorphisms (catamorphisms defined on hierarchical data types, in which we distinguish the components which act on the different levels) are defined inductively as follows.

(6.2) Definition (Hierarchical Catamorphism) *A hierarchical catamorphism defined on a hierarchical data type $(\!|\phi_n, \ldots, \phi_0|\!)_{\ddagger_n\underline{A}} : A\mathsf{L}_1 \ldots \mathsf{L}_n \to B$ is defined as follows. If $n = 0$ then $(\!|\phi_n, \ldots, \phi_0|\!) = \phi_0$, and for $n \geq 1$ we define*

$$(\!|\phi_n, \ldots, \phi_0|\!)_{\ddagger_n\underline{A}} \ = \ (\!|\phi_n|\!)_{\dagger_n\underline{B}} \cdot (\!|\phi_{n-1}, \ldots, \phi_0|\!)_{\ddagger_{n-1}\underline{A}}\mathsf{L}_n \ ,$$

where we assume that $A\mathsf{L}_1 \ldots \mathsf{L}_0 = A$.

By definition of hierarchical catamorphisms, we have the following characterisation of hierarchical catamorphisms.

(6.3) Theorem (Hierarchical Catamorphism Characterisation) *For $n \geq 1$,*

$$h = \llparenthesis \phi_n, \ldots, \phi_0 \rrparenthesis_{\ddagger_n \underline{A}} \quad \equiv \quad h \cdot in_{\ddagger_n \underline{A}} = \phi_n \cdot h \dagger_n \llparenthesis \phi_{n-1}, \ldots, \phi_0 \rrparenthesis_{\ddagger_{n-1} \underline{A}} \ .$$

Proof

$$\begin{array}{ll}
 & h = \llparenthesis \phi_n, \ldots, \phi_0 \rrparenthesis_{\ddagger_n \underline{A}} \\
\equiv & \qquad \text{Hierarchical Catamorphism (6.2)} \\
 & h = \llparenthesis \phi_n \rrparenthesis_{\dagger_n \underline{B}} \cdot \llparenthesis \phi_{n-1}, \ldots, \phi_0 \rrparenthesis_{\ddagger_{n-1} \underline{A}} \mathsf{Ln} \\
\equiv & \qquad \text{Factorisation (2.52)} \\
 & h = \llparenthesis \phi_n \cdot id \dagger_n \llparenthesis \phi_{n-1}, \ldots, \phi_0 \rrparenthesis_{\ddagger_{n-1} \underline{A}} \rrparenthesis_{\ddagger_n \underline{A}} \\
\equiv & \qquad \text{Catamorphism (2.34)} \\
 & h \cdot in_{\ddagger_n \underline{A}} = \phi_n \cdot id \dagger_n \llparenthesis \phi_{n-1}, \ldots, \phi_0 \rrparenthesis_{\ddagger_{n-1} \underline{A}} \cdot h \dagger_n id \\
\equiv & \qquad \text{property of (binary) functors} \\
 & h \cdot in_{\ddagger_n \underline{A}} = \phi_n \cdot h \dagger_n \llparenthesis \phi_{n-1}, \ldots, \phi_0 \rrparenthesis_{\ddagger_{n-1} \underline{A}} \ .
\end{array}$$

□

Fusion is an easy consequence of this theorem.

(6.4) Theorem (Hierarchical Fusion) *Let $\psi_0 = h_0 \cdot \phi_0$, and suppose that for m with $n \geq m \geq 1$, h_m satisfies*

$$h_m \cdot \phi_m \quad = \quad \psi_m \cdot h_m \dagger_m h_{m-1} \ ,$$

on the image of $\llparenthesis \phi_m, \ldots, \phi_0 \rrparenthesis_{\ddagger_m \underline{A}} \dagger_m \llparenthesis \phi_{m-1}, \ldots, \phi_0 \rrparenthesis_{\ddagger_{m-1} \underline{A}}$. Then we have

$$h_n \cdot \llparenthesis \phi_n, \ldots, \phi_0 \rrparenthesis_{\ddagger_n \underline{A}} \quad = \quad \llparenthesis \psi_n, \ldots, \psi_0 \rrparenthesis_{\ddagger_n \underline{A}} \ .$$

By means of Factorisation (2.52), we can write every hierarchical catamorphism as a catamorphism in the usual sense. The other way around is also possible, that is, every catamorphism on a hierarchical data type can be written as a hierarchical catamorphism on that data type. Given a catamorphism defined on a hierarchical data type of n levels, we want to distinguish the n components which act on the different levels of the hierarchical data type. We have

(6.5) Theorem (Factorisation of catamorphisms) *Let the catamorphism $(\![\phi]\!)$ be defined on a hierarchical data type of level n. Then there exist $\psi_n, \ldots, \psi_0$ such that*

$$(\![\phi]\!) \quad = \quad (\![\psi_n, \ldots, \psi_0]\!) \; .$$

Proof If we take $\psi_0 = id$, $\psi_n = \phi$ and $\psi_m = in$ for all m with $n > m \geq 1$, we can prove by induction that $(\![\psi_{n-1}, \ldots, \psi_0]\!) = id$ and we can apply the definition of hierarchical catamorphisms (6.2) to obtain the above result. □

Note that the choices made for ψ in the proof of the theorem above are not very useful: in the cases in which we distinguish the levels of a hierarchical catamorphism the components are usually more interesting.

6.2 The data type array

In this section we give a definition of the hierarchical data type *array*. We compare the resulting data type with other approaches to arrays. Furthermore, we define some auxiliary functions on arrays.

6.2.1 Arrays as a hierarchical data type

In this section we give a definition of the hierarchical data type *array*. We give a definition of the data type *zero-dimensional array*, the set of elements of which is denoted by $A{\star}_0$; of the data type *one-dimensional array*, denoted by $(A{\star}, \square \triangledown {+\!\!<})$, and we define the data type *n-dimensional array* ($n > 1$), denoted by $(A{\star}_n, \square \triangledown {+\!\!<})$, in terms of the data type *$(n-1)$-dimensional array*.

The data type *zero-dimensional array*

Elements of the data type *zero-dimensional array*, the set of elements of which is denoted by $A{\star}_0$, are scalars: elements of data types like *bool* and *natural number*.

The data type *one-dimensional array*

The data type *one-dimensional array* over base type A, denoted by $(A{\star}, \square \triangledown {+\!\!<})$, is considered to be the data type *snoc-list* as defined in Section 2.7, that is

$$(A{\star}, \square \triangledown {+\!\!<}) \quad = \quad \mu(\mathsf{K}_A)$$

where the monofunctor K_A is defined by $\mathsf{K}_A = \underline{1} + \mathsf{I} \times \underline{A}$.

The data type *two-dimensional array*

A two-dimensional array is modelled by a list of lists. We will speak about a list of lists as a list of columns; the converse interpretation via rows is of course just as valid. For example, the list of lists $[[2,4,0],[3,0,1]]$ denotes the matrix

$$\begin{pmatrix} 2 & 3 \\ 4 & 0 \\ 0 & 1 \end{pmatrix}$$

Intuitively, only a list of equal-length columns is a proper two-dimensional array. We have not been able to define a pair of transformers that models this condition on the elements of an arbitrary $\mathsf{K}_{A\star}$-algebra, and therefore we forget about this restriction. The data type *two-dimensional array* is defined as an initial algebra in the category of $\mathsf{K}_{A\star}$-algebras, and hence a list with columns of different lengths is also an array. After the following definition of the data type *n-dimensional array* and the definition of a hierarchical left-reduction, we will define a family of functions $propar_n$ of which the component $propar_2$ determines, given a two-dimensional array, whether or not it is a proper two-dimensional array, i.e., whether or not all its columns have the same length.

The data type *n-dimensional array*

In general, the data type *n-dimensional array* ($n > 1$), denoted by $(A\star_n, \Box \triangledown \dashv\!<)$, is defined as an initial algebra in the category of $\mathsf{K}_{A\star_{n-1}}$-algebras.

(6.6) Definition *The data type* zero-dimensional array*, denoted by $A\star_0$, is any simple data type like* bool *and* nat. *The data type* one-dimensional array*, denoted by $(A\star, \Box \triangledown \dashv\!<)$, is the data type* snoc-list. *Let $(A\star_{n-1}, \Box \triangledown \dashv\!<)$ be the data type* $(n-1)$-dimensional array. *Then the data type* n-dimensional array *is an initial algebra in the category of $\mathsf{K}_{A\star_{n-1}}$-algebras, and is denoted by $(A\star_n, \Box \triangledown \dashv\!<)$, where $A\star_n = A\star_{n-1}\star$.*

Hierarchical left-reductions

A hierarchical catamorphism defined on n-dimensional arrays, called a *hierarchical left-reduction*, is defined as follows.

(6.7) Definition (Hierarchical Left-reduction) *A hierarchical left-reduction defined on the data type* n-dimensional array *is denoted by*

$$(\oplus_n, \ldots, \oplus_1, f) \not\to_n (e_n, \ldots, e_1) \quad : \quad A\star_n \to B \ ,$$

and defined as follows. If $n = 0$, then define $(\oplus_n, \ldots, \oplus_1, f) \not\to_n (e_n, \ldots, e_1) = f$, and for $n \geq 1$ define

$$(\oplus_n, \ldots, \oplus_1, f) \not\to_n (e_n, \ldots, e_1) \quad = \quad \oplus_n \not\to e_n \cdot ((\oplus_{n-1}, \ldots, \oplus_1, f) \not\to_{n-1} (e_{n-1}, \ldots, e_1))\star\ .$$

A hierarchical left-reduction can be characterised in a fashion similar to Hierarchical Catamorphism Characterisation, Theorem 6.3. The following theorem is one of the most important results for the derivation of hierarchies of algorithms on arrays.

(6.8) Theorem (Hierarchical Left-reduction Characterisation)

$$f_n \quad = \quad (\otimes_n, \ldots, \otimes_1, f_0) \not\to_n (e_n, \ldots, e_1)$$

if and only if for all m with $n \geq m \geq 1$,

$$\begin{array}{lcl} f_m\,[\,] & = & e_m \\ f_m\,(x \mathbin{-\!\!\!\prec} a) & = & (f_m\, x) \otimes_m (f_{m-1}\, a)\ , \end{array}$$

for some family of values e_n and some family of operators $\otimes_n$.

Proof By induction on n. The base case is valid since, by definition (6.7), if $n = 0$, then $(\otimes_0, \ldots, \otimes_1, f_0) \not\to_n (e_0, \ldots, e_1) = f_0$.

It remains to prove the induction step. The induction hypothesis is

$$f_{n-1} \quad = \quad (\otimes_{n-1}, \ldots, \otimes_1, f_0) \not\to_{n-1} (e_{n-1}, \ldots, e_1)$$

if and only if for all m with $n{-}1 \geq m \geq 1$,

$$\begin{array}{lcl} f_m\,[\,] & = & e_m \\ f_m\,(x \mathbin{-\!\!\!\prec} a) & = & (f_m\, x) \otimes_m (f_{m-1}\, a)\ , \end{array}$$

for some family of values e_m and some family of operators $\otimes_m$. We have

$$\begin{array}{ll} & f_n = (\otimes_n, \ldots, \otimes_1, f_0) \not\to_n (e_n, \ldots, e_1) \\ \equiv & \qquad \text{definition of hierarchical left-reduction (6.7), induction hypothesis} \\ & f_n = \otimes_n \not\to e_n \cdot f_{n-1}\star \\ \equiv & \qquad \text{Factorisation 2.52} \\ & f_n = (\otimes_n \cdot id \times f_{n-1}) \not\to e_n \\ \equiv & \qquad \text{Characterisation of left-reductions} \\ & f_n\,[\,] = e_n \ \wedge\ f_n\,(x \mathbin{-\!\!\!\prec} a) = (f_n\, x) \otimes_n (f_{n-1}\, a)\ , \end{array}$$

which proves the induction step. $\square$

An example of a hierarchical left-reduction, denoted by $rev_n : A\star_n \to A\star_n$, is described by Turner in [132]. Every catamorphism in this hierarchy reverses all levels of the array to which it is applied. The function reverse on *snoc-list* is the left-reduction $\oplus\not\to[\,]$, where operator $\oplus$ is defined by

$$x \oplus a \;=\; [a] \mathbin{+\!\!\!+\!\!<} x \;.$$

Define function rev_0 by $rev_0 = id$. the family of functions rev_n is defined as the following hierarchical left-reduction.

$$rev_n \;=\; (\oplus, \ldots, \oplus, id)\not\to_n([\,], \ldots, [\,]) \;.$$

The family of functions $me_n : A\star_n \to nat\star$ (me abbreviates 'measure') is another example of a hierarchical left-reduction. Function me_1 returns a list containing the length of its argument. Function $me_2 : A\star_2 \to nat\star$ returns, when applied to a proper two-dimensional array, a list containing two elements: the height and the width of the array. When applied to to a non-proper two-dimensional array it returns some arbitrary value. The components of the family of functions me_n return the measure of an n-dimensional array as a list. Define function me_0 by $me_0\, a = [\,]$. The family of functions me_n is defined by the equalities

$$\begin{aligned} me_n\,[\,] \quad &= \quad [0] \\ me_n\,(x \mathbin{-\!\!<} a) \quad &= \quad (me_{n-1}\, a) \mathbin{-\!\!<} ((lt\; me_n\, x) + 1) \;, \end{aligned}$$

for all m with $n \geq m \geq 1$. Hence, if we define for m with $n \geq m \geq 1$, $e_m = [0]$, and

$$x \otimes_m a \;=\; a \mathbin{-\!\!<} ((lt\, x) + 1) \;, \tag{6.9}$$

then, applying Hierarchical Left-reduction Characterisation, me_n is defined by

$$me_n \;=\; (\otimes_n, \ldots, \otimes_1, \underline{[\,]})\not\to_n(e_n, \ldots, e_1) \;. \tag{6.10}$$

For the definition of the family of functions $propar_n : A\star_n \to bool$ (for 'proper array') we have to develop some more theory. The mth component of the family of functions $propar_n$ determines, when applied to an element of $A\star_m$, whether or not the element is a proper m-dimensional array. An example of a component of this family of functions is the function $propar_2$, which, given a lists of lists, determines whether or not it is a list of equal-length lists. The family of functions $propar_n$ is defined by means of the family of functions me_n. The tuple of functions $propar_n \vartriangle me_n$ is a hierarchical left-reduction. To prove this fact, we use the following notions and theorem.

Mutumorphisms on arrays

We say that the family of functions f_n is *catamorphic modulo* the family of functions g_n if there exists a family of operators $\oplus_n$ such that for all m with $n \geq m \geq 1$

$$f_m\,(x \mathbin{+\!\!\!<} a) \;=\; (f_m\,x, g_m\,x) \oplus_m (f_{m-1}\,a, g_{m-1}\,a)\ .$$

If the family of functions g_n is catamorphic modulo the family of functions f_n too, we call the families of functions f_n and g_n mutumorphisms. For mutumorphisms f_n and g_n we prove the following theorem.

(6.11) Theorem (Array Mutumorphisms) *Suppose the families of functions f_n and g_n are catamorphic modulo each other, that is, there exist families of operators $\oplus_n$ and $\ominus_n$ such that for all m with $n \geq m \geq 1$*

$$\begin{aligned} f_m\,(x \mathbin{+\!\!\!<} a) &= (f_m\,x, g_m\,x) \oplus_m (f_{m-1}\,a, g_{m-1}\,a) \\ g_m\,(x \mathbin{+\!\!\!<} a) &= (f_m\,x, g_m\,x) \ominus_m (f_{m-1}\,a, g_{m-1}\,a)\ . \end{aligned}$$

Then $f_n \vartriangle g_n$ is a hierarchical left-reduction:

$$f_n \vartriangle g_n \;=\; (\odot_n, \ldots, \odot_1, f_0 \vartriangle g_0) \not\rightarrow_n (e_n, \ldots, e_1)\ ,$$

where for all m with $n \geq m \geq 1$, $e_m = (f_m\,[\,], g_m\,[\,])$, and operator $\odot_m$ is defined by

$$(x, y) \odot_m (a, b) \;=\; ((x, y) \oplus_m (a, b), (x, y) \ominus_m (a, b))\ .$$

Proof The proof of this theorem is a simple application of Hierarchical Left-reduction Characterisation. We have

$$f_n \vartriangle g_n \;=\; (\odot_n, \ldots, \odot_1, f_0 \vartriangle g_0) \not\rightarrow_n (e_n, \ldots, e_1)\ ,$$

if and only if for all m with $n \geq m \geq 1$

$$\begin{aligned} (f_m \vartriangle g_m)\,[\,] &= e_m \\ (f_m \vartriangle g_m)\,(x \mathbin{+\!\!\!<} a) &= (f_m \vartriangle g_m)\,x \odot_m (f_{m-1} \vartriangle g_{m-1})\,a\ . \end{aligned}$$

The first condition is satisfied by definition of the family of values e_n. For the second condition we have

$$\begin{aligned} & (f_m \vartriangle g_m)\,(x \mathbin{+\!\!\!<} a) \\ = \quad & \text{definition of } \vartriangle \\ & (f_m\,(x \mathbin{+\!\!\!<} a), g_m\,(x \mathbin{+\!\!\!<} a)) \\ = \quad & \text{conditions of the theorem} \end{aligned}$$

$$
\begin{array}{ll}
& ((f_m\,x, g_m\,x) \oplus_m (f_{m-1}\,a, g_{m-1}\,a), (f_m\,x, g_m\,x) \ominus_m (f_{m-1}\,a, g_{m-1}\,a)) \\
= & \quad \text{definition of } \odot_m \\
& (f_m\,x, g_m\,x) \odot_m (f_{m-1}\,a, g_{m-1}\,a) \\
= & \quad \text{definition of } \vartriangle \\
& (f_m \vartriangle g_m)\,x \odot_m (f_{m-1} \vartriangle g_{m-1})\,a\ .
\end{array}
$$

□

The families of functions $propar_n$ and me_n are mutumorphisms. For all m with $n \geq m \geq 1$ we have

$$me_m\,(x \mathrel{-\!\!\!<} a) \quad = \quad (me_m\,x, propar_m\,x) \oplus_m (me_{m-1}\,a, propar_{m-1}\,a)\ ,$$

where operator $\oplus_m$ is defined by

$$(x, y) \oplus_m (a, b) \quad = \quad x \otimes_m a\ ,$$

and

$$propar_m\,(x \mathrel{-\!\!\!<} a) \quad = \quad (me_m\,x, propar_m\,x) \ominus_m (me_{m-1}\,a, propar_{m-1}\,a)\ ,$$

where operator $\ominus_m$ is defined by

$$(x, y) \ominus_m (a, b) \quad = \quad \begin{cases} \mathit{false} & \text{if } y = \mathit{false} \ \vee\ b = \mathit{false} \ \vee\ itx \neq a \\ \mathit{true} & \text{otherwise}\ . \end{cases}$$

Applying Hierarchical Mutumorphisms we obtain

$$me_n \vartriangle propar_n \quad = \quad (\odot_n, \ldots, \odot_1, \underline{[\,]} \vartriangle \underline{true}) \nrightarrow_n (e_n, \ldots, e_1)\ ,$$

where operator $\odot$ is defined by

$$(x, y) \odot_m (a, b) \quad = \quad ((x, y) \oplus_m (a, b), (x, y) \ominus_m (a, b))\ .$$

Fusion on arrays

We have the following fusion theorem on the data type *array*.

(6.12) Theorem (Array Fusion) *Suppose that the family of functions f_n and the families of operators $\otimes_n$ and $\oplus_n$ satisfy for m with $n \geq m \geq 1$,*

$$f_m\,(x \otimes_m a) \quad = \quad (f_m\,x) \oplus_m (f_{m-1}\,a)\ ,$$

where

$$
\begin{array}{lcl}
x & = & ((\otimes_m, \ldots, \otimes_1, g_0) \nrightarrow_m (e_m, \ldots, e_1))\,y \\
a & = & ((\otimes_{m-1}, \ldots, \otimes_1, g_0) \nrightarrow_{m-1} (e_{m-1}, \ldots, e_1))\,b\ ,
\end{array}
$$

for some y and b. Then

$$f_n \cdot (\otimes_n, \ldots, \otimes_1, g_0) \not\rightarrow_n (e_n, \ldots, e_1) \quad = \quad (\oplus_n, \ldots, \oplus_1, f_0 \cdot g_0) \not\rightarrow_n (f_n\, e_n, \ldots, f_1\, e_1) \; .$$

Proof Again, the proof of this theorem is an application of Hierarchical Left-reduction Characterisation. We have

$$f_n \cdot (\otimes_n, \ldots, \otimes_1, g_0) \not\rightarrow_n (e_n, \ldots, e_1) \quad = \quad (\oplus_n, \ldots, \oplus_1, f_0 \cdot g_0) \not\rightarrow_n (f_n\, e_n, \ldots, f_1\, e_1) \; .$$

if and only if for all m with $n \geq m \geq 1$

$$\begin{array}{lcl} f_m\, h_m\, [\,] & = & f_m\, e_m \\ f_m\, h_m\, (y \rightarrow\!\!\!< b) & = & f_m\, h_m\, y \oplus_m f_{m-1}\, h_{m-1}\, b \; , \end{array}$$

where $h_m = (\otimes_m, \ldots, \otimes_1, g_0) \not\rightarrow_m (e_m, \ldots, e_1)$. For the first condition we have by definition of hierarchical left-reduction $f_m\, h_m\, [\,] = f_m\, e_m$. For the second condition we have

$$\begin{array}{ll} & f_m\, h_m\, (y \rightarrow\!\!\!< b) \\ = & \quad \text{definition of } h_m\text{, definition of hierarchical left-reduction} \\ & f_m\, (h_m\, y \otimes_m h_{m-1}\, b) \\ = & \quad \text{condition of the theorem} \\ & f_m\, h_m\, y \oplus_m f_{m-1}\, h_{m-1}\, b \; . \end{array}$$

□

Other definitions of arrays

Other proposals for the description of arrays have been given in the literature. We briefly discuss five of these.

Bird and Malcolm give definitions for the data type *two-dimensional array*. In [17], Bird gives a definition of the data type *two-dimensional array* over base type A (denoted by $A\mathsf{M}$). He defines the data type *two-dimensional array* by means of three constructors: the singleton constructor $\tau : A \to A\mathsf{M}$, the besides constructor $\phi : A\mathsf{M} \times A\mathsf{M} \to A\mathsf{M}$, and the above constructor $\ominus : A\mathsf{M} \times A\mathsf{M} \to A\mathsf{M}$. The expressions $x \ominus y$ (respectively $x \phi y$) are defined only if x and y have equal width (respectively height). The constructors $\ominus$ and ϕ are associative, and they satisfy the so-called abides-law:

$$(x \ominus y) \phi (u \ominus v) \quad = \quad (x \phi u) \ominus (y \phi v) \; ,$$

for x, y, u, and v of the correct form. Formally, define the monofunctor $\ddagger\underline{A}$ by

$$\ddagger\underline{A} \quad = \quad (\mathsf{I} \times \mathsf{I}) + (\mathsf{I} \times \mathsf{I}) + \underline{A} \; .$$

Then the data type *two-dimensional array* is defined as a kind of subtype of

$$(A\mathsf{M}, \ominus \triangledown \phi \triangledown \tau) \quad = \quad \mu(\ddagger\underline{A}) \ .$$

The subtype contains those elements of $A\mathsf{M}$ which are proper arrays: $x \, \phi \, y$ may only occur if x and y have equal height, $x \ominus y$ may only occur if x and y have equal width. However, there is still some partiality present in the abides-law, and it is difficult to get rid of this partiality (Jeffrey [69] discusses this topic).

In the definition of the data type *two-dimensional array* as given by Malcolm [88] there are infinitely many constructor rules. For each triple (i, j, k) of positive natural numbers we have a constructor $\ominus_{i\,j\,k}$ and a constructor $\phi_{i\,j\,k}$. For example, we have a constructor $\phi_{i\,j\,k} : A\mathsf{M}(i,j) \times A\mathsf{M}(i,k) \to A\mathsf{M}(i, j+k)$, which takes two arrays of respectively measure $[i,j]$ and $[i,k]$, and places them besides each other to obtain an array of measure $[i, j+k]$. Associativity of ϕ is translated into infinitely many laws like

$$(x \, \phi_{i\,j\,k} \, y) \, \phi_{i\,(j+k)\,l} \, z \quad = \quad x \, \phi_{i\,j\,(k+l)} \, (y \, \phi_{i\,k\,l} \, z) \ ,$$

and translating the abides law is even more complicated. Thus we obtain infinitely many data types $A\mathsf{M}(i,j)$. In the case of higher-dimensional arrays or even a hierarchy of data types, the number of subscripts rises accordingly.

Banger [11] defines the data type *array* as an initial algebra in the category of B_A-algebras, where functor B_A is defined by

$$\mathsf{B}_A \quad = \quad \underline{A^\infty} + \underline{\mathrm{nat}} \times \mathsf{I} \ ,$$

where the elements of A^∞ are infinite lists over base type A. It follows that an array is a list with as starting element an infinite list, followed by natural numbers. The natural numbers are used to describe the shape or measures of the array.

An approach to the data type *array* in which the data type array is not defined as an initial algebra, is described by Mullin in [107] and by Hains and Mullin in [62]. Define a list x to be *at most* a list y if x and y have equal length, and if for each position i in these lists, the element of x at position i is at most the element of y at position i. The definition of an array x consists of two parts: the measure $me \, x$ of the array, and a function which returns an array element when applied to a list of integers which is at most $me \, x$. The advantage of this approach is that some equalities which are difficult to prove in our setting can be proved easily. On the other hand, the form of some nontrivial functions becomes disagreeably complex in Mullin's setting.

Wise [137] represents n-dimensional arrays by 2^n-ary trees and describes array algorithms on trees.

The elements in the data types *array* given by Banger, Mullin and Wise are more representations of arrays. Bird's arrays seem to link up nicely with the intuition behind arrays, but there is a serious problem in defining it as an initial algebra. Because of the proliferating subscripts, Malcolm's arrays do not seem to be a good alternative.

Finally, note that if the data type *snoc-list* is replaced by the data type *join-list* in the definition of the data type *n-dimensional array*, we obtain a definition of the data type *n-dimensional array* based on the data type *join-list*. Thus we obtain constructors of the form $+\!\!\!+_m$, which concatenate two m-dimensional arrays.

6.2.2 Auxiliary functions

In this subsection we define some auxiliary functions, some of which reappear later in our derivations.

The function $\#_n$

The function $\#_n$ returns the number of elements in an n-dimensional array. It is defined as the following hierarchical left-reduction.

$$\#_n \;=\; (+,\ldots,+,\underline{1})\not\rightarrow_n(0,\ldots,0)\ .$$

One would expect the following equality to hold on the domain of proper arrays.

$$\#_n \;=\; \times/\cdot me_n\ . \tag{6.13}$$

Apply Array Fusion to obtain this equality.

$$\begin{array}{ll}
 & \times/\cdot me_n = \#_n \\
= & \quad \text{definition of } me_n \text{ and } \#_n \\
 & \times/\cdot(\otimes_n,\ldots,\otimes_1,\underline{[\,]})\not\rightarrow_n(e_n,\ldots,e_1) = (+,\ldots,+,\underline{1})\not\rightarrow_n(0,\ldots,0) \\
\Leftarrow & \quad \text{Array Fusion, } \times/\cdot\underline{[\,]} = \underline{1},\ \times/\,e_i = 0 \\
 & \times/\,(x\otimes_m a) = \times/\,x + \times/\,a\ ,
\end{array}$$

for $x = me_m\,y$, and $a = me_{m-1}\,b$, with, since $y -\!\!\!< b$ is a proper array, $it\; me_m\,y = me_{m-1}\,b$, so $it\,x = a$. The condition obtained from the above calculation is satisfied, since for all m with $n \geq m \geq 1$ we have

$$\begin{array}{ll}
 & \times/\,(x\otimes_m a) \\
= & \quad \text{definition of } \otimes_m \text{ (6.9)} \\
 & \times/\,(a -\!\!\!< (lt\,x+1)) \\
= & \quad \text{definition of reduction} \\
 & (\times/\,a)\times(lt\,x+1) \\
= & \quad \text{definition of } \times
\end{array}$$

$$
\begin{array}{ll}
 & (\times/\ a \times \mathit{lt}\,x) + (\times/\ a) \\
= & \quad a = \mathit{it}\,x \\
 & (\times/\ \mathit{it}\,x \times \mathit{lt}\,x) + (\times/\ a) \\
= & \quad \text{definition of reduction} \\
 & \times/\ x + \times/\ a\ .
\end{array}
$$

Transpose

A two-dimensional array can be *transposed* using the function $\mathit{tr} : A{\star}_2 \rightarrow A{\star}_2$. For example,

$$
\mathit{tr}\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} .
$$

Define

$$
\mathit{tr} \;=\; \oslash \not\rightarrow [\,]\ ,
$$

where operator $\oslash$ is defined by

$$
x \oslash a \;=\; \begin{cases} \tau\star\, a & \text{if } x = [\,] \\ x\ \Upsilon_{+\!\!<}\ a & \text{otherwise} \end{cases} .
$$

Transpose is its own identity.

Inproduct and multiplication

The *inproduct* of two vectors (lists or one-dimensional arrays) a and b with $\#\,a = \#\,b$, written $a \circ b$, is defined by

$$
a \circ b \;=\; +/\,(a\ \Upsilon_{\times}\ b)\ .
$$

The inproduct is used in the definition of *multiplication* on *two-dimensional array*. The multiplication of two-dimensional arrays x and y with $\mathit{lt}\ \mathit{me}_2\,x = \mathit{hd}\ \mathit{me}_2\,y$, written $x\ \mathsf{X}\ y$, is defined by

$$
\begin{array}{lcl}
(x\mathsf{X}) & = & \oslash_x \not\rightarrow [\,] \\
y \oslash_x a & = & y +\!\!\!+\!\!<\ (a\circ)\star\ \mathit{tr}\ x\ .
\end{array}
$$

It would be interesting (and likely very difficult) to derive Strassen's algorithm for matrix multiplication, see [128], from this specification.

The operators $\ominus$ and ϕ

The constructors $\ominus$ and ϕ used by Bird can be expressed in our terms as follows. Suppose x and y satisfy $x =_{hd \cdot me_2} y$, then

$$x \mathbin{\phi} y \quad = \quad x \mathbin{+\!\!\!\!+\!\!<} y\ .$$

The definition of $\ominus$ is somewhat more difficult, because the arrays we define are constructed from left to right. Suppose x and y satisfy $x =_{lt \cdot me_2} y$, then

$$x \ominus y \quad = \quad x \, \Upsilon_{+\!\!\!\!+\!\!<} \, y$$

A large number of other operators and functions defined on arrays can be found in Mullin [107].

6.3 Subarray problems

In this section we discuss subarray problems. Subarray problems can be viewed as the extension of segment problems to the data type *array*. We derive a hierarchical version of Horner's rule. This theorem could have been derived via a hierarchical version of the *segs*-Fusion Theorem, but since we do not give hierarchical versions of the Sliding Tails Theorem and the Hopping Tails Theorem we just give a derivation of Horner's rule itself. The hierarchical version of Horner's rule is applied to the maximum subarray sum problem, to obtain a hierarchy of efficient algorithms for this problem. Furthermore, we briefly discuss the conditions a family of predicates p_n has to satisfy in order to derive hierarchies of efficient algorithms for the problem of finding the largest subarray satisfying p_n of a given array (largest-p_n subarray problems).

6.3.1 Subarrays

In this section we give a definition of a hierarchical left-reduction which returns, among others, all subarrays of an array. The family of functions which returns just all subarrays is denoted by $suba_n$, where n is the level in the hierarchy: the dimension of the arrays on which the particular instance of *suba* is defined. For example, the function $suba_1$ returns all segments of its argument, so $suba_1$ might be defined as the function *segs*. The family of functions $suba_n$ can be defined elegantly using an auxiliary family of functions $tails_n$, just as function *segs* has been defined elegantly using function *tails*. Define the family of functions $suta_n$ by

$$suta_n \quad = \quad suba_n \vartriangle tails_n\ . \tag{6.14}$$

We enumerate subarrays in a snoc-list for reasons that will become clear later. Consider as an example two-dimensional arrays. Let x and y be the the following two-dimensional arrays.

$$x = \left(\begin{array}{ccc} 5 & 2 & 3 \\ 6 & 4 & 0 \\ 9 & 0 & 1 \end{array} \right) \quad , \quad y = \left(\begin{array}{cc} 5 & 2 \\ 6 & 4 \\ 9 & 0 \end{array} \right) .$$

Then $x = y \mathbin{+\!\!\!<} a$, where $a = [3, 0, 1]$. The two-dimensional subarrays of x consist of the two-dimensional subarrays of y, together with all subarrays of $y \mathbin{+\!\!\!<} a$ containing one or more contiguous elements from the one-dimensional array a. All contiguous elements from a are obtained by applying suba_1 to a. All subarrays of $y \mathbin{+\!\!\!<} a$ containing a contiguous part of column a are obtained by appending the elements of $\mathit{suba}_1\, a$ to the 'corresponding tails' (of the same height, and occurring at the same position) of y. The tails of y are obtained by means of the function tails_2. Informally, we define a subarray v of an array w to end in w if v, when drawn in two dimensions, occurs at the right end of w. The function tails_2 applied to an array w returns a list of lists, in which each list consists of subarrays of w, ending at the same position in w. One of the lists in $\mathit{tails}_2\, y$ is the list

$$[\left(\begin{array}{cc} 6 & 4 \\ 9 & 0 \end{array} \right), \left(\begin{array}{c} 4 \\ 0 \end{array} \right)] ,$$

and to obtain two-dimensional subarrays of x ending in x, both of these arrays should be appended with list $[0, 1]$. Any two arrays occurring in one of the lists of tails_2 are of equal height, but of different breadth. The lists in the list of lists are enumerated in the same order of height as the subarrays in suba_1. We define suba_n by

$$\begin{array}{lcl} \mathit{suba}_n & : & A\star_n \to A\star_n\star \\ \mathit{suba}_n\, [\,] & = & [[\,]] \\ \mathit{suba}_n\, (x \mathbin{+\!\!\!<} a) & = & (\mathit{suba}_n\, x) \mathbin{+\!\!\!+\!\!\!<} (\mathbin{+\!\!\!+\!\!\!<}/\ \mathit{tails}_n\, (x \mathbin{+\!\!\!<} a)) \ , \end{array} \tag{6.15}$$

and the function tails_n by

$$\begin{array}{lcl} \mathit{tails}_n & : & A\star_n \to A\star_n\star\star \\ \mathit{tails}_n\, [\,] & = & [\,] \\ \mathit{tails}_n\, (x \mathbin{+\!\!\!<} a) & = & (\mathit{tails}_n\, x) \oslash (\mathit{suba}_{n-1}\, a) \ , \end{array} \tag{6.16}$$

where operator $\oslash$ is defined by

$$\begin{array}{lcl} \oslash & : & A\star_n\star\star \times A\star_{n-1}\star \to A\star_n\star\star \\ x \oslash a & = & \left\{ \begin{array}{ll} (\tau \cdot \tau)\star\, a & \text{if } x = [\,] \\ x \, \Upsilon_{\odot}\, a & \text{otherwise} \ , \end{array} \right. \end{array} \tag{6.17}$$

where operator $\odot$ is defined by

$$\begin{array}{lcl} \odot & : & A\star_n\star \times A\star_{n-1} \to A\star_n\star \\ x \odot a & = & (\mathbin{+\!\!\!<} a)\star\, x \mathbin{+\!\!\!+\!\!\!<} [[a]] \ . \end{array} \tag{6.18}$$

Function $tails_1$ equals function $tails$ defined in equation (5.11) except for two things: function $tails_1$ doesn't return the empty tail, and there is a type difference: $tails_1$ enumerates tails in a list of lists instead of a set. From equations (6.15) and (6.16) it follows that functions $suba_n$ and $tails_n$ are mutumorphisms. Apply Array Mutumorphisms to obtain a hierarchical left-reduction for the tuple of functions $suba_n \vartriangle tails_n$. Define the families of operators $\oplus_n$ and $\ominus_n$ by

$$\begin{aligned}(x,y) \oplus_m (a,b) &= x \mathbin{+\!\!\!+\!\!\!<} \mathbin{+\!\!\!+\!\!\!<}/\,(y \oslash a)\\ (x,y) \ominus_m (a,b) &= y \oslash a\ ,\end{aligned}$$

for all m with $n \geq m \geq 1$. Then

$$\begin{aligned}suba_m\,(x \mathbin{-\!\!\!<} a) &= (suba_m\,x, tails_m\,x) \oplus_m (suba_{m-1}\,a, tails_{m-1}\,a)\\ tails_m\,(x \mathbin{-\!\!\!<} a) &= (suba_m\,x, tails_m\,x) \ominus_m (suba_{m-1}\,a, tails_{m-1}\,a)\ ,\end{aligned}$$

and hence we have

$$\begin{aligned}suta_n &: A\star_n \rightarrow A\star_n\star \times A\star_n\star\star\\ suta_n &= (\otimes_n, \ldots, \otimes_1, suta_0) \not\rightarrow_n (e_n, \ldots, e_1)\ ,\end{aligned} \tag{6.19}$$

where the family of values e_n is defined by $e_m = ([[\,]],[\,])$, and the family of operators $\otimes_n$ is defined by

$$\begin{aligned}\otimes_m &: (A\star_m\star \times A\star_m\star\star) \times (A\star_{m-1}\star \times A\star_{m-1}\star\star) \rightarrow A\star_m\star \times A\star_m\star\star\\ (x,y) \otimes_m (a,b) &= ((x,y) \oplus_m (a,b), (x,y) \ominus_m (a,b))\ ,\end{aligned} \tag{6.20}$$

for all m with $n \geq m \geq 1$. Define $suta_0 : A \rightarrow A\star \times A\star\star$ by $\tau \vartriangle (\tau \cdot \tau)$. The function $suta_n$ cannot be implemented directly in a strongly typed language such as Miranda: the type of the function $suta_n$ depends on its subscript. Nevertheless, for each specific n, function $suta_n$ is easily translated into a Miranda program.

6.3.2 A hierarchical version of Horner's rule

Subarray problems

Given a family of functions f_n of type $A\star_n \rightarrow B_n$ and a family of operators $\oplus_n$ of type $B_n \times B_n \rightarrow B_n$, we consider the problem

$$h_n = \oplus_n/ \cdot f_n\star \cdot suba_n\ . \tag{6.21}$$

Function h_n is called a *subarray problem.* A lot of interesting problems can be specified as the composition of a catamorphism and the function $suba_n$; examples are the pattern-matching problem and the maximum subarray sum problem. Our goal is to derive a hierarchical left-reduction for this problem. The derivation is very similar to the derivations of the *segs*-Fusion Theorem and Horner's rule in Section 5.2. In the derivation we calculate a hierarchical left-reduction for the specification tupled with some extra functions. The definition of these extra functions is prescribed by the form of the expressions that appear in the calculation.

Deriving a hierarchical left-reduction for a subarray problem

A straightforward implementation of the family of functions h_n as defined in equation (6.21), given a family of operators $\oplus_n$ and a family of functions f_n, is an inefficient program. We want to derive an algorithm that can be implemented as an efficient program for h_n. Since the tuple of functions $suba_n \vartriangle tails_n$ can be defined elegantly as a hierarchical left-reduction, we want to tuple function $suba_n$ with function $tails_n$ in the specification, and then we try to apply Array Fusion. Using equation 2.14 we derive for arbitrary family of functions g_n

$$
\begin{array}{ll}
 & \oplus_n/ \cdot f_n\star \cdot suba_n \\
= & \qquad \text{definition of } suta_n \\
 & \oplus_n/ \cdot f_n\star \cdot exl \cdot suta_n \\
= & \qquad \text{equation 2.14} \\
 & exl \cdot (\oplus_n/ \cdot f_n\star) \times g_n \cdot suta_n \ .
\end{array}
$$

For g_n we can choose any family of functions that suits us, and in the subsequent derivation a natural candidate will emerge, just as in the derivation of the *segs*-Fusion Theorem. Define

$$j_n \ = \ (\oplus_n/ \cdot f_n\star) \times g_n \ .$$

Function $suta_n$ is a hierarchical left-reduction, so we can apply Array Fusion to the expression $j_m \cdot suta_n$. Array Fusion gives

$$j_n \cdot suta_n \ = \ (\ominus_n, \ldots, \ominus_1, t) \not\rightarrow_n (e_n, \ldots, e_1) \ ,$$

for $t = j_0 \cdot suta_0$, and for some family of values e_n and some family of operators $\ominus_n$, provided

$$e_m \ = \ j_m\,([[\,]], [\,]) \ = \ (f_m\,[\,], g_m\,[\,]) \tag{6.22}$$

$$j_m\,((x, y) \otimes_m (a, b)) \ = \ (j_m\,(x, y)) \ominus_m (j_{m-1}\,(a, b)) \ , \tag{6.23}$$

for all m with $n \geq m \geq 1$. It follows that l is defined by $f_0 \vartriangle (g_0 \cdot \tau \cdot \tau)$.

The second condition can be satisfied provided we tuple with some auxiliary functions. These functions will appear in the derivation. An operator $\ominus_m$ satisfying condition (6.23) is synthesised as follows.

$$
\begin{array}{ll}
 & j_m\,((x, y) \otimes_m (a, b)) \\
= & \qquad \text{definition of } \otimes_m \text{ (6.20)}, \ z = y \oslash a \\
 & j_m\,(x \mathbin{+\!\!\!+\!\!<} (+\!\!\!+\!\!</\, z), z) \\
= & \qquad \text{definition of } j_m \\
 & ((\oplus_m/ \cdot f_m\star) \times g_m)\,(x \mathbin{+\!\!\!+\!\!<} (+\!\!\!+\!\!</\, z), z) \\
= & \qquad \text{map and reduction on } \textit{snoc-list}
\end{array}
$$

$$
\begin{array}{ll}
 & ((\oplus_m/\, f_m\star\, x) \oplus_m (\oplus_m/\, f_m\star \cdot +\!\!\!\!\!\!\nwarrow/\, z), g_m\, z) \\
= & \quad \text{Fusion on } \textit{snoc-list} \\
 & ((\oplus_m/\, f_m\star\, x) \oplus_m (\oplus_m/(\oplus_m/\cdot f_m\star)\star\, z), g_m\, z)\ .
\end{array}
$$

In the last expression of this calculation we can distinguish three subexpressions:

$$
\begin{array}{l}
\oplus_m/\, f_m\star\, x \\
\oplus_m/\, (\oplus_m/\cdot f_m\star)\star\, z \\
g_m\, z\ ,
\end{array}
$$

where $z = y \oslash a$. In view of the desired expression the first subexpression $\oplus_m/\, f_m\star\, x$ need not be developed any further. The subexpressions $\oplus_m/(\oplus_m/\cdot f_m\star)\star\, z$ and $g_m\, z$, where $z = y \oslash a$, have to be expressed in terms of $g_m\, y$ and $\oplus_{m-1}/\, f_{m-1}\star\, a$. Hence a reasonable choice for the family of functions g_n, also suggested by the *segs*-Fusion Theorem, taking into account the type difference, seems to be

$$
g_n \;=\; (\oplus_n/\cdot f_n\star)\star\ .
$$

Note that with this definition of g_n,

$$
\begin{array}{lcl}
t & = & f_0 \vartriangle (g_0 \cdot \tau \cdot \tau) \;=\; f_0 \vartriangle (\tau \cdot f_0) \\
e_m & = & (f_m\, [\,], [\,])\ ,
\end{array}
$$

for all m with $n \geq m \geq 1$. We have

$$
\begin{array}{lcl}
j_m\, ((x, y) \otimes_m (a, b)) & = & (s \oplus_m (\oplus_m/\, w), w) \\
 & & \quad \textbf{where } (s, t) = j_m\, (x, y) \\
 & & \qquad\quad w = (\oplus_m/\cdot f_m\star)\star\, (y \oslash a)
\end{array}
\tag{6.24}
$$

The remaining task is now to express $(\oplus_m/\cdot f_m\star)\star\, (y \oslash a)$ in terms of $(\oplus_m/\cdot f_m\star)\star\, y$ and $\oplus_{m-1}/\, f_{m-1}\star\, a$. Distinguish the two cases in the definition of operator $\oslash$: $y = [\,]$ and $y \neq [\,]$. This case distinction is equivalent to the case distinction $(\oplus_m/\cdot f_m\star)\star\, y = [\,]$ and $(\oplus_m/\cdot f_m\star)\star\, y \neq [\,]$.

- If $y = [\,]$, then $y \oslash a = (\tau \cdot \tau)\star\, a$, and hence

$$
\begin{array}{ll}
 & (\oplus_m/\cdot f_m\star)\star\, (y \oslash a) \\
= & \quad \text{case assumption} \\
 & (\oplus_m/\cdot f_m\star)\star\, (\tau \cdot \tau)\star\, a \\
= & \quad \text{map-distributivity} \\
 & (\oplus_m/\cdot f_m\star \cdot \tau \cdot \tau)\star\, a \\
= & \quad \text{map and reduction on } \textit{snoc-list} \\
 & (f_m \cdot \tau)\star\, a \\
= & \quad \textbf{assumption} \text{ (satisfied if } f \text{ is a hierarchical left-reduction)} \\
 & (\phi_m \cdot f_{m-1})\star\, a\ .
\end{array}
$$

The assumption used to obtain the last expression, $f_m \cdot \tau = \phi_m \cdot f_{m-1}$, is satisfied if f_n is a hierarchical left-reduction $(\circledequal_n, \dots, \circledequal_1, f_0) \nrightarrow_n (u_n, \dots, u_1)$, since then $f_m\, \tau\, a = u_m \circledequal_m f_{m-1}\, a$, so $\phi_m = (u_m \circledequal_m)$.

- If $y \neq [\,]$ we have, applying the two Zip Laws (3.34) and (3.35)

$$
\begin{array}{ll}
 & (\oplus_m/ \cdot f_m\star)\star\,(y \oslash a) \\
= & \quad \text{case assumption} \\
 & (\oplus_m/ \cdot f_m\star)\star\,(y\; \Upsilon_{\odot}\; a) \\
= & \quad \text{Zip law (3.34)} \\
 & y\; \Upsilon_{\oplus_m/\cdot f_m\star\cdot\odot}\; a \\
= & \quad \text{equation (6.25) below} \\
 & y\; \Upsilon_{\obar_m\cdot(\oplus_m/\cdot f_m\star)\times f_{m-1}}\; a \\
= & \quad \text{Zip Law (3.35)} \\
 & (\oplus_m/ \cdot f_m\star)\star\, y\; \Upsilon_{\obar_m}\; f_{m-1}\star\, a\ .
\end{array}
$$

Equation (6.25) applied in this calculation reads

$$
\oplus_m/ \cdot f_m\star \cdot \odot \quad = \quad \obar_m \cdot (\oplus_m/ \cdot f_m\star) \times f_{m-1}\ . \tag{6.25}
$$

This equation is derived by means of the following calculation.

$$
\begin{array}{ll}
 & \oplus_m/\, f_m\star\,(x \odot a) \\
= & \quad \text{definition of } \odot \\
 & \oplus_m/\, f_m\star\,((\mathbin{-\!\!\!\prec} a)\star\, x \mathbin{+\!\!\!+\!\!\!\prec} [[a]]) \\
= & \quad \text{definition of map and reduction on } \textit{snoc-list} \\
 & \oplus_m/\, f_m\star\,(\mathbin{-\!\!\!\prec} a)\star\, x \oplus_m f_m\,[a] \\
= & \quad \text{Corollary 2.65, } \textbf{assumption} \text{ (see below)} \\
 & (\circledequal_m f_{m-1}\, a) \oplus_m/\, f_m\star\, x \oplus_m \phi_m\, f_{m-1}\, a \\
= & \quad \text{introduction of } \obar_m \\
 & \oplus_m/\, f_m\star\, x\; \obar_m\; f_{m-1}\, a\ ,
\end{array}
$$

where operator $\obar_m$ is defined by

$$
x \obar_m a \quad = \quad (x \circledequal_m a) \oplus_m \phi_m\, a\ . \tag{6.26}
$$

The condition of Corollary 2.65 is satisfied if we assume that the family of functions f_n is a hierarchical left-reduction $(\circledequal_n, \dots, \circledequal_1, f_0) \nrightarrow_n (u_n, \dots, u_1)$ such that section $(\circledequal_m c)$ is $(\oplus_m, \oplus_m)$-fusable for all m with $n \geq m \geq 1$, since then $\oplus_m/ \cdot f_m\star \cdot (\mathbin{-\!\!\!\prec} a)\star = (\circledequal f_{m-1}\, a) \cdot \oplus_m/ \cdot f_m\star$.

Using this case distinction, equation (6.24) is transformed into the following equation.

$$j_m\,((x,y)\otimes_m(a,b)) \;=\; (s\oplus_m(\oplus_m/w),w) \qquad (6.27)$$
$$\textbf{where } (s,t)=j_m\,(x,y)$$
$$w=\begin{cases}\phi_m\star f_{m-1}\star\,a & \text{if } t=[\,]\\ t\,\Upsilon_{\oplus_m}\,f_{m-1}\star\,a & \text{otherwise}\end{cases}\;.$$

Expression $f_{m-1}\star\,a$ cannot be extracted from $j_m\,(x,y)$ or $j_{m-1}\,(a,b)$. It follows that we have not succeeded in finding an operator $\ominus_m$ satisfying equation (6.23).

An extended specification

From expression (6.27) it follows that we have not succeeded in finding an operator $\ominus_m$ satisfying equation (6.23), but it also follows that $j_n\cdot suta_n$ is catamorphic modulo $k_n\cdot suta_n$, where the family of functions k_n is defined by

$$k_n \;=\; f_n\star\times f_n\star\star\;,$$

that is, if we define for all m with $n\geq m\geq 1$ operator $\ominus_m$ by

$$((s,t),(x,y))\ominus_m((a,b),(c,d)) \;=\; (s\oplus_m(\oplus_m/\,w),w)$$
$$\textbf{where } w=\begin{cases}\phi_m\star\,c & \text{if } y=[\,]\\ t\,\Upsilon_{\oplus_m}\,c & \text{otherwise}\end{cases}\;,$$

then for all m with $n\geq m\geq 1$

$$j_m\,suta_m\,(x \mathbin{-\!\!\!\!<} a) \;=\; (j_m\,suta_m\,x, k_m\,suta_m\,x)\ominus_m(j_{m-1}\,suta_{m-1}\,a, k_{m-1}\,suta_{m-1}\,a)\;.$$

To apply Array Mutumorphisms, Theorem 6.11, to obtain a hierarchical left-reduction for the tuple of functions

$$(j_n\cdot suta_n)\vartriangle(k_n\cdot suta_n)\;, \qquad (6.28)$$

it remains to show that function $k_n\cdot suta_n$ is catamorphic modulo $j_n\cdot suta_n$. In fact, function $k_n\cdot suta_n$ is a hierarchical left-reduction, and therefore catamorphic modulo $j_n\cdot suta_n$, see equation (2.48). Note that for the original specification of a subarray problem we have

$$\begin{array}{ll} & \oplus_n/\cdot f_n\star\cdot suba_n\\ = & \quad\text{definition of } j_n \text{ and } suta_n\\ & exl\cdot j_n\cdot suta_n\\ = & \quad\text{equation (2.16)}\\ & exl\cdot exl\cdot(j_n\cdot suta_n)\vartriangle(k_n\cdot suta_n)\;,\end{array}$$

so the tuple of functions given in equation (6.28) is an extension of the original specification. We prove as follows that the family of functions $k_n \cdot suta_n$ equals a hierarchical left-reduction. Since $suta_n$ is a hierarchical left-reduction, Array Fusion gives

$$k_n \cdot suta_n \;=\; (\oslash_n, \ldots, \oslash_1, l) \not\to_n (u_n, \ldots, u_1) \ ,$$

for $l = k_0 \cdot suta_0$, and for some family of values u_n and some family of operators $\oslash_n$ provided

$$\begin{array}{lcl} u_m & = & k_m\,([[\,]], [\,]) \\ k_m \cdot \otimes_m & = & \oslash_m \cdot k_m \times k_{m-1} \ , \end{array} \tag{6.29}$$

for all m with $n \geq m \geq 1$. It follows that $u_m = ([f_m\,[\,]], [\,])$, and that $l = suta_0 \cdot f_0$. A family of operators satisfying equation (6.29) is synthesised as follows.

$$\begin{array}{ll} & k_m\,((x,y) \otimes_m (a,b)) \\ = & \quad \text{definition of } \otimes_m \text{ (6.20)}, z = y \oslash a \\ & k_m\,(x \mathbin{+\!\!\!\!+\!\!\!\!\!<} (\mathbin{+\!\!\!\!+\!\!\!\!\!<}/\ z), z) \\ = & \quad \text{definition of } k_n \text{ and } \times \\ & (f_m\star\,(x \mathbin{+\!\!\!\!+\!\!\!\!\!<} (\mathbin{+\!\!\!\!+\!\!\!\!\!<}/\ z)), f_m\star\star\ z) \\ = & \quad \text{map and Fusion on } \textit{snoc-list} \\ & (f_m\star\,x \mathbin{+\!\!\!\!+\!\!\!\!\!<} \mathbin{+\!\!\!\!+\!\!\!\!\!<}/\ f_m\star\star\ z, f_m\star\star\ z) \ . \end{array}$$

In this last expression subexpression $f_m\star\star\ z$, where $z = y \oslash a$, needs to be developed further. Distinguish the two cases in the definition of operator $\oslash$: $y = [\,]$ and $y \neq [\,]$.

- If $y = [\,]$, then $y \oslash a = (\tau \cdot \tau)\star\ a$, and hence

$$\begin{array}{ll} & f_m\star\star\,(y \oslash a) \\ = & \quad \text{case assumption} \\ & f_m\star\star\,(\tau \cdot \tau)\star\ a \\ = & \quad \text{map-distributivity} \\ & (f_m\star \cdot \tau \cdot \tau)\star\ a \\ = & \quad \text{definition of map on } \textit{snoc-list} \\ & (\tau \cdot f_m \cdot \tau)\star\ a \\ = & \quad \textbf{assumption} \text{ on } f \\ & (\tau \cdot \phi_m \cdot f_{m-1})\star\ a \ . \end{array}$$

- If $y \neq [\,]$, then

$$
\begin{array}{ll}
 & f_m\star\star\,(y \oslash a) \\
= & \quad \text{definition of } \oslash \\
 & f_m\star\star\,(y \mathbin{\Upsilon_{\odot}} a) \\
= & \quad \text{Zip Law (3.34)} \\
 & y \mathbin{\Upsilon_{f_m\star\cdot\odot}} a \\
= & \quad \text{equation (6.30) below} \\
 & y \mathbin{\Upsilon_{\oparallel_m\cdot f_m\star\times f_m}} a \\
= & \quad \text{Zip Law (3.35)} \\
 & f_m\star\star\, y \mathbin{\Upsilon_{\oparallel_m}} f_m\star\, a \ .
\end{array}
$$

Equation (6.30) applied in this calculation reads

$$
f_m\star\cdot\odot \quad = \quad \oparallel_m \cdot f_m\star \times f_m \ . \tag{6.30}
$$

This equation is derived by means of the following calculation.

$$
\begin{array}{ll}
 & f_m\star\,(x \odot a) \\
= & \quad \text{definition of } \odot \\
 & f_m\star\,((\mathbin{+\!\!\prec} a)\star\, x \mathbin{+\!\!\!+\!\!\prec} [[a]]) \\
= & \quad \text{definition of map, map-distributivity} \\
 & (f_m \cdot (\mathbin{+\!\!\prec} a))\star\, x \mathbin{+\!\!\!+\!\!\prec} [f_m\,[a]] \\
= & \quad \textbf{assumption} \text{ on } f \\
 & ((\circledequal_m f_{m-1}\, a)\cdot f_m)\star\, x \mathbin{+\!\!\!+\!\!\prec} [\phi_m\, f_{m-1}\, a] \\
= & \quad \text{introduction of } \oparallel_m \\
 & f_m\star\, x \oparallel_m f_{m-1}\, a \ ,
\end{array}
$$

where operator $\oparallel_m$ is defined by

$$
x \oparallel_m a \quad = \quad (\circledequal_m a)\star\, x \mathbin{+\!\!\!+\!\!\prec} [\phi_m\, a] \ .
$$

Using this case distinction define, for all m with $n \geq m \geq 1$, operator $\oslash_m$ by

$$
\begin{array}{lll}
(x, y) \oslash_m (a, b) & = & (x \mathbin{+\!\!\!+\!\!\prec} \mathbin{+\!\!\!+\!\!\prec}/\, w, w) \\
 & & \quad \textbf{where } w = \begin{cases} (\tau\cdot\phi_m)\star\, a & \text{if } y = [\,] \\ y \mathbin{\Upsilon_{\oparallel_m}} a & \text{otherwise} \end{cases} .
\end{array}
$$

We have shown that functions $j_n \cdot suta_n$ and $k_n \cdot suta_n$ are mutumorphisms. Applying Theorem 6.11, we obtain the following theorem, in which we use the abbreviation $exl \cdot exl = exl^2$.

(6.31) Theorem (Hierarchical Horner's rule) *Suppose f_n is a hierarchical left-reduction*

$$f_n \;=\; (\circledequal_n, \ldots, \circledequal_1, f_0) \nrightarrow_n (u_n, \ldots, u_1)\ ,$$

such that $(\circledequal_m b)$ is $(\oplus_m, \oplus_m)$-fusable for all m with $n \geq m \geq 1$ and b. Then

$$\oplus_n/ \cdot f_n\star \cdot suba_n \;=\; exl^2 \cdot (\ominus_n, \ldots, \ominus_1, t) \nrightarrow_n (e_n, \ldots, e_1)\ ,$$

where t is the function $(id \vartriangle \tau) \vartriangle suta_0 \cdot f_0$, and where the family of values e_n is defined by $e_m = ((u_m, [\,]), ([u_m], [\,]))$ for all m with $n \geq m \geq 1$, and the family of operators $\ominus_n$ is defined by

$$\begin{array}{rcl} ((x,y),(z,w)) \ominus_m ((i,j),(k,l)) & = & ((x \oplus_m (\oplus_m/\, r), r), (z \mathbin{+\!\!+\!\!<} (\mathbin{+\!\!+\!\!<}/\, s), s)) \\ & & \textbf{where}\quad \begin{array}{rcl} r & = & \begin{cases} \phi_m\star k & \text{if } y = [\,] \\ y\, \Upsilon_{\obar_m}\, k & \text{otherwise} \end{cases} \\ s & = & \begin{cases} (\tau \cdot \phi_m)\star k & \text{if } y = [\,] \\ w\, \Upsilon_{\Obar_m}\, k & \text{otherwise} \end{cases} \end{array} \end{array} ,$$

where $\phi_m = (u_m \circledequal_m)$, and where the families of operators $\obar_n$ and $\Obar_n$ are defined by

$$\begin{array}{rcl} x \obar_m a & = & (x \circledequal_m a) \oplus_m \phi_m\, a \\ x \Obar_m a & = & ((\circledequal_m a)\star x) \mathbin{+\!\!+\!\!<} [\phi_m\, a]\ , \end{array}$$

for all m with $n \geq m \geq 1$.

6.3.3 The maximum subarray sum

The maximum subarray sum problem requires finding a subarray with maximal sum of a given array. In this section we give a hierarchical left-reduction for finding the maximum subarray sum of an array, by means of the hierarchical version of Horner's rule derived in the previous subsection. Instantiated with one-dimensional arrays or snoc-lists, we obtain the well-known linear-time maximum segment sum algorithm described in Section 5.2. The equivalent of the $O(n^{\frac{3}{2}})$ algorithm we obtain for the data type *two-dimensional array* (n is the number of elements in the input array) with 'max' replaced by 'min' is described by Dijkstra [36]. The algorithm given by Smith [125] for the same problem is a catamorphism on the Bird/Malcolm data type two-dimensional array with the same time-complexity. I am not aware of other work on maximum subarray sum algorithms on arrays of higher dimensions. For three-dimensional arrays we obtain an $O(n^{\frac{5}{3}})$ algorithm, and, in general, for m-dimensional arrays we obtain an $O(n^{\frac{2m-1}{m}})$ algorithm.

The specification of the maximum subarray sum problem reads as follows. Define the hierarchical left-reduction sum_n returning the sum of an array by

$$sum_n \;=\; (+, \ldots, +, id) \nrightarrow_n (0, \ldots, 0)\ .$$

The maximum subarray sum problem mss_n is specified by

$$mss_n \;=\; \uparrow/ \cdot sum_n\star \cdot suba_n \;.$$

Since $(+b)$ is $(\uparrow, \uparrow)$-fusable for all b apply Hierarchical Horner's rule to obtain a hierarchical left-reduction for mss_n. We have

$$mss_n \;=\; exl^2 \cdot (\ominus_n, \ldots, \ominus_1, g) \not\rightarrow_n (e_n, \ldots e_1) \;,$$

where g is the function $(id \vartriangle \tau) \vartriangle suta_0$, and where the family of values e_n is defined by $e_m = ((0, [\,]), ([0], [\,]))$ for all m with $n \geq m \geq 1$, and the family of operators $\ominus_n$ is defined by

$$((x,y),(z,w)) \ominus_m ((i,j),(k,l)) \;=\; ((x \uparrow (\uparrow/\, r), r), (z \mathbin{+\!\!\!+\!\!\!<} (\mathbin{+\!\!\!+\!\!\!<}/\, s), s))$$
$$\textbf{where} \quad \begin{array}{rcl} r & = & \begin{cases} k & \text{if } y = [\,] \\ y \,\Upsilon_{\obar_m}\, k & \text{otherwise} \end{cases} \\ s & = & \begin{cases} \tau\star k & \text{if } y = [\,] \\ w \,\Upsilon_{\Obar_m}\, k & \text{otherwise} \end{cases} \end{array} ,$$

where the families of operators $\obar_n$ and $\Obar_n$ are defined by

$$\begin{aligned} x \obar_m a &= (x + a) \uparrow a \\ x \Obar_m a &= ((+a)\star x) \mathbin{+\!\!\!+\!\!\!<} [a] \;. \end{aligned}$$

for all m with $n \geq m \geq 1$.

Let us now discuss the complexity of the programs obtained by implementing the elements in this hierarchy of algorithms. We first discuss the *one-dimensional array* or *snoc-list* algorithm. Note that for the *one-dimensional array* algorithm $exl^2 \cdot (\ominus_1, h) \not\rightarrow_1 e_1$ we do not need the last component of the pair returned by $(\ominus_1, h) \not\rightarrow_1 e_1$, and therefore we may omit this component (it is used in the higher-dimensional algorithms). We have obtained the well-known linear-time maximum segment sum algorithm that has been derived in Section 5.2.

For the *two-dimensional array* algorithm $exl^2 \cdot (\ominus_2, \ominus_1, h) \not\rightarrow_2 (e_2, e_1)$ note that again the last component of $(\ominus_2, \ominus_1, h) \not\rightarrow_2 (e_2, e_1)$ is only used in the higher-dimensional algorithms (this is a property shared by all array algorithms in the hierarchy). Let T_2 be the function which, given the measure $[h, w]$ of a two-dimensional array x (w is the width of x, h is the height of x), returns the time spent by the program corresponding to the algorithm $exl^2 \cdot (\ominus_2, \ominus_1, h) \not\rightarrow_2 (e_2, e_1)$ on x. T_2 satisfies the following recurrence equation

$$T_2\,(h, w) \;=\; T_2\,(h, w{-}1) + ch^2 \;,$$

where c is a constant. The function $T_2\,(h, w) = cw \times h^2$ is a solution of this equation. If we suppose that the input array is a square array of size m, we have $T_2\,(\sqrt{m}, \sqrt{m}) = O(m^{\frac{3}{2}})$. The algorithm we have derived is the same as the algorithm described by Dijkstra [36] for finding the minimum subarray sum of an array. Smith [125] gives an algorithm for the maximum segment sum problem of the same time complexity on a Bird/Malcolm-like data type *two-dimensional arrays*.

Suppose now T_n is the function which, given the measure $[s_1, \ldots, s_n]$ of an n-dimensional array x, returns the time spent by the program corresponding to the algorithm $exl^2 \cdot (\ominus_n, \ldots, \ominus_1, h) \not\to_n (e_n, \ldots, e_1)$ on x. T_n satisfies the following recurrence equation.

$$T_n\,(s_1, \ldots, s_n) \;=\; T_n\,(s_1, \ldots, s_{n-1}, s_n - 1) + c(s_1 \times \ldots \times s_{n-1})^2 \;,$$

which has as a solution $T_n\,(s_1, \ldots, s_n) = cs_n \times (s_1 \times \ldots \times s_{n-1})^2$. If we suppose that the input array is an n-cube of size m, we get $T_n\,(m^{\frac{1}{n}}, \ldots, m^{\frac{1}{n}}) = O(m^{\frac{2n-1}{n}}) < O(m^2)$.

6.3.4 Largest-p_n subarray problems

In this section we consider a subclass of the class of subarray problems specified in (6.21). Suppose p_n is a family of predicates defined on arrays. Given an array, we want to find a largest subarray satisfying p_n. This problem can be specified by

$$h_n \;=\; \uparrow_{\#_n}/ \cdot p_n\triangleleft \cdot suba_n \;,$$

and such a problem is called a *largest-p_n subarray* problem.

We claim that several results very similar to the results derived for longest-p segment problems in Chapter 5 can be derived for this class of problems. However, none of the calculations is given, and this section lacks the formality and rigorousness of the previous sections.

It is possible to derive a hierarchical version of the *segs*-Fusion Theorem, Theorem 5.18, and we name this theorem the *suba*-Fusion Theorem. The *suba*-Fusion Theorem is obtained if we do not apply Corollary 2.65 in the derivation of Hierarchical Horner's rule, but instead let the desired equalities be the conditions of the theorem. Just as Horner's rule, the Sliding Tails Theorem and the Hopping Tails Theorem are corollaries of the *segs*-Fusion Theorem, we can derive hierarchical versions of these three results as corollaries of the *suba*-Fusion Theorem.

The hierarchical versions of the Hopping Tails Theorem and the Sliding Tails Theorem state that if the family of predicates p_n is prefixen-closed (defined below), then there exists a hierarchical left-reduction, that can be implemented as a hierarchy of efficient programs, for the largest-p_n subarray problem. The definition of prefixen-closed families of predicates is a generalisation of the notion prefix-closed defined in equation (2.82) to arrays. A family of predicates p_n is *prefixen-closed* if for all m with $n \geq m \geq 1$, m-dimensional arrays x and $(m-1)$-dimensional arrays a,

$$\begin{array}{lcl} p_m\,[\,] & = & true \\ p_m\,(x \mathbin{+\!\!\!<} a) & \Rightarrow & (p_m\,x) \wedge (p_{m-1}\,a)\;. \end{array} \tag{6.32}$$

An example of a prefixen-closed family of predicates is the family of predicates which expresses that an array is ascending in all dimensions.

Just as $p?_\otimes$ is a left-reduction if predicate p is prefix-closed, see Theorem 2.84, the family of functions $p_n?_{\otimes_n}$ is a hierarchical left-reduction if the family of functions p_n is prefixen-closed. Let $\omega_n = \nu_{\otimes_n}$.

(6.33) Theorem *Suppose the family of predicates p_n is prefixen-closed. Then*

$$p_n?_{\otimes_n} \;=\; (\ominus_n, \ldots, \ominus_1, p_0?_{\otimes_0}) \not\to_n ([\,], \ldots, [\,])\ ,$$

where the family of operators $\ominus_n$ is defined by

$$x \ominus_m a \;=\; \begin{cases} x \mathbin{+\!\!<} a & \text{if } x \neq \omega_m \ \wedge\ a \neq \omega_{m-1} \ \wedge\ p_m\,(x \mathbin{+\!\!<} a) \\ \omega_m & \text{otherwise}\ , \end{cases}$$

for all m with $n \geq m \geq 1$.

Proof Using Hierarchical Left-reduction Characterisation we have

$$p_n?_{\otimes_n} \;=\; (\ominus_n, \ldots, \ominus_1, p_0?_{\otimes_0}) \not\to_n ([\,], \ldots, [\,])\ ,$$

if and only if for all m ith $n \geq m \geq 1$,

$$\begin{aligned} p_m?_{\otimes_m}\,[\,] &= [\,] \\ p_m?_{\otimes_m}\,(x \mathbin{+\!\!<} a) &= (p_m?_{\otimes_m}\,x) \ominus_m (p_{m-1}?_{\otimes_{m-1}}\,a)\ . \end{aligned}$$

Both of these conditions follow from the fact that the family of predicates p_n is prefixen-closed. The first condition is an immediate consequence, and for the second condition we reason as follows. If $p_m\,(x \mathbin{+\!\!<} a)$ holds, then $p_m?_{\otimes_m}\,(x \mathbin{+\!\!<} a) = x \mathbin{+\!\!<} a$, and, since $p_m\,(x \mathbin{+\!\!<} a) \Rightarrow p_m\,x \ \wedge\ p_{m-1}\,a$,

$$\begin{aligned} & (p_m?_{\otimes_m}\,x) \ominus_m (p_{m-1}?_{\otimes_{m-1}}\,a) \\ = \quad & \quad p_m\,x \ \wedge\ p_{m-1}\,a \\ & x \ominus_m a \\ = \quad & \quad \text{definition of } \ominus_m \\ & x \mathbin{+\!\!<} a\ . \end{aligned}$$

If $\neg p_m\,(x \mathbin{+\!\!<} a)$ holds, then $p_m?_{\otimes_m}\,(x \mathbin{+\!\!<} a) = \omega_m$, and by definition of operator $\ominus_m$, $(p_m?_{\otimes_m}\,x) \ominus_m (p_{m-1}?_{\otimes_{m-1}}\,a) = \omega_m$ too. □

6.4 Pattern matching on arrays

The pattern-matching problem, discussed in the context of lists in Chapter 5, can be posed on all 'structured' data types. For example, if P is a u by v rectangular two-dimensional array, and S is an m by n array of the same type, the problem is to find a pair (i, j) such that for all k and l such that $u \geq k \geq 1$ and $v \geq l \geq 1$

$$S[i-u+k, j-v+l] \quad = \quad P[k, l] \,. \tag{6.34}$$

This description of the two-dimensional pattern-matching problem is taken from Baker [10]. For higher dimensional arrays similar definitions of matching can be given. In this section we derive a hierarchy of algorithms for pattern matching on arrays. The algorithms for pattern matching from Knuth et al. [85] and Aho and Corasick [2] can be used in an algorithm for two-dimensional pattern matching. This algorithm is described by Bird [13] and Baker [10]. Baker also notices the existence of the hierarchy of algorithms we derive. Karp, Miller and Rosenberg [81] give an algorithm for finding repeated occurrences of square submatrices. This algorithm can be adjusted to deal with pattern matching. Although this algorithm is well suited for parallel implementation, see Crochemore and Rytter [33], it is inefficient compared with the aforementioned sequential algorithms.

Given a pattern P and a subject S, the pattern-matching problem requires to find the positions in S at which P matches. The pattern-matching problem is specified as a subarray problem. To obtain a hierarchy of algorithms for pattern matching, the first step made is to transform the function given in the specification into a family of functions. The next step is to not enumerate all subarrays, but only those of which the init of their measure equals the init of the measure of P. Using the pattern-matching algorithm from Aho and Corasick a hierarchy of efficient algorithms is derived.

This section is organised as follows. The first subsection gives the specification as a subarray problem of the pattern-matching problem on arrays. The second subsection derives a hierarchy of efficient algorithms for pattern matching, the final form of which is presented in the third subsection.

6.4.1 The specification

In this subsection we give a specification as a family of functions for pattern matching on arrays.

A first specification

The specification of the pattern-matching problem on *array* resembles the specification of the pattern-matching problem on *snoc-list*, see Section 5.5. Given an N-dimensional array P, the pattern-matching problem requires finding an occurrence of P in an N-dimensional array, or, if there are no such occurrences, the longest prefix of P occurring in the given array. Recall that array P is a list of $(N-1)$-dimensional arrays, so function *inits* may be applied to P.

$$pm_N \quad = \quad \uparrow_{\#_N}/ \cdot (\in \mathit{inits}\, P)\triangleleft \cdot \mathit{suba}_N \ , \tag{6.35}$$

where the filter expression $(\in \mathit{inits}\, P)\triangleleft$ is defined on *snoc-list* instead of on *set*. This definition of filter is omitted. Note that, except for the type difference, pm_1 is the specification of the pattern-matching problem on *snoc-list* given in Chapter 5. For the purpose of applying the theory developed in the previous subsections, we want to specify the pattern-matching problem as a family of functions instead of a function defined just on N-dimensional arrays, that is, we want the functions $\uparrow_{\#_N}/$, $(\in \mathit{inits}\, P)$, and suba_N to be dimension-dependent.

A second specification

To obtain a family of functions as the specification of the pattern-matching problem replace $\uparrow_{\#_N}/$ by $\uparrow_{\#_n}/$ and suba_N by suba_n. This leaves function $(\in \mathit{inits}\, P)$ to be replaced by a dimension-dependent equivalent. We define a family of functions set_n such that $\mathit{set}_{N-n} : A\star_N\wr \to A\star_n\wr$, that is, the $(N-n)$th component of this family of functions turns a set of N-dimensional arrays into a set of n-dimensional arrays. For example, if $N = 2$ and $n = 1$, then set_1 turns a set of two-dimensional arrays in a set of one-dimensional arrays, containing the columns of the arrays in the set of two-dimensional arrays.

$$\begin{aligned} \mathit{set}_n \quad &: \quad A\star_N\wr \to A\star_{N-n}\wr \\ \mathit{set}_0 \quad &= \quad \mathit{id} \\ \mathit{set}_n \quad &= \quad \cup/ \cdot \mathit{setify}* \cdot \mathit{set}_{n-1} \ , \end{aligned} \tag{6.36}$$

where function *setify* returns given a snoc-list the set with the elements from the snoc-list; it is the left-reduction $(\cup \cdot \mathit{id} \times \sigma)\not\to\{\,\}$. The family of functions pm_n is specified by

$$pm_n \quad = \quad \uparrow_{\#_n}/ \cdot (\in \cup/\ \mathit{inits}*\ \mathit{set}_{N-n}\, \{P\})\triangleleft \cdot \mathit{suba}_n \ . \tag{6.37}$$

Function *inits* is defined on the data type *n-dimensional array* for $n \geq 1$; on the data type *zero-dimensional array* we assume function *inits* to be the identity function. Function $\cup/$ is not defined on the data type $A\wr$ if A is not of the form $B\wr$ for some B, and in this case we define $\cup/$ to be the identity function on *set*. Consider the following example. Let $P = [[2,3],[5,4]]$ be a two-dimensional array pattern. Then the family of functions pm_n consists of three components:

$$\begin{aligned} pm_2 \quad &= \quad \uparrow_{\#_2}/ \cdot (\in \{[\,],[[2,3]],[[2,3],[5,4]]\})\triangleleft \cdot \mathit{suba}_2 \\ pm_1 \quad &= \quad \uparrow_{\#_1}/ \cdot (\in \{[\,],[2],[2,3],[5],[5,4]\})\triangleleft \cdot \mathit{suba}_1 \\ pm_0 \quad &= \quad \uparrow_{\#_0}/ \cdot (\in \{2,3,4,5\})\triangleleft \cdot \mathit{suba}_0 \ . \end{aligned}$$

Function pm_1 is not exactly the specification of the pattern-matching problem on *snoc-list*: the set $\cup/\, inits{*}\, set_{N-1}\,\{P\}$ might contain the inits of more than one list. Function pm_1 is a specification of the problem of pattern matching with probably more than one pattern. Note that for function pm_0 we have

$$\begin{array}{ll} & pm_0 \\ = & \quad \text{definition of } pm_0 \\ & \uparrow_{\#_0}/ \cdot (\in \{2,3,4,5\}) \triangleleft \cdot suba_0 \\ = & \quad \text{definition of } suba_0 \text{ and filter on } \textit{snoc-list} \\ & \uparrow_{\#_0}/ \cdot (\in \{2,3,4,5\})?_{\uparrow_{\#_0}} \star \cdot \tau \\ = & \quad \text{definition of map and reduction on } \textit{snoc-list} \\ & (\in \{2,3,4,5\})?_{\uparrow_{\#_0}} \ . \end{array}$$

We abbreviate the set $\cup/\, inits{*}\, set_{N-n}\,\{P\}$ to Q_n, and the predicate $\in Q_n$ to q_n.

$$\begin{array}{lll} Q_n & = & \cup/\, inits{*}\, set_{N-n}\,\{P\} \\ q_n & = & \in Q_n \ . \end{array}$$

Note that $q_N = (\in\ inits P)$, and that q_n is prefixen-closed, and hence prefix-closed. A derivative $\delta_x\, a$ of q_n is defined by

$$\delta_x\, a \quad = \quad a \in (hd \cdot (\#\, x \hookrightarrow)){*}\, (x \in inits) \triangleleft Q_n \ . \tag{6.38}$$

A third specification

Compared with the pattern-matching problem on *snoc-list*, there is a new aspect in the pattern-matching problem on arrays. Consider the function pm_2. Let the two-dimensional pattern P be

$$P \quad = \quad \begin{pmatrix} 5 & 2 \\ 6 & 4 \end{pmatrix} .$$

Then all of the elements of the set $inits\, P$, except for the empty list, have height 2, and each element of $suba_2$ with height not equal to 2 does not match with any of the elements of $inits\, P$. Therefore, given an argument x, say

$$x \quad = \quad \begin{pmatrix} 5 & 2 \\ 6 & 4 \\ 9 & 0 \end{pmatrix} ,$$

all elements of $suba_2\, x$ with height not equal to 2 can immediately be discarded. So from the elements of $suba_2\, x$ the remaining candidates are

$$[[\,], \begin{pmatrix} 5 \\ 6 \end{pmatrix}, \begin{pmatrix} 5 & 2 \\ 6 & 4 \end{pmatrix}, \begin{pmatrix} 6 \\ 9 \end{pmatrix}, \begin{pmatrix} 6 & 4 \\ 9 & 0 \end{pmatrix}]$$

This argument can be repeated for the higher-dimensional functions pm_n, with height replaced by me_{n-1}-value. We define a family of functions $subm_n$ the elements of which enumerate only subarrays of a given measure, and we replace $suba_n$ by $subm_n$ in the specification of the pattern-matching problem.

Let the family of functions $subm_n$ enumerate all submatrices of its argument with me_{n-1}-value equal to me_{n-1}-value of a given pattern. The family of functions $subm_n$ can be defined elegantly using an auxiliary family of functions $taim_n$ which enumerates the 'tails' with me_{n-1}-value equal to me_{n-1}-value of the pattern. The families of functions $subm_n$ and $taim_n$ are very similar to the family of functions $suba_n$ and $tails_n$. Let P be an N-dimensional array with measure p, with $all\,(\neq 0)\,p = true$. Define the family of functions $stam_n$ by

$$stam_n \;=\; subm_n \vartriangle taim_n \tag{6.39}$$

Apply Array Mutumorphisms to obtain a hierarchical left-reduction for $stam_n$. Define $stam_0$ by $\tau \vartriangle (\tau \cdot \tau)$. The family of functions $subm_n$ is characterised by

$$\begin{array}{lcl} subm_n & : & A{\star}_n \rightarrow A{\star}_n{\star} \\ subm_n\,[\,] & = & [\,] \\ subm_n\,(x \mathbin{-\!\!\!<} a) & = & (subm_n\,x) \mathbin{+\!\!\!+\!\!\!<} (\mathbin{+\!\!\!+\!\!\!<}/\; taim_n\,(x \mathbin{-\!\!\!<} a))\ , \end{array} \tag{6.40}$$

and the function $taim_n$ by

$$\begin{array}{lcl} taim_n & : & A{\star}_n \rightarrow A{\star}_n{\star}{\star} \\ taim_n\,[\,] & = & [\,] \\ taim_n\,(x \mathbin{-\!\!\!<} a) & = & taim_n\,x \oslash ((=(n{-}1) \rightharpoonup p) \cdot me_{n-1}) \triangleleft subm_{n-1}\,a\ , \end{array} \tag{6.41}$$

where operator $\oslash$ is defined by

$$x \oslash a \;=\; \begin{cases} (\tau \cdot \tau){\star}\, a & \text{if } x = [\,] \\ x\;\Upsilon_{\odot}\;a & \text{otherwise}\ , \end{cases} \tag{6.42}$$

where operator $\odot$ is defined by

$$x \odot a \;=\; (\mathbin{-\!\!\!<} a){\star}\,x \mathbin{+\!\!\!+\!\!\!<} [[a]]\ . \tag{6.43}$$

Note that the only differences between functions $subm_n$ and $taim_m$ and functions $suba_n$ and $tails_n$ are the difference between the expressions for $subm_n\,[\,]$ and $suba_n\,[\,]$, and the difference between the expressions for $tails_n\,(x \mathbin{-\!\!\!<} a)$ and $taim_n\,(x \mathbin{-\!\!\!<} a)$. Define the family of predicates p_n by

$$p_m \;=\; (= m \rightharpoonup p) \cdot me_m\ ,$$

for all m with $0 \leq m \leq n$, and note that $p_0 = \underline{true}$. If we define the families of operators $\oplus_n$ and $\ominus_n$ by

$$\begin{array}{lcl} (x, y) \oplus_m (a, b) & = & x \mathbin{+\!\!\!+\!\!\!<} \mathbin{+\!\!\!+\!\!\!<}/(y \oslash p_{m-1} \triangleleft a) \\ (x, y) \ominus_m (a, b) & = & y \oslash p_{m-1} \triangleleft a\ , \end{array}$$

Then

$$\begin{array}{lcl} \mathit{subm}_m\,(x \mathbin{+\!\!\!<} a) & = & (\mathit{subm}_m\,x, \mathit{taim}_m\,x) \oplus_m (\mathit{subm}_{m-1}\,a, \mathit{taim}_{m-1}\,a) \\ \mathit{taim}_m\,(x \mathbin{+\!\!\!<} a) & = & (\mathit{subm}_m\,x, \mathit{taim}_m\,x) \ominus_m (\mathit{subm}_{m-1}\,a, \mathit{taim}_{m-1}\,a)\ , \end{array}$$

for all m with $n \geq m \geq 1$, and applying Array Mutumorphisms we obtain

$$\mathit{stam}_n \;=\; (\otimes_n, \ldots, \otimes_1, \tau \vartriangle (\tau \cdot \tau)) \not\rightarrow_n (e_n, \ldots, e_1)\ , \tag{6.44}$$

where the family of values e_n is defined by $e_m = ([\,],[\,])$, and the family of operators $\otimes_n$ by

$$\begin{array}{lcl} \otimes_m & : & (A\star_m\star \times A\star_m\star\star) \times (A\star_{m-1}\star \times A\star_{m-1}\star\star) \to A\star_m\star \times A\star_m\star\star \\ (x,y) \otimes_m (a,b) & = & ((x,y) \oplus_m (a,b), (x,y) \ominus_m (a,b))\ , \end{array} \tag{6.45}$$

for all m with $n \geq m \geq 1$. Specification (6.37) is transformed into

$$pm_n \;=\; \uparrow_{\#_n}/ \cdot q_n\triangleleft \cdot \mathit{subm}_n\ . \tag{6.46}$$

6.4.2 The derivation

In this section we derive, starting with specification (6.46) for pattern matching on *array* given in the previous subsection, a hierarchy of efficient algorithms for pattern matching. The structure of the derivation is similar to the structure of the derivation of the hierarchical version of Horner's rule: specification pm_n (6.46) will be extended with the computation of some extra information to obtain a hierarchical left-reduction followed by a projection function for pm_n.

Deriving a hierarchical left-reduction for pm_n

Since the straightforward implementation of specification (6.46) of pm_n is an inefficient program, we try to derive an algorithm that can be implemented as a more efficient program for pm_n. Consider specification (6.46) of pm_n. We strive for an application of Array Fusion, and for that purpose we derive for arbitrary family of functions functions g_n

$$\begin{array}{ll} & pm_n \\ = & \quad \text{specification (6.46) of } pm_n \\ & \uparrow_{\#_n}/ \cdot q_n\triangleleft \cdot \mathit{subm}_n \\ = & \quad \text{definition of } \mathit{stam}_n \\ & \uparrow_{\#_n}/ \cdot q_n\triangleleft \cdot \mathit{exl} \cdot \mathit{stam}_n \\ = & \quad \text{equation (2.14)} \\ & \mathit{exl} \cdot (\uparrow_{\#_n}/ \cdot q_n\triangleleft) \times g_n \cdot \mathit{stam}_n\ . \end{array}$$

The derivations in Section 5.2 and Section 6.3.3 suggest to define g_n by $(\uparrow_{\#_n}/ \cdot q_n\triangleleft)\star$. Abbreviate the resulting product of functions by

$$j_n \;=\; (\uparrow_{\#_n}/ \cdot q_n\triangleleft) \times (\uparrow_{\#_n}/ \cdot q_n\triangleleft)\star \;. \tag{6.47}$$

Since stam_n is a hierarchical left-reduction we can apply Array Fusion to $j_n \cdot \mathit{stam}_n$. Array Fusion gives

$$j_n \cdot \mathit{stam}_n \;=\; (\ominus_n, \ldots, \ominus_1, h)\nrightarrow_n(e_n, \ldots, e_1) \;, \tag{6.48}$$

for $h = j_0 \cdot \mathit{stam}_0$, and for some family of values e_n and some family of operators $\ominus_n$, provided

$$e_m \;=\; j_m\,([\,],[\,]) \tag{6.49}$$

$$j_m\,((x,y)\otimes_m(a,b)) \;=\; j_m\,(x,y)\ominus_m j_{m-1}\,(a,b)\;, \tag{6.50}$$

for all m with $n \geq m \geq 1$ and for all (x,y) in the image of stam_m and all (a,b) in the image of stam_{m-1}. It follows that h is defined by $\mathit{id} \vartriangle \tau \cdot q_0?_{\uparrow_{\#_0}}$, and e_m is defined by $(\omega_m,[\,])$, where $\omega_m = \nu_{\uparrow_{\#_m}}$.

The second condition can be satisfied provided we tuple with some auxiliary functions. These functions will appear in the derivation. An operator $\ominus_m$ satisfying condition (6.50) is synthesised as follows.

$$\begin{aligned}
& j_m\,((x,y)\otimes_m(a,b)) \\
= \quad & \quad \text{definition of } j_m \text{ and } \otimes_m,\ z = y \oslash p_{m-1}\triangleleft a \\
& ((\uparrow_{\#_m}/ \cdot q_m\triangleleft)\times(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star)\,(x \mathbin{+\!\!\!+\!\!<} \mathbin{+\!\!\!+\!\!<}/\, z, z) \\
= \quad & \quad \text{filter and reduction on } \textit{snoc-list} \\
& (\uparrow_{\#_m}/\, q_m\triangleleft x \uparrow_{\#_m} \uparrow_{\#_m}/\, q_m\triangleleft\mathbin{+\!\!\!+\!\!<}/\, z, (\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star z) \\
= \quad & \quad \text{Fusion on } \textit{snoc-list} \\
& (\uparrow_{\#_m}/\, q_m\triangleleft x \uparrow_{\#_m} \uparrow_{\#_m}/\,(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star z, (\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star z)\;.
\end{aligned}$$

In this last expression, the subexpression $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star z$, where $z = y \oslash p_{m-1}\triangleleft a$ needs to be developed further; subexpression $\uparrow_{\#_m}/\, q_m\triangleleft x$ equals $\mathit{exl}\, j_m\,(x,y)$. We have

$$\begin{aligned}
j_m\,((x,y)\otimes_m(a,b)) \;=\; & (s \uparrow_{\#_m} (\uparrow_{\#_m}/\, w), w) \\
& \quad \textbf{where } (s,t) = j_m\,(x,y) \\
& \qquad\quad w = (\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star(y \oslash p_{m-1}\triangleleft a)\;.
\end{aligned} \tag{6.51}$$

In view of condition (6.50) express $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star(y\oslash(p_{m-1}\triangleleft a))$ in terms of $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star y$ and $\uparrow_{\#_{m-1}}/\, q_{m-1}\triangleleft a$. In the following derivation we use the fact that the family of predicates q_n is prefixen-closed. Since the family of predicates q_n is prefixen-closed, we can apply Theorem 6.33 to obtain a hierarchical left-reduction for function $q_n?_{\uparrow_{\#_n}}$. We have

$$q_n?_{\uparrow_{\#_n}} \;=\; (\oplus_n, \ldots, \oplus_1, q_0?_{\uparrow_{\#_0}})\nrightarrow_n([\,], \ldots, [\,])\;, \tag{6.52}$$

where the family of operators $\oplus_n$ is defined by

$$x \oplus_m a \;=\; \begin{cases} x \mathbin{-\!\!\!\prec} a & \text{if } x \neq \omega_m \;\wedge\; a \neq \omega_{m-1} \;\wedge\; q_m\,(x \mathbin{-\!\!\!\prec} a) \\ \omega_m & \text{otherwise} \;, \end{cases}$$

for all m with $n \geq m \geq 1$. Distinguish the two cases in the definition of operator $\oslash$. Let $\omega_m = \nu_{\uparrow_{\#_m}}$.

- If $y = [\,]$, or equivalently $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star\, y = [\,]$, then

$$\begin{array}{ll} & (\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star\,(y \oslash p_{m-1}\triangleleft a) \\ = & \quad \text{case assumption, definition of } \oslash \\ & (\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star\,(\tau \cdot \tau)\star\, p_{m-1}\triangleleft a \\ = & \quad \text{map-distributivity} \\ & (\uparrow_{\#_m}/ \cdot q_m\triangleleft \cdot \tau \cdot \tau)\star\, p_{m-1}\triangleleft a \\ = & \quad \text{derivation below} \\ & (([\,]\oplus_m) \cdot q_{m-1}?_{\uparrow_{\#_{m-1}}})\star\, p_{m-1}\triangleleft a \\ = & \quad \text{map-distributivity} \\ & ([\,]\oplus_m)\star\, q_{m-1}?_{\uparrow_{\#_{m-1}}}\star\, p_{m-1}\triangleleft a \;. \end{array}$$

The derivation of the equality applied in the third step reads as follows.

$$\begin{array}{ll} & \uparrow_{\#_m}/ \cdot q_m\triangleleft \cdot \tau \cdot \tau \\ = & \quad \text{filter on } \textit{snoc-list} \\ & \uparrow_{\#_m}/ \cdot q_m?_{\uparrow_{\#_m}}\star \cdot \tau \cdot \tau \\ = & \quad \text{map and reduction on } \textit{snoc-list} \\ & q_m?_{\uparrow_{\#_m}} \cdot \tau \\ = & \quad \text{equation (6.52)} \\ & (\oplus_m, \ldots, \oplus_1, q_0?_{\uparrow_{\#_0}}) \not\rightarrow_m ([\,], \ldots [\,]) \cdot \tau \\ = & \quad \text{Hierarchical Left-reduction, definition 6.7, map on } \textit{snoc-list} \\ & \oplus_m \not\rightarrow [\,] \cdot \tau \cdot q_{m-1}?_{\uparrow_{\#_{m-1}}} \\ = & \quad \text{definition of left-reduction} \\ & ([\,]\oplus_m) \cdot q_{m-1}?_{\uparrow_{\#_{m-1}}} \;. \end{array}$$

The expression obtained for $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star\,(y \oslash p_{m-1}\triangleleft a)$ in case $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star\, y = [\,]$ is not of the desired form. This is resolved after the following case.

- If $(\uparrow_{\#_m}/ \cdot q_m\triangleleft)\star\, y \neq [\,]$ we calculate as follows.

$$
\begin{array}{ll}
 & (\uparrow_{\#_m}/\cdot\, q_m\triangleleft)\star(y \oslash p_{m-1}\triangleleft a) \\
= & \quad \text{case assumption, definition of } \oslash \\
 & (\uparrow_{\#_m}/\cdot\, q_m\triangleleft)\star(y \curlyvee_{\odot} p_{m-1}\triangleleft a) \\
= & \quad \text{map-zip law (3.34)} \\
 & y \curlyvee_{\uparrow_{\#_m}/\cdot q_m\triangleleft\cdot\odot}\ p_{m-1}\triangleleft a \\
= & \quad \text{equation (6.53) below} \\
 & y \curlyvee_{⦷_m\cdot(\uparrow_{\#_m}/\cdot q_m\triangleleft)\times q_{m-1}?_{\uparrow_{\#_{m-1}}}}\ p_{m-1}\triangleleft a \\
= & \quad \text{zip-map law (3.35)} \\
 & (\uparrow_{\#_m}/\cdot\, q_m\triangleleft)\star y \curlyvee_{⦷_m} q_{m-1}?_{\uparrow_{\#_{m-1}}}\star p_{m-1}\triangleleft a\ .
\end{array}
$$

Equation (6.53) applied in this calculation reads

$$
\uparrow_{\#_m}/\cdot\, q_m\triangleleft\cdot\odot \;=\; ⦷_m\cdot(\uparrow_{\#_m}/\cdot\, q_m\triangleleft)\times q_{m-1}?_{\uparrow_{\#_{m-1}}}\ . \tag{6.53}
$$

This equation is derived by means of the following calculation. Let w and b be elements of respectively y and $p_{m-1}\triangleleft a$.

$$
\begin{array}{ll}
 & \uparrow_{\#_m}/\, q_m\triangleleft(w \odot b) \\
= & \quad \text{definition of } \odot \\
 & \uparrow_{\#_m}/\, q_m\triangleleft((\mathbin{-\!\!<} b)\star w \mathbin{+\!\!\!<} [[b]]) \\
= & \quad \text{filter and reduction on } \textit{snoc-list} \\
 & \uparrow_{\#_m}/\, q_m\triangleleft(\mathbin{-\!\!<} b)\star w \uparrow_{\#_m} q_m?_{\uparrow_{\#_m}}[b] \\
= & \quad \text{equation (6.52)} \\
 & \uparrow_{\#_m}/\, q_m\triangleleft(\mathbin{-\!\!<} b)\star w \uparrow_{\#_m} ([\,]\oplus_m q_{m-1}?_{\uparrow_{\#_{m-1}}} b) \\
= & \quad \text{introduction of } \obar_m \text{ below} \\
 & (\obar_m q_{m-1}?_{\uparrow_{\#_{m-1}}} b) \uparrow_{\#_m}/\, q_m\triangleleft w \uparrow_{\#_m} ([\,]\oplus_m q_{m-1}?_{\uparrow_{\#_{m-1}}} b) \\
= & \quad \text{introduction of } ⦷_m \text{ below} \\
 & \uparrow_{\#_m}/\, q_m\triangleleft w\ ⦷_m\ q_{m-1}?_{\uparrow_{\#_{m-1}}} b\ ,
\end{array}
$$

where operator $⦷_m$ is defined by

$$
s\ ⦷_m\ c \;=\; (s \obar_m c) \uparrow_{\#_m} ([\,]\oplus_m c)\ .
$$

It remains to give a definition of operator $\obar_m$. For expression $\uparrow_{\#_m}/\, q_m\triangleleft(\mathbin{-\!\!<} b)\star w$ we return to Chapter 5. Since (x, y) is an element in the image of $\textit{stam}_m$,

$$
y \;=\; \textit{taim}_m\, v\ .
$$

for some v. Function $\textit{taim}_m$ returns, given an array, a list of lists, in which each list contains the tails (considered as the elements returned by function *tails* defined on *snoc-list*) of an array of a specific measure, so that all elements in such a list have equal me_{m-1}-value, but different me_m-value. Array w, an element of y, is such a list. Since all elements in w have equal me_{m-1}-value, selector $\uparrow_{\#_m}$ equals selector $\uparrow_{\#}$ on w. It follows that we have an expression of the form $\uparrow_{\#}/\ q_m\triangleleft(\mathbin{-\!\!<} b)\star w$, where $w = \mathit{tails}\ u$ (actually, this is not correct: the elements of w are enumerated in some order (w is a list), and this order is not present in $\mathit{tails}\ u$ ($\mathit{tails}\ u$ is a set); but it is not difficult to repeat the development given in Section 5.3 for function *tails* returning a list instead of a set) for some list (array) u, and where predicate q_m is prefixen-closed and hence prefix-closed. For such an expression it has been calculated in Chapter 5, see equations (5.33) and (5.35), that

$$\uparrow_{\#}/\ q_m\triangleleft(\mathbin{-\!\!<} b)\star w \quad = \quad (\obar_m b)\uparrow_{\#}/\ q_m\triangleleft w\ , \tag{6.54}$$

where operator $\obar_m$ is defined by

$$w \obar_m b \quad = \quad \begin{cases} w \mathbin{-\!\!<} b & \text{if } \delta_w\ b \\ (\mathit{lpt}\ \mathit{tl}\ w) \obar_m b & \text{if } \neg\,\delta_w\ b\ \wedge\ w \neq [\,] \\ \omega_m & \text{otherwise}\ , \end{cases} \tag{6.55}$$

where function *lpt* is the function $\uparrow_{\#}/\cdot\ q_m\triangleleft\cdot\ \mathit{tails}$. A definition of a derivative δ of predicate q_m is given in equation (6.38). Since q_m is prefixen-closed,

$$q_m\,(w \mathbin{-\!\!<} b) \qquad \Rightarrow \qquad q_{m-1}\ b\ ,$$

so b may be preceded by a test on q_{m-1}. It follows that

$$\begin{array}{ll} & \uparrow_{\#}/\ q_m\triangleleft(\mathbin{-\!\!<} b)\star w \\ = & \qquad q_m\,(w \mathbin{-\!\!<} b)\ \Rightarrow\ q_{m-1}\ b \\ & \uparrow_{\#}/\ q_m\triangleleft(\mathbin{-\!\!<} q_{m-1}?_{\uparrow_{\#_{m-1}}}\ b)\star w \\ = & \qquad \text{equation (6.54)} \\ & (\obar_m\, q_{m-1}?_{\uparrow_{\#_{m-1}}}\ b)\uparrow_{\#}/\ q_m\triangleleft w\ , \end{array}$$

where operator $\obar_m$ is defined by

$$w \obar_m b \quad = \quad \begin{cases} w \mathbin{-\!\!<} b & \text{if } b \neq \omega_{m-1}\ \wedge\ \delta_w\ b \\ (\mathit{lpt}\ \mathit{tl}\ w) \obar_m b & \text{if } b \neq \omega_{m-1}\ \wedge\ \neg\,\delta_w\ b\ \wedge\ w \neq [\,] \\ \omega_m & \text{otherwise}\ . \end{cases} \tag{6.56}$$

In both cases of the definition of operator $\oslash$, $(\uparrow_{\#_m}/\cdot\ q_m\triangleleft)\star(y\oslash p_{m-1}\triangleleft a)$ has been expressed in terms of $(\uparrow_{\#_m}/\cdot\ q_m\triangleleft)\star y$ and $q_{m-1}?_{\uparrow_{\#_{m-1}}}\star p_{m-1}\triangleleft a$. Equation (6.51) is transformed into

$$\begin{array}{lcl} j_m\,((x,y)\otimes_m(a,b)) & = & (s\uparrow_{\#_m}\uparrow_{\#_m}/\ w, w) \\ & & \quad \textbf{where}\ (s,t) = j_m\,(x,y) \\ & & \qquad\quad w = \begin{cases} ([\,]\oplus_m)\star q_{m-1}?_{\uparrow_{\#_{m-1}}}\star p_{m-1}\triangleleft a & \text{if } t = [\,] \\ t\ \Upsilon_{\obar_m}\ q_{m-1}?_{\uparrow_{\#_{m-1}}}\star p_{m-1}\triangleleft a & \text{if } t \neq [\,]\ . \end{cases} \end{array} \tag{6.57}$$

It follows that we have not been able to find an operator $\ominus_m$ satisfying (6.50).

An extended specification

From equation (6.57) it follows that we have not been able to find an operator $\ominus_m$ satisfying (6.50), but it also follows that function $j_m \cdot \mathit{stam}_m$ is catamorphic modulo l_m, where the family of functions l_n is defined by

$$\begin{array}{lcl} l_m & = & k_m \cdot \mathit{subm}_m \\ k_m & = & q_m?_{\uparrow_{\#_m}}\star \cdot p_m\triangleleft \ , \end{array}$$

that is, if we define the family of operators $\ominus_n$ for m with $n \geq m \geq 1$ by

$$((x,y),z) \ominus_m ((a,b),c) \;=\; (x \uparrow_{\#_m} \uparrow_{\#_m}/\ s, s) \quad \textbf{where } s = \begin{cases} ([\,]\oplus_m)\star c & \text{if } y = [\,] \\ y \ \Upsilon_{\obar_m}\ c & \text{otherwise} \end{cases} ,$$

then for all m with $n \geq m \geq 1$

$$j_m\, \mathit{stam}_m\, (x \mathbin{-\!\!\!<} a) \;=\; (j_m\, \mathit{stam}_m\, x, l_m\, x) \ominus_m (j_{m-1}\, \mathit{stam}_{m-1}\, a, l_{m-1}\, a) \ .$$

For the purpose of applying Array Mutumorphisms, Theorem 6.11 to obtain a hierarchical left-reduction for the tuple of functions

$$(j_n \cdot \mathit{stam}_n) \vartriangle l_n \ , \tag{6.58}$$

we show that l_m is catamorphic modulo $j_m \cdot \mathit{stam}_m$. Note that for the original specification of the pattern-matching problem we have

$$\begin{array}{ll} & \mathit{pm}_n \\ = & \quad \text{definition of } j_n \text{ and } \mathit{stam}_n \\ & \mathit{exl} \cdot j_n \cdot \mathit{stam}_n \\ = & \quad \text{equation (2.16)} \\ & \mathit{exl} \cdot \mathit{exl} \cdot (j_n \cdot \mathit{stam}_n) \vartriangle l_n \ , \end{array}$$

so the tuple of functions (6.58) is an extension of the original specification. We show that l_m is catamorphic modulo $j_m \cdot \mathit{stam}_m$, that is, we construct a family of operators $\ominus_n$ such that

$$l_m\, (x \mathbin{-\!\!\!<} a) \;=\; (j_m\, \mathit{stam}_m\, x, l_m\, x) \ominus_m (j_{m-1}\, \mathit{stam}_{m-1}\, x, l_{m-1}\, a) \ , \tag{6.59}$$

for all m with $n \geq m \geq 1$. First, note that

$$\begin{array}{ll} & l_0 \\ = & \quad \text{definition of } l_0 \\ & k_0 \cdot \mathit{subm}_0 \\ = & \quad \text{definition of } \mathit{subm}_0 \end{array}$$

$$\begin{array}{ll}
 & k_0 \cdot \tau \\
= & \quad \text{definition of } k_0 \\
 & q_0?_{\uparrow\#_0}\star \cdot p_0\triangleleft \cdot \tau \\
= & \quad p_0 = \underline{\mathit{true}} \\
 & q_0?_{\uparrow\#_0}\star \cdot \tau \\
= & \quad \text{definition of map on } \textit{snoc-list} \\
 & \tau \cdot q_0?_{\uparrow\#_0} \ .
\end{array}$$

Furthermore,

$$\begin{array}{ll}
 & l_m\,[\,] \\
= & \quad \text{definition of } l_m \\
 & k_m\ \mathit{subm}_m\,[\,] \\
= & \quad \text{definition of } \mathit{subm}_m \\
 & k_m\,[\,] \\
= & \quad \text{definition of } k_m \\
 & q_m?_{\uparrow\#_m}\star\, p_m\triangleleft\,[\,] \\
= & \quad \text{definition of filter and map on } \textit{snoc-list} \\
 & [\,]\ ,
\end{array}$$

for all m with $n \geq m \geq 1$. For the construction of a family of operators $\ominus_n$ satisfying equation (6.59), we reason as follows. Previous derivations show that

$$k_m\ \mathit{subm}_m\,(x \mathbin{-\!\!<} a) \quad = \quad k_m\ \mathit{subm}_m\ x \mathbin{+\!\!\!+\!\!<} \mathbin{+\!\!\!+\!\!<}/\ k_m\star\,(\mathit{taim}_m\ x \oslash p_{m-1}\triangleleft\,\mathit{subm}_{m-1}\ a)\ .$$

It remains to express value $k_m\star\,(\mathit{taim}_m\ x \oslash p_{m-1}\triangleleft\,\mathit{subm}_{m-1}\ a)$ in terms of values $k_m\ \mathit{subm}_m\ x$, $j_m\ \mathit{stam}_m\ x$, and $k_{m-1}\ \mathit{subm}_{m-1}\ a$. Distinguish the two cases in the definition of operator $\oslash$. Let $y = \mathit{taim}_m\ x$, and $c = \mathit{subm}_{m-1}\ a$.

- If $y = [\,]$, calculate as follows.

$$\begin{array}{ll}
 & k_m\star\,(y \oslash p_{m-1}\triangleleft\, c) \\
= & \quad \text{case assumption, definition of } \oslash \\
 & k_m\star\,(\tau\cdot\tau)\star\, p_{m-1}\triangleleft\, c \\
= & \quad \text{map-distributivity} \\
 & (k_m\cdot\tau\cdot\tau)\star\, p_{m-1}\triangleleft\, c\ .
\end{array}$$

If $\mathit{lt}\,(m \rightharpoonup p) = 1$, then $p_m\triangleleft\cdot\tau\cdot\tau$ may be replaced by $\tau\cdot\tau$ in the above expression. So

$$
\begin{array}{ll}
 & (k_m \cdot \tau \cdot \tau)\star\, p_{m-1} \lhd c \\
= & \quad \text{definition of } k_m \\
 & (q_m ?_{\uparrow_{\#_m}} \star \cdot p_m \lhd \cdot \tau \cdot \tau)\star\, p_{m-1} \lhd c \\
= & \quad \text{case assumption} \\
 & (q_m ?_{\uparrow_{\#_m}} \star \cdot \tau \cdot \tau)\star\, p_{m-1} \lhd c \\
= & \quad \text{map on } \textit{snoc-list} \\
 & (\tau \cdot q_m ?_{\uparrow_{\#_m}} \cdot \tau)\star\, p_{m-1} \lhd c \\
= & \quad \text{equation (6.52)} \\
 & (\tau \cdot ([\,]\oplus_m) \cdot q_{m-1} ?_{\uparrow_{\#_{m-1}}})\star\, p_{m-1} \lhd c \\
= & \quad \text{map-distributivity, definition of } k_{m-1} \\
 & (\tau \cdot ([\,]\oplus_m))\star\, k_{m-1}\ c\ .
\end{array}
$$

If $lt\,(m \rightharpoonup p) \neq 1$, then $p_m \lhd \cdot \tau \cdot \tau = \underline{[\,]}$, and hence

$$
\begin{array}{ll}
 & (k_m \cdot \tau \cdot \tau)\star\, p_{m-1} \lhd c \\
= & \quad \text{definition of } k_m \\
 & (q_m ?_{\uparrow_{\#_m}} \star \cdot p_m \lhd \cdot \tau \cdot \tau)\star\, p_{m-1} \lhd c \\
= & \quad \text{case assumption} \\
 & \underline{[\,]}\star\, p_{m-1} \lhd c \\
= & \quad \text{definition of constant function} \\
 & \underline{[\,]}\star\, k_{m-1}\ c\ .
\end{array}
$$

- If $y \neq [\,]$, calculate as follows.

$$
\begin{array}{ll}
 & k_m\star\,(y \oslash p_{m-1} \lhd c) \\
= & \quad \text{case assumption, definition of } \oslash \\
 & k_m\star\,(y\ \Upsilon_{\odot}\ p_{m-1} \lhd c) \\
= & \quad \text{map-zip law (3.34)} \\
 & y\ \Upsilon_{k_m \cdot \odot}\ p_{m-1} \lhd c \\
= & \quad \text{equation 6.60 below} \\
 & y\ \Upsilon_{\ominus_m \cdot (\uparrow_{\#_m}/ \cdot q_m \lhd) \times q_{m-1} ?_{\uparrow_{\#_{m-1}}}}\ p_{m-1} \lhd c \\
= & \quad \text{zip-map law (3.35), definition of } k_{m-1} \\
 & (\uparrow_{\#_m}/ \cdot q_m \lhd)\star\, y\ \Upsilon_{\ominus_m}\ k_{m-1}\ c\ .
\end{array}
$$

Equation (6.60) applied in this calculation reads

$$
k_m \cdot \odot \;=\; \ominus_m \cdot (\uparrow_{\#_m}/ \cdot q_m \lhd) \times q_{m-1} ?_{\uparrow_{\#_{m-1}}}\ . \tag{6.60}
$$

Let w and b be elements of respectively y and $p_{m-1} \triangleleft c$. By definition of $\odot$ and k_m we have

$$k_m\,(w \odot b) \;=\; q_m?_{\uparrow_{\#_m}} \star p_m \triangleleft ((-\!\!\!< b) \star w +\!\!\!+\!\!\!< [[b]]) \;.$$

Since w is an element of y, and $y = \mathit{taim}_m\,x$, w is a list of the form $\mathit{tails}\,u$ for some m-dimensional array u, with $\#\,u = \#\,x$. Furthermore, by definition of taim_m,

$$\mathit{it\,me}_m\,u \;=\; (p{-}1) \rightharpoonup p \;,$$

and hence for all elements $e \in \mathit{tails}\,u$ we have

$$\mathit{it\,me}_m\,e \;=\; (p{-}1) \rightharpoonup p \;.$$

Since for all elements e in $\mathit{tails}\,u$, $\#\,e \leq \#\,u$,

$$\begin{aligned}
&\#\,e\\
\leq\quad & \quad e \in \mathit{tails}\,u\\
&\#\,u\\
=\quad & \quad \text{definition of } \mathit{me}\\
&\mathit{lt\,me}_m\,u \;.
\end{aligned}$$

The longest element in $w \odot b$ is $u -\!\!\!< b$. If $\mathit{lt\,me}_m\,u < \mathit{lt}\,(m \rightharpoonup p) - 1$, or equivalently $\mathit{lt\,me}_m\,x < \mathit{lt}\,(m \rightharpoonup p) - 1$, then $p_m \triangleleft (w \odot b) = [\,]$, and hence $k_m\,(w \odot b) = [\,]$. If $\mathit{lt\,me}_m\,x \geq \mathit{lt}\,(m \rightharpoonup p) - 1$, then the list $p_m \triangleleft (w \odot b)$ contains one element. It follows that we need the length of the argument x in order to determine the value of function $k_m \cdot \mathit{subm}_m$, so $k_m \cdot \mathit{subm}_m$ is catamorphic modulo the length of the argument, and function $\#$ is tupled with function $k_m \cdot \mathit{subm}_m$. The definition of $\mathbin{⊜}_m$ distinguishes three cases, depending on the values $\#\uparrow_{\#_m}/\; q_m \triangleleft w$ and $\mathit{lt}\,(m \rightharpoonup p)$, which is abbreviated to lp_m. Note that $\#\uparrow_{\#_m}/\; q_m \triangleleft w \leq \mathit{lp}_m$.

- Suppose $\#\uparrow_{\#_m}/\; q_m \triangleleft w < \mathit{lp}_m - 1$. Then, since q_n is prefixen-closed, no element in $w \odot b$ satisfies p_m, and hence operator $\mathbin{⊜}_m$ is defined by $s \mathbin{⊜}_m c = [\omega_m]$
- Suppose $\#\uparrow_{\#_m}/\; q_m \triangleleft w = \mathit{lp}_m - 1$. Then the only element of measure $m \rightharpoonup p$ in $w \odot b$ is $\uparrow_{\#_m}/\; q_m \triangleleft w -\!\!\!< b$, so

$$\begin{aligned}
&q_m?_{\uparrow_{\#_m}} \star p_m \triangleleft (w \odot b)\\
=\quad & \quad \text{case assumption, definition of } p_m\\
&q_m?_{\uparrow_{\#_m}} \star [\uparrow_{\#_m}/\; q_m \triangleleft w -\!\!\!< b]\\
=\quad & \quad \text{definition of map on } \textit{snoc-list}\\
&[q_m?_{\uparrow_{\#_m}}\,(\uparrow_{\#_m}/\; q_m \triangleleft w -\!\!\!< b)]\\
=\quad & \quad \text{equation (6.52)}\\
&[\uparrow_{\#_m}/\; q_m \triangleleft w \oplus_m q_{m-1}?_{\uparrow_{\#_{m-1}}}\, b] \;,
\end{aligned}$$

and operator $⊜_m$ is defined by $s ⊜_m c = [s \oplus_m c]$.

- Suppose $\# \uparrow_{\#_m} / \, q_m \triangleleft w = lp_m$. Then the only element of measure $m \rightharpoonup p$ in $w \odot b$ is $tl \uparrow_{\#_m} / \, q_m \triangleleft w \mathbin{-\!\!\!<} b$, so

  $$k_m \, (w \odot b) \;=\; [q_m ?_{\uparrow_{\#_m}} (tl \uparrow_{\#_m} / \, q_m \triangleleft w \mathbin{-\!\!\!<} b)] \; ,$$

  and by distinguishing the possible cases for the value $lpt \; tl \uparrow_{\#_m} / \, q_m \triangleleft w$, where $lpt = \uparrow_{\#} / \cdot q_m \triangleleft \cdot \, tails$, we obtain the following definition for operator $⊜_m$

  $$s ⊜_m c = lpt \; tl \; s ⊜_m c \; .$$

So if operator $⊜_m$ is defined by

$$s ⊜_m c \;=\; \begin{cases} [s \oplus_m c] & \text{if } \# s = lp_m - 1 \\ (lpt \; tl \; s) ⊜_m c & \text{if } \# s = lp_m \\ [\omega_m] & \text{if } \# s < lp_m - 1 \; , \end{cases}$$

then equation (6.60) is satisfied.

In both cases of the definition of operator $\oslash$, $k_m \, (y \oslash p_{m-1} \triangleleft c)$ has been expressed in terms of $j_m \; stam_m \; x$ and $k_{m-1} \; suba_m \; a$ and if the family of operators $\ominus_n$ defined by

$$((x, y), z) \ominus_m ((a, b), c) \;=\; z \mathbin{+\!\!\!+\!\!\!<} \mathbin{+\!\!\!+\!\!\!<} / \, t$$
$$\textbf{where}$$
$$t = \begin{cases} (\tau \cdot ([\,] \oplus_m)) \star c & \text{if } y = [\,] \;\wedge\; lp_m = 1 \\ \underline{[\,]} \star c & \text{if } y = [\,] \;\wedge\; lp_m \neq 1 \\ y \, \Upsilon_{⊜_m} \, c & \text{otherwise} \; , \end{cases}$$

for all m with $n \geq m \geq 1$, then equation (6.59) is satisfied.

6.4.3 The hierarchy of algorithms

In this subsection we present the final result of the derivation of a hierarchy of efficient algorithms for pattern matching given in the previous section, and we give a final optimisation of this algorithm. The first two components of the following hierarchical left-reduction correspond to the function $j_m \cdot stam_m$, the third component to the function $k_m \cdot subm_m$, and the fourth component to the function $\#$.

We have

$$pm_n \;=\; exl^2 \cdot (\ominus_n, \ldots, \ominus_1, h) \not\rightarrow_n (e_n, \ldots, e_1) \; , \tag{6.61}$$

where h is defined by $(id \vartriangle \tau \cdot q_0 ?_{\uparrow_{\#_0}}) \vartriangle (\tau \cdot q_0 ?_{\uparrow_{\#_0}}) \vartriangle \underline{1}$, e_m is defined by $((\omega_m, [\,]), [\,], 0)$, and operator $\ominus_m$ is defined by

$$((u, v), w, z) \ominus_m ((a, b), c, d) \;=\; ((u \uparrow_{\#_m} \uparrow_{\#_m} / \, s, s), w \mathbin{+\!\!\!+\!\!\!<} \mathbin{+\!\!\!+\!\!\!<} / \, t, z{+}1)$$

$$\textbf{where}$$
$$s = \begin{cases} ([\,]\oplus_m)\star c & \text{if } v = [\,] \\ v\;\Upsilon_{\circledparallel_m}\; c & \text{otherwise} \end{cases}$$
$$t = \begin{cases} (\tau \cdot ([\,]\oplus_m))\star c & \text{if } w = [\,] \;\wedge\; lp_m = 1 \\ \underline{[\,]}\star c & \text{if } w = [\,] \;\wedge\; lp_m \neq 1 \\ v\;\Upsilon_{\circledequal_m}\; c & \text{otherwise} \;\;, \end{cases}$$

where the operators $\oplus_m$, $\circledparallel_m$ and $\circledequal_m$ are defined by

$$x \oplus_m a \;\;=\;\; \begin{cases} x \mathbin{-\!\!\!\prec} a & \text{if } x \neq \omega_m \;\wedge a \neq \omega_{m-1} \;\wedge\; q_m\,(x \mathbin{-\!\!\!\prec} a) \\ \omega_m & \text{otherwise} \;\;, \end{cases}$$

operator $\circledparallel_m$ is defined by

$$s \circledparallel_m c \;\;=\;\; (s \obar_m c) \uparrow_{\#_m} ([\,] \oplus_m c) \;,$$

where

$$s \obar_m c \;\;=\;\; \begin{cases} s \mathbin{-\!\!\!\prec} c & \text{if } c \neq \omega_{m-1} \;\wedge\; \delta_s\, c \\ (\mathit{lpt}\;\mathit{tl}\;s) \obar_m c & \text{if } c \neq \omega_{m-1} \;\wedge\; \neg\delta_s\, c \;\wedge\; s \neq [\,] \\ \omega_m & \text{otherwise} \;\;, \end{cases}$$

and operator $\circledequal_m$ is defined by $s \circledequal_m c = [\,]$ if $z < lp_m - 1$, and if $z \geq lp_m - 1$, then

$$s \circledequal_m c \;\;=\;\; \begin{cases} [s \oplus_m c] & \text{if } \#\, s = lp_m - 1 \\ (\mathit{lpt}\;\mathit{tl}\;s) \circledequal_m c & \text{if } \#\, s = lp_m \\ [\omega_m] & \text{if } \#\, s < lp_m - 1 \;, \end{cases}$$

for all m with $n \geq m \geq 1$.

The two remaining problems that have to be addressed are the computation of $\delta_s\, c$ and the computation of $\mathit{lpt}\;\mathit{tl}\;s$.

Concerning the computation of these values we have the following. Function δ takes as arguments an m-dimensional array s and an $(m-1)$-dimensional array c, and checks whether $s \mathbin{-\!\!\!\prec} c$ is an element of Q_m, that is, whether $q_m\,(s \mathbin{-\!\!\!\prec} c)$ holds. It is defined in equation (6.38) by

$$\delta_s\, c \;\;=\;\; c \in (\mathit{hd} \cdot (\#\, s \hookrightarrow))\ast (s \in \mathit{inits}) \triangleleft Q_m \;.$$

Checking whether or not $c \in (\mathit{hd} \cdot (\#\, s \hookrightarrow))\ast (s \in \mathit{inits}) \triangleleft Q_m$ is expensive and superfluous. We explain the situation for two-dimensional arrays. Consider the set of two-dimensional array patterns Q_2 defined by $\cup/\; \mathit{inits}\ast\; \mathit{set}_{N-2}\, \{P\}$. The two-dimensional pattern-matching algorithm with the set of patterns Q_2 uses the one-dimensional pattern-matching algorithm with the set of patterns Q_1. The columns of the two-dimensional arrays in Q_2 are one-dimensional arrays in Q_1. Assign a unique identifier, a natural number say, to each element in Q_1. Let the one-dimensional pattern-matching algorithm return the unique identifier instead of the one-dimensional array in case of a match, and replace the columns in the set of patterns Q_2 by the unique identifiers of the columns. Thus the two-dimensional pattern-matching problem has been reduced to a one-dimensional pattern-matching algorithm. This idea is easily generalised to n-dimensional arrays. We do not give the definitions of the functions involved in working out this idea. We have reduced the problem of computing $\delta_s\ c$ for an m-dimensional array s and an $(m{-}1)$-dimensional array c to computing $\delta_s\ c$ for a one-dimensional array s and an element c. The efficient computation of this value, and of the value $lpt\ tl\ s$, in the context of pattern-matching on lists has been discussed in the section on pattern matching in Section 5.5.

Algorithm (6.61) can be implemented as a program that is linear in the sum of the size of the pattern and the size of the argument. This follows from the observation that each algorithm in the hierarchy of algorithms is in fact the pattern-matching algorithm of Aho and Corasick on the data type *snoc-list*.

6.5 Conclusions

This chapter discusses the data type *array*. Using a general approach to hierarchical data types, it defines a hierarchy of data types for the data type *array*, based on the data type *snoc-list*, in which the data type *n-dimensional array* is expressed in terms of the data type $(n{-}1)$-*dimensional array*. All elements in the hierarchy of data types *array* are initial algebras. A catamorphism defined on the data type *n-dimensional array* can be factored into a composition of catamorphisms defined on the data type *m-dimensional array* for $m < n$, and therefore it is called a hierarchical catamorphism. We develop a theory for constructing hierarchical catamorphisms on *array*. Besides standard theorems like Array Fusion and Array Mutumorphisms, we also derive a generator fusion theorem for the family of generators $suba_n$, the components of which return all subarrays of a given array. Using this theorem, we construct a hierarchy of algorithms for finding the maximum subarray sum. The last section of this chapter derives a hierarchy of algorithms for pattern matching on *array*. Although we use succinct notations, the results still require much space, and one of the topics for future research is to try to automatically construct a hierarchy of algorithms given just the one-dimensional algorithm, the beginning of such a hierarchy. Another interesting topic for further research is the construction of other hierarchies of algorithms, for example for tiling problems on arrays and for compressing arrays, see [4].

Chapter 7

Incremental algorithms

There are a number of reasons why more problems will be solvable on a computer in the future. First, the processor speed of computers is still increasing. Second, more efficient algorithms may be derived for problems for which the existence of an optimal algorithm has not yet been shown. Third, for some classes of problems other kinds of algorithms, such as parallel or incremental, may be derived. This chapter discusses incremental algorithms.

If a computation is performed repeatedly on slightly changed data, it is often profitable to describe the computation by means of an incremental algorithm. An incremental algorithm describes how to compute the required value depending on the slightly changed data from the old value, the changes in the data, and perhaps some other information. Examples of computations which are performed repeatedly on slightly changed data can be found in interactive programs such as program development environments and spreadsheet programs. For example, in a spreadsheet program often lists of numbers are summed. If a number of the list is changed from a to b, the sum s should be changed to $s + (b - a)$. This change does not require the summation of all numbers in the lists.

The form of an incremental algorithm depends on the data type on which it is defined, and the edit model used. We propose a method for the description and derivation of incremental algorithms on several data types; the emphasis will be on the data type list.

Besides lots of articles on incremental algorithms for specific problems, like for example the algorithm for incrementally computing the minimum spanning tree of Frederickson [49] (one of the many incremental algorithms on graphs, for other examples see La Poutré [114]), the algorithm for incremental or adaptive Huffman coding of Knuth [84] and others, and the algorithm for pattern matching with a dynamically changing set of patterns of Meyer [100], several proposals for the derivation and description of incremental algorithms have been given in the literature. The language INC, designed by Yellin and Strom [141], automatically transforms algorithms in an FP-like syntax to incremental algorithms. For each construct in FP an incremental version is given, and since every algorithm consists of a series of FP constructs it can be incrementalised by incrementalising its components. This approach has the disadvantage, shared with all automatic methods for formal program derivation, that not always the most efficient incremental algorithm will be derived. Furthermore, the only data types INC can handle are *bag* and finite cartesian product, which is rather restrictive. Another approach to incrementality, called finite differencing, is described by Paige [110]. So-called invariants, equalities of the form $E = f\,(x_1, \ldots, x_n)$, are maintained by means of code which describes how to find the value of E if one or more of the arguments are changed. The approach is generic, and does not distinguish incrementality on different data types. The approach sketched in this chapter can be compared with the work of Reps, Teitelbaum, and Demers [117] on incremental attribute evaluators. They give an incremental algorithm in an interactive program development environment (a tree editor) for the evaluation of the attributes of a tree. Some of the incremental algorithms given in this paper can also be found in the Views System [113]. This chapter is based on Jeuring [75, 76].

This chapter is organised as follows. Section 7.1 discusses the definition of incremental algorithms on various data types, emphasising the data type list. Section 7.2 derives some involved examples of incremental algorithms: respectively an incremental algorithm for coding a text with respect to a dictionary, and an incremental algorithm for formatting a text (breaking a paragraph into lines). Finally, Section 7.3 gives some conclusions.

7.1 Definitions of incremental algorithms

This section defines incremental algorithms on various data types, and develops some theory for the derivation of incremental algorithms. It is divided into four subsections on respectively incremental algorithms on data types without laws; incremental algorithms on bags and sets; incremental algorithms on lists; and incremental algorithms on trees.

7.1.1 Incremental algorithms on data types without laws

Consider an initial object (L, in) in the category of F-algebras. Without any further knowledge of the type, the only means to manipulate objects of the type is via the functions in and out. In the case of *snoc-list*, this implies that the only possible modifications are appending an element a to a list x obtaining $x \mathbin{+\!\!\!<} a$, and removing the last element from a nonempty list, obtaining y from $y \mathbin{+\!\!\!<} b$. Let f be a function defined on *snoc-list*. An incremental algorithm should prescribe how to find value $f\,(x \mathbin{+\!\!\!<} a)$ given values $f\,x$ and a, and how to find value $f\,y$ given value $f\,(y \mathbin{+\!\!\!<} b)$. Speaking in terms of the data type (L, in), the first requirement on incremental algorithms amounts to showing that there exists a ϕ such that $f = (\![\phi]\!)$, and the second requirement amounts to showing that there exists a ψ such that $f\mathsf{F} \cdot out = \psi \cdot f$. Define

(7.1) Definition (Incremental algorithm) *An incremental algorithm is a pair* (f, ψ) *such that for some* ϕ

$$\begin{array}{lcl} f & = & (\![\phi]\!) \\ f\mathsf{F} \cdot out & = & \psi \cdot f \ . \end{array}$$

It follows that if $f = (\![\phi]\!)$ and ϕ has left-inverse ψ, then (f, ψ) is an incremental algorithm, since

$$\begin{array}{ll} & f\mathsf{F} \cdot out = \psi \cdot f \\ \equiv & \qquad in \cdot out = id \text{ and } out \cdot in = id \\ & f\mathsf{F} = \psi \cdot f \cdot in \\ \equiv & \qquad f = (\![\phi]\!), \text{ Catamorphism Characterisation} \\ & f\mathsf{F} = \psi \cdot \phi \cdot f\mathsf{F} \\ \equiv & \qquad \psi \cdot \phi = id \\ & true \ . \end{array}$$

Note that there may be different ψ such that (f, ψ) is an incremental algorithm as long as ψ is the left-inverse of ϕ on the image of $f\mathsf{F}$. The definition of incremental algorithms is rather restrictive: even a simple function like summing a snoc-list, $+\not\rightarrow 0$, cannot be expressed as an incremental algorithm, since the operator $+$ has no left-inverse.

Let f be a catamorphism $(\![\phi]\!)$. Suppose ϕ has no left-inverse, so that there does not exist an incremental algorithm (f, ψ). We now address the question of how to extend the computation of f in a natural way to a function g such that there exists an incremental algorithm (g, ψ). Note that each function is a paramorphism, that is, each function is catamorphic modulo id. A first attempt is to extend f to $id \vartriangle f$. We have, applying Theorem 2.47,

$$id \vartriangle f \quad = \quad (\![(in \cdot exl\mathsf{F}) \vartriangle (\phi \cdot exr\mathsf{F})]\!) \ ,$$

and if we define ψ by

$$\psi \quad = \quad (id \vartriangle f)\mathsf{F} \cdot out \cdot exl \ ,$$

then

$$(id \vartriangle f)\mathsf{F} \cdot out \quad = \quad \psi \cdot id \vartriangle f \ ,$$

and we have obtained an incremental algorithm $(id \vartriangle f, \psi)$. However, if we operationally interpret ψ, the function that satisfies $(id \vartriangle f)\mathsf{F} \cdot out = \psi \cdot id \vartriangle f$, it decomposes its argument into its constituents and recomputes the f-value of each of these components from scratch, and is therefore highly inefficient. We construct another incremental algorithm (g, ν) such that $f = \mathit{exr} \cdot g$. Suppose we store the f-values of the constituents of the argument in some structure. Then retrieving the f-values of the constituents of the argument may be done efficiently (with respect to time; the amount of space required increases). Meertens [98] gives a generic way to store all constituents of the argument. Given monofunctor F, define binary functor $\dagger$ by

$$\begin{aligned} X \dagger Y &= (X \times Y)\mathsf{F} \\ f \dagger g &= (f \times g)\mathsf{F} \, , \end{aligned}$$

and define $(M, \mathit{jn}) = \mu(\dagger\underline{L})$, where $(L, \mathit{in}) = \mu(\mathsf{F})$. Let jout be jn^{-1}. The constituents of the argument are now obtained by means of the function preds defined as a paramorphism, see (2.43), by

$$\mathit{preds} \;=\; [\![\mathit{jn}]\!] \, .$$

For example, if (L, in) is the data type *natural number*, function preds returns a list with the numbers less than the argument in ascending order, that is,

$$\mathit{preds}\, n \;=\; [0, 1, \ldots, n{-}1] \, .$$

The f-values of the constituents of an argument are obtained by means of the function $f{\star}\cdot \mathit{preds}$, where $\star$ is the map-functor corresponding to functor $(\dagger\underline{A})$, that is, for $f : A \to B$, we define $f\star : A\star \to B\star$ by $f\star = (\![\mathit{jn}_{(\dagger\underline{B})} \cdot \mathit{id}_{B\star} \dagger f]\!)_{(\dagger\underline{A})}$. The computation of $f\star \cdot \mathit{preds}$ is extended with the computation of the f-value of the argument itself (which is not contained in preds). The extended problem reads

$$g \;=\; f\star \times f \cdot \mathit{preds} \vartriangle \mathit{id} \, .$$

We show that there exists a ν such that (g, ν) is an incremental algorithm. For that purpose, we first show that g is a catamorphism $(\![\eta]\!)$. It is easily proved that paramorphism preds and function id are mutumorphisms. Applying Theorem 2.47 it follows that

$$\mathit{preds} \vartriangle \mathit{id} \;=\; (\![\mathit{jn} \vartriangle (\mathit{in} \cdot \mathit{exr}\mathsf{F})]\!) \, .$$

Apply Fusion to prove that g is a catamorphism.

$$\begin{array}{ll}
 & g = (\![\eta]\!) \\
\Leftarrow & \quad \text{Fusion} \\
 & f\star \times f \cdot \mathit{jn} \vartriangle (\mathit{in} \cdot \mathit{exr}\mathsf{F}) = \eta \cdot (f\star \times f)\mathsf{F} \\
= & \quad \text{equation (2.12), Catamorphism Characterisation for } f = (\![\phi]\!) \\
 & (f\star \cdot \mathit{jn}) \vartriangle (\phi \cdot f\mathsf{F} \cdot \mathit{exr}\mathsf{F}) = \eta \cdot (f\star \times f)\mathsf{F} \\
\Leftarrow & \quad \textbf{assume } \eta = \chi \vartriangle (\phi \cdot \mathit{exr}\mathsf{F}) \text{, equation (2.13)} \\
 & (f\star \cdot \mathit{jn}) \vartriangle (\phi \cdot f\mathsf{F} \cdot \mathit{exr}\mathsf{F}) = (\chi \cdot (f\star \times f)\mathsf{F}) \vartriangle (\phi \cdot \mathit{exr}\mathsf{F} \cdot (f\star \times f)\mathsf{F}) \\
= & \quad \text{equation (2.19)}
\end{array}$$

$$\begin{array}{ll}
& f\star \cdot jn = \chi \cdot (f\star \times f)\mathsf{F} \ \wedge \ \phi \cdot f\mathsf{F} \cdot exr\mathsf{F} = \phi \cdot exr\mathsf{F} \cdot (f\star \times f)\mathsf{F} \\
= & \quad \text{property of functor, equations (2.15) and (2.19)} \\
& f\star \cdot jn = \chi \cdot (f\star \times f)\mathsf{F} \\
= & \quad \text{definition of map} \\
& jn \cdot id \dagger f \cdot f\star \dagger id = \chi \cdot (f\star \times f)\mathsf{F} \\
= & \quad \text{property of functor, definition of } \dagger \\
& jn \cdot (f\star \times f)\mathsf{F} = \chi \cdot (f\star \times f)\mathsf{F} \\
= & \quad \textbf{assume } \chi = jn \\
& true \ .
\end{array}$$

It follows that $g = (\!|jn \vartriangle (\phi \cdot exr\mathsf{F})|\!)$. It remains to find a left-inverse ν of $jn \vartriangle (\phi \cdot exr\mathsf{F})$, that is,

$$\nu \cdot (jn \vartriangle (\phi \cdot exr\mathsf{F})) \ = \ id \ .$$

Choosing $\nu = jout \cdot exl$ suffices. We have found that (g, ν) is an incremental algorithm. Usually, the components of this incremental algorithm are more efficient than the corresponding components of the incremental algorithm $(id \vartriangle f, \psi)$.

7.1.2 Incremental algorithms on *bag*

The data types *set* and *bag* do not belong to the class of data types discussed in the previous section: the constructors of *bag* and *set* satisfy laws. We discuss incremental algorithms on the data type *bag*; incremental algorithms on the data type *set* can be dealt with analogously. Well-known incremental algorithms on *set* are algorithms that construct data structures for set-representations, such as AVL-trees. These algorithms provide efficient means with which elements can be inserted in or deleted from such a structure.

The definition of an incremental algorithm on *bag*

The first component of an incremental algorithm on the data type *bag* is required to be a catamorphism on *bag*. The second component of an incremental algorithm prescribes what to do in the case corresponding to the case $f\mathsf{F} \cdot out = \psi \cdot f$ described in the previous section. By definition of the data type *bag*, the function $\langle\,\rangle \triangledown \rho \triangledown \uplus$ is not injective, and hence a function *out* does not exist. However, we do have an operator $\ominus$ which deletes elements from a bag. Given a bag b and a bag c contained in b, operator $\ominus$ removes the elements from c in b. It is characterised by

$$b \ominus c = d \ \equiv \ d \uplus c = b \ .$$

An incremental algorithm should prescribe how to find the value of $f\,(b \ominus c)$ given the values $f\ b$ and c. The formal definition of an incremental algorithm on *bag* reads as follows.

(7.2) Definition (Incremental Algorithm on *bag***)** *An incremental algorithm on the data type* bag *is a pair* $(f, \oplus)$ *such that there exist an operator* $\odot$ *and a function* r *satisfying*

$$\begin{array}{lcl} f & = & \odot/ \cdot r* \\ f\,(x \ominus c) & = & (f\ x) \oplus c\ . \end{array}$$

Note that for an incremental algorithm $(f, \oplus)$, and for $x = y \uplus c$, section $(\oplus c)$ is the left-inverse of section $(\odot(f\ c))$ on the image of f, since

$$f\ x \;=\; f\,(y \uplus c) \;=\; (f\ y) \odot (f\ c)\ .$$

The operator $\odot$ itself need not be invertible. We call function f *incremental* if there exists an incremental algorithm $(f, \oplus)$.

Examples of incremental algorithms on *bag*

We give some examples of incremental algorithms.

- Each function $f*$ is incremental: we have $f* = \uplus/ \cdot (\rho \cdot f)*$, and $f*\,(x \ominus c) = (f*\,x) \ominus (f*\,c)$, so $(f*, \ominus \cdot id \times f*)$ is an incremental algorithm.
- Each function $p\triangleleft$ is incremental: we have $p\triangleleft = \uplus/ \cdot p\S*$, and $p\triangleleft\,(x \ominus c) = (p\triangleleft\,x) \ominus (p\triangleleft\,c)$, so $(p\triangleleft, \ominus \cdot id \times p\triangleleft)$ is an incremental algorithm.
- Given an element a, the function $a\in$ determines whether a is an element of the argument. We have $a\in\ = \vee/ \cdot (= a)*$, but we cannot find an operator $\oplus$ such that $a \in (x \ominus c) = (a \in x) \oplus c$. The function $a\in$ can be extended to the function *no a* which returns the number of occurrences of a in its argument. We have

  $$a\in \;=\; (\geq 1) \cdot no\ a\ .$$

  Function *no a* is a catamorphism $+/ \cdot f_a*$, where $f_a\ a = 1$ and $f_a\ b = 0$ for all $b \neq a$. Furthermore, $no\ a\,(x \ominus c) = (no\ a\ x) - (no\ a\ c)$, so $(no\ a, - \cdot id \times (no\ a))$ is an incremental algorithm.
- If operator $\oplus$ is such that section $(\oplus(\oplus/\ c))$ has a left-inverse $(\otimes(\oplus/\ c))$, then $(\oplus/, \otimes \cdot id \times \oplus/)$ is an incremental algorithm. Operator $+$ is such an operator.

A consequence of these results is that if $f = \odot/ \cdot r*$ and operator $\odot$ is such that section $(\odot(\odot/\, r*\, c))$ has a left-inverse $(\otimes(\odot/\, r*\, c))$ on the image of f, then

$$(f, \otimes \cdot id \times (\odot/ \cdot r*))\ , \tag{7.3}$$

is an incremental algorithm, since

$$\begin{array}{ll}
 & \odot/\, r*\, (x \ominus c) \\
= & \qquad (r*, \ominus \cdot id \times r*) \text{ is an incremental algorithm} \\
 & \odot/\, (r*\, x \ominus r*\, c) \\
= & \qquad (\odot/, \otimes \cdot id \times \odot/) \text{ is an incremental algorithm} \\
 & (\odot/\, r*\, x) \otimes (\odot/\, r*\, c)\ .
\end{array}$$

Other examples of incremental algorithms on *bag* are given by Yellin and Strom [141].

Incremental sorting

Suppose we want to sort a bag incrementally, that is, we want to construct an incremental algorithm $(sorting, \obar)$. In Section 4.2 we have derived

$$sorting \quad = \quad \oslash/ \cdot \tau*\ ,$$

where operator $\oslash$ is defined by $[\,] \oslash x = x \oslash [\,] = x$, and

$$(x \mathbin{-\!\!<} a) \oslash (y \mathbin{-\!\!<} b) \quad = \quad \begin{cases} (x \oslash (y \mathbin{-\!\!<} b)) \mathbin{-\!\!<} a & \text{if } a \geq b \\ ((x \mathbin{-\!\!<} a) \oslash y) \mathbin{-\!\!<} b & \text{otherwise}\ . \end{cases}$$

According to the remark made after the list of examples above it suffices to show that there exists an operator $\otimes$ such that $(\otimes(\oslash/\, \tau*\, c))$ is a left-inverse of $(\oslash(\oslash/\, r*\, c))$ on the image of *sorting*. Intuitively it is clear what operator $\otimes$ should do: $x \otimes c$ removes the elements of the sorted list c from the sorted list x. Define operator $\otimes$ by

$$\begin{array}{lcl}
[\,] \otimes y & = & [\,] \\
x \otimes [\,] & = & x \\
(x \mathbin{-\!\!<} a) \otimes (y \mathbin{-\!\!<} b) & = & \begin{cases} x \otimes y & \text{if } a = b \\ (x \otimes (y \mathbin{-\!\!<} b)) \mathbin{-\!\!<} a & \text{otherwise}\ . \end{cases}
\end{array}$$

Section $(\otimes(\oslash/\, \tau*\, c))$ is a left-inverse of section $(\oslash(\oslash/\, r*\, c))$ on the image of *sorting*. We prove by induction to the length of sorted list y that for all sorted lists x, $(x \oslash y) \otimes y = x$. For the base case $\#\, y = 0$, or equivalently, $y = [\,]$, we have

$$\begin{array}{ll}
 & (x \oslash y) \otimes y \\
= & \qquad \text{definition of } \oslash
\end{array}$$

$$\begin{aligned}
& x \otimes y \\
= \quad & \quad \text{definition of } \otimes \\
& x \;.
\end{aligned}$$

Suppose for all lists w with $\# w \leq n$ we have $(x \oslash w) \otimes w = x$. This is the first induction hypothesis. Let $y = z \mathbin{-\!\!\!<} b$ with $\# z = n$. We prove by induction to the length of x that

$$(x \oslash y) \otimes y \;\;=\;\; x \;.$$

In this proof we use the fact that for all lists v

$$v \otimes v \;\;=\;\; [\,]\;, \tag{7.4}$$

a result easily proved by induction to the length of v. For the base case $\# x = 0$, or equivalently, $x = [\,]$, we have

$$\begin{aligned}
& (x \oslash y) \otimes y \\
= \quad & \quad \text{definition of } \oslash \\
& y \otimes y \\
= \quad & \quad \text{equation (7.4)} \\
& [\,] \;.
\end{aligned}$$

Suppose for all lists v with $\# v = n$ we have $(v \oslash y) \otimes y = v$. This is the second induction hypothesis. Let $x = u \mathbin{-\!\!\!<} a$, with $\# u = n$. Distinguish the two cases $a > b$ (recall $b = \mathit{lt}\, y$) and $a \leq b$. Suppose $a > b$.

$$\begin{aligned}
& ((u \mathbin{-\!\!\!<} a) \oslash y) \otimes y \\
= \quad & \quad \text{definition of } \oslash,\ a > \mathit{lt}\, y \\
& ((u \oslash y) \mathbin{-\!\!\!<} a) \otimes y \\
= \quad & \quad \text{definition of } \otimes,\ a > \mathit{lt}\, y \\
& ((u \oslash y) \otimes y) \mathbin{-\!\!\!<} a \\
= \quad & \quad \text{second induction hypothesis} \\
& u \mathbin{-\!\!\!<} a \;.
\end{aligned}$$

If $a \leq b$, we use the fact that operator $\oslash$ is commutative on the domain of sorted lists, that is, if x and y are sorted, then $x \oslash y = y \oslash x$.

$$\begin{aligned}
& ((u \mathbin{-\!\!\!<} a) \oslash y) \otimes y \\
= \quad & \quad \oslash \text{ is commutative} \\
& (y \oslash (u \mathbin{-\!\!\!<} a)) \otimes y \\
= \quad & \quad \text{definition of } \oslash,\ y = z \mathbin{-\!\!\!<} b,\ a \leq b
\end{aligned}$$

$$
\begin{array}{ll}
 & ((z \oslash (u \mathbin{-\!\!\!\prec} a)) \mathbin{-\!\!\!\prec} b) \otimes (z \mathbin{-\!\!\!\prec} b) \\
= & \quad \text{definition of } \otimes \\
 & (z \oslash (u \mathbin{-\!\!\!\prec} a)) \otimes z \\
= & \quad \oslash \text{ is commutative} \\
 & ((u \mathbin{-\!\!\!\prec} a) \oslash z) \otimes z \\
= & \quad \text{first induction hypothesis} \\
 & u \mathbin{-\!\!\!\prec} a \; .
\end{array}
$$

It follows that $(x \oslash y) \otimes y = x$ for all sorted lists x and y. If operator $\obar$ is defined by $\obar = \otimes \cdot id \times (\oslash/ \cdot \tau*)$, then $(sorting, \obar)$ is an incremental algorithm.

7.1.3 Incremental algorithms on lists

Incremental algorithms on lists can be (and sometimes are) used in interactive programs such as text-editors, spreadsheet programs, etc. This section sketches an approach to incrementality on the data type *join-list*. It gives a definition of a basic incremental algorithm, and several examples of problems for which a basic incremental algorithm exists. Then it gives a more general definition of an incremental algorithm, and, using this new definition, it shows that there exists an incremental algorithm for every catamorphism on *join-list*.

The edit model

Let f be a function defined on *join-list*: $f : A* \to B$. Suppose that we want to find the f-value of a list, and that we are interactively editing this list. A description of interactive programs in a functional setting has been given by Thompson [130]. When editing a piece of data, a text, a program, or a list of numbers from a spreadsheet program, a cursor is moved through the data. Suppose the data is represented as a list. The cursor is always positioned in between two elements. If the cursor is positioned somewhere in the data, two lists can be distinguished: the part of the data in front of the position of the cursor, and the part after the position of the cursor. We want several actions to be possible:

- moving the cursor left or right;
- deleting or inserting one or more elements;
- splitting a piece of data in two;
- concatenating two pieces of data.

Note that when a piece of data is split in two pieces of data we obtain two cursors, and that when two pieces of data are concatenated one of the two cursors disappears. We are not interested in the exact positions of the cursors in these cases. This list of edit actions is our edit-model. It is incomplete, but it does comprise the basic components of an editor. Most of the other components of editors consists of compositions of these actions. After each edit-action, we want the result of f applied to the resulting list(s) immediately available. This implies that we have to adapt the interactive program we are working in. After an edit-action, the interactive program should also, besides for example showing the result of the edit-action on the screen, update the f-value(s). We now describe what should happen after each of the edit-actions.

The definition of a basic incremental algorithm

When two pieces of data, say x and y, are concatenated, the value of $f\ (x +\!\!+\ y)$ has to be determined from the values $f\ x$ and $f\ y$. The first, tentative, assumption we make about incremental algorithms is that there exists an operator $\odot$ such that $f\ (x +\!\!+ y) = (f\ x) \odot (f\ y)$. It follows that f is a *join-list* catamorphism $\odot/\cdot r*$ where function r is defined by $r\ a = f\ [a]$. This assumption is almost inevitable if we want to deal with insertion and deletion properly, but it is also reasonable. Many functions, possibly tupled with some extra information, are catamorphisms. If the data is split into two pieces of data, say again x and y, the values of $f\ x$ and $f\ y$ have to be determined from the value $f\ (x +\!\!+\ y) = (f\ x) \odot (f\ y)$. If operator $\odot$ is invertible this is easy; however, most binary associative operators are not invertible. In general, there is no other way to find the values of $f\ x$ and $f\ y$ than to compute them from scratch or to tuple the computation with the computation of the f-value of the list in front of the cursor $(f\ x)$, and the f-value of the list after the cursor $(f\ y)$. We have chosen this last option. Splitting the data in two at the point where the cursor is located is now simple: the f-values of the constituents are immediately available. Concluding, we have assumed that the interactive program is extended with the computation of a triple of values: the f-value of the list in front of the cursor, the f-value of the argument list, and the f-value of the list after the cursor.

The form of incremental algorithms described above provides an elegant way to deal with insertion and deletion of one or more elements. Suppose a list z is inserted in between the two lists x and y, so the triple

$$(f\ x\ , f\ (x +\!\!+\ y)\ , f\ y) \tag{7.5}$$

should be transformed into

$$(f\ (x +\!\!+\ z)\ , f\ (x +\!\!+\ z +\!\!+\ y)\ , f\ y)\ . \tag{7.6}$$

Note that we have assumed that after an insertion the cursor is positioned just after the last inserted element. The other choice: after an insertion the cursor is positioned just before the first inserted element, yields similar methods. To obtain triple (7.6) from triple (7.5): split $x +\!\!+\ y$ and compute $f\ z$, and then compute $(f\ x) \odot (f\ z)$ and $(f\ x) \odot (f\ z) \odot (f\ y)$. If a segment z is deleted from $x +\!\!+\ z +\!\!+\ y$ we have to construct the triple

$$(f\ x\ , f\ (x +\!\!+\ y)\ , f\ y) \tag{7.7}$$

from the triple

$$(f\ x\ , f\ (x +\!\!+\ z +\!\!+\ y)\ , f\ (z +\!\!+\ y))\ . \tag{7.8}$$

To obtain triple (7.7): first split $x +\!\!+ z +\!\!+ y$ into x and $z +\!\!+ y$ ('set the first mark'), and then split $z +\!\!+ y$ in z and y ('set the second mark'). If the setting order of marks is reversed similar methods are obtained. Since the values of $f\ x$ and $f\ y$ are now available, the triple $(f\ x\ , f\ (x +\!\!+ y)\ , f\ y)$ can be computed.

Finally, we have to deal with cursor movements. Suppose the cursor is positioned in between two lists, of which the left one is nonempty, say lists $x +\!\!+ [a]$ and y, and the cursor is moved left. Then it is required to construct the triple

$$(f\ x\ , f\ (x +\!\!+ [a] +\!\!+ y)\ , f\ ([a] +\!\!+ y)) \tag{7.9}$$

from the triple

$$(f\ (x +\!\!+ [a])\ , f\ (x +\!\!+ [a] +\!\!+ y)\ , f\ y)\ . \tag{7.10}$$

Since we assume that f is a catamorphism $\odot/ \cdot r*$, we have $f\ ([a] +\!\!+ y) = (r\ a) \odot f\ y$. Furthermore, we also have $f\ (x +\!\!+ [a]) = (f\ x) \odot (r\ a)$. So, if there exists an operator $\ominus$ such that

$$((f\ x) \odot (r\ a)) \ominus a \quad = \quad f\ x\ ,$$

then we can express $f\ x$ in terms of $f\ (x +\!\!+ [a])$ by means of $f\ x = (f\ (x +\!\!+ [a])) \ominus a$. For incremental algorithms we require the existence of such an operator $\ominus$. When the cursor is moved right it is required to construct the triple

$$(f\ (x +\!\!+ [b])\ , f\ (x +\!\!+ [b] +\!\!+ y)\ , f\ y) \tag{7.11}$$

from the triple

$$(f\ x\ , f\ (x +\!\!+ [b] +\!\!+ y)\ , f\ ([b] +\!\!+ y))\ . \tag{7.12}$$

For incremental algorithms we require the existence of an operator $\ominus$ satisfying

$$a \ominus ((r\ a) \odot (f\ x)) \quad = \quad f\ x\ .$$

(7.13) Definition (Basic incremental algorithm) *A basic incremental algorithm is a triple $(f, \ominus, \ominus)$ such that there exists an operator $\odot$ and a function r such that*

$$\begin{array}{lcl} f & = & \odot/ \cdot r* \\ ((f\ x) \odot (r\ a)) \ominus a & = & f\ x \\ a \ominus ((r\ a) \odot (f\ x)) & = & f\ x\ . \end{array}$$

We say a function f is *incremental* if there exists a basic incremental algorithm $(f, \ominus, \ominus)$.

Examples of basic incremental algorithms

We give some examples of incremental functions.

- Each function $f*$ is incremental. The triple $(f*, it \cdot exl, tl \cdot exr)$ satisfies the conditions on basic incremental algorithms.
- Each function $p\triangleleft$ is incremental. The triple $(p\triangleleft, \ominus, \circleddash)$, where operators $\ominus$ and $\circleddash$ are defined by

$$x \ominus a \;=\; \begin{cases} it\, x & \text{if } p\ a \\ x & \text{otherwise} \end{cases} \tag{7.14}$$

$$a \circleddash x \;=\; \begin{cases} tl\, x & \text{if } p\ a \\ x & \text{otherwise} \ , \end{cases} \tag{7.15}$$

  satisfies the conditions on basic incremental algorithms.
- Function $\oplus/$ is incremental, provided sections $(\oplus a)$ and $(a\oplus)$ are invertible. An example of an incremental reduction is $+/$. The basic incremental algorithm is $(+/, -, - \cdot swap)$.
- Function *parts* defined as a catamorphism $\oplus/ \cdot (\sigma \cdot \tau \cdot \tau)*$ in (4.36) is incremental. Suppose functions *git*, see (3.9), and *gtl* are defined on *join-list* instead of *snoc-list*. Then

$$\begin{aligned} parts\, x &= git{*}\, parts\, (x +\!\!+ [a]) \\ parts\, x &= gtl{*}\, parts\, ([a] +\!\!+ x)\ . \end{aligned} \tag{7.16}$$

  It follows that $(parts, git* \cdot exl, gtl* \cdot exr)$ is a basic incremental algorithm.

Fusion for basic incremental algorithms

Let $(f\ , \ominus\ , \circleddash)$ be a basic incremental algorithm, and let g be a function. We want to find conditions on g such that the existence of a basic incremental algorithm $(g \cdot f, \obar, \circledvert)$ is guaranteed.

(7.17) Theorem (Basic Incremental Algorithm Fusion) *Let $(f, \ominus, \circleddash)$ be a basic incremental algorithm, where $f = \odot/ \cdot r*$, and let function g be $(\odot, \obslash)$–fusable after f such that*

$$\begin{aligned} g\,(f\,x \ominus a) &= (g\,f\,x) \obar a \\ g\,(a \circleddash f\,x) &= a \circledvert (g\,f\,x)\ , \end{aligned}$$

for some operators $\obar$ and $\circledvert$. Then $(g \cdot f, \obar, \circledvert)$ is a basic incremental algorithm, where $g \cdot f = \obslash/ \cdot (g \cdot r)*$.

Proof Since g is $(\odot, \obslash)$–fusable after f it follows using Fusion on *join-list*, Corollary 2.62, that $g \cdot f = \obslash/ \cdot (g \cdot r)*$. Furthermore, we have by calculation

$$\begin{array}{ll}
 & ((g\,f\;x) \obslash (g\;r\;a)) \obar a \\
= & \quad g \text{ is } (\odot, \obslash)\text{–fusable after } f \\
 & (g\,((f\;x) \odot (r\;a))) \obar a \\
= & \quad (g\;y) \obar a = g\,(y \ominus a) \\
 & g\,(((f\;x) \odot (r\;a)) \ominus a) \\
= & \quad (f, \ominus, \ominus) \text{ is a basic incremental algorithm} \\
 & g\,f\;x\ .
\end{array}$$

Similarly, $a \obar ((g\;r\;a) \obslash (g\,f\;x)) = g\,f\;x$. It follows that $(g \cdot f, \oslash, \ominus)$, where $g \cdot f = \obslash/ \cdot (g \cdot r)*$ is a basic incremental algorithm. □

More examples of basic incremental algorithms

A basic incremental algorithm for the problem of finding the shortest partition into ascending lists of a string is obtained as follows. The problem is specified by

$$saap \;=\; \downarrow_{\#}/ \cdot (all\;asc)\triangleleft \cdot parts\ . \tag{7.18}$$

For example, $saap\,[1,3,2,1,4] = [[1,3],[2],[1,4]]$. Since predicate asc is segment-closed and robust, and holds for singletons, we can apply Corollary 4.79 to obtain a catamorphism on *join-list* for $saap$.

$$saap \;=\; \oslash/ \cdot (\tau \cdot \tau)*\ ,$$

where operator $\oslash$ is defined by

$$ts \oslash us \;=\; \begin{cases} \xi & \text{if } ts = \xi \;\vee\; us = \xi \\ \downarrow_{\#}/\,(all\,p)\triangleleft(ts \ominus us) & \text{if } all\,p\;ts \;\wedge\; all\,p\;us \\ \text{anything} & \text{otherwise}\ , \end{cases}$$

where operator $\ominus$ is defined in (4.38). In fact, since predicate asc holds for singletons, and since the result of $ts \oslash us$ satisfies $all\,p$ if $all\,p\;ts$ and $all\,p\;us$ hold, operator $\oslash$ satisfies

$$ts \oslash us \;=\; \begin{cases} it\;ts \mathbin{+\!\!+} [lt\;ts \mathbin{+\!\!+} hd\;us] \mathbin{+\!\!+} tl\;us & \text{if } lt\;lt\;ts \le hd\;hd\;us \\ ts \mathbin{+\!\!+} us & \text{otherwise}\ . \end{cases}$$

The other components of the basic incremental algorithm are obtained as follows. Abbreviate function $\tau \cdot \tau$ by r. By definition of operator $\oslash$ we have

$$\oslash/\;r*\;ts \oslash [[a]] \;=\; \begin{cases} it\,\oslash/\;r*\;ts \mathbin{+\!\!+} [lt\;ts \mathbin{+\!\!+} [a]] & \text{if } lt\;lt\;ts \le a \\ \oslash/\;r*\;ts \mathbin{+\!\!+} [[a]] & \text{otherwise}\ . \end{cases}$$

It follows that

$$\mathit{git}\,(\oslash/\, r{*}\, ts \oslash r\ a) \quad = \quad \oslash/\, r{*}\, ts\ ,$$

and similarly

$$\mathit{gtl}\,(r\ a \oslash \oslash/\, r{*}\, ts) \quad = \quad \oslash/\, r{*}\, ts\ .$$

Define operators $\mathbin{\bigcirc\mkern-15mu\rightarrow}$ and $\mathbin{\bigcirc\mkern-15mu\leftarrow}$ by

$$x \mathbin{\bigcirc\mkern-15mu\rightarrow} a \quad = \quad \mathit{git}\, x \tag{7.19}$$
$$a \mathbin{\bigcirc\mkern-15mu\leftarrow} x \quad = \quad \mathit{gtl}\, x\ . \tag{7.20}$$

We find that $(\oslash/ \cdot (\tau \cdot \tau){*}, \mathbin{\bigcirc\mkern-15mu\rightarrow}, \mathbin{\bigcirc\mkern-15mu\leftarrow})$ is a basic incremental algorithm for *saap*.

Without proof we note that for any segment-closed and robust predicate p that holds for singletons, the above construction yields a basic incremental algorithm for problem $\downarrow_{\#}/ \cdot (\mathit{all}\, p) \triangleleft \cdot \mathit{parts}$.

Non-incremental algorithms

An example of a function f for which there does not exist a basic incremental algorithm $(f, \otimes, \oplus)$ is the reduction $\uparrow/$. For $\uparrow/$ there do not exist operators $\oplus$ and $\otimes$ satisfying respectively $a \oplus (a \uparrow (\uparrow/\, x)) = \uparrow/\, x$ and $((\uparrow/\, x) \uparrow a) \otimes a = \uparrow/\, x$. Another problem for which no basic incremental algorithm exists, simply because there does not exist a catamorphism equal to the problem, is the maximum segment sum problem. It is easy to show that there does not exist a catamorphism for the maximim segment sum problem *mss*, see (5.22). Suppose $\mathit{mss} = \odot/ \cdot r{*}$ for some operator $\odot$ and some function r. We derive a contradiction. Let $x = [3, -1]$, $x' = [3]$, and $y = [1]$. Note that $\mathit{mss}\, x = \mathit{mss}\, x'$. We have

$$\begin{array}{ll}
 & 3 \\
= & \qquad \text{definition of } \mathit{mss} \\
 & \mathit{mss}\,(x \mathbin{+\!\!+} y) \\
= & \qquad \mathit{mss} = \odot/ \cdot r{*} \text{ (wrong assumption)} \\
 & \mathit{mss}\, x \odot \mathit{mss}\, y \\
= & \qquad \mathit{mss}\, x = \mathit{mss}\, x' \\
 & \mathit{mss}\, x' \odot \mathit{mss}\, y \\
= & \qquad \mathit{mss} = \odot/ \cdot r{*} \\
 & \mathit{mss}\,(x' \mathbin{+\!\!+} y) \\
= & \qquad \text{definition of } \mathit{mss} \\
 & 4\ .
\end{array}$$

It follows that there does not exist a catamorphism equal to *mss*. Since we do want to have incremental algorithms for these problems we generalise the definition of incremental algorithms.

The definition of an incremental algorithm

Given a function f which is required to be incremental, the interactive program in which f is computed is extended with the computation of a triple $(f\,x\,, f\,(x \mathbin{+\!\!+} y)\,, f\,y)$, where x (y) is the list in front of (after) the cursor. Instead of the triple $(f\,x\,, f\,(x \mathbin{+\!\!+} y)\,, f\,y)$ we extend the interactive program with the computation of the triple

$$(g\,x\,, f\,(x \mathbin{+\!\!+} y)\,, h\,y)\ ,$$

and we suppose there exist (efficiently computable) functions α and β such that

$$\begin{array}{lcl} f & = & \alpha \cdot g \\ f & = & \beta \cdot h\ . \end{array}$$

Furthermore, to deal with cursor movements, we suppose there exist operators $\otimes$ and $\oslash$ such that

$$\begin{array}{lcl} g\,(x \mathbin{+\!\!+} [a]) & = & g\,x \otimes a \\ (x \otimes a) \oslash a & = & x\ . \end{array}$$

Abusing the left-reduction (right-reduction) notation (since left-reductions (right-reductions) are defined on the data type *snoc-list* (*cons-list*) instead of on the data type *join-list*) we require function g to be a left-reduction $\otimes\!\not\rightarrow\! e$ such that there exists an operator $\oslash$ satisfying

$$(x \otimes a) \oslash a \ = \ x\ ,$$

and function h is a right-reduction $\oplus\!\not\leftarrow\! u$ such that there exists an operator $\ominus$ satisfying

$$a \ominus (a \oplus x) \ = \ x\ .$$

Note that g and h play a dual role.

(7.21) Definition (Incremental Algorithm) *An incremental algorithm for f is a triple (f, g, h) such that*

- *g and h generalise f, that is, there exist functions α and β such that*

$$\begin{array}{lcl} f & = & \alpha \cdot g \\ f & = & \beta \cdot h\ , \end{array}$$

- *f is a catamorphism, g is a left-reduction, and h is a right-reduction, that is, there exist operators $\odot$, $\otimes$, and $\oplus$, function r and values e and u such that*

$$\begin{aligned} f &= \odot/ \cdot r* \\ g &= \otimes\!\not\rightarrow\! e \\ h &= \oplus\!\not\leftarrow\! u \ , \end{aligned}$$

- *section $(\otimes a)$, where $\otimes$ is the operator from the left-reduction for g, has a left-inverse $(\oslash a)$ on the image of g, and section $(a\oplus)$, where $\oplus$ is the operator from the right-reduction for h, has a left-inverse $(a\ominus)$ on the image of h, that is,*

$$\begin{aligned} (g\,x \otimes a) \oslash a &= g\,x \\ a \ominus (a \oplus h\,x) &= h\,x \ . \end{aligned}$$

Properties of incremental algorithms

A basic incremental algorithm $(f, \oslash, \ominus)$, where $f = \odot/ \cdot r*$, can be extended to an incremental algorithm (f, g, h) by taking

$$\begin{aligned} g &= \otimes\!\not\rightarrow\! \nu_{\otimes} \\ h &= \oplus\!\not\leftarrow\! \nu_{\oplus} \end{aligned}$$

where operators $\otimes$ and $\oplus$ are defined by

$$\begin{aligned} x \otimes a &= x \odot (r\,a) \\ a \oplus x &= (r\,a) \odot x \ . \end{aligned}$$

In fact, for every catamorphism there exists an incremental algorithm. This is expressed by the following theorem.

(7.22) Theorem *Let f be a catamorphism. Then*

$$(f, f* \cdot \mathit{inits}, f* \cdot \mathit{tails})$$

is an incremental algorithm.

Proof Functions $f* \cdot \mathit{inits}$ and $f* \cdot \mathit{tails}$ satisfy the requirements for an incremental algorithm. First

$$\begin{aligned} \mathit{lt} \cdot f* \cdot \mathit{inits} &= f \\ \mathit{hd} \cdot f* \cdot \mathit{tails} &= f \ , \end{aligned}$$

since

$$\begin{array}{lcl} lt \cdot f* & = & f \cdot lt \\ lt \cdot inits & = & id \ , \end{array}$$

and similarly for *lt* and *inits* replaced by respectively *hd* and *tails*. Furthermore, $f* \cdot inits$ is a left-reduction $\otimes\!\not\rightarrow\! e$ such that $it\,(x \otimes a) = x$, so operator $\oslash$ may be defined by

$$x \oslash a \;=\; it\,x \ ,$$

and $f* \cdot tails$ is a right-reduction $\oplus\!\not\leftarrow\! u$ such that $tl\,(a \oplus x) = x$, so operator $\ominus$ may be defined by

$$a \ominus x \;=\; tl\,x \ .$$

□

A slight generalisation of the maximum segment problem specified in equation (5.22) by

$$mss \;=\; \uparrow/ \cdot +/* \cdot segs \ ,$$

is a catamorphism (tuple with the maximum sum among the *tails*, the maximum sum among the *inits*, and the sum of the argument list, see Smith [125]). Hence Theorem 7.22 gives an efficient incremental algorithm for finding the maximum segment sum of a list.

A fusion theorem like Basic Incremental Algorithm Fusion, Theorem 7.17, for incremental algorithms reads as follows.

(7.23) Theorem (Incremental Algorithm Fusion) *Let (f, g, h) be an incremental algorithm for f. Then $(j \cdot f, k \cdot g, l \cdot h)$ is an incremental algorithm for $j \cdot f$, provided*

- *There exist functions γ, δ, k, and l satisfying*

$$\begin{array}{lcl} j \cdot \alpha & = & \gamma \cdot k \\ j \cdot \beta & = & \delta \cdot l \ , \end{array}$$

- *There exist operators $\odot$, $\obar$ and $\oparallel$ such that function j is $(\odot, \obslash)$-fusable, and*

$$\begin{array}{lcl} k\,(x \otimes a) & = & k\,x \obar a \\ l\,(a \oplus x) & = & a \oparallel l\,x \ , \end{array}$$

- *There exist a section $(\ominus a)$ that is the left-inverse of section $(\obar a)$ on the image of k, and a section $(a \ominus)$ that is the left-inverse of section $(a \oparallel)$ on the image of l, that is,*

$$\begin{array}{lcl} (k\,x \obar a) \ominus a & = & k\,x \\ a \ominus (a \oparallel l\,x) & = & l\,x \ . \end{array}$$

Proof By definition of incremental algorithm $(j \cdot f, k \cdot g, l \cdot h)$ is an incremental algorithm for $j \cdot f$, provided

- $k \cdot g$ and $l \cdot h$ generalise $j \cdot f$. We have

$$\begin{array}{ll}
 & j \cdot f \\
= & \qquad f = \alpha \cdot g \\
 & j \cdot \alpha \cdot g \\
= & \qquad j \cdot \alpha = \gamma \cdot k \\
 & \gamma \cdot k \cdot g \ ,
\end{array}$$

  and

$$\begin{array}{ll}
 & j \cdot f \\
= & \qquad f = \beta \cdot h \\
 & j \cdot \beta \cdot h \\
= & \qquad j \cdot \beta = \delta \cdot l \\
 & \delta \cdot l \cdot h \ .
\end{array}$$

  It follows that $k \cdot g$ and $l \cdot h$ generalise $j \cdot f$,

- $j \cdot f$ is a catamorphism, $k \cdot g$ is a left-reduction, and $l \cdot h$ is a right-reduction. Since f is a catamorphism $\odot/ \cdot r*$, and function j is $(\odot, \obslash)$-fusable, it follows that $j \cdot f = \obslash/ \cdot (j \cdot r)*$. Since function g is a left-reduction $\otimes\nrightarrow e$, and since there exists an operator $\obar$ such that $k\,(x \otimes a) = k\,x \obar a$, apply Fusion on *snoc-list* to obtain

$$k \cdot \otimes\nrightarrow e \quad = \quad \obar\nrightarrow(k\ e) \ ,$$

  Similarly, since function h is a right-reduction $\oplus\nleftarrow u$, and since there exists an operator $⦷$ such that $l\,(x \oplus a) = l\,x ⦷ a$, apply Fusion on *snoc-list* to obtain

$$l \cdot \oplus\nleftarrow u \quad = \quad ⦷\nrightarrow(l\ u) \ .$$

- section $(\obar a)$ has a left-inverse $(\ominus a)$ on the range of $k \cdot g$, and section $(a ⦷)$ has a left-inverse $(a \ominus)$ on the image of $l \cdot h$. These are assumptions of the theorem.

□

7.1.4 Incremental algorithms on trees

Incremental algorithms on trees are used for example in systems for generating language-based editors. Each editor represents a file as an attributed tree of a specific attribute grammar. When editing operations modify the tree, consistent attribute values have to be reestablished throughout the tree.

Each different data type for trees, see Gibbons [57, 58] for some definitions, permits several ways of describing incremental algorithms on it. One edit-model has been given by Reps, Teitelbaum, and Demers in [117]. Important parts of an incremental algorithm on a data type *tree* are the parts that correspond to pruning a tree from a tree and grafting a tree onto a tree. Investigating the merits of the different definitions of incremental algorithms on a data type *tree* remains to be done.

7.2 Incremental algorithms for partition problems

This section derives an incremental algorithm for a class of partition problems. Consider the class of partition problems *fap* defined in equation (4.41).

$$fap \;=\; \downarrow_f / \cdot (all\, p) \triangleleft \cdot parts \;,$$

where f is an arbitrary function, and p is an arbitrary predicate. Examples of problems that belong to this class are the text formatting problem and the problem of coding a text with respect to a dictionary. This section derives an efficient incremental algorithm for a subclass of *fap* where $\downarrow_f$ and p satisfy some conditions, and it shows that the two example problems mentioned above meet these conditions.

This section is organised as follows. The first subsection specifies the problem of coding a text with respect to a dictionary. The second subsection specifies the text formatting problem. The third subsection derives conditions an element of *fap* has to meet in order to derive an incremental algorithm for it. The fourth and fifth subsection show that the two example problems meet the conditions derived in the third subsection.

7.2.1 Coding with respect to a dictionary

This section specifies the problem of coding a text with respect to a dictionary, a data compression problem.

Consider the case in which an organisation (for example Oxford University Press) maintains a large text (the law) that has to be sent to many places (all courts). The organisation edits the text regularly, and sends the updated text to the users. To speed up data transmission, a compressed version of the text is sent. For this purpose, the text has to be compressed regularly. Usually, the updated text does not differ much from its original, and for some compression methods, the same holds for their compressed versions. Hence it might pay to use an incremental algorithm for data compression.

Data compression is used in data transmission and data storage. Many aspects are described in Leweler and Hirschberg [87], and Storer [127]. An important technique for data compression is textual substitution. Textual substitution is a technique that identifies repeated substrings and replaces some or all substrings by pointers to another copy. Here we consider a specific textual substitution method: coding a text with respect to a dictionary. Suppose a dictionary is given, consisting of all 128 ASCII characters together with the 32640 most common substrings of two or more characters of printed English text. The problem of coding a text with respect to this dictionary is to partition the text into as few strings as possible, each of which is contained in the dictionary, and to replace each string in this partition by a 15 bit pointer corresponding to the entry in the dictionary.

To our knowledge, our algorithm is the first incremental algorithm for coding a text with respect to a dictionary. By means of this algorithm, the coding of a text is available immediately after it is edited. Many algorithms incrementally build a dictionary using so-called dynamic dictionary methods, see Storer [127], but no algorithms have been given for incrementally computing the coding of a text with respect to a static dictionary. Katajainen and Mäkinen [82] describe methods for incrementally coding trees, but they do not use textual substitution.

The problem of coding a text with respect to a dictionary D is specified as follows. The argument is a list of symbols. This list is partitioned into a list of lists where each of the lists in the list of lists is an element in the dictionary D. All possible partitions satisfying this condition are obtained by applying function $(\mathit{all} \in D)\triangleleft \cdot \mathit{parts}$ to the argument. The best coding is the shortest among those partitions, so the specification of the problem of coding a text with respect to a dictionary D reads as follows.

$$\mathit{coding} \;=\; \mathit{repl}{*} \cdot \mathit{cod}\ , \tag{7.24}$$

where function *cod* is defined by

$$\mathit{cod} \;=\; \downarrow_{\#}/ \cdot (\mathit{all} \in D)\triangleleft \cdot \mathit{parts}\ , \tag{7.25}$$

and *repl* is a function that replaces a word by a pointer. The precise definition of *repl* is not given. This specification can easily be implemented in a functional language, but its implementation is very inefficient. Given an incremental algorithm (cod, g, h), it is not difficult to find an incremental algorithm (coding, j, k). In the sequel we will deal with function *cod* only.

7.2.2 Formatting

This section specifies the problem of breaking a paragraph into lines (text-formatting).

When formatting a document, one of the tasks is to break each paragraph into individual lines such that the result looks nice. There are many aspects to this task. In the detailed study on the breaking of paragraphs into lines by Knuth and Plass [86], many of these aspects are treated.

One of the aspects of text-formatting is the formalisation of 'nicely looking'. Knuth and Plass [86] describe several functions, which, given a formatted text, determine the 'badness' of that solution. These functions are called waste functions. Given a waste function, it is then required to find a solution which minimises the amount of waste. On-line algorithms for this problem have been given by Knuth and Plass [86], Achugbue [1], and Bird [15].

The subsequent sections derive an incremental text-formatting algorithm. This algorithm combines two paragraphs in constant time. The text-formatting algorithm of Knuth and Plass gives different results when applied to a given text in some cases. Combining two paragraphs using their algorithm requires time linear in the length of the second paragraph.

Since two formatted paragraphs can be combined in constant time, the algorithm we obtain for formatting a paragraph is linear time, so the algorithm we obtain has the same time complexity as the on-line algorithms. Furthermore, deleting or inserting a piece of text and breaking the resulting paragraph into lines can be done in time linear in the length of the deleted or inserted piece of text. A consequence of these results is that a text can be formatted while it is edited. This facilitates WYSIWYG-editing.

We give a formal specification of the problem of breaking a paragraph into lines. Suppose a list of natural numbers is given, representing the lengths of the words in the text. It is assumed that punctuation marks are glued to the words on which they follow. This list of numbers has to be broken into lists that all fit on a given line length such that a given waste function is minimised. Suppose the line length is given by a constant C, and that all natural numbers in the input are at most this value C. For example, the following sentence from 'Pride and Prejudice' by Jane Austen

> It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

is represented by the list

$$[2,2,1,5,11,13,4,1,6,3,2,10,2,1,4,8,4,2,2,4,2,1,5]\ .$$

If $C = 25$, one of the many formattings is the following.

```
It is a truth universally
acknowledged, that a
single man in possession                                  (7.26)
of a good fortune, must
be in want of a wife .
```

All possible ways to break a list of (lengths of) words x into a list of lines of words (a paragraph) are obtained by applying function *parts* to x. Since the maximum line-length is bounded by a constant C, we filter *parts x* with predicate *all fit*, where predicate *fit* is defined by

$$\mathit{fit}\, x \;=\; (+/\, x) \leq C \;. \tag{7.27}$$

To obtain a paragraph with as little as possible space wasted apply reduction $\downarrow_{waste}/$, a definition of which is given later, to $(\mathit{all\,fit}) \triangleleft \mathit{parts}\, x$. We specify the problem to be solved, *format*, by

$$\begin{array}{lcl} \mathit{format} & : & A* \to A** \\ \mathit{format} & = & \downarrow_{waste}/ \cdot (\mathit{all\,fit}) \triangleleft \cdot \mathit{parts} \;. \end{array} \tag{7.28}$$

For the reduction $\downarrow_{waste}/$, function *waste* has to be defined, and the definition of $\downarrow_{waste}$ on objects with equal *waste*-value has to be given. For function *waste* various choices can be made. We choose one of the simplest. This choice is guided by the conditions, derived in the subsequent section, the components of our specification have to satisfy in order to derive an incremental algorithm. Other, more sophisticated, choices of *waste* are given by Bird [15] and Knuth and Plass [86]. It is not clear whether incremental algorithms can be derived if one of these other definitions of *waste* is chosen

$$\begin{array}{lcl} \mathit{waste} & = & +/ \cdot \mathit{was}* \\ \mathit{was}\, x & = & \left\{ \begin{array}{ll} C - (+/\, x) & \text{if } +/\, x \leq C \\ \infty & \text{otherwise} \;. \end{array} \right. \end{array} \tag{7.29}$$

If $C = 25$, the *waste*-value of the solution given in (7.26) is 12. For an argument x, 'function' f thus defined may have several different solutions, all with equal *waste*-value, and one of the important themes in previous papers (see Bird [15] and Fokkinga [42]) is to resolve this nondeterminism. In fact, we could specify a relation instead of a function, and thus avoid having to define $\downarrow_{waste}$ on lists with equal *waste*-value. However, we have chosen to stay within the functional framework. A derivation of one of the on-line algorithms for breaking a paragraph into lines in a relational framework is carried out by De Moor [102]. Here, the nondeterminism is resolved by trying to satisfy the conditions the selector $\downarrow_{waste}$ has to satisfy in order to give an incremental algorithm for formatting a text. This concludes the specification of the problem. If we apply the specification to Jane Austen's sentence we obtain the solution given in (7.26).

7.2.3 A derivation of an incremental algorithm for *fap*

This section constructs an incremental algorithm for *fap*, provided selector $\downarrow_f$ and predicate p satisfy some conditions. We want to use the derived incremental algorithm for the problem of coding with respect to a dictionary and for the text-formatting problem.

In Section 7.1.3 a basic incremental algorithm is derived for problems of the form $\downarrow_{\#}/\cdot(\mathit{all}\,p)\triangleleft\cdot\mathit{parts}$, where predicate p is segment-closed and robust and holds for singletons. Since the components of the problem of coding a text with respect to a dictionary, respectively predicate $\in D$ and selector $\downarrow_{\mathit{waste}}$ are, respectively, not robust and not equal to the selector $\downarrow_{\#}$, we cannot apply the construction of Section 7.1.3 to obtain a basic incremental algorithm for *fap*. We follow another approach.

The derivation of conditions the constituents $\downarrow_f$ and p of *fap* have to meet in order to be able to derive an incremental algorithm consists of three phases. In the first phase we try to derive a catamorphism that can be implemented as an efficient program for *fap*. We do not succeed, but the derivation suggests an extension of function *fap*. In this phase a condition is imposed on the components of *fap*. In the second phase we try to derive a catamorphism that can be implemented as an efficient program for this extension, called *fapg*, of function *fap*. Again, we do not succeed, but for certain predicates p the derivation suggests a restriction of function *fapg* for which we can derive a catamorphism that can be implemented as an efficient program. This derivation, together with the description of the other components of the incremental algorithm for *faph*, the restriction of function *fapg*, is given in the third phase of the derivation.

Trying to construct a catamorphism for *fap*

The first and most difficult problem that has to be solved is the construction of a catamorphism for *fap*, that is, we want to find a function $\odot/\cdot l*$, that can be implemented as an efficient program, such that

$$\downarrow_f/\cdot(\mathit{all}\,p)\triangleleft\cdot\mathit{parts} \quad=\quad \odot/\cdot l*\ .$$

To obtain a catamorphism on *join-list* for *fap*, we can apply results from the sections on *parts*-Fusion on *join-list*, see Section 4.3.4 and Section 4.3.5. If we apply *parts*-Fusion on *join-list* I, Theorem 4.71, to function *fap* we obtain a number of conditions on the components $\downarrow_f$ and p, which are not satisfied for the components of the text-formatting example and the coding with respect to a dictionary example. Hence we try to apply the results from the other section on *parts*-Fusion on *join-list*, Section 4.3.5. Section 4.3.5 does not derive a catamorphism for *fap*, but it does derive the following equality, see (4.94).

$$\mathit{fap}\,(x +\!\!+\, y) \quad=\quad \downarrow_f/\,((\mathit{fap}\times \mathit{id})*\,\mathit{splits2}\;x \;\mathbin{\raisebox{0pt}{χ}}_{\oplus}\; (\mathit{id}\times \mathit{fap})*\,\mathit{splits2}\;y)\ , \qquad (7.30)$$

provided operator $\oplus$ satisfies

$$\downarrow_f/\cdot(\mathit{all}\,p)\triangleleft\cdot\odot \quad=\quad \oplus\cdot((\downarrow_f/\cdot(\mathit{all}\,p)\triangleleft)\times \mathit{id})\times(\mathit{id}\times\downarrow_f/\cdot(\mathit{all}\,p)\triangleleft)\ ,$$

on the image of $(\mathit{parts}\times \mathit{id})\times(\mathit{id}\times\mathit{parts})$, where operator $\odot$ is defined in equation (4.90). A definition of operator $\oplus$ is obtained as follows.

$$
\begin{array}{ll}
 & {\downarrow_f}/\,(\mathit{all}\,p)\triangleleft((x,y)\odot(u,v)) \\
= & \quad \text{definition of } \odot \text{ (4.90)} \\
 & {\downarrow_f}/\,(\mathit{all}\,p)\triangleleft(x \,\chi_{\ominus_{y+\!\!+u}}\, v) \\
= & \quad \text{Cross Law (3.27)} \\
 & {\downarrow_f}/\,(x \,\chi_{(\mathit{all}\,p)?_{\downarrow_f}\cdot\ominus_{y+\!\!+u}}\, v) \\
= & \quad \text{equation (7.31) below} \\
 & {\downarrow_f}/\,(x \,\chi_{\obar_{y+\!\!+u}\cdot(\mathit{all}\,p)?_{\downarrow_f}\times(\mathit{all}\,p)?_{\downarrow_f}}\, v) \\
= & \quad \text{Cross Law (3.28)} \\
 & {\downarrow_f}/\,((\mathit{all}\,p)?_{\downarrow_f}*\,x \,\chi_{\obar_{y+\!\!+u}}\, (\mathit{all}\,p)?_{\downarrow_f}*\,v) \\
= & \quad \text{Cross Law (3.31), see proviso below} \\
 & ({\downarrow_f}/\,(\mathit{all}\,p)?_{\downarrow_f}*\,x)\obar_{y+\!\!+u}({\downarrow_f}/\,(\mathit{all}\,p)?_{\downarrow_f}*\,v) \\
= & \quad \text{definition of } \obar \text{ below} \\
 & ({\downarrow_f}/\,(\mathit{all}\,p)?_{\downarrow_f}*\,x,y)\obar(u,{\downarrow_f}/\,(\mathit{all}\,p)?_{\downarrow_f}*\,v)\ ,
\end{array}
$$

where operator $\obar$ is defined by

$$(x,y)\obar(u,v) \quad = \quad x\obar_{y+\!\!+u}v\ .$$

Equation (7.31) applied in this calculation reads as follows.

$$(\mathit{all}\,p)?_{\downarrow_f}\cdot\ominus_{y+\!\!+u} \quad = \quad \obar_{y+\!\!+u}\cdot(\mathit{all}\,p)?_{\downarrow_f}\times(\mathit{all}\,p)?_{\downarrow_f}\ . \tag{7.31}$$

Define operator $\obar_c$ by

$$x\obar_c y \quad = \quad \begin{cases} x+\!\!+[c]+\!\!+y & \text{if } p\,c\ \wedge\ x,y\neq\nu_{\downarrow_f} \\ \nu_{\downarrow_f} & \text{otherwise}\ . \end{cases} \tag{7.32}$$

It is easily verified, using the fact that predicate *all* p is segment-closed for all predicates p, that operator $\obar_c$ satisfies equation (7.31). Cross Law 3.31 that is applied in the calculation requires

sections $(\obar_c a)$ and $(b\obar_c)$ are $(\downarrow_f,\downarrow_f)$-fusable after ${\downarrow_f}/\cdot(\mathit{all}\,p)\triangleleft\cdot(\in \mathit{parts}\,u)\triangleleft$,

for all u. These are the first two conditions on the components of *fap*.

We have not been able to construct a catamorphism for *fap*, but equation (7.30) suggests that it might be possible to derive a catamorphism for the following extension of *fap*.

$$\mathit{fapg} \quad = \quad (\mathit{fap}\times\mathit{id})*\vartriangle(\mathit{id}\times\mathit{fap})*\cdot\mathit{splits2}\ .$$

We derive a catamorphism for *fapg* in the following subsection.

Constructing a catamorphism for *fapg*

Equation (7.30) and the wish to construct a catamorphism that can be implemented as a linear-time program suggest to extend the specification (7.25) of *fap* to a specification of *fapg*:

$$fapg \;=\; (fap \times id)* \vartriangle (id \times fap)* \cdot splits2 \;,$$

or, for notational reasons

$$\begin{aligned} fapg &= fiif \cdot splits2 && (7.33)\\ fiif &= (fap \times id)* \vartriangle (id \times fap)* \;. && (7.34)\end{aligned}$$

We assume that function *splits2* is a function that returns a list instead of a set. This choice for a specific form of function *splits2* will turn out to be convenient later. Function *splits2* is defined as a catamorphism on *join-list* in equation (4.80) by

$$splits2 \;=\; \oplus/ \cdot r* \;,$$

where function r is defined by $r\,a = [([\,],[a]),([a],[\,])]$, and operator $\oplus$ is defined by

$$\begin{aligned} x \oplus y &= (j\;y)*\;it\;x \mathbin{+\!\!+} (k\;x)*\;y\\ j\;y &= id \times (\mathbin{+\!\!+}\, exr\;hd\;y)\\ k\;x &= (exl\;lt\;x \mathbin{+\!\!+}) \times id \;.\end{aligned}$$

Function *fap* is expressed in terms of *fapg* by means of

$$exr \cdot hd \cdot exr \cdot fapg \;=\; fap \;=\; exl \cdot lt \cdot exl \cdot fapg \;.$$

Since $exr \cdot hd \cdot exr$ is a cheap function, the remaining task is to construct an incremental algorithm $(fapg, g, h)$ that can be implemented as an efficient program. First, we derive a catamorphism for function *fapg*. The structure of *fapg*, a function composed with a catamorphism, suggests to apply Fusion, Corollary 2.62. Applying Fusion we obtain

$$fiif \cdot splits2 \;=\; \odot/ \cdot (fiif \cdot r)* \;,$$

provided operator $\odot$ satisfies

$$fiif\;(x \oplus y) \;=\; fiif\;x \odot fiif\;y \;, \qquad (7.35)$$

for x and y in the image of *splits2*. For the composition of functions $fiif \cdot r$ we have

$$fiif\;r\;a \;=\; ([([\,],[a]),([[a]],[\,])],[([\,],[[a]]),([a],[\,])]) \;.$$

The remainder of this subsection is devoted to the synthesis of operator $\odot$ satisfying (7.35). In the synthesis of operator $\odot$ conditions are imposed upon the components of *fap*. To obtain a definition of operator $\odot$, we try to express $fiif\;(x \oplus y)$ in terms of $fiif\;x$ and $fiif\;y$, where we may assume that x are y are elements in the image of *splits2*, that is, there exist lists w and z such that

$$\begin{array}{lcl} x & = & \mathit{splits2}\ w \\ y & = & \mathit{splits2}\ z\ . \end{array}$$

Using the symmetry in the definitions of *fiif* and operator $\oplus$, we only derive an expression for $\mathit{exl}\,\mathit{fiif}\ (x \oplus y)$; subsequently we just give operator $\odot$.

$$\begin{array}{ll} & \mathit{exl}\,\mathit{fiif}\ (x \oplus y) \\ = & \quad \text{definition of } \mathit{fiif} \text{ and } \oplus \\ & \mathit{exl}\,((\mathit{fap} \times \mathit{id}){*} \vartriangle (\mathit{id} \times \mathit{fap}){*})\ ((j\ y){*}\ \mathit{it}\ x \mathbin{+\!\!+} (k\ x){*}\ y) \\ = & \quad \text{equation (2.16), definition of map} \\ & (\mathit{fap} \times \mathit{id}){*}\,(j\ y){*}\ \mathit{it}\ x \mathbin{+\!\!+} (\mathit{fap} \times \mathit{id}){*}\,(k\ x){*}\ y\ . \end{array}$$

Since $j\ y = \mathit{id} \times (\mathbin{+\!\!+}\mathit{exr}\ \mathit{hd}\ y)$ commutes with $\mathit{fap} \times \mathit{id}$ the left-hand argument of $\mathbin{+\!\!+}$ is easily expressed in terms of $\mathit{fiif}\ x$ and $\mathit{fiif}\ y$.

$$\begin{array}{ll} & (\mathit{fap} \times \mathit{id}){*}\,(j\ y){*}\ \mathit{it}\ x \\ = & \quad f{*} \cdot \mathit{it} = \mathit{it} \cdot f{*},\ j\ y \text{ commutes with } \mathit{fap} \times \mathit{id}, \text{ equation (2.9)} \\ & \mathit{it}\,(j\ y){*}\,(\mathit{fap} \times \mathit{id}){*}\,x \\ = & \quad \text{definition of } j \text{ and } \mathit{fiif} \\ & \mathit{it}\,(\mathit{id} \times (\mathbin{+\!\!+}\mathit{exr}\ \mathit{hd}\ y)){*}\ \mathit{exl}\,\mathit{fiif}\ x \\ = & \quad \text{proviso below} \\ & \mathit{it}\,(\mathit{id} \times (\mathbin{+\!\!+}\mathit{exl}\ \mathit{lt}\ \mathit{exr}\,\mathit{fiif}\ y)){*}\ \mathit{exl}\,\mathit{fiif}\ x\ . \end{array}$$

The last step in this calculation is justified, using the fact that y is an element in the image of *splits2*, as follows.

$$\begin{array}{ll} & \mathit{exr}\ \mathit{hd}\ y \\ = & \quad y = \mathit{splits2}\ z, \text{ equation (4.84)} \\ & z \\ = & \quad \text{equation (2.14)} \\ & \mathit{exl}\,(\mathit{id} \times \mathit{fap})\ (z, [\,]) \\ = & \quad (z, [\,]) = \mathit{lt}\ \mathit{splits2}\ z = \mathit{lt}\ y,\ f{*} \cdot \mathit{lt} = \mathit{lt} \cdot f \\ & \mathit{exl}\ \mathit{lt}\ (\mathit{id} \times \mathit{fap}){*}\ y \\ = & \quad \text{definition of } \mathit{fiif} \\ & \mathit{exl}\ \mathit{lt}\ \mathit{exr}\,\mathit{fiif}\ y\ . \end{array}$$

It is also possible, but much more difficult, to express the right-hand argument of $+\!\!+$ in terms of *fiif* x and *fiif* y. The resulting expression is rather complicated.

$$
\begin{array}{ll}
 & (\mathit{fap} \times \mathit{id})\!*\,(k\;x)\!*\,y \\
= & \quad \text{definition of } k \text{, equation (2.9)} \\
 & ((\mathit{fap} \cdot (\mathit{exl}\;\mathit{lt}\;x{+\!\!+})) \times \mathit{id})\!*\,y \\
= & \quad \text{equation (2.11), equation (3.40)} \\
 & (\mathit{fap} \cdot (\mathit{exl}\;\mathit{lt}\;x{+\!\!+}) \cdot \mathit{exl})\!*\,y \;\Upsilon\; \mathit{exr}\!*\,y\ .
\end{array}
$$

Consider the arguments of zip in the above expression separately. For the right-hand argument of zip, $\mathit{exr}\!*\,y$, we have

$$
\begin{array}{ll}
 & \mathit{exr}\!*\,y \\
= & \quad \text{equation (2.15), map-distributivity} \\
 & \mathit{exr}\!*\,(\mathit{fap} \times \mathit{id})\!*\,y \\
= & \quad \text{equation (2.16)} \\
 & \mathit{exr}\!*\,\mathit{exl}\;\mathit{fiif}\;y\ .
\end{array}
$$

It follows that the only remaining task for expressing $\mathit{exl}\;\mathit{fiif}\;(x \oplus y)$ in terms of *fiif* x and *fiif* y is to express the left-hand argument of zip in terms of *fiif* x and *fiif* y. For that purpose we first consider the composition of functions $\mathit{fap} \cdot (\mathit{exl}\;\mathit{lt}\;x{+\!\!+})$. Applying equation (7.30) we obtain

$$
\begin{array}{ll}
 & \mathit{fap}\;(\mathit{exl}\;\mathit{lt}\;x +\!\!+ v) \\
= & \quad \text{equation (7.30)} \\
 & \downarrow_f/\;((\mathit{fap} \times \mathit{id})\!*\,\mathit{splits2}\;\mathit{exl}\;\mathit{lt}\;x \;\mathsf{X}_{\oplus}\; (\mathit{id} \times \mathit{fap})\!*\,\mathit{splits2}\;v) \\
= & \quad \mathit{splits2}\;\mathit{exl}\;\mathit{lt}\;x = \mathit{splits2}\;w = x\text{, definition of } \mathit{fiif} \\
 & \downarrow_f/\;(\mathit{exl}\;\mathit{fiif}\;x \;\mathsf{X}_{\oplus}\; (\mathit{id} \times \mathit{fap})\!*\,\mathit{splits2}\;v)\ ,
\end{array}
$$

provided equation (7.30) is applicable, that is, provided sections $(\oplus_c a)$ and $(b \oplus_c)$, where operator $\oplus$ is defined in equation (7.32), are $(\downarrow_f, \downarrow_f)$-fusable after $\downarrow_f/\cdot(\mathit{all}\;p)\triangleleft\cdot(\in \mathit{parts}\;u)\triangleleft$ for all u. These are the conditions we imposed on the components of *fap* in the part on constructing a catamorphism for *fap*. Note that we used the fact that x is an element in the image of function *splits2* in the above calculation. For notational reasons we define

$$
k \otimes l \;=\; \downarrow_f/\;(k \;\mathsf{X}_{\oplus}\; l)\ . \qquad (7.36)
$$

For the left-hand argument of zip $(\mathit{fap} \cdot (\mathit{exl}\;\mathit{lt}\;x{+\!\!+}) \cdot \mathit{exl})\!*\,y$ we now obtain the following expression, using the above calculation for the composition of functions $\mathit{fap} \cdot (\mathit{exl}\;\mathit{lt}\;x{+\!\!+})$.

$$
(\mathit{fap} \cdot (\mathit{exl}\;\mathit{lt}\;x{+\!\!+}) \cdot \mathit{exl})\!*\,y \;=\; (\mathit{exl}\;\mathit{fiif}\;x\otimes)\!*\,((\mathit{id} \times \mathit{fap})\!* \cdot \mathit{splits2} \cdot \mathit{exl})\!*\,y\ .
$$

Finally, we express $((\mathit{id} \times \mathit{fap})\!* \cdot \mathit{splits2} \cdot \mathit{exl})\!*\,y$ in terms of *fiif* y as follows.

$$
\begin{array}{ll}
 & (id \times fap){*}{*}\, splits2{*}\, exl{*}\, y \\
= & \quad y = splits2\ z,\ \text{definition of } inits \\
 & (id \times fap){*}{*}\, splits2{*}\, inits\ z \\
= & \quad \textbf{assume } splits2{*} \cdot inits = \phi \cdot splits2,\ \phi \text{ is defined below} \\
 & (id \times fap){*}{*}\, \phi\, splits2\ z \\
= & \quad \textbf{assume } (id \times fap){*}{*} \cdot \phi = \psi \cdot (id \times fap){*},\ \psi \text{ is defined below} \\
 & \psi\, (id \times fap){*}\, splits2\ z \\
= & \quad splits2\ z = y,\ \text{definition of } fiif \\
 & \psi\ exr\ fiif\ y\ .
\end{array}
$$

Functions ϕ and ψ satisfying, respectively,

$$
\begin{aligned}
splits2{*} \cdot inits &= \phi \cdot splits2 && (7.37)\\
(id \times fap){*}{*} \cdot \phi &= \psi \cdot (id \times fap){*}\ , && (7.38)
\end{aligned}
$$

are defined as follows. Define function ϕ by

$$
\begin{aligned}
\phi\,[\,] &= [\,]\\
\phi\,(x \mathbin{+\!\!+} [a]) &= (id \times it){*}{*}\, \phi\, x \mathbin{+\!\!+} [x \mathbin{+\!\!+} [a]]\ ,
\end{aligned}
$$

then equation (7.37) is satisfied. The simple proof of this fact is left to the reader. Suppose function *fap* satisfies

$$
fap \cdot it = git \cdot fap\ , \qquad (7.39)
$$

then function ψ defined by

$$
\begin{aligned}
\psi\,[\,] &= [\,]\\
\psi\,(v \mathbin{+\!\!+} [d]) &= (id \times git){*}{*}\, \psi\, v \mathbin{+\!\!+} [v \mathbin{+\!\!+} [d]]\ ,
\end{aligned}
$$

satisfies equation (7.38). The straightforward proof is left to the reader.

We collect the results showing that $exl\ fiif\ (x \oplus y)$ can be expressed in terms of $fiif\ x$ and $fiif\ y$ together. Suppose that sections $(\oplus_c a)$ and $(b \oplus_c)$ are $(\downarrow_f, \downarrow_f)$-fusable after $\downarrow_f / \cdot (all\ p) \triangleleft \cdot (\in parts\ u) \triangleleft$ for all u, and suppose function *fap* satisfies equation (7.39). Then

$$
\begin{aligned}
exl\ fiif\ (x \oplus y) &= it\,(id \times (\mathbin{+\!\!+} exl\ lt\ u)){*}\, v' \mathbin{+\!\!+} ((v' \otimes){*}\, \psi\, u\ \Upsilon\ exr{*}\, u')\\
&\quad \textbf{where } (u', u) = fiif\ y\\
&\qquad\qquad (v', v) = fiif\ x\ .
\end{aligned}
$$

Similarly, for $exr\ fiif\ (x \oplus y)$ we have, assuming that

$$
fap \cdot tl = gtl \cdot fap
$$

and defining ψ' by

$$\begin{array}{lcl} \psi'\,[\,] & = & [\,] \\ \psi'\,[[d] \mathbin{+\!\!+} v] & = & [[d] \mathbin{+\!\!+} v] \mathbin{+\!\!+} (\mathit{gtl} \times \mathit{id}){*}{*}\,\psi'\,v\ , \end{array}$$

that

$$\begin{array}{lcl} \mathit{exr}\,\mathit{fiif}\,(x \oplus y) & = & (\mathit{exl}{*}\,v \curlyvee (\otimes u){*}\,\psi'\,v') \mathbin{+\!\!+} \mathit{tl}\,((\mathit{exr}\,\mathit{hd}\,v' \mathbin{+\!\!+}) \times \mathit{id}){*}\,u \\ & & \textbf{where } (u', u) = \mathit{fiif}\ y \\ & & \qquad\quad (v', v) = \mathit{fiif}\ x\ , \end{array}$$

resulting in the following definition of operator $\odot$.

$$\begin{array}{lcll} (v', v) \odot (u', u) & = & (s, t) & (7.40) \\ & & \textbf{where} & \\ & & s = \mathit{it}\,(\mathit{id} \times (\mathbin{+\!\!+} \mathit{exl}\,\mathit{lt}\,u)){*}\,v' \mathbin{+\!\!+} ((v'\otimes){*}\,\psi\,u \curlyvee \mathit{exr}{*}\,u') & (7.41) \\ & & t = (\mathit{exl}{*}\,v \curlyvee (\otimes u){*}\,\psi'\,v') \mathbin{+\!\!+} \mathit{tl}\,((\mathit{exr}\,\mathit{hd}\,v' \mathbin{+\!\!+}) \times \mathit{id}){*}\,u\ , & (7.42) \end{array}$$

and

$$\mathit{fapg} \quad = \quad \odot/ \cdot (\mathit{fiif} \cdot r){*}\ .$$

The catamorphism $\odot/ \cdot (\mathit{fiif} \cdot r){*}$ we have obtained for *fapg* cannot be implemented as a linear-time program, because the evaluations of ψ, ψ', $\otimes$, and zip require time at least linear in the length of their arguments. This comes as no surprise, since the length of the result of *fapg* is quadratic in the length of the argument. To obtain a catamorphism that can be implemented as a linear-time program, we specify yet another function *faph* which is closely related to function *fapg*.

Constructing a incremental algorithm for *faph*

A predicate p is called *length-bounded* if there is an upperbound on the lengths of the lists that satisfy p, i.e., if there exists an L such that

$$p\,x \quad \Rightarrow \quad \#\,x < L\ .$$

An example of a length-bounded predicate is the predicate $\in D$, where D is a set of lists. In this case take $L = \uparrow/\,\#{*}\,D + 1$. Another example of a length-bounded predicate is the predicate *fit* defined in (7.27) if we assume that all natural numbers in the argument list are positive. For this example L may be defined by $L = C + 1$.

Suppose p is a length-bounded predicate. Consider equation (7.30).

$$\mathit{fap}\,(x \mathbin{+\!\!+} y) \quad = \quad \downarrow_f/\,((\mathit{fap} \times \mathit{id}){*}\,\mathit{splits2}\,x \between_{\oplus} (\mathit{id} \times \mathit{fap}){*}\,\mathit{splits2}\,y)\ ,$$

where operator $\obar$ is defined by

$$(x, y) \obar (u, v) \quad = \quad \begin{cases} x \mathbin{+\!\!+} [y \mathbin{+\!\!+} u] \mathbin{+\!\!+} v & \text{if } p\,(y \mathbin{+\!\!+} u) \\ \nu_{\downarrow_f} & \text{otherwise} \ . \end{cases}$$

Since predicate p is length-bounded all elements (a, b) of *exl fapg* x with $\# \, b \geq L$ can be discarded since $b \mathbin{+\!\!+} c$ is too long to satisfy p, and $\obar$ evaluates to $\nu_{\downarrow_f}$ for these elements. Similarly, all elements (c, d) of *exr fapg* y with $\# \, c \geq L$ can be discarded. It follows that operator $\otimes$ defined in equation (7.36) satisfies

$$u \otimes v \quad = \quad (L \leftharpoonup u) \otimes (L \rightharpoonup v) \ , \tag{7.43}$$

for $u = \mathit{exl\,fapg}\,x$ and $v = \mathit{exr\,fapg}\,y$ for some x and y. Hence for the computation of *fap* it suffices to have the last L elements of *exl fapg* x and the first L elements of *exr fapg* y available. Function *faph* is specified in terms of function *fapg* as follows.

$$\mathit{faph} \quad = \quad (L \leftharpoonup) \times (L \rightharpoonup) \cdot \mathit{fapg} \ . \tag{7.44}$$

Function *fap* is expressed in terms of *faph* by means of

$$\mathit{exr} \cdot \mathit{hd} \cdot \mathit{exr} \cdot \mathit{faph} \quad = \quad \mathit{fap} \quad = \quad \mathit{exl} \cdot \mathit{lt} \cdot \mathit{exl} \cdot \mathit{faph} \ .$$

This subsection consists of five parts. The first part derives a catamorphism for function *faph*. The second part presents a function h that is a right-reduction $\oplus\!\not\leftarrow\! u$, such that there exists a function β, which is constructed in the third part, satisfying $\mathit{faph} = \beta \cdot h$. The fourth part constructs an operator $\ominus$ satisfying $a \ominus (a \oplus h\,x) = h\,x$. Finally, the fifth part presents the complete incremental algorithm for *faph*.

Constructing a catamorphism for *faph*

This part derives a catamorphism for function *faph*. Since under suitable assumptions *fapg* is a catamorphism $\odot/ \cdot (\mathit{fiif} \cdot r)*$, apply Fusion to obtain a catamorphism for *faph*. We have

$$\mathit{faph} \quad = \quad \oslash/ \cdot ((L \leftharpoonup) \times (L \rightharpoonup) \cdot \mathit{fiif} \cdot r)* \ ,$$

provided

$$((L \leftharpoonup) \times (L \rightharpoonup))\,((v', v) \odot (u', u)) \quad = \quad (L \leftharpoonup v', L \rightharpoonup v) \oslash (L \leftharpoonup u', L \rightharpoonup u) \ ,$$

for (v', v) and (u', u) in the image of *fapg*. We assume that $L \geq 2$, and hence, by definition of r we have

$$(L \leftharpoonup) \times (L \rightharpoonup) \cdot \mathit{fiif} \cdot r \quad = \quad \mathit{fiif} \cdot r \ .$$

By definition of operator $\odot$,

$$((L \leftharpoonup) \times (L \rightharpoonup))\,((v', v) \odot (u', u)) \quad = \quad (L \leftharpoonup s, L \rightharpoonup t)\ ,$$

where s and t are defined in respectively (7.41) and (7.42). We express both $L \leftharpoonup s$ and $L \rightharpoonup t$ in terms of $L \leftharpoonup v'$, $L \rightharpoonup v$, $L \leftharpoonup u'$, and $L \rightharpoonup u$. Expression $L \leftharpoonup s$ is expressed in terms of these four components as follows. Distinguish the two cases $\# u \geq L$ and $\# u < L$.

Case $\# u \geq L$ or equivalently $\# (L \rightharpoonup u) = L$.

$$\begin{array}{ll}
 & L \leftharpoonup s \\
= & \quad \text{definition of } s \\
 & L \leftharpoonup (it\,(id \times (+\!\!+ exl\ lt\ u))* v' +\!\!+ ((v'\otimes)* \psi\, u \,\Upsilon\, exr* u')) \\
= & \quad \# \psi\, u \geq \# u \geq L, \text{ definition of } \leftharpoonup \\
 & L \leftharpoonup ((v'\otimes)* \psi\, u \,\Upsilon\, exr* u') \\
= & \quad \text{property (3.41) of } \Upsilon \\
 & (L \leftharpoonup (v'\otimes)* \psi\, u) \,\Upsilon\, (L \leftharpoonup exr* u')\ .
\end{array}$$

The arguments of Υ are dealt with separately. For the right-hand argument of zip we use the fact that $\leftharpoonup$ commutes with $f*$ for all functions f, see (3.15), to prove that

$$L \leftharpoonup exr* u' \quad = \quad exr* (L \leftharpoonup u')\ .$$

For the left-hand argument of Υ we have

$$\begin{array}{ll}
 & L \leftharpoonup (v'\otimes)* \psi\, u \\
= & \quad \text{property (7.43) of } \otimes \\
 & L \leftharpoonup (((L \leftharpoonup v')\otimes) \cdot (L \rightharpoonup))* \psi\, u \\
= & \quad \text{map-distributivity} \\
 & L \leftharpoonup ((L \leftharpoonup v')\otimes)* (L \rightharpoonup)* \psi\, u \\
= & \quad \leftharpoonup \text{ commutes with map} \\
 & ((L \leftharpoonup v')\otimes)* (L \leftharpoonup (L \rightharpoonup)* \psi\, u) \\
= & \quad \text{Lemma 7.45 below} \\
 & ((L \leftharpoonup v')\otimes)* \gamma\, L\, (L \rightharpoonup u)\ ,
\end{array}$$

where Lemma 7.45 reads

(7.45) Lemma *For v in the image of $exr \cdot fapg$ we have*

$$N \leftharpoonup (L \rightharpoonup)* \psi\, v \quad = \quad \gamma\, N\, (L \rightharpoonup v)\ ,$$

where function γ is defined by

$$\gamma\, n\, v \;=\; \begin{cases} [\,] & \mathit{if}\ n = 0 \ \vee\ v = [\,] \\ ((\mathit{id} \times \mathit{git}){*}{*}\, \gamma\, (n{-}1)\, v) \mathbin{+\!\!+} [v] & \mathit{if\ exr\ lt}\ v \neq [\,] \\ ((\mathit{id} \times \mathit{git}){*}{*}\, \gamma\, (n{-}1)\, (\mathit{it}\ v)) \mathbin{+\!\!+} [v] & \mathit{otherwise}\ . \end{cases}$$

Proof by induction to N. Note first that if $N = 0$ or $v = [\,]$, then the equality holds by definition of functions ψ and γ. This proves the base case. Suppose $v \neq [\,]$. The induction hypothesis is

$$N \leftharpoonup (L \rightharpoonup){*}\, \psi\, v \;=\; \gamma\, N\, (L \rightharpoonup v)\ .$$

Calculate as follows.

$$\begin{array}{ll}
 & (N{+}1) \leftharpoonup (L \rightharpoonup){*}\, \psi\, v \\
= & \qquad v = w \mathbin{+\!\!+} [a]\text{, definition of } \psi \\
 & (N{+}1) \leftharpoonup (L \rightharpoonup){*}\, ((\mathit{id} \times \mathit{git}){*}{*}\, \psi\, w \mathbin{+\!\!+} [w \mathbin{+\!\!+} [a]]) \\
= & \qquad \text{definition of map and } \leftharpoonup \\
 & (N \leftharpoonup (L \rightharpoonup){*}\, (\mathit{id} \times \mathit{git}){*}{*}\, \psi\, w) \mathbin{+\!\!+} [L \rightharpoonup (w \mathbin{+\!\!+} [a])] \\
= & \qquad \text{properties of } \leftharpoonup \text{ and } \rightharpoonup \\
 & (\mathit{id} \times \mathit{git}){*}{*}\, (N \leftharpoonup (L \rightharpoonup){*}\, \psi\, w) \mathbin{+\!\!+} [L \rightharpoonup (w \mathbin{+\!\!+} [a])] \\
= & \qquad \text{induction hypothesis} \\
 & ((\mathit{id} \times \mathit{git}){*}{*}\, \gamma\, N\, (L \rightharpoonup w)) \mathbin{+\!\!+} [L \rightharpoonup (w \mathbin{+\!\!+} [a])]\ .
\end{array}$$

We may assume that v is an element in the image of function $\mathit{exr} \cdot \mathit{fapg}$, that is, $v = (\mathit{id} \times \mathit{fap}){*}\, \mathit{splits2}\ z$ for some list z. Distinguish the two cases $\#\, v > L$ and $\#\, v \leq L$. If $\#\, v > L$, then $L \rightharpoonup w = L \rightharpoonup (w \mathbin{+\!\!+} [a])$, and $\mathit{exr\ lt}\ L \rightharpoonup (w \mathbin{+\!\!+} [a]) \neq [\,]$. Hence

$$\begin{array}{ll}
 & ((\mathit{id} \times \mathit{git}){*}{*}\, \gamma\, N\, (L \rightharpoonup w)) \mathbin{+\!\!+} [L \rightharpoonup (w \mathbin{+\!\!+} [a])] \\
= & \qquad \text{case assumption} \\
 & ((\mathit{id} \times \mathit{git}){*}{*}\, \gamma\, N\, (L \rightharpoonup (w \mathbin{+\!\!+} [a]))) \mathbin{+\!\!+} [L \rightharpoonup (w \mathbin{+\!\!+} [a])] \\
= & \qquad \mathit{exr}\ a \neq [\,]\text{, definition of } \gamma \\
 & \gamma\, (N{+}1)\, (L \rightharpoonup (w \mathbin{+\!\!+} [a]))\ .
\end{array}$$

If $\#\, v \leq L$, then $\mathit{exr\ lt}\, (L \rightharpoonup (w \mathbin{+\!\!+} [a])) = \mathit{exr}\ a = [\,]$, and hence

$$\begin{array}{ll}
 & ((\mathit{id} \times \mathit{git}){*}{*}\, \gamma\, N\, (L \rightharpoonup w)) \mathbin{+\!\!+} [L \rightharpoonup (w \mathbin{+\!\!+} [a])] \\
= & \qquad \text{case assumption}
\end{array}$$

$$\begin{array}{ll}
& ((id \times git){*}{*}\, \gamma\, N\, w) \mathbin{+\!\!+} [w \mathbin{+\!\!+} [a]] \\
= & \qquad exr\; a = [\,],\ \text{definition of } \gamma \\
& \gamma\, (N{+}1)\, (w \mathbin{+\!\!+} [a]) \\
= & \qquad \text{definition of } \rightharpoonup \\
& \gamma\, (N{+}1)\, (L \rightharpoonup (w \mathbin{+\!\!+} [a]))\ .
\end{array}$$

This proves the induction step. □

Case $\#\,(L \rightharpoonup u) < L$. Calculate as follows.

$$\begin{array}{ll}
& L \leftharpoonup s \\
= & \qquad \text{definition of } s \\
& L \leftharpoonup (it\, (id \times (\mathbin{+\!\!+} exl\; lt\; u)){*}\, v' \mathbin{+\!\!+} ((v' \otimes){*}\, \psi\, u \;\Upsilon\; exr{*}\, u')) \\
= & \qquad \#\,(L \leftharpoonup u) < L,\ \text{definition of } \leftharpoonup \\
& ((L{-}\#\, u) \leftharpoonup it\, (id \times (\mathbin{+\!\!+} exl\; lt\; u)){*}\, v') \mathbin{+\!\!+} ((v' \otimes){*}\, \psi\, u \;\Upsilon\; exr{*}\, u')
\end{array}$$

Since $\#\,(L \rightharpoonup u) < L$ all occurrences of u and u' may be replaced by $L \rightharpoonup u$ and $L \rightharpoonup u'$, respectively. Furthermore, by definition of $\leftharpoonup$,

$$(L{-}\#\, u) \leftharpoonup it\, (id \times (\mathbin{+\!\!+} q)){*}\, v' \;=\; (L{-}\#\, u) \leftharpoonup it\, (id \times (\mathbin{+\!\!+} q)){*}\, (L \leftharpoonup v')\ ,$$

where $q = exl\; lt\; u$. For the right-hand argument of zip we apply equation (7.43) for $\otimes$ to replace $(v'\otimes)$ by $((L \leftharpoonup v')\otimes)$. This concludes the case $\#\,(L \rightharpoonup u) < L$. In both cases depending on the length of $L \rightharpoonup u$, we have expressed $L \leftharpoonup s$ in terms of $L \leftharpoonup u'$, $L \rightharpoonup u$, $L \leftharpoonup v$, and $L \rightharpoonup v$. Expression $L \rightharpoonup t$ is expressed in terms of these four components similarly. We have obtained the following definition of operator $\oslash$.

$$(v', v) \oslash (u', u) \;=\; (s, t)\ ,$$

where the components s and t are the following complicated expressions.

$$\begin{array}{lll}
s & = & \left\{ \begin{array}{ll} (v'\otimes){*}\, \gamma\, L\, u \;\Upsilon\; exr{*}\, u' & \text{if } \#\, u = L \\ ((L{-}\#\, u) \leftharpoonup it\, (id \times (\mathbin{+\!\!+} q)){*}\, v') \mathbin{+\!\!+} ((v'\otimes){*}\, \psi\, u \;\Upsilon\; exr{*}\, u') & \text{if } \#\, u < L \end{array} \right. \\
& & \textbf{where } q = exl\; lt\; u \\
t & = & \left\{ \begin{array}{ll} exl{*}\, v \;\Upsilon\; (\otimes u){*}\, \gamma'\, L\, v' & \text{if } \#\, v' = L \\ (exl{*}\, v \;\Upsilon\; (\otimes u){*}\, \psi'\, v') \mathbin{+\!\!+} ((L{-}\#\, v') \rightharpoonup tl\, ((r \mathbin{+\!\!+}) \times id){*}\, u) & \text{if } \#\, v' < L \end{array} \right. \\
& & \textbf{where } r = exr\; hd\; v'\ ,
\end{array}$$

where function γ', the counterpart of function γ, is defined by

$$\gamma'\, n\, v \;=\; \left\{ \begin{array}{ll} [\,] & \text{if } v = [\,] \;\vee\; n = 0 \\ ((gtl \times id){*}{*}\, \gamma'\, (n{-}1)\, v) \mathbin{+\!\!+} [v] & \text{if } exl\; hd\; v \neq [\,] \\ ((gtl \times id){*}{*}\, \gamma'\, (n{-}1)\, (tl\; v)) \mathbin{+\!\!+} [v] & \text{otherwise}\ . \end{array} \right.$$

We claim that each evaluation of operator $\oslash$ requires constant time when implemented, and hence that the catamorphism for *faph* requires linear time when implemented. Note that $\# \, u$, $\# \, u'$, $\# \, v$, $\# \, v' \leq L$, so evaluating the maps and zips in s and t requires constant time. Furthermore, since the arguments of $\otimes$ have length at most L, evaluating $\otimes$ requires at most time L^2, provided $\downarrow_f$ can be evaluated in time linear in its argument.

To obtain an incremental algorithm (faph, g, h), functions g and h should satisfy the following. Function g is a left-reduction $\ominus\!\not\rightarrow\! e$ such that there exists an operator $\ominus$ satisfying

$$(((\ominus\!\not\rightarrow\! e)\, x) \ominus a) \ominus a \quad = \quad (\ominus\!\not\rightarrow\! e)\, x \ ,$$

and there exists a function α such that $\mathit{faph} = \alpha \cdot g$. Function h is a right-reduction $\oplus\!\not\leftarrow\! u$ such that there exists an operator $\ominus$ satisfying

$$a \ominus (a \oplus ((\oplus\!\not\leftarrow\! u)\, x)) \quad = \quad (\oplus\!\not\leftarrow\! u)\, x \ ,$$

and there exists a function β such that $\mathit{faph} = \beta \cdot h$. For an efficient incremental algorithm the components α, $\ominus$, $\ominus$, β, $\oplus$, $\ominus$ should be such that their implementations can be evaluated in constant time.

An obvious candidate for both function g and function h is function *faph* itself. The characterisation of *faph* as a catamorphism immediately provides characterisations of *faph* as a left-reduction and as a right-reduction. Functions α and β are the identity function. The only problem is the definition of the operators $\ominus$ and $\ominus$. For our examples, we have not been able to construct an operator $\ominus$ that, given the codings of the longest L tails of $[a] +\!\!+\, y$, returns the codings of the longest L tails of y, and that can be implemented as a function that can be evaluated in constant time. Instead of function *faph* we take the following variants of function *fapg* as the other components of the incremental algorithm. Functions g and h that do satisfy the conditions imposed upon incremental algorithms are specified by

$$g \quad = \quad (\mathit{exl}*) \times (L \rightharpoonup) \cdot \mathit{fapg} \qquad (7.46)$$

$$h \quad = \quad (L \leftharpoonup) \times (\mathit{exr}*) \cdot \mathit{fapg} \ . \qquad (7.47)$$

We discuss the equations h is supposed to satisfy.

Constructing a left-reduction for h

This part constructs an operator $\oplus$ satisfying

$$h\,([a] +\!\!+\, z) \quad = \quad a \oplus h\, z \ .$$

A consequence is that $h = \oplus\!\not\leftarrow\!(h\,[\,])$. Recall that *fapg*, in terms of which h is defined, is a catamorphism $\odot/ \cdot (\mathit{fiif} \cdot r)*$. Consider the left and the right component of h separately.

$$\begin{array}{ll}
 & \mathit{exl}\, h\,([a] +\!\!+\, z) \\
= & \quad \text{definition of } h \text{ and } \mathit{faph} \\
 & \mathit{exl}\, \mathit{faph}\,([a] +\!\!+\, z) \\
= & \quad \mathit{faph} = \oslash/ \cdot (\mathit{fiif} \cdot r)* \\
 & \mathit{exl}\,(\mathit{fiif}\ r\ a \oslash \mathit{faph}\, z) \\
= & \quad \text{introduction of } \beta \\
 & \mathit{exl}\,(\mathit{fiif}\ r\ a \oslash \beta\, h\, z)\ ,
\end{array}$$

where we assume that function β satisfies $\mathit{faph} = \beta \cdot h$. A definition of function β is constructed in the subsequent part. The above calculation shows that the left-component of $h\,([a] +\!\!+\, z)$ can be expressed in terms of a and $h\, z$. For the right component of $h\,([a] +\!\!+\, z)$ we calculate as follows.

$$\begin{array}{ll}
 & \mathit{exr}\, h\,([a] +\!\!+\, z) \\
= & \quad \text{definition of } h \\
 & \mathit{exr}*\, \mathit{exr}\, \mathit{fapg}\,([a] +\!\!+\, z) \\
= & \quad \mathit{fapg} = \odot/ \cdot (\mathit{fiif} \cdot r)* \\
 & \mathit{exr}*\, \mathit{exr}\,(\mathit{fiif}\ r\ a \odot \mathit{fapg}\, z)\ .
\end{array}$$

Let $(v', v) = \mathit{fiif}\ r\ a$, $(u', u) = \mathit{fapg}\, z$, and let s and t be defined in, respectively, (7.41) and (7.42). Then

$$\begin{array}{ll}
 & \mathit{exr}*\, \mathit{exr}\,(\mathit{fiif}\ r\ a \odot \mathit{fapg}\, z) \\
= & \quad \text{introduced abbreviations} \\
 & \mathit{exr}*\,((\mathit{exl}*\, v\ \Upsilon\ (\otimes u)*\, \psi'\, v') +\!\!+\, \mathit{tl}\,((\mathit{exr}\, \mathit{hd}\, v' +\!\!+) \times \mathit{id})*\, u) \\
= & \quad \text{definition of map, equation (3.4)} \\
 & \mathit{exr}*\,(\mathit{exl}*\, v\ \Upsilon\ (\otimes u)*\, \psi'\, v') +\!\!+\, \mathit{tl}\ \mathit{exr}*\,((\mathit{exr}\, \mathit{hd}\, v' +\!\!+) \times \mathit{id})*\, u\ .
\end{array}$$

The right-hand argument $\mathit{tl}\ \mathit{exr}*\,((\mathit{exr}\, \mathit{hd}\, v' +\!\!+) \times \mathit{id})*\, u$ of $+\!\!+$ is expressed in terms of $h\, z$ by means of the following calculation.

$$\begin{array}{ll}
 & \mathit{tl}\ \mathit{exr}*\,((\mathit{exr}\, \mathit{hd}\, v' +\!\!+) \times \mathit{id})*\, u \\
= & \quad \text{map-distributivity, equation (2.15)} \\
 & \mathit{tl}\ \mathit{exr}*\, u \\
= & \quad \text{definition of } u \\
 & \mathit{tl}\ \mathit{exr}*\, \mathit{exr}\, \mathit{fapg}\, z \\
= & \quad \text{equation (2.15)}
\end{array}$$

$$\begin{array}{ll}
 & \mathit{tl}\ \mathit{exr}\ ((L \leftharpoonup) \times (\mathit{exr}*))\ \mathit{fapg}\ z \\
= & \quad \text{definition of } h \\
 & \mathit{tl}\ \mathit{exr}\ h\ z\ .
\end{array}$$

For the left-hand argument $\mathit{exr}*\,(\mathit{exl}*\,v\ \Upsilon\ (\otimes u)*\,\psi'\ v')$ of $+\!\!+$ we calculate as follows.

$$\begin{array}{ll}
 & \mathit{exr}*\,(\mathit{exl}*\,v\ \Upsilon\ (\otimes u)*\,\psi'\ v') \\
= & \quad \text{law 3.38} \\
 & (\otimes u)*\,\psi'\ v' \\
= & \quad \text{equation (7.43)} \\
 & ((\otimes(L \rightharpoonup u)) \cdot (L \leftharpoonup))*\,\psi'\ v' \\
= & \quad \text{definition of } u\text{, map-distributivity} \\
 & (\otimes(L \rightharpoonup \mathit{exr}\ \mathit{fapg}\ z))*\,(L \leftharpoonup)*\,\psi'\ v' \\
= & \quad \text{equation (2.15)} \\
 & (\otimes(\mathit{exr}\ ((L \rightharpoonup) \times (L \leftharpoonup))\ \mathit{fapg}\ z))*\,(L \leftharpoonup)*\,\psi'\ v' \\
= & \quad \text{definition of } \mathit{faph} \\
 & (\otimes(\mathit{exr}\ \mathit{faph}\ z))*\,(L \leftharpoonup)*\,\psi'\ v' \\
= & \quad \text{definition of } h \text{ and } \beta \\
 & (\otimes(\mathit{exr}\ \beta\ h\ z))*\,(L \leftharpoonup)*\,\psi'\ v' \\
= & \quad L \geq 2 = \#\,v' \\
 & (\otimes(\mathit{exr}\ \beta\ h\ z))*\,\psi'\ v'\ .
\end{array}$$

It follows that operator $\oplus$ is defined by

$$\begin{array}{lll}
a \oplus z & = & (\mathit{exl}\ ((v', v) \oslash \beta\ z), (\otimes \mathit{exr}\ \beta\ z)*\,\psi'\ v' +\!\!+ \mathit{tl}\ \mathit{exr}\ z) \\
 & & \quad \textbf{where } (v', v) = \mathit{fiif}\ r\ a\ .
\end{array}$$

Constructing a function β such that $\mathit{faph} = \beta \cdot h$

This part constructs a function β such that $\mathit{faph} = \beta \cdot h$. The definition of β is calculated as follows.

$$\begin{array}{ll}
 & \mathit{faph} = \beta \cdot h \\
= & \quad \text{definition of } h\text{, definition of } \mathit{faph} \\
 & (L \leftharpoonup) \times (L \rightharpoonup) \cdot \mathit{fapg} = \beta \cdot (L \leftharpoonup) \times (\mathit{exr}*) \cdot \mathit{fapg} \\
\Leftarrow & \quad \textbf{assume } \beta = \mathit{id} \times ((L \rightharpoonup) \cdot \eta)
\end{array}$$

$$\begin{array}{ll}
& \mathit{exr} \cdot \mathit{fapg} = \eta \cdot \mathit{exr}* \cdot \mathit{exr} \cdot \mathit{fapg} \\
= & \quad \text{definition of } \mathit{fapg}\text{, map-distributivity, equations (2.17), (2.15)} \\
& (\mathit{id} \times \mathit{fap})* \cdot \mathit{splits2} = \eta \cdot (\mathit{fap} \cdot \mathit{exr})* \cdot \mathit{splits2} \\
= & \quad \text{equations (2.11), (3.40)} \\
& \Upsilon \cdot \mathit{exl}* \vartriangle (\mathit{fap} \cdot \mathit{exr})* \cdot \mathit{splits2} = \eta \cdot (\mathit{fap} \cdot \mathit{exr})* \cdot \mathit{splits2} \\
\Leftarrow & \quad \textbf{assume } \eta = \Upsilon \cdot \xi \vartriangle \mathit{id} \\
& \mathit{exl}* \cdot \mathit{splits2} = \xi \cdot (\mathit{fap} \cdot \mathit{exr})* \cdot \mathit{splits2} \\
= & \quad \text{map-distributivity, equations (5.4), (5.5)} \\
& \mathit{inits} = \xi \cdot \mathit{fap}* \cdot \mathit{tails} \\
= & \quad \mathit{ghd} \cdot \mathit{fap} = \mathit{hd},\ \mathit{it} \cdot \mathit{hd}* \cdot \mathit{tails} = \mathit{id} \\
& \xi = \mathit{inits} \cdot \mathit{it} \cdot \mathit{ghd}* \ .
\end{array}$$

Collecting the assumptions in the above calculation together, we have

$$\beta \quad = \quad \mathit{id} \times ((L \rightharpoonup) \cdot \Upsilon \cdot (\mathit{inits} \cdot \mathit{it} \cdot \mathit{ghd}*) \vartriangle \mathit{id}) \ .$$

Since

$$\begin{array}{lcl}
(L \rightharpoonup) \cdot \Upsilon & = & \Upsilon \cdot (L \rightharpoonup) \times (L \rightharpoonup) \\
(L \rightharpoonup) \cdot \mathit{inits} \cdot \mathit{it} & = & \mathit{inits} \cdot \mathit{it} \cdot (L \rightharpoonup) \\
(L \rightharpoonup) \cdot f* & = & f* \cdot (L \rightharpoonup) \ ,
\end{array}$$

properties that are easily verified, we may rewrite β as

$$\beta \quad = \quad \mathit{id} \times (\Upsilon \cdot (\mathit{inits} \cdot \mathit{it} \cdot \mathit{ghd}*) \vartriangle \mathit{id} \cdot (L \rightharpoonup)) \ .$$

Constructing an operator $\ominus$ such that $a \ominus (a \oplus h\,x) = h\,x$

This part constructs an operator $\ominus$ such that

$$a \ominus (a \oplus h\,x) \quad = \quad h\,x \ . \tag{7.48}$$

Since $a \oplus h\,x = h\,([a] \mathbin{+\!\!+} x)$, we try to express $h\,x$ in terms of $h\,([a] \mathbin{+\!\!+} x)$.

$$\begin{array}{ll}
& h\,x \\
= & \quad \text{definition of } h \\
& ((L \leftharpoonup) \times \mathit{exr}*)\, \mathit{fiif}\ \mathit{splits2}\ x \\
= & \quad \text{equation (4.88)} \\
& ((L \leftharpoonup) \times \mathit{exr}*)\, \mathit{fiif}\ (\mathit{tl} \times \mathit{id})*\ \mathit{tl}\ \mathit{splits2}\ ([a] \mathbin{+\!\!+} x) \\
= & \quad \textbf{assume } \mathit{fiif} \cdot (\mathit{tl} \times \mathit{id})* \cdot \mathit{tl} = \delta \cdot \mathit{fiif}\text{, } \delta \text{ is defined below}
\end{array}$$

$$\begin{array}{ll} & ((L \leftharpoonup) \times exr*)\,\delta\, \mathit{fiif}\ \mathit{splits2}\ ([a] +\!\!\!+\, x) \\ = & \quad \textbf{assume } ((L \leftharpoonup) \times exr*) \cdot \delta = \epsilon \cdot ((L \leftharpoonup) \times exr*), \epsilon \text{ is defined below} \\ & \epsilon\, ((L \leftharpoonup) \times exr*)\, \mathit{fiif}\ \mathit{splits2}\ ([a] +\!\!\!+\, x) \\ = & \quad \text{definition of } h \\ & \epsilon\, h\, ([a] +\!\!\!+\, x)\ . \end{array}$$

So if we assume that there exist functions δ and ϵ satisfying

$$\mathit{fiif} \cdot (tl \times id)* \cdot tl \quad = \quad \delta \cdot \mathit{fiif} \tag{7.49}$$

$$((L \leftharpoonup) \times exr*) \cdot \delta \quad = \quad \epsilon \cdot ((L \leftharpoonup) \times exr*)\ , \tag{7.50}$$

then $h\,x = \epsilon\, h\, ([a] +\!\!\!+\, x)$, and operator $\ominus$ is defined by

$$a \ominus z \quad = \quad \epsilon\, z\ .$$

It remains to give functions δ and ϵ satisfying, respectively, equations (7.49) and (7.50). Function δ is obtained as follows.

$$\begin{array}{ll} & \mathit{fiif} \cdot (tl \times id)* \cdot tl \\ = & \quad \text{definition of } \mathit{fiif} \\ & (fap \times id)* \vartriangle (id \times fap)* \cdot (tl \times id)* \cdot tl \\ = & \quad \text{equation (2.13), map-distributivity, equation (2.9)} \\ & (((fap \cdot tl) \times id)* \cdot tl) \vartriangle ((tl \times fap)* \cdot tl) \\ = & \quad \text{assumption: } fap \cdot tl = gtl \cdot fap \text{, equation (2.9), map-distributivity} \\ & ((gtl \times id)* \cdot (fap \times id)* \cdot tl) \vartriangle ((tl \times id)* \cdot (id \times fap)* \cdot tl) \\ = & \quad \text{equations (3.4), (2.12), definition of } \mathit{fiif} \\ & ((gtl \times id)* \cdot tl) \times ((tl \times id)* \cdot tl) \cdot \mathit{fiif}\ , \end{array}$$

so function δ is defined by

$$\delta \quad = \quad ((gtl \times id)* \cdot tl) \times ((tl \times id)* \cdot tl)\ .$$

Function ϵ satisfying equation (7.50) is obtained as follows.

$$\begin{array}{ll} & (L \leftharpoonup) \times exr* \cdot \delta \\ = & \quad \text{definition of } \delta \\ & (L \leftharpoonup) \times exr* \cdot ((gtl \times id)* \cdot tl) \times ((tl \times id)* \cdot tl) \\ = & \quad \text{equations (2.9), (3.15), map-distributivity} \\ & ((gtl \times id)* \cdot (L \leftharpoonup) \cdot tl) \times ((exr \cdot tl \times id)* \cdot tl) \\ = & \quad \text{equations (2.15), (3.4)} \end{array}$$

$$\begin{array}{ll} & ((\mathit{gtl} \times \mathit{id})* \cdot (L \leftharpoonup) \cdot \mathit{tl}) \times (\mathit{tl} \cdot \mathit{exr}*) \\ = & \quad \textbf{assume } (L \leftharpoonup) \cdot \mathit{tl} = \chi \cdot (L \leftharpoonup),\ \chi \text{ is defined below, equation (2.9)} \\ & ((\mathit{gtl} \times \mathit{id})* \cdot \chi) \times \mathit{tl} \cdot (L \leftharpoonup) \times \mathit{exr}*\ , \end{array}$$

so if we assume that there exists a function χ such that

$$(L \leftharpoonup) \cdot \mathit{tl} \quad = \quad \chi \cdot (L \leftharpoonup)\ ,$$

then function ϵ defined by

$$\epsilon \quad = \quad ((\mathit{gtl} \times \mathit{id})* \cdot \chi) \times \mathit{tl}\ ,$$

satisfies equation (7.50).

For the definition of χ we reason as follows. We have

$$L \leftharpoonup \mathit{tl}\, x \quad = \quad \begin{cases} L \leftharpoonup x & \text{if } \# x > L \\ \mathit{tl}\,(L \leftharpoonup x) & \text{if } \# x \leq L\ . \end{cases}$$

The argument of χ is $L \leftharpoonup x$, so we cannot make the case distinction of the above equation. However, the context in which χ is applied in the above calculation supplies enough information to determine a similar case distinction. Function χ is applied to an argument of the form $\mathit{exl}\, h\,([a] +\!\!+\, z) = L \leftharpoonup \mathit{exl}\,\mathit{fapg}\,([a] +\!\!+\, z)$, so we have to find a function χ such that

$$\chi\,(L \leftharpoonup \mathit{exl}\,\mathit{fapg}\,([a] +\!\!+\, z)) \quad = \quad L \leftharpoonup \mathit{tl}\ \mathit{exl}\,\mathit{fapg}\,([a] +\!\!+\, z)\ .$$

By definition of *fapg* we have

$$\#\, \mathit{exl}\,\mathit{fapg}\,([a] +\!\!+\, z) \leq L \quad \equiv \quad \mathit{exl}\,\mathit{hd}\,(L \leftharpoonup \mathit{exl}\,\mathit{fapg}\,([a] +\!\!+\, z)) = [\,]\ ,$$

and hence function χ is defined by

$$\chi\, x \quad = \quad \begin{cases} x & \text{if } \mathit{exl}\,\mathit{hd}\, x \neq [\,] \\ \mathit{tl}\, x & \text{if } \mathit{exl}\,\mathit{hd}\, x = [\,]\ . \end{cases}$$

It follows that operator $\ominus$ satisfying (7.48) is defined by

$$a \ominus (x, y) \quad = \quad \begin{cases} ((\mathit{gtl} \times \mathit{id})*\, x,\ \mathit{tl}\, y) & \text{if } \mathit{exl}\,\mathit{hd}\, x \neq [\,] \\ ((\mathit{gtl} \times \mathit{id})*\, \mathit{tl}\, x,\ \mathit{tl}\, y) & \text{otherwise}\ . \end{cases}$$

The incremental algorithm (*fapg*, g, h)

This part gives the incremental algorithm (faph, g, h). If sections $(\oplus_c a)$ and $(b \oplus_c)$ are $(\downarrow_f, \downarrow_f)$-fusable after $\downarrow_f / \cdot (\mathit{all}\, p) \triangleleft \cdot (\in \mathit{parts}\, z) \triangleleft$ for all lists z, where operator $\oplus_c$ is defined by

$$x \oplus_c y \quad = \quad \begin{cases} x +\!\!+ [c] +\!\!+ y & \text{if } p\ c\ \wedge\ x, y \neq \nu_{\downarrow_f} \\ \nu_{\downarrow_f} & \text{otherwise}\ , \end{cases}$$

if predicate p is length-bounded with constant L, and if function *fap* satisfies

$$\begin{aligned} \mathit{fap} \cdot \mathit{it} &= \mathit{git} \cdot \mathit{fap} \\ \mathit{fap} \cdot \mathit{tl} &= \mathit{gtl} \cdot \mathit{fap} \ , \end{aligned}$$

then

$$\mathit{faph} \ = \ \oslash/ \cdot (\mathit{fiif} \cdot r){*} \ ,$$

where operator $\oslash$ is defined by

$$(v', v) \oslash (u', u) \ = \ (s, t) \ ,$$

where the components s and t are the following expressions.

$$\begin{aligned} s \ &= \ \begin{cases} (v'\otimes){*}\, \gamma\, L\, u\, \Upsilon\, \mathit{exr}{*}\, u' & \text{if } \#\, u = L \\ ((L{-}\#\, u) \leftharpoonup \mathit{it}\, (\mathit{id} \times ({+\!\!+}q)){*}\, v') +\!\!+ ((v'\otimes){*}\, \psi\, u\, \Upsilon\, \mathit{exr}{*}\, u') & \text{if } \#\, u < L \end{cases} \\ &\ \ \textbf{where } q = \mathit{exl}\ \mathit{lt}\ u \\ t \ &= \ \begin{cases} \mathit{exl}{*}\, v\, \Upsilon\, (\otimes u){*}\, \gamma'\, L\, v' & \text{if } \#\, v' = L \\ (\mathit{exl}{*}\, v\, \Upsilon\, (\otimes u){*}\, \psi'\, v') +\!\!+ ((L{-}\#\, v') \rightharpoonup \mathit{tl}\, ((r{+\!\!+}) \times \mathit{id}){*}\, u) & \text{if } \#\, v' < L \end{cases} \\ &\ \ \textbf{where } r = \mathit{exr}\ \mathit{hd}\ v' \ , \end{aligned}$$

Function $g = (\mathit{exl}{*}) \times (L \rightharpoonup) \cdot \mathit{fapg}$ is a left-reduction $\ominus\!\not\rightarrow\! e$, where $e = ([[\,]], [([\,], [\,])])$, and operator $\ominus$ is defined by

$$\begin{aligned} z \ominus a \ &= \ (\mathit{it}\ \mathit{exl}\ z +\!\!+ (\mathit{exl}\, \alpha\, z \otimes){*}\, \psi\, u, \mathit{exr}\, (\alpha\, z \oslash (v', v))) \\ &\qquad \textbf{where } (v', v) = \mathit{fiif}\ r\ a \ . \end{aligned}$$

Function α satisfying $\mathit{faph} = \alpha \cdot g$ is defined by

$$\alpha \ = \ (\Upsilon \cdot \mathit{id} \vartriangle (\mathit{tails} \cdot \mathit{tl} \cdot \mathit{gtl}{*}) \cdot (L \leftharpoonup)) \times \mathit{id} \ ,$$

and operator $\ominus$ satisfying $(g\, x \ominus a) \ominus a = h\, x$ is defined by

$$(x, y) \ominus a \ = \ \begin{cases} (\mathit{it}\, x, (\mathit{id} \times \mathit{git}){*}\, y) & \text{if } \mathit{exr}\ \mathit{lt}\ y \neq [\,] \\ (\mathit{it}\, x, (\mathit{id} \times \mathit{git}){*}\, \mathit{it}\, y) & \text{otherwise} \ . \end{cases}$$

Function $h = (L \leftharpoonup) \times \mathit{exr}{*} \cdot \mathit{fapg}$ is a right-reduction $\oplus\!\not\leftarrow\! u$ where $u = ([([\,], [\,])], [[\,]])$, and operator $\oplus$ is defined by

$$\begin{aligned} a \oplus z \ &= \ (\mathit{exl}\, ((v', v) \oslash \beta\, z), (\otimes \mathit{exr}\, \beta\, z){*}\, \psi'\, v' +\!\!+ \mathit{tl}\ \mathit{exr}\ z) \\ &\qquad \textbf{where } (v', v) = \mathit{fiif}\ r\ a \ . \end{aligned}$$

Function β satisfying $\mathit{faph} = \beta \cdot h$ is defined by

$$\beta \ = \ \mathit{id} \times (\Upsilon \cdot (\mathit{inits} \cdot \mathit{it} \cdot \mathit{ghd}{*}) \vartriangle \mathit{id} \cdot (L \rightharpoonup)) \ ,$$

and operator $\ominus$ satisfying $a \ominus (a \oplus h\, x) = h\, x$ is defined by

$$a \ominus (x, y) \ = \ \begin{cases} ((\mathit{gtl} \times \mathit{id}){*}\, x, \mathit{tl}\, y) & \text{if } \mathit{exl}\ \mathit{hd}\ x \neq [\,] \\ ((\mathit{gtl} \times \mathit{id}){*}\, \mathit{tl}\, x, \mathit{tl}\, y) & \text{otherwise} \ . \end{cases}$$

This completes the description of an incremental algorithm (faph, g, h).

7.2.4 Incremental coding

This section shows that the conditions listed in the last part of the previous subsection hold for the components of specification of *cod* (7.25), and hence that there exists an incremental algorithm for coding a text with respect to a dictionary.

For the first requirement we have to show that for all a and b, sections $(\oplus_c a)$ and $(b \oplus_c)$ are $(\downarrow_\#, \downarrow_\#)$-fusable after $\downarrow_\# / \cdot (\mathit{all} \in D) \triangleleft \cdot (\in \mathit{parts}\, z) \triangleleft$ for all lists z, where operator $\oplus$ is defined by

$$x \oplus_c y \quad = \quad \begin{cases} x +\!\!+ [c] +\!\!+ y & \text{if } c \in D \ \wedge \ x, y \neq \nu_{\downarrow_\#} \\ \nu_{\downarrow_\#} & \text{otherwise} \ . \end{cases}$$

Since

$$\nu_{\downarrow_\#} \oplus_c a \quad = \quad b \oplus_c \nu_{\downarrow_\#} \quad = \quad \nu_{\downarrow_\#} \ ,$$

it suffices to show that

$$\begin{aligned} (x \downarrow_\# y) \oplus_c v \quad &= \quad (x \oplus_c v) \downarrow_\# (y \oplus_c v) \\ v \oplus_c (x \downarrow_\# y) \quad &= \quad (v \oplus_c x) \downarrow_\# (v \oplus_c y) \ , \end{aligned}$$

for all x, y and v, where x and y are elements in the image of function $\downarrow_\# / \cdot (\mathit{all} \in D) \triangleleft \cdot (\in \mathit{parts}\, z) \triangleleft$ for all lists z. If at least one of x and y is equal to $\nu_{\downarrow_\#}$, or $c \notin D$ holds, these equalities trivially hold. Let, for $x, y \neq \nu_{\downarrow_\#}$, $x \downarrow_\# y = x$, the other case being symmetrical, and suppose that $c \in D$ holds. We have to prove that the two equalities

$$\begin{aligned} (x \oplus_c v) \downarrow_\# (y \oplus_c v) \quad &= \quad (x \oplus_c v) \\ (v \oplus_c x) \downarrow_\# (v \oplus_c y) \quad &= \quad (v \oplus_c x) \ , \end{aligned}$$

hold. Distinguish the two cases $x <_\# y$ and $x =_\# y \ \wedge \ x \neq y$. If $x <_\# y$, then $x \oplus_c v <_\# y \oplus_c v$, and $v \oplus_c x <_\# v \oplus_c y$, and the above equalities immediately follow. For the case $x =_\# y \ \wedge \ x \neq y$ we have to decide how $\downarrow_\#$ is defined on equal-length but different arguments. Since x and y are both elements of *parts* z, they are completely determined by $\# * x$ and $\# * y$. We choose the following lexicographical variant of $\downarrow_\#$.

$$x \downarrow_\# y = x \quad \equiv \quad x <_\# y \ \vee \ (x =_\# y \ \wedge \ \mathit{rev}\, \# * x \leq_L \mathit{rev}\, \# * y) \ ,$$

that is, if two partitions have equal length, $\downarrow_\#$ returns the one with the shortest lists at the right end of the list. It is easily verified that if $x =_\# y \ \wedge \ x \neq y$, then equalities

$$\begin{aligned} \mathit{rev}\, \# * (x \oplus_c v) \quad &\leq_L \quad \mathit{rev}\, \# * (y \oplus_c v) \\ \mathit{rev}\, \# * (v \oplus_c x) \quad &\leq_L \quad \mathit{rev}\, \# * (v \oplus_c x) \ , \end{aligned}$$

also hold. It follows that sections $(\oplus_c a)$ and $(b \oplus_c)$ are $(\downarrow_\#, \downarrow_\#)$-fusable after $\downarrow_\# / \cdot (\mathit{all}\, \mathit{fit}) \triangleleft \cdot (\in \mathit{parts}\, z) \triangleleft$ for all lists z.

The second condition that has to be satisfied in order to derive an incremental algorithm, by means of the construction given in the previous subsection, for function *cod* holds, since predicate $\in D$ is length-bounded with constant $\uparrow/\,\#*\,D+1$.

Finally, for the third requirement we have to show that

$$\begin{array}{lcl} \mathit{cod}\cdot\mathit{it} & = & \mathit{git}\cdot\mathit{cod} \end{array} \qquad (7.51)$$

$$\begin{array}{lcl} \mathit{cod}\cdot\mathit{tl} & = & \mathit{gtl}\cdot\mathit{cod}\ . \end{array} \qquad (7.52)$$

For the proof of these equations we use the fact that by Corollary 4.51, *cod* is a left-reduction

$$\mathit{cod} \;=\; \oplus\!\not\to[\,]\ ,$$

where operator $\oplus$ is defined by $\nu_{\downarrow\#}\oplus a = \nu_{\downarrow\#}$, and

$$\begin{array}{lcl} [\,]\oplus a & = & [[a]] \\ (zs \mathbin{+\!\!+} [z])\oplus a & = & \left\{\begin{array}{ll} zs \mathbin{+\!\!+} [z \mathbin{+\!\!+} [a]] & \text{if } z \mathbin{+\!\!+} [a]\in D \\ zs \mathbin{+\!\!+} [z] \mathbin{+\!\!+} [[a]] & \text{otherwise}\ , \end{array}\right. \end{array}$$

provided predicate $\in D$ is segment-closed and holds for singletons, so assume all singletons are elements of D, and

$$x\in D \;\Rightarrow\; \mathit{segs}\,x\in D\ .$$

Operator $\oplus$ satisfies

$$\mathit{git}\cdot(\oplus a) \;=\; \mathit{id} \qquad (7.53)$$

$$\mathit{gtl}\cdot(\oplus a) \;=\; (\oplus a)\cdot\mathit{gtl}\ , \qquad (7.54)$$

if we assume that function *gtl* and operator $\oplus$ are defined such that $\mathit{gtl}\,[\,]\oplus a=[\,]$ for all a. The proofs of these equalities are easy and omitted. For equation (7.51) we calculate as follows.

$$\begin{array}{ll} & \mathit{cod}\ \mathit{it}\,(x \mathbin{-\!\!\!\!<} a) \\ = & \qquad \text{definition of } \mathit{it} \\ & \mathit{cod}\,x \\ = & \qquad \text{equation (7.53)} \\ & \mathit{git}\,(\mathit{cod}\,x\oplus a) \\ = & \qquad \text{definition of } \oplus \\ & \mathit{git}\ \mathit{cod}\,(x \mathbin{-\!\!\!\!<} a)\ . \end{array}$$

If we assume that functions *git* and *cod* are defined such that $\mathit{git}\,[\,]=\mathit{cod}\ \mathit{it}\,[\,]$ then equation (7.51) holds. Equation (7.52) is proved by induction. For the base case assume that functions *gtl* and *cod* are defined such that $\mathit{gtl}\,[\,] = \mathit{cod}\ \mathit{tl}\,[\,]$. For singletons we trivially have

$$\mathit{gtl}\ \mathit{cod}\,[a] \quad = \quad \mathit{cod}\ \mathit{tl}\,[a]\ .$$

The induction hypothesis is: for all nonempty lists x

$$\mathit{gtl}\ \mathit{cod}\ x \quad = \quad \mathit{cod}\ \mathit{tl}\ x\ .$$

Calculate as follows for the inductive step. Assume $x \neq [\,]$.

$$\begin{array}{ll}
 & \mathit{gtl}\ \mathit{cod}\,(x \mathbin{+\!\!\!<} a) \\
= & \qquad \mathit{cod} = \oplus\!\not\rightarrow\![\,] \\
 & \mathit{gtl}\,(\mathit{cod}\ x \oplus a) \\
= & \qquad \text{equation (7.54)} \\
 & (\mathit{gtl}\ \mathit{cod}\ x) \oplus a \\
= & \qquad \text{induction hypothesis, } x \text{ is nonempty} \\
 & (\mathit{cod}\ \mathit{tl}\ x) \oplus a \\
= & \qquad \mathit{cod} = \oplus\!\not\rightarrow\![\,] \\
 & \mathit{cod}\,(\mathit{tl}\ x \mathbin{+\!\!\!<} a) \\
= & \qquad \text{definition of } \mathit{tl} \\
 & \mathit{cod}\ \mathit{tl}\,(x \mathbin{+\!\!\!<} a)\ .
\end{array}$$

This concludes the proof of equation (7.52). It follows that function *cod* satisfies the conditions that are sufficient to derive an incremental algorithm for it. The incremental algorithm for *cod* is obtained by instantiating the free variables of the incremental algorithm given at the end of the previous section with the components of *cod*.

7.2.5 Incremental text formatting

This section shows that the conditions listed in the last part of the previous subsection hold for the components of specification *format* (7.28), and hence that there exists an incremental algorithm for formatting a text.

For the first requirement we have to show that for all a and b, sections $(\obar_c a)$ and $(b \obar_c)$ are $(\downarrow_{\mathit{waste}}, \downarrow_{\mathit{waste}})$-fusable after $\downarrow_{\mathit{waste}}/ \cdot (\mathit{all}\,\mathit{fit})\triangleleft \cdot (\in \mathit{parts}\ z)\triangleleft$ for all lists z, where operator $\obar$ is defined by

$$x \obar_c y \quad = \quad \begin{cases} x \mathbin{+\!\!+} [c] \mathbin{+\!\!+} y & \text{if } \mathit{fit}\ c\ \wedge\ x, y \neq \nu_{\downarrow_{\mathit{waste}}} \\ \nu_{\downarrow_{\mathit{waste}}} & \text{otherwise}\ . \end{cases}$$

Since

$$\nu_{\downarrow_{waste}} \obar_c a \;=\; b \obar_c \nu_{\downarrow_{waste}} \;=\; \nu_{\downarrow_{waste}} \;,$$

it suffices to show that

$$\begin{aligned} (x \downarrow_{waste} y) \obar_c v &= (x \obar_c v) \downarrow_{waste} (y \obar_c v) \\ v \obar_c (x \downarrow_{waste} y) &= (v \obar_c x) \downarrow_{waste} (v \obar_c y) \;, \end{aligned}$$

for all x, y and v, where x and y are elements in the image of function $\downarrow_{waste}/ \cdot (\mathit{all\,fit})\triangleleft \cdot (\in \mathit{parts}\, z)\triangleleft$. If at least one of x and y is equal to $\nu_{\downarrow_{waste}}$, or $\neg \mathit{fit}\; c$ holds, these equalities trivially hold. Let, for $x, y \neq \nu_{\downarrow_{waste}}$, $x \downarrow_{waste} y = x$, the other case being symmetrical, and suppose that $\mathit{fit}\; c$ holds. We have to prove that the two equalities

$$\begin{aligned} (x \obar_c v) \downarrow_{waste} (y \obar_c v) &= (x \obar_c v) \\ (v \obar_c x) \downarrow_{waste} (v \obar_c y) &= (v \obar_c x) \;, \end{aligned}$$

hold. Distinguish the two cases $x <_{waste} y$ and $x =_{waste} y \;\wedge\; x \neq y$. If $x <_{waste} y$, then $(x \obar_c v) <_{waste} (y \obar_c v)$, and $(v \obar_c x) <_{waste} (v \obar_c y)$, and it follows that the above equalities hold in case $x <_{waste} y$. Suppose $x =_{waste} y \;\wedge\; x \neq y$. We have to decide how operator $\downarrow_{waste}$ is defined on different arguments with equal *waste*-value. Since x and y are both elements of *parts* z, they are completely determined by $\# * x$ and $\# * y$. We choose the following lexicographical variant of $\downarrow_{waste}$, analogous to the definition of $\downarrow_{\#}$ in equation (4.50).

$$x \downarrow_{waste} y = x \;\;\equiv\;\; x <_{waste} y \;\vee\; (x =_{waste} y \;\wedge\; \mathit{rev}\,\# * x \leq_L \mathit{rev}\,\# * y) \;,$$

that is, if two partitions have equal *waste*-value, operator $\downarrow_{waste}$ returns the one with the shortest lists at the right end of the list. It is easily verified that if $x =_{waste} y \;\wedge x \neq y$ equalities

$$\begin{aligned} \mathit{rev}\,\# * (x \obar_c v) &\leq_L \mathit{rev}\,\# * (y \obar_c v) \\ \mathit{rev}\,\# * (v \obar_c x) &\leq_L \mathit{rev}\,\# * (v \obar_c y) \;, \end{aligned}$$

also hold. It follows that sections $(\obar_c a)$ and $(b \obar_c,)$ are $(\downarrow_{waste}, \downarrow_{waste})$-fusable after $\downarrow_{waste}/ \cdot (\mathit{all\,fit})\triangleleft \cdot (\in \mathit{parts}\, z)\triangleleft$ for all lists z.

The second condition that has to be satisfied in order to derive an incremental algorithm, by means of the construction of the previous subsection, for function *format* holds, since predicate *fit* defined by $\mathit{fit}\, x = +/\, x \leq C$ is length-bounded with constant $L = C + 1$.

Finally, for the third requirement we have to show that

$$\begin{aligned} \mathit{format} \cdot \mathit{it} &= \mathit{git} \cdot \mathit{format} \\ \mathit{format} \cdot \mathit{tl} &= \mathit{gtl} \cdot \mathit{format} \;. \end{aligned}$$

The proof of these equalities is very similar to the proof of equalities (7.51) and (7.52), and is omitted.

Since the components of function *format* satisfy the conditions that are sufficient to derive an incremental algorithm for *format*, we can instantiate the free variables in the incremental algorithm given at the end of the previous subsection with the components of *format* to obtain an incremental algorithm for *format*.

7.3 Conclusions

This chapter discusses the derivation and description of incremental algorithms on various data types, with an emphasis on the data type *list*. An incremental algorithm is defined relative to an edit model. We give, among others, two different definitions of an incremental algorithm on the data type *list*, and we show that the second definition in particular is well suited for the derivation of a large body of incremental algorithms.

The theory developed is exemplified with the derivation of incremental algorithms for two larger examples: coding a text with respect to a dictionary and formatting a text. The derivation of these incremental algorithms is rather substantial, and shows the need for the development of more theory.

It is desirable to abstract from the different definitions of an incremental algorithm we have given on data types with laws. Another wish is to let the definition of an incremental algorithm be dependent on the edit model only, instead of it being dependent on an edit model and a data type. These are topics for future research. Another topic for future research is the derivation of useful other examples of incremental algorithms, such as the derivation of an incremental algorithm for constructing a suffix tree.

Chapter 8

In conclusion

The aspiration of the constructive-algorithmics approach to program construction is to cover, eventually, large parts of the 'tricks of the trade' of the practice of computing, and to provide a body of concepts, notations and theories with which the methods and results for certain data types or classes of problems can be described in a systematic way.

What is the contribution of this thesis to achieving this goal?

Of course, considering the ambitious goal of Constructive Algorithmics, the contributions can be no else than limited. Part I of this thesis contains the general theory of the Bird-Meertens calculus developed over the last ten years. Except for the laws for cross and zip, this theory has been developed by other people, and it is more or less common knowledge within the Constructive Algorithmics community.

Part II of this thesis applies the Bird-Meertens calculus in different situations. Almost all results in this part are new. Part II consists of four chapters, of which the first one is used in each of the three other chapters. The last three chapters contain the developments of specialised calculational theories. Chapter 5 develops a specialised theory the results of which apply to a specific class of problems, namely the class of segment problems. Chapter 6 develops a specialised theory the results of which apply to a specific data type, namely the data type *array*. Chapter 7 develops a specialised theory the results of which give a specific kind of algorithms, namely incremental algorithms.

Chapter 4 discusses so-called generator fusion theorems. A generator is usually a polymorphic function that generates a set of elements that are related in a specific manner to the argument. A generator fusion theorem lists sufficiency conditions a catamorphism has to satisfy in order to fuse the catamorphism with the generator. Most of the derivations of generator fusion theorems are short and straightforward. I particularly like the 'hierarchy' of fusion theorems for the generator *segs*, comprising the *segs*-Fusion Theorem, the Sliding Tails Theorem, the Hopping Tails Theorem, and two corollaries of the Hopping Tails Theorem, see Chapter 5. Concerning generator fusion theorems we note the following two points. First, the similarity of some of the derivations of generator fusion theorems indicates the existence of one or more general theorems each of which can be specialised to a generator fusion theorem. Indeed, such theorems exist, see De Moor and Bird [105], and similar theorems should be formulated. Second, some problems arise in some of the applications of generator fusion theorems. One of these problems can be found in, for example, the derivations of the corollaries of the *parts*-Fusion Theorem where the catamorphism is of the form $\downarrow_{\#}/ \cdot (all\, p)\triangleleft$. The derivations are long and tedious, and especially the derivations of selectors $\downarrow_{\#}$ such that the conditions of the *parts*-Fusion Theorem are satisfied are complicated. I have found no way that avoids these unattractive calculations. A relational calculus may be useful in deferring these calculations to a later stage of the development of a theory, thus separating the concerns of developing theory and of constructing programs. However, when constructing a program, a precise definition of the selector is required, and a calculation similar to the ones we give has to be performed.

Chapter 5 develops a calculational theory for solving segment problems. Using the hierarchy of fusion theorems for the generator *segs*, we derive about ten algorithms for example problems. The examples show that it is indeed possible to develop theory that yields calculations for several non-trivial problems like the pattern-matching problem, and the problem of finding palindromes in a list. The calculations also show that derivations can be quite complicated. And the calculational nature of the inventive step that has to be made in several examples (tupling with some extra functions) is questionable. Another, related, problem is the 'rigidity' of data types. In the larger examples of derivations of algorithms for segment problems we have to tuple with some extra functions to obtain algorithms that can be implemented as efficient programs. For example, the algorithm for the *iani*-problem given in Section 5.5 corresponds to five functions. Compared with the derivation of an algorithm for the same problem by Van der Woude [138], our derivation is much more complicated. The difference between the two derivations is caused by the difference between the definitions of the data type list. Van der Woude defines a list as a function with source the natural numbers, and he allows random access in the list, that is, retrieving a value from a list costs constant time. We do not have random access in a list, and retrieving a value from a list may therefore cost time linear in the length of a list. This is the reason why we tuple with a number of extra functions. An extra theory of storing and retrieving previously computed values of a function may be of use here.

The chapter on the data type *array* shows that it can be defined as a hierarchy of initial algebras (nested lists). Results on the data type *list* are extended to the data type *array* by means of straightforward but rather long calculations. Although the extension of results from the data type list to the data type *array* is free of surprises, the formulation of the results requires much space and many symbols. Future research should be directed towards the automatic construction of hierarchies of algorithms on *array* from algorithms on *list*.

Finally, the chapter on incremental algorithms gives a formal definition of incremental algorithms on various data types, and it derives several incremental algorithms. To my knowledge this is the first *calculational* approach to incremental algorithms. The second part of this chapter derives an incremental algorithm that can be used for text-formatting and data compression with respect to a dictionary. There are two points that deserve attention. First, it is desirable to abstract from the different definitions of an incremental algorithm we have given on data types with law. Another wish is to let the definition of an incremental algorithm be dependent on the edit model only instead of it being dependent on an edit model and a data type. Second, most parts of the derivation in the second part of this chapter are straightforward, but the complete derivation presented in Section 7.2 is rather long: eighteen pages. This derivation, and other derivations in Part II show the need for a computer program, a proofeditor, that supports calculations. Using a proofeditor with a library of laws reduces the chance of making mistakes. Furthermore, if such a proofeditor possesses a mechanism for suppressing parts of proofs, derivations can be presented in a prettier way.

Concluding: we have developed several specialised theories for algorithm calculation. The theories contain the first steps towards a comprehensive theory for algorithm calculation. Parts of the theories are quite elegant and useful in the calculation of algorithms, other parts are rather unattractive and laborious when applied to some examples.

Index

Bibliography

[1] J.O. Achugbue. On the line breaking problem in text formatting. *ACM SIGOA Newsletter*, 2(1 & 2):117–121, 1981.

[2] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[4] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In J.A. Storer and M. Cohn, editors, *Proceedings Data Compression Conference, March 24–27, 1992, Snowbird, Utah*, pages 279–288, 1992.

[5] A. Apostolico and Z. Galil, editors. *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*. Springer–Verlag, 1985.

[6] R.C. Backhouse. An exploration of the Bird–Meertens formalism. Technical Report Computing Science Notes CS 8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[7] R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.

[8] R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Polynomial relators. In *Proceedings second Conference on Algebraic Methodology and Software Technology*, 1991.

[9] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[10] T.P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978.

[11] C. Banger. Arrays with categorical type constructors. To appear in [61], 1993.

[12] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[13] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[14] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–505, 1984. Addendum: Ibid. 7(3):490–492, 1985.

[15] R.S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6:159–189, 1986.

[16] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.

[17] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989.

[18] R.S. Bird. A calculus of functions for program derivation. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 287–308. Addison-Wesley Publishing Company, 1990.

[19] R.S. Bird. Small specification exercises. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty Is Our Business, A Birthday Salute to Edsger W. Dijkstra*, pages 390–398. Springer-Verlag, 1990.

[20] R.S. Bird. Functional pearls: On removing duplicates. *Journal of Functional Programming*, 1(2):235–243, 1991.

[21] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[22] R.S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International, 1988.

[23] E.A. Boiten. Intersection of sets and bags of extended substructures — a class of problems. In B. Möller, editor, *Constructing Programs from Specifications*, pages 33–48. North-Holland, 1991.

[24] K.S. Booth. Lexicographically least circular substrings. *Information Processing Letters*, 10(4,5):240–242, 1980.

[25] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[26] P.J. de Bruin. Naturalness of polymorphism. Technical Report CS8916, Department of Mathematics and Computing Science, University of Groningen, 1989.

[27] R.M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[28] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up tree pattern matching. In *Proceedings* 15^{th} *Colloquium on Trees in Algebra and Programming*, pages 72–86, 1990. LNCS 431.

[29] A. Cayley. A memoir on the theory of matrices. *Philosophical Transactions of the Royal Society of London*, 148, 1858.

[30] P. Chisholm. Calculation by computer. *The Squiggolist*, 1(4):64–67, 1990.

[31] P. Chisholm. Calculation by computer: Overview. Technical Report Computing Science Notes CS 9007, Department of Mathematics and Computing Science, University of Groningen, 1990.

[32] S.N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, C-18(4):349–365, 1969.

[33] M. Crochemore and W. Rytter. Parallel computations on strings and arrays. In C. Choffrut and T. Lengauer, editors, *7th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 415, pages 109–125, 1990.

[34] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[35] E.W. Dijkstra. *A Discipline of Programming*. Prentice–Hall, 1976.

[36] E.W. Dijkstra. Minsegsumtwodim. EWD900a, 1984.

[37] E.W. Dijkstra and W.H.J. Feyen. *A Method of Programming*. Addison–Wesley, 1988.

[38] J.P.H.W. van den Eijnde. Left-bottom and right-top segments. *Science of Computer Programming*, 15:79–94, 1990.

[39] P.C. Fischer and M.S. Paterson. String matching and other products. In R.M. Karp, editor, *SIAM AMS Proceedings on Complexity of Computation*, volume 7, pages 113–126, 1974.

[40] M.M. Fokkinga. Selector guards. *The Squiggolist*, 1(4):55–60, 1990.

[41] M.M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4):81–82, 1990.

[42] M.M. Fokkinga. Using underspecification in the derivation of some optimal partition algorithms. CWI, Amsterdam, 1990.

[43] M.M. Fokkinga. Calculate categorically! In J. van Leeuwen, editor, *Proceedings SION Computing Science in the Netherlands*, pages 211–230, 1991.

[44] M.M. Fokkinga. Datatype laws without signatures. In J. van Leeuwen, editor, *Proceedings SION Computing Science in the Netherlands*, pages 231–248, 1991.

[45] M.M. Fokkinga. An exercise in transformational programming—backtracking and branch-and-bound. *Science of Computer Programming*, 16(1):19–48, 1991.

[46] M.M. Fokkinga. A gentle introduction to category theory — the calculational approach —. In Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics, Utrecht University, 1992.

[47] M.M. Fokkinga. *Law and order in algorithmics*. PhD thesis, Twente University, 1992.

[48] M.M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, 1991.

[49] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[50] Z. Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proceedings of the Eighth Annual Symposium on Theory of Computing*, pages 161–173, 1976.

[51] Z. Galil. Two fast simulations which imply some fast string matching and palindrome-recognition algorithms. *Information Processing Letters*, 4:85–87, 1976.

[52] Z. Galil. Palindrome recognition in real time by a multitape Turing machine. *Journal of Computers and Systems Sciences*, 16:140–157, 1978.

[53] Z. Galil. String matching in real time. *Journal of the ACM*, 28(1):134–149, 1981.

[54] Z. Galil and J. Seiferas. A linear-time on-line recognition algorithm for ‘Palstar’. *Journal of the ACM*, 25(1):102–111, 1978.

[55] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP–Completeness*. W.H. Freeman and company, 1979.

[56] J. Gibbons. A new view of binary trees. Programming Research Group, Oxford University, 1988.

[57] J. Gibbons. *Algebras for Tree Algorithms*. PhD thesis, Programming Research Group, Oxford University, 1991. Technical Monograph PRG-94.

[58] J. Gibbons. Upwards and downwards accumulations on trees. In C.C. Morgan and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, 1992.

[59] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[60] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[61] G. Hains, editor. *Proceedings of the Second international workshop on array structures*. 1993. To appear.

[62] G. Hains and L.M.R. Mullin. An algebra of multidimensional arrays. submitted for publication, 1992.

[63] C.A.R. Hoare. Notes on an approach to category theory for computer scientists. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989.

[64] C.M. Hoffmann and M.J. O'Donnel. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[65] R.R. Hoogerwoord. *The design of functional programs: a calculational approach*. PhD thesis, Eindhoven University of Technology, 1989.

[66] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[67] P. Hudak and Ph. Wadler Eds. Report on the functional programming language haskell. Technical Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, 1988.

[68] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communication of the ACM*, 20(5):350–353, 1977.

[69] A. Jeffrey. Soft arrays. *The Squiggolist*, 1(4):74–75, 1990.

[70] J. Jeuring. Finding palindromes. In *Proceedings SION Computing Science in the Netherlands*, pages 123–140, 1988.

[71] J. Jeuring. Deriving algorithms on binary labelled trees. In P.M.G. Apers, D. Bosman, and J. van Leeuwen, editors, *Proceedings SION Computing Science in the Netherlands*, pages 229–249, 1989.

[72] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[73] J. Jeuring. The largest ascending substree — an exercise in nub theory. *The Squiggolist*, 1:36–44, 1990.

[74] J. Jeuring. The derivation of hierarchies of algorithms on matrices. In B. Möller, editor, *Constructing Programs from Specifications*, pages 9–32. North-Holland, 1991.

[75] J. Jeuring. Incremental algorithms on lists. In J. van Leeuwen, editor, *Proceedings SION Computing Science in the Netherlands*, pages 315–335, 1991.

[76] J. Jeuring. Incremental data compression —abstract—. In J.A. Storer and M. Cohn, editors, *Proceedings Data Compression Conference, March 24–27, 1992, Snowbird, Utah*, page 411, 1992.

[77] J. Jeuring. The derivation of a hierarchy of algorithms for pattern matching on arrays. To appear in [61], 1993.

[78] J. Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. To appear in Algorithmica, 1993.

[79] J. Jeuring and L. Meertens. The least-effort cabinet formation. *The Squiggolist*, 1(2), 1989.

[80] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[81] R. Karp, R.E. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings* 4^{th} *Annual ACM Symposium on Theory of Computing*, pages 125–136, 1972.

[82] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1(4):425–447, 1990.

[83] D.E. Knuth. *The art of computer programming Volume 3, Sorting and Searching*. Addison-Wesley Publishing Company, 1977.

[84] D.E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[85] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1978.

[86] D.E. Knuth and M.F. Plass. Breaking paragraphs into lines. *Software: Practice & Experience*, 11(11):1119–1184, 1981.

[87] D.A. Leweler and D.S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[88] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989. LNCS 375.

[89] G. Malcolm. *Algebraic data types and program transformation.* PhD thesis, University of Groningen, 1990.

[90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[91] G. Malcolm. Squiggoling in context. *The Squiggolist*, 1(3):30–35, 1990.

[92] G. Manacher. A new linear–time 'on–line' algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22:346–351, 1975.

[93] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[94] L. Meertens. Some more examples of algorithmic developments. IFIP WG2.1 Working paper ADP-7, Pont-à-Mousson, France, 1984.

[95] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[96] L. Meertens. First steps towards the theory of rose trees. CWI, Amsterdam, 1987.

[97] L. Meertens. Variations on trees. In International Summer School on Constructive Algorithmics, Hollum, Ameland, 1989.

[98] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[99] L.G.L.T. Meertens, editor. *Program specification and transformation.* North-Holland, 1987.

[100] B. Meyer. Incremental string matching. *Information Processing Letters*, 21:219–227, 1985.

[101] B. Möller, editor. *Constructing Programs from Specifications.* North-Holland, 1991.

[102] O. de Moor. *Categories, relations and dynamic programming.* PhD thesis, Oxford University, 1992. Technical Monograph PRG-98.

[103] O. de Moor and R.S. Bird. Lecture notes on nub theory. Lecture Notes International Summer School on Constructive Algorithmics, Hollum-Ameland, The Netherlands, 1989.

[104] O. de Moor and R.S. Bird. List partitions. To appear in Formal Aspects of Computing, 1992.

[105] O. de Moor and R.S. Bird. Solving optimization problems with catamorphisms. To appear in Mathematics of Program Construction, 1992.

[106] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[107] L.M.R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, Syracuse, New York, 1988.

[108] L.M.R. Mullin. Psi: The indexing function a basis for FFP with arrays. In L.M.R. Mullin et al., editor, *International Workshop on Arrays, Functional Languages and Parallelism*. Kluwer, 1990.

[109] L.M.R. Mullin, editor. *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991.

[110] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, 1986.

[111] H.A. Partsch and F.A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2:109–122, 1990.

[112] H.A. Partsch and N. Völker. Another case study on reusability of transformational developments — pattern matching according to knuth, morris, and pratt. In M. Broy and M. Wirsing, editors, *Methods of Programming*, pages 35–48. Springer-Verlag, 1991. LNCS 544.

[113] S. Pemberton. Views: An open-architecture user-interface system. to appear in: Proceedings of Interacting with computers, preparing for the nineties, Noordwijkerhout, The Netherlands, 1990.

[114] J.A. La Poutré. *Dynamic Graph Algorithms and Data Structures*. PhD thesis, Utrecht University, 1991.

[115] M. Rem. Small programming exercises. *Science of Computer Programming*, 3:217–222, 1983.

[116] M. Rem. Small programming exercises 16. *Science of Computer Programming*, 8:203–211, 1987.

[117] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[118] C. Runciman. Another look at labelled trees. Presented at the Fortieth meeting of IFIP WG2.1, Lost Valley, Colorado, USA, 1989.

[119] A. Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

[120] J.I. Seiferas. Iterative arrays with direct central control. *Acta Informatica*, 8:177–192, 1977.

[121] Y. Shiloach. A fast equivalence checking algorithm for circular lists. *Information Processing Letters*, 8(5):236–238, 1979.

[122] D.B. Skillicorn. Architecture independent parallel programming. *IEEE Computer*, 23(12):38–51, 1990.

[123] A.O. Slisenko. Recognizing a symmetry predicate by multihead Turing machines with input. In V.P. Orverkov and N.A. Sonin, editors, *Proceedings of the Steklov Institue of Mathematics*, number 129, pages 25–208, 1973.

[124] D.R. Smith. The design of divide-and-conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[125] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.

[126] D.R. Smith. On the design of generate-and-test algorithms: subspace generators. In L.G.L.T. Meertens, editor, *Program specification and transformation*, pages 207–220. North-Holland, 1987.

[127] J.A. Storer. *Data Compression; Methods and Theory*. Computer Science Press, 1988.

[128] V. Strassen. Gaussian elimination is not optimal. *Numerische Math.*, 13:354–356, 1969.

[129] S.D. Swierstra and O. de Moor. Virtual data structures. Technical Report RUU-CS-92-16, Utrecht University, 1992. To appear elsewhere.

[130] S. Thompson. Interactive functional programs, a method and a formal semantics. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 249–285. Addison-Wesley Publishing Company, 1990.

[131] J. Traugott. Deductive synthesis of sorting programs. *J. Symbolic Computation*, 7:533–572, 1989.

[132] D.A. Turner. Duality and De Morgan principles for lists. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty Is Our Business, A Birthday Salute to Edsger W. Dijkstra*, pages 390–398. Springer-Verlag, 1990.

[133] D.A. Turner. An overview of Miranda. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 1–16. Addison-Wesley Publishing Company, 1990.

[134] N. Verwer. Homomorphisms, factorisation, and promotion. *The Squiggolist*, 1(3):45–54, 1990.

[135] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989. FPCA '89, Imperial College, London.

[136] P. Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, volume 14, pages 1–11, 1973.

[137] D.S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture*, pages 134–153, 1987. LNCS 274.

[138] J. van der Woude. Playing with patterns, searching for strings. *Science of Computer Progamming*, 12:177–190, 1989.

[139] J.C.S.P. van der Woude. Optimal segmentations. Technical Report 89/15, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1989.

[140] A.C. Yao. A lower bound to palindrome recognition by probabilistic Turing machines. Technical Report STAN-CS-77-647, Computer Science Department, Stanford University, 1977.

[141] D. Yellin and R. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, 1991.

[142] H. Zantema. Longest segment problems. *Science of Computer Programming*, 18:39–66, 1992.

Samenvatting

Met deze samenvatting hoop ik zowel collega's als mensen van buiten het vakgebied Informatica een indruk te geven van het onderwerp en de resultaten van dit proefschrift.

Programma's

Dit proefschrift gaat over het produceren van programma's voor computers. Een computerprogramma bevat instrukties die een computer 'vertellen' wat hij moet berekenen. Als we bijvoorbeeld alle priemgetallen[1] kleiner dan duizend willen weten, dan kunnen we een computerprogramma schrijven om met een computer die priemgetallen te berekenen. Dit programma is geen Engelse of Nederlandse zin, zoals 'Bereken alle priemgetallen kleiner dan duizend', maar is geschreven in een taal die een computer kan verwerken. Een zin als 'Bereken alle priemgetallen kleiner dan duizend' heet een *specificatie* van het probleem. We leggen twee eisen op aan een computerprogramma voor het berekenen van alle priemgetallen kleiner dan duizend:

- Als we het programma op een computer uitvoeren, dan moet de computer de priemgetallen kleiner dan duizend berekenen, en niets anders. Een programma dat er voor zorgt dat de computer precies de priemgetallen kleiner dan duizend berekent heet *correct* met betrekking tot de specificatie.

- Als we het programma op een computer uitvoeren, dan moet de computer de priemgetallen kleiner dan duizend binnen afzienbare tijd opleveren: het programma is *efficiënt*.

Een programma dat niet aan deze twee eisen voldoet, bijvoorbeeld omdat het naast alle priemgetallen kleiner dan duizend ook nog een ander getal oplevert (bijvoorbeeld het getal 100), of omdat het één priemgetal per jaar oplevert, is waardeloos.

[1]Een priemgetal is een geheel getal groter dan 1 dat alleen deelbaar is door 1 en zichzelf, dus 2, 3, 5, 7, 11, enzovoort.

Het construeren van programma's

Hoe kunnen we er voor zorgen dat alle programma's die geschreven worden correct met betrekking tot hun specificatie en efficiënt zijn? Voor het construeren van correcte en efficiënte programma's zijn verschillende programmeermethodologieën ontworpen. Als we (goed) gebruik maken van een programmeermethodologie bij het construeren van een programma uit een gegeven specificatie, dan is de correctheid van het geconstrueerde programma gegarandeerd.

Transformationeel Programmeren

Eén van de programmeermethodologieën die momenteel in de belangstelling staan is 'Transformationeel Programmeren'. We zullen deze methodologie uitleggen aan de hand van een voorbeeld.

De volgende 'algebra' van breuken heeft iedereen op de lagere school gezien. Een breuk wordt als volgt gedefinieerd (in de volgende formules wordt zowel het symbool - als het symbool : gebruikt voor delen door). Een breuk $\frac{a}{b}$ is gelijk aan een getal c dan en slechts dan als $b \times c$ gelijk is aan a.

$$(\frac{a}{b} = c) \quad \equiv \quad (a = b \times c) \; . \tag{8.1}$$

De volgende vier gelijkheden kunnen uit de bovenstaande definitie afgeleid worden. We zullen de bewijzen van deze gelijkheden niet geven. Voor alle getallen a geldt

$$\frac{a}{1} \quad = \quad a \; . \tag{8.2}$$

Breuken worden als volgt met elkaar vermenigvuldigd. Voor alle a, b, c, en d

$$\frac{a}{b} \times \frac{c}{d} \quad = \quad \frac{a \times c}{b \times d} \; . \tag{8.3}$$

Delen door een breuk is vermenigvuldigen met het omgekeerde.

$$\frac{a}{b} : \frac{c}{d} \quad = \quad \frac{a}{b} \times \frac{d}{c} \; . \tag{8.4}$$

Gelijke factoren in de noemer en de deler van een breuk mogen tegen elkaar weggestreept worden.

$$\frac{a \times c}{b \times c} \quad = \quad \frac{a}{b} \tag{8.5}$$

Gebruik makend van de bovenstaande gelijkheden kunnen we laten zien dat $\frac{3}{2} : \frac{3}{4} = 2$.

$$
\begin{array}{ll}
 & \frac{3}{2} : \frac{3}{4} \\
= & \quad \text{gelijkheid (8.4)} \\
 & \frac{3}{2} \times \frac{4}{3} \\
= & \quad \text{gelijkheid (8.3)} \\
 & \frac{3\times4}{2\times3} \\
= & \quad \text{definitie van vermenigvuldigen: } 3 \times 4 = 2 \times 6,\ 2 \times 3 = 1 \times 6 \\
 & \frac{2\times6}{1\times6} \\
= & \quad \text{gelijkheid (8.5)} \\
 & \frac{2}{1} \\
= & \quad \text{gelijkheid (8.2)} \\
 & 2\ .
\end{array}
$$

In dit proefschrift rekenen we op dezelfde manier met programma's in plaats van getallen.

Voor programma's kunnen definities worden gegeven die qua vorm lijken op definitie (8.1), en kunnen wetten worden afgeleid die qua vorm lijken op de wetten (8.2), (8.3), (8.4), en (8.5). In de programmeermethodologie die we gebruiken worden zulke definities en wetten voor programma's gegeven. Deze programmeermethodologie werkt volgens het volgende principe. Vaak kan uit een specificatie op eenvoudige wijze een correct programma geconstrueerd worden, dat meestal erg inefficiënt is als het wordt uitgevoerd op een computer. Met behulp van wetten die gelden voor programma's of onderdelen van programma's 'transformeren' we nu dit programma (in het bovenstaande voorbeeld $\frac{3}{2} : \frac{3}{4}$) in een efficiënt programma (in het bovenstaande voorbeeld 2). Omdat elke wet de correctheid van het programma bewaart (in het bovenstaande voorbeeld betekent dit dat iedere tussenliggende expressie gelijk is aan 2) verkrijgen we zo een correct en efficiënt programma. Stel dat programma A gelijk is aan programma B op grond van een wet. We schrijven dan:

$$
\begin{array}{ll}
 & A \\
= & \quad \text{wet} \\
 & B\ .
\end{array}
$$

Zo kunnen we programma's berekenen op een manier die iedereen al op de lagere school heeft gezien.

Constructieve Algoritmiek

De verschillende methodologieën voor Transformationeel Programmeren onderscheiden zich in de formele taal die gebruikt wordt om programma's en wetten uit te drukken. In dit proefschrift wordt een methodologie gebruikt waarmee korte programma's en krachtige wetten beschreven kunnen worden. Deze methodologie wordt 'Constructieve Algoritmiek' of 'het Bird-Meertens formalisme' genoemd, naar Bird en Meertens die deze theorie in de tachtiger jaren ontwikkeld hebben.

Constructieve Algoritmiek is een wiskundige theorie waarmee gelijkheden tussen verschillende elementen van de wiskundige taal bewezen kunnen worden. Er bestaat een voor de hand liggende vertaling van de meeste wiskundige functies naar uitvoerbare programma's. We noemen de wiskundige functies algoritmen.

De specificatie 'Bereken alle priemgetallen kleiner dan duizend' kan nu formeel geschreven worden als een compositie van twee functies toegepast op het getal 1000:

$$(f \cdot g)\, 1000\ .$$

Dit betekent: pas eerst functie g toe op het argument 1000, en pas dan functie f toe op het resultaat. De functies f en g worden als volgt gedefinieerd. Functie g levert, gegeven een natuurlijk getal n, de verzameling van alle getallen kleiner dan n op. Dus als g wordt uitgevoerd met argument 6, dan:

$$g\, 6 = \{1, 2, 3, 4, 5\}\ .$$

Functie f neemt als argument een verzameling getallen en heeft als resultaat dezelfde verzameling waaruit alle getallen die niet priem zijn zijn weggelaten. Bijvoorbeeld:

$$f\, \{1, 2, 3, 4, 5\} \quad = \quad \{2, 3, 5\}\ .$$

Uit de voorbeelden volgt dat:

$$(f \cdot g)\, 6 \quad = \quad \{2, 3, 5\}\ .$$

Ook voor het volgende probleem willen we een formele specificatie hebben. Een palindroom is een rijtje letters dat niet verandert als je het omkeert. Voorbeelden van palindromen zijn 'madam' en 'parterretrap'. Alle rijtjes met slechts één letter zijn palindromen, en het lege rijtje, dat wordt genoteerd als λ, is ook een palindroom. Gegeven een rijtje letters, dan willen we daarin het langste opeenvolgende rijtje letters hebben dat een palindroom is. In 'zorro' is dat bijvoorbeeld 'orro'. Dit probleem wordt als volgt formeel gespecificeerd.

$$l \cdot p \cdot s\ .$$

De functie s levert de verzameling van alle aaneengesloten deelrijen van een gegeven rijtje letters op. Zo'n deelrij noemen we een *segment*. Bijvoorbeeld:

$$s\, (brr) \quad = \quad \{b, br, brr, r, rr, r, \lambda\}\ .$$

Functie p neemt als argument een verzameling van segmenten, en heeft als resultaat dezelfde verzameling met daaruit weggelaten alle segmenten die geen palindroom zijn:

$$p\,\{b, br, brr, r, rr, r, \lambda\} \quad = \quad \{b, r, rr, r, \lambda\}\ .$$

Functie l, tenslotte, neemt als argument een verzameling segmenten, en heeft als resultaat het langste segment, dus bijvoorbeeld,

$$l\,\{b, r, rr, r, \lambda\} \quad = \quad rr\ .$$

Uit de voorbeelden volgt dat:

$$(l \cdot p \cdot s)\,brr \quad = \quad rr\ ,$$

dat wil zeggen, het langste palindroom in het rijtje 'brr' is het rijtje 'rr'.

Datatypen

Eén van de belangrijkste begrippen in de Constructieve Algoritmiek is het begrip *datatype*. Een datatype is een collectie elementen die een verwante structuur hebben. Voorbeelden van datatypen zijn: de natuurlijke getallen, de rijtjes van letters, de verzamelingen van rijtjes van letters, de verzamelingen van natuurlijke getallen, enzovoort. Datatypen ontlenen hun belang aan onder meer een speciale klasse van functies die op datatypen is gedefinieerd: de klasse van zogenaamde catamorfismen.

Catamorfismen

Een *catamorfisme* is een functie, gedefinieerd op een datatype, met een aantal bijzonder prettige eigenschappen. Een voorbeeld van een catamorfisme is de functie l die het langste segment van een verzameling segmenten oplevert (zie boven). Een catamorfisme kan in het algemeen eenvoudig als een efficiënt programma worden geschreven, en daarnaast gelden een aantal wetten voor catamorfismen die het manipuleren van en rekenen met catamorfismen gemakkelijk maken. Eén van deze wetten is de zogenaamde 'Fusiestelling'. De Fusiestelling geeft condities waaronder de compositie van een functie en een catamorfisme geschreven kan worden als een (ander) catamorfisme.

Theorieën voor het berekenen van algoritmen

Het doel van Constructieve Algoritmiek is het ontwikkelen van concepten, notaties, en theorieën waarmee de methoden en resultaten voor het berekenen van algoritmen die voorkomen in de praktijk van het programmeren op een systematische manier beschreven kunnen worden. We onderscheiden de volgende drie klassen van algoritmen:

- algoritmen voor speciale klassen van problemen, zoals bijvoorbeeld de klasse van algoritmen voor problemen die met behulp van de functie s gespecificeerd worden;
- algoritmen op speciale datatypen, zoals bijvoorbeeld algoritmen op het datatype 'rijtjes van letters';
- speciale soorten algoritmen, zoals bijvoorbeeld parallelle algoritmen of incrementele algoritmen.

In dit proefschrift worden drie theorieën voor het berekenen van algoritmen ontwikkeld. Iedere theorie behandelt een voorbeeld uit één van de bovenstaande klassen van algoritmen. De bijdrage van dit proefschrift aan de Constructieve Algoritmiek is de ontwikkeling van deze theorieën; de benodigde basistheorie van de Constructieve Algoritmiek is in de afgelopen tien jaar door andere mensen ontwikkeld. De basistheorie van de Constructieve Algoritmiek wordt beschreven in het eerste gedeelte van dit proefschrift; de theorieën voor het berekenen van algoritmen worden beschreven in het tweede gedeelte van dit proefschrift. Bij de ontwikkeling van de theorieën voor het berekenen van algoritmen gebruiken we een theorie voor het construeren van zogenaamde generator-fusiestellingen. De constructie van generator-fusiestellingen wordt beschreven in Hoofdstuk 4.

Generatoren

Een generator is een functie die, gegeven een argument, een verzameling elementen genereert die op de één of andere manier aan het argument gerelateerd zijn. Een voorbeeld van een generator is de functie s die gegeven een rijtje letters de verzameling van alle segmenten van dat rijtje genereert. De functie g die gegeven een natuurlijk getal de verzameling van alle getallen kleiner dan dat getal genereert is een ander voorbeeld van een generator. Vaak is een generator een catamorfisme of de compositie van een eenvoudig te berekenen functie met een catamorfisme.

Specificaties met behulp van generatoren

Veel problemen kunnen worden gespecificeerd als de compositie van een catamorfisme met een generator. Als voorbeelden dienen weer de twee voorbeelden die we hebben gegeven. Functie f is een catamorfisme en functie g is een generator, dus de specificatie van het probleem van het vinden van de priemgetallen kleiner dan duizend is de compositie van een catamorfisme met een generator. Ook de specificatie van het probleem van het vinden van het langste segment dat een palindroom is in een rijtje letters is de compositie van een catamorfisme (namelijk de compositie van functies $l \cdot p$) en een generator (namelijk de functie s).

Generator-fusiestellingen

De fusiestelling voor catamorfismen geeft condities waaronder de compositie van een functie f en een catamorfisme c_1 geschreven kan worden als een catamorfisme c_2:

$$f \cdot c_1 \quad = \quad c_2$$

Een generator-fusiestelling geeft condities waaronder de compositie van een catamorfisme c_1 en een generator g geschreven kan worden als een catamorfisme c_2 of als de compositie van een eenvoudig te berekenen functie e en een catamorfisme c_3:

$$\begin{aligned} c_1 \cdot g &\quad = \quad c_2 \quad \text{òf} \\ c_1 \cdot g &\quad = \quad e \cdot c_3 \;. \end{aligned}$$

Omdat veel specificaties van de vorm de compositie van een catamorfisme en een generator zijn, en omdat een catamorfisme vaak eenvoudig naar een efficiënt programma vertaald kan worden, leveren generator-fusiestellingen efficiënte programma's voor specificaties. In dit proefschrift wordt voor een tiental generatoren een generator-fusiestelling geconstrueerd. De generator-fusiestellingen worden toegepast op specifieke problemen, zoals het probleem van het vinden van het langste segment van een rijtje letters dat een palindroom is. Verder worden generator-fusiestellingen gebruikt in de ontwikkeling van de drie volgende theorieën voor het berekenen van algoritmen.

Segmentproblemen

Er zijn een heel aantal problemen die gespecificeerd kunnen worden met behulp van de functie s die gegeven een rijtje letters de verzameling van alle segmenten van dat rijtje genereert. Voorbeelden van dit soort problemen zijn: vind het langste stijgende segment; vind de positie van het segment dat gelijk is aan een gegeven patroon (het patroonherkenningsprobleem); vind het segment waarvan de som van de elementen (elementen zijn in dit geval integers) maximaal is; etcetera. Al deze problemen kunnen worden gespecificeerd als de compositie van een catamorfisme met de functie s. Problemen die gespecificeerd worden als de compositie van een catamorfisme met de functie s noemen we *segmentproblemen*. Voor de generator s leiden we een fusiestelling af: de s-Fusiestelling. De s-Fusiestelling geeft condities waaronder de compositie van een catamorfisme c_1 en de functie s geschreven kan worden als de compositie van een eenvoudig te berekenen functie e en een catamorfisme c_2.

$$c_1 \cdot s \quad = \quad e \cdot c_2 \;.$$

De condities van de s-Fusiestelling zijn heel algemeen, en voor specifieke voorbeelden vergt het verifiëren van de condities vaak nog veel werk. Over specifieke deelklassen van catamorfismen echter kunnen we meer zeggen. Een voorbeeld van zo'n deelklasse vormen de catamorfismen van de vorm: vind het langste rijtje dat aan één of andere eigenschap voldoet.

Als die eigenschap aan bepaalde condities voldoet, dan is aan de eisen van de s-Fusiestelling voldaan. In Hoofdstuk 5 beschouwen we vijf deelklassen van catamorfismen, en voor elke deelklasse formuleren we tenminste één bijbehorend gevolg van de s-Fusiestelling. Deze stellingen worden op een tiental problemen toegepast.

Uitbreidingen naar arrays

De meeste voorbeelden van problemen die we hebben gegeven hebben als argument een element van het datatype 'rijtjes van symbolen'. Het datatype 'rijtjes van symbolen' wordt ook wel het datatype 'arrays' genoemd. In de wis- en natuurkunde komt het datatype matrices, of twee-dimensionale arrays, veel voor. In Hoofdstuk 6 wordt voor willekeurige $n \geq 0$ een definitie van het datatype *n-dimensionale arrays* gegeven, en worden catamorfismen op dit datatype gedefinieerd. Verder blijkt dat een aantal van de voorbeeldproblemen gegeven in het hoofdstuk over segmentproblemen op een voor de hand liggende wijze uitgebreid kan worden naar problemen die gedefinieerd zijn op arrays van willekeurige dimensie. Voor de uitgebreide problemen worden met behulp van generator-fusiestellingen algoritmen geconstrueerd, die efficiënt zijn als ze op een computer uitgevoerd worden. Een van de algoritmen die we construeren in het hoofdstuk over problemen op het datatype arrays is een algoritme voor het vinden van een gegeven n-dimensionaal array (het patroon) in een ander n-dimensionaal array.

Incrementele algoritmen

We hebben de ontwikkeling van theorieën voor het berekenen van algoritmen voor speciale klassen van problemen (namelijk de klasse van segmentproblemen) en het berekenen van algoritmen voor problemen op een specifiek datatype (namelijk het datatype arrays) behandeld. In het laatste hoofdstuk ontwikkelen we een theorie voor het berekenen van een speciaal soort algoritmen, de zogenaamde *incrementele algoritmen*. Stel, een groot bedrijf wil de som van de leeftijd van alle medewerkers bijhouden. Als iemand jarig is, dan is de som van de leeftijden veranderd, en moet de som dus opnieuw berekend worden. In plaats van alle leeftijden van de medewerkers op te tellen, is het efficiënter om bij de som die we al hadden één op te tellen. Een algoritme dat bij het verjaren van een medewerker in plaats van de som opnieuw te berekenen één bij de oude som optelt heet een incrementeel algoritme. In Hoofdstuk 7 geven we een definitie van incrementele algoritmen op verschillende datatypen, en ontwikkelen we theorie voor het rekenen met incrementele algoritmen. Eén van de stellingen die we bewijzen is een fusiestelling voor incrementele algoritmen. De fusiestelling voor incrementele algoritmen geeft condities waaronder de compositie van een functie f en een incrementeel algoritme $\imath_1$ weer een incrementeel algoritme $\imath_2$ is:

$$f \cdot \imath_1 \;=\; \imath_2 \,.$$

Naast het ontwikkelen van theorie voor het rekenen met incrementele algoritmen leiden we ook algoritmen af voor twee praktische problemen, namelijk het probleem van het opslaan van data met zo weinig mogelijk gebruik van geheugen, en het probleem van het afbreken van regels van een tekst zodanig dat zo weinig mogelijk papier wordt verbruikt.

Conclusie

In dit proefschrift hebben we verschillende theorieën voor het berekenen van algoritmen ontwikkeld. De theorieën vormen de eerste stap naar een algemene theorie voor het berekenen van algoritmen. Sommige gedeelten van de theorieën zijn elegant en zeer geschikt voor het rekenen met algoritmen; andere gedeelten van de theorieën, waarin het rekenen met algoritmen moeizaam gaat, zijn minder attractief. De ontwikkelde theorieën dienen in de toekomst uitgebreid en op sommige plaatsen aangepast te worden.

Curriculum Vitae

Johan Theodoor Jeuring

12 augustus 1965:	geboren te Oude Pekela
1977 – 1983:	VWO op de Scholengemeenschap Jan van Arkel te Hardenberg en het Wessel Gansfortcollege te Groningen
1983 – 1987:	Studie wiskunde aan de Rijksuniversiteit Groningen
1988 – 1992:	Onderzoeker in opleiding op het NFI-project STOP: 'Specification and Transformation of Programs' gedetacheerd bij het CWI te Amsterdam en de Rijksuniversiteit Utrecht
6 juli 1992:	vervangende dienstplicht aan de Rijksuniversiteit Utrecht