Data Structures and Amortized Complexity in a Functional Setting

Berry Schoenmakers

Doctoral Dissertation from
Eindhoven University of Technology

September 1992

Acknowledgements

First of all, I would like to thank Anne Kaldewaij for his continuous support and guidance over the past four years. We both enjoyed working on the subject of amortization and learned a lot of interesting new things. Also, I very much appreciate his advice to write and explain things in a relaxed pace (and which he demonstrated to me every so often).

Lex Bijlsma, Victor Dielissen, and Wim Nuij are gratefully acknowledged for discussing and reading together with Anne and me drafts of this thesis and related work. In particular, Wim Nuij is acknowledged for providing me with the basis for Lemma 11.6.

Ronald Heutinck is acknowledged for his initial study of Fibonacci heaps which led to the notion of "diagonal bags" [15]. The late Peter van den Hurk is acknowledged for his study (see [18, Chapter 3]) of the path/insert problem [19], which triggered the research reported in Section 3.2.

I thank Wim Kloosterhuis for his assistance with some of the mathematics in the analyses of bottom-up melding and splaying.

Finally, Daniel D. Sleator is acknowledged for his referee's report of [21] in which he presented an improved analysis of top-down melding.

Contents

Introduction

This thesis has been written from the point of view of an "algorithm designer" and as such it is primarily concerned with two issues, viz. the correctness and the performance of algorithms (or programs). Starting from an abstract specification, the design of an algorithm typically passes through a number of stages, each stage concluded with another approximation of the algorithm, and eventually resulting in a correct and efficient solution. The starting-point of this thesis is that we regard this process of "stepwise refinement" as a *manual* job of a highly *mathematical* nature which is mostly done by a *small* group of people—as is the case with many mathematical activities.

A major distinction between algorithm design and other mathematical activities lies, however, in the kind of efficiency requirements imposed upon algorithms. Although such requirements as time and space limitations tend to complicate the design of algorithms significantly, they are at the same time what makes the field so fascinating; indeed, the beauty of many algorithms lies in the way they are designed to attain the degree of efficiency. The goal of this thesis is to develop a calculational style of programming which deals with all relevant aspects of algorithm design[1] regarding correctness and performance, and which—at the same time—does justice to the beauty of the field.

By a calculational style of algorithm design we mean a design style in which design *decisions* are alternated by *calculations* of the consequences of these decisions. The design decisions constitute the important choices made in a design, whereas the calculations should not contain any significant choices: ideally, it should be possible to reconstruct a calculational design from its design decisions, the calculational details being redundant. Furthermore, to prevent errors, the calculations should be as precise as possible, which requires a suitable formalization of the design. Such a formalization may also help in clarifying design decisions by revealing several alternatives and, in addition, by motivating a particular choice from these alternatives. It should be borne in mind, however, that the way one formalizes things already constitutes a design decision by itself, since particular formalizations may exclude particular solutions.

Combined with the method of stepwise refinement, a calculational design style results in a development of an algorithm in which each next refinement is

[1] Algorithm design includes data structure design, since we view a data structure as a set of *co-operating* algorithms that share the same data type.

calculated from the previous one after some design decisions have been made. In this way, a large design may be divided into parts of manageable sizes. Starting with Back's work [2], considerable effort has been spent on formalizing the method of stepwise refinement within the framework of imperative programming. In this work one distinguishes a simple kind of refinement, often called *algorithmic refinement*, in which both the refined program and its refinement operate on the same state space. A more general—and indeed more powerful—kind of refinement is *data refinement*. It permits one to begin with a program operating on variables of abstract types (like sets and bags), to establish its correctness, and, subsequently, to replace the abstract types by concrete ones and the statements referring to variables of these types by concrete statements.

In the current literature on data refinement (e.g., [11, 25, 4]) we observe, however, that the rules for data refinement involve too many algorithmic details. We believe it is better to make a clear distinction between algorithmic refinement and data refinement. For example, there are rules for data refining such constructs as selections and repetitions, but these rules are rather complicated. Yet, many data refinements are of a much simpler nature, to wit a number of "pointwise" substitutions of a type and connected operations by some other type and corresponding operations. In order to separate such simultaneous substitutions from the overall development of an algorithm, we encapsulate a type and a number of operations in an *algebra*. Such a data refinement step then corresponds to an *algebra refinement*, in which the abstract algebra serves as specification and the concrete algebra as implementation. This dual role of algebras can be recognized also in the use of the terms "abstract data types" and "data structures" in the literature: the former term tends to be preferred on the level of specifications, the latter on the level of implementations.

In addition to a notion of correctness for algebras, we need a notion of their performance. For algorithms, traditional performance measures are the time complexity, the space complexity, and also the length of a program. Similar measures are in order for algebras but, in this thesis, we will concentrate on the time complexity of the operations of an algebra.

A well-known complexity measure for algorithms is their worst-case time complexity. Although this is a useful measure for algorithms on their own, it is, in general, not suitable for the operations of an algebra. The reason is that the worst-case complexities of the operations of an algebra do not always give a clue to the worst-case complexities of the algorithms in which they are applied: for instance, the worst-case complexity of a composition like $g \circ f$ can, in general, not be expressed in terms of the worst-case complexities of the components f and g. This observation led Sleator and Tarjan to the introduction of *amortized costs* for operations of algebras [31][2] in addition to the *actual costs*.

The idea of amortization is to choose the amortized costs for the operations

[2]Tarjan describes amortization as "the *averaging* of the running time of the operations in a sequence over the sequence." Amortized analyses should however not be confused with so-called average-case analyses. In an analysis of the latter type one determines, for instance, the average running time over all inputs of size n, as a function of n. Average-case analysis

such that, for all possible sequences of operations, the sum of the amortized costs is (as good as) equal to the sum of the actual costs of the operations. Moreover, the trick is to do this such that the sum of the *worst-case amortized complexities* of the operations equals the *worst-case complexity*[3] of the entire sequence. Because of correlations between the successive operations that occur in a sequence of operations, the worst-case amortized complexity of an operation may then be significantly smaller than the worst-case complexity of the operation; the importance of amortized complexity measures is, thus, that there exist algebras which are efficient in the amortized sense but not in the traditional worst-case sense.[4]

To determine the amortized costs of an operation we have to study sequences of operations—instead of operations in isolation, as in a traditional worst-case analysis. Sleator and Tarjan devised two techniques to alleviate the task of determining amortized complexities, viz. the *banker's method* and the *physicist's method* [31]. In this thesis we concentrate on the latter method because it lends itself better to a formalization of a calculational way of analyzing algorithms. Briefly, the physicist's method comprises the notion of a *potential function*, which is a real-valued function on the possible values of the data structure (or on the state space of a program). The amortized costs are then defined in terms of this potential function, namely as the actual costs plus the potential difference caused by the operation. Therefore, once the potential function has been chosen, the amortized costs of the operations are fixed. An aspect of data structure design that will receive much attention in this thesis is the derivation of potential functions for which the corresponding amortized costs are low (ideally, as low as possible).

Having elucidated the first two parts of the title of this thesis, we now arrive at the final part, the "*functional* setting". This part comprises the programming style or, more concretely, the program notation. In choosing a program notation there are two important criteria, viz. (a) mathematical tractability of the notation, and (b) concreteness of the notation, i.e., the extent to which it is clear how programs written in the notation are executed on *random access machines*. Criterion (a) is of importance because it determines the ease of proving correctness and also the extent to which it supports a calculational design style. Criterion (b) is of importance because the performance of an algorithm is usually measured relative to a random access machine model; that is, we want to find out how much it costs—whatever that means—to execute an algorithm on a random access machine. (See, for instance, [23] for a formal definition of such a model.) As for criterion (a), we favour a functional style of programming. However, to cope with (b), some imperative features, such as the use of arrays and pointers, are added so as to compensate some deficiencies of purely-functional

---

is often presented as counterpart of worst-case analysis, but amortized analysis can better be considered as a sophisticated form of worst-case analysis.

[3] In full: worst-case *actual* complexity.

[4] Needless to say, such algebras cannot be used in so-called *real-time* situations where *every* application of an operation has to be fast. Such real-time aspects are ignored in this thesis.

languages.

To keep the design of data structures as much as possible within the "functional realm", we will use intermediate algebras to confine the use of imperative features—most notably, to encapsulate the use of arrays and pointers. That is, in many designs, we will introduce a suitable type of trees along with some operations; this gives "customized" tree algebras in terms of which implementations of data structures are described at a relatively high level. The implementation of the tree algebras is done separately. Since this task is usually an order of magnitude simpler than the rest of the design of a data structure, it will not receive much attention in this thesis.

An important aspect of this approach is that the tree algebras serve as formal counterparts of the pictures commonly used in the design of data structures. Instead of bridging the gap between specifications and pointer implementations by means of (usually incomplete) pictures, we will present a precise description of the implementation in terms of formally defined tree algebras. What is more, the complexity analysis can be done at this level as well, which benefits the calculational derivation of potential functions.

Overview

This thesis is divided into two parts. Part I is the more theoretical part in which the above mentioned subjects are elaborated and illustrated with simple examples. Part II consists of a number of case studies which contain more advanced applications of the theory presented in Part I. A major criterion for the selection of the cases in Part II has been the extent to which they serve to illustrate our way of deriving potential functions.

Part I starts with the definition of an algebra as a collection of data types (sets) and operations (relations) in Chapter 1. Notations for important data types and operations are also introduced in this chapter, and it is shown by a number of examples how well-known data structures can be viewed as algebras. Furthermore, a restricted type of algebras, called monoalgebras, is introduced, for which some theory is developed throughout the remainder of Part I.

In Chapter 2, a notion of refinement is defined for monoalgebras. To prepare for this definition, first a notion of data refinement of functions is introduced, which is in turn introduced as a generalization of algorithmic refinement. Our main reason for introducing a notion of algebra refinement is that it enables us to formulate a specification of a data structure as a quest for a concrete refinement of a given abstract algebra. In connection with this, two properties of algebras, called surjectivity and injectivity, are studied at the end of this chapter.

Chapters 3 and 5 are devoted to amortization. In Chapter 3, amortization is first described in an imperative framework. Subsequently, the principle of amortization is discussed in a more abstract setting and it is argued that the physicist's and the banker's methods are equally powerful. Since the former method lends itself better to formalization, we go on to show in Chapter 5 how

potential functions can be used to analyze functional programs in a compositional way. Also a general scheme is presented for monoalgebras according to which amortized costs of various types of operations can be defined.

Chapter 4 introduces the functional program notation used throughout the thesis. In addition to algebras normally provided by purely-functional languages, we will also use algebras involving arrays and pointers in our functional programs. For reasons of efficiency, operations of these algebras are implemented destructively. This brings along some restrictions on the use of these algebras. As will be explained briefly, the well-known method of eager evaluation suffices as simple and efficient evaluation method for all programs in this thesis.

Chapter 6 concludes Part I. This chapter introduces some typical tree algebras, which serve as intermediate algebras in the data structure designs in Part II. Dependent on the set of operations required in a design, a suitable view (read "type") of trees is chosen so that each operation can be defined concisely. In this chapter we also deal with the implementation of these algebras (at pointer level), so that we do not need to address this issue in Part II.

Part II starts with the presentation of some *tricky representations of sets and arrays* in Chapter 7. Basically, an implementation of arrays is presented that achieves $O(1)$ cost for the initialization of arrays of length N (instead of $O(N)$ cost). The standard array implementation achieves $O(1)$ amortized cost for this operation. We use these array implementations to implement bounded sets (subsets of $[0..N)$).

In Chapter 8, *maintaining the minimum of a list*, we present implementations of several list algebras, all of which include an operation for the computation of the minimum value of a list. It is shown how the complexity of these implementations increases as the set of list operations gets more advanced.

Chapter 9 presents two nice implementations of mergeable priority queues, viz. our versions of the top-down and bottom-up *skew heaps* of Sleator and Tarjan. A large part of this chapter is devoted to the calculational derivation of suitable potential functions for these data structures. Our results reduce the bounds obtained by Sleator and Tarjan [30] by more than a factor of two.

In Chapter 10 we present a generalization of *Fibonacci heaps*, which implement mergeable priority queues extended with the so-called "decrease key" operation. This algebra of priority queues plays a central role in some important graph algorithms. A pleasing result of this chapter is that our formal description of Fibonacci heaps easily fits on one page.

Chapter 11 is centered around the derivation of potential functions for three similar operations on trees, viz. *path reversal*, *splaying*, and *pairing*. Although the bounds obtained in this chapter do not improve the bounds obtained by the inventors of these operations, we have included our analyses because we have been able to *derive* the required potential functions to a large extent.

General notions and notation

Besides today's standard notations, many of our notations for general notions such as function application, quantifications, predicates, and equations have been adopted from [6]. Below, we introduce some of these notations; more specific notations will be given throughout the thesis.

Instead of the more traditional notations $f(x)$, $f\ x$, or f_x, we often use $f.x$ to denote function application. For binary operator $\oplus$ and expression E of appropriate type, $(E\oplus)$ denotes the function satisfying $(E\oplus).x = E \oplus x$. The function $(\oplus E)$ is defined similarly. In this notation, doubling may be denoted as $(2*)$ and, of course, also as $(*2)$. Note that $(\oplus E) = (E\oplus)$ whenever $\oplus$ is symmetric (commutative).

The notations for quantifications and constructions all follow the pattern:

<quantifier|constructor> <dummies> : <predicate> : <expression>

enclosed by a pair of delimiters. As quantifiers we use $\forall$, $\exists$, Σ, $\#$, Min, and Max. An example of a quantification is $(\#\, x : x \subseteq \{0,1,2\} : 1 \in x)$, which denotes the number of subsets of $\{0,1,2\}$ containing 1. We use λ as function constructor; for example, $(\lambda\, x : x \subseteq \{0,1,2\} : \{0,1,2\} \setminus x)$ denotes the function that maps the subsets of $\{0,1,2\}$ onto their complements. Using set construction, this function may be denoted as $\{x : x \subseteq \{0,1,2\} : (x, \{0,1,2\} \setminus x)\}$. Hence, in case of set construction it is the pair of delimiters that distinguishes it from other quantifications and constructions. The conventional notation for set construction $\{E \mid P\}$ will be used as abbreviation for $\{x : P : E\}$ when it is clear that x is a dummy.

Similarly, equations are denoted in the form $x : P$, in which unknown x is explicitly mentioned in front of predicate P. The set of solutions of equation $x : P$ equals $\{x : P : x\}$. The construction $(\mu\, x : P : E)$ denotes the smallest solution of $x :\ P\ \wedge\ x = E$.

Finally, it turns out that the number $\phi = (1 + \sqrt{5})/2$ (≈ 1.618) often occurs as base for logarithms in upper bounds for the amortized costs of operations. This number is known as the "golden ratio" and it is the unique positive real number satisfying $\phi - 1 = 1/\phi$.

Part I

Theoretical Aspects

Chapter 1

Algebras

The diversity of studies of data structures, abstract data types, many-sorted algebras, etc. provides evidence for the fundamental role of such structures in computing science. Indeed, many designs of sophisticated algorithms hinge on the right choice of data structures and efficient implementation thereof. Such structures, consisting of a number of data types and operations, are the subject of this monograph; we call them *algebras* to reflect our mathematical view of these structures.

In descriptions of programming languages one generally opposes the data structures of the language to its control structures. This natural division between "data" and "control" can be traced back to primitive models of computation such as Turing machines. Taking Church's thesis for granted, Turing machines are powerful enough to describe any algorithm; hence, any algorithm can be considered as a composition from a relatively small repertoire of operations that manipulate data. A Turing machine consists of a (finite) control part and an (infinite) tape used to store data. Combined with a tape head, the tape forms a simple data structure with operations "read", "write", "move left" and "move right". By means of these operations the initial contents of the tape are transformed into the corresponding output according to the program in the control part. The distinction between control structures and data structures can also be recognized in the Von Neumann style of computers in use nowadays. Abstract models of this type of computers are known as "random access machines" (see e.g. [23]); the primitive data structure used in this model is an infinite array of cells, called memory, whose contents can be modified by means of move instructions and arithmetic instructions.

So, "data" and "control" can be viewed as two more or less orthogonal aspects, and therefore they are often studied independently. For instance, in semantics of imperative programs, the types of variables and the forms of expressions are usually ignored, but one concentrates on iteration and recursion constructs, say. In this thesis, we concentrate on data structures and to us it is important that a programming language provides—among other things—a number of algebras, which are abstractions of the more primitive algebras provided

by machine languages.

The remainder of this chapter consists of two sections. In the next section we introduce some frequently used data types and operations. These types and operations are used in two ways: simply as mathematical objects, and as components of algebras or, more generally, as parts of algorithms. In the other section we introduce our notion of algebras and we provide some examples of typical algebras which can be implemented in many programming languages. Furthermore, a restricted form of algebras, called *monoalgebras*, is introduced for which some theory is developed in later chapters.

1.1 Basic data types and operations

In addition to standard notations for well-known mathematical objects such as sets and arithmetical operators, we introduce some home-grown notations, in particular for structured types like finite lists. We would like to stress that only notation matters to us in this section, not foundations; the concepts we use have been founded rigorously elsewhere, and we take that as a starting-point.

1.1.1 Tuples

We consider *tuples* as elements of cartesian products. The *empty tuple*, the only member of the empty cartesian product, is denoted by $\perp$. Nonempty tuples are either enclosed by a pair of parentheses or by a pair of angular brackets. To select the respective components of a nonempty tuple t we write $t.0$ to denote the first component, $t.1$ to denote the second one, and so on. For example, $\langle x, y\rangle.1 = y$.

1.1.2 Relations and functions

A *(binary) relation* is a set of ordered pairs. For relation R, we denote its *domain* $\{x, y : (x, y) \in R : x\}$ by $\operatorname{dom} R$, and its *range* $\{x, y : (x, y) \in R : y\}$ by $\operatorname{rng} R$. For $x \in \operatorname{dom} R$, we use $R.x$ (in addition to $R(x)$ and $R\ x$) to denote the *application of R to x*, which means that $R.x$ denotes *a* solution of the equation $y : (x, y){\in}R$. In accordance with this notation for relation application—in which the "input" stands to the right of the relation's name—we introduce yRx as a shorthand for $(x, y){\in}R$. This notation goes well with the conventional notation for relation composition in which the rightmost relation is applied first:

$$
\begin{array}{ll}
z\,(S \circ R)\,x \;\equiv\; (\exists\, y :: zSy \;\wedge\; yRx) & \textit{composition } S \circ R \textit{ of } R \textit{ and } S \\
y\,R^{\star}\,x \;\equiv\; xRy & \textit{dual } R^{\star} \textit{ of } R \\
y\,R{\restriction}X\,x \;\equiv\; yRx \;\wedge\; x \in X & \textit{restriction } R{\restriction}X \textit{ of } R \textit{ to } X.
\end{array}
$$

A relation R satisfying $yRx \Rightarrow y{=}x$ for all x and y, is called an *identity (relation)*. Hence, $=$ is an identity itself. A relation whose domain is a singleton set is said to have *no inputs*.

A relation f is called a *function* when it satisfies *Leibniz's rule*, which states that $x{=}y \Rightarrow f.x{=}f.y$ for all $x, y \in \mathrm{dom}\, f$. In other words, for a function f, yfx equivales $x{\in}\mathrm{dom}\, f \wedge y{=}f.x$. With "relation" replaced by "function", the above introduced nomenclature for relations is also used for functions. A function without inputs is also called a *constant (function)*. Note that identities are functions. By $f[x{:=}y]$, with $x \in \mathrm{dom}\, f$, we denote the function equal to f except that x's image is y.

For sets X and Y, we distinguish the following kinds of relations and functions:

$$\begin{array}{ll} \mathcal{P}(X{\times}Y) = \{R \mid \mathrm{dom}\, R{\subseteq}X \ \wedge\ \mathrm{rng}\, R{\subseteq}Y\} & \text{relations on } X \text{ and } Y \\ X{\rightharpoonup}Y = \{f \mid \mathrm{dom}\, f{\subseteq}X \ \wedge\ \mathrm{rng}\, f{\subseteq}Y\} & \text{partial functions from } X \text{ to } Y \\ X{\to}Y = \{f \mid \mathrm{dom}\, f{=}X \ \wedge\ \mathrm{rng}\, f{\subseteq}Y\} & \text{(total) functions from } X \text{ to } Y. \end{array}$$

Note that $X{\to}Y \subseteq X{\rightharpoonup}Y \subseteq \mathcal{P}(X{\times}Y)$, and also that $\emptyset$ is a (total) function of type $\emptyset{\to}Y$, a partial function of type $X{\rightharpoonup}Y$, and a relation of type $\mathcal{P}(X{\times}Y)$, for every X and Y.

Remark 1.1
Apart from the different meaning of yRx we have adhered to conventional notations for relations and functions as much as possible. By using yRx as shorthand for $(x, y) \in R$, our notation is such that data flows from right to left on the *entire* level of application, while it flows in the opposite direction on the typing level; for instance, we write $z = g.(f.x)$ for functions $f \in X \to Y$ and $g \in Y \to Z$.
□

1.1.3 Simple data types

To begin with, we have the *unit type* $\{\bot\}$. This trivial type serves as the domain of relations without inputs. These relations are of the form $\{\bot\} \times T$ with T nonempty, and we denote them by $?_T$ (subscript $_T$ is omitted when the type is clear from the context). We also use $?_T$ as abbreviation for $?_T.\bot$; in that case $?_T$ denotes an arbitrary value of type T. This value is not fixed, so we do not know whether $?_T = ?_T$—unless T is a singleton type, of course. Note that $?_T$ is a constant if T is a singleton type; in that case we write—as usual—c instead of $?_T$, where c is the unique element of T.

Another finite type is the set of *booleans* $\{\mathrm{false}, \mathrm{true}\}$, denoted by Bool. Boolean operators are denoted by the same symbols used for predicates, which are after all boolean-valued functions.

Finally, there are the *numerical types* Nat (naturals), Int (integers), and Real (reals), together with the well-known arithmetical operators. For these types we use standard notations, except perhaps for our notation for *intervals*: $[a..b)$ denotes set $\{x \mid a \leq x < b\}$, in which the type of dummy x depends on the context; the three other variations are $[a..b]$, $(a..b]$, and $(a..b)$, which speak for themselves.

In connection with numerical types we use the values ∞ and $-\infty$. As is common practice, it depends on the context whether a numerical type contains these values or not. For example, $+$ and $\max$ are both arithmetic operators of type $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$, but usually Int is equal to $(-\infty..\infty)$ in case of $+$, while it equals $[-\infty..\infty]$ for $\max$.

1.1.4 Structured data types

Let T be a nonempty type. In order of increasing structure, we have the finite sets over T, the finite bags over T, the finite lists over T, and the finite binary trees over T, or "sets", "bags", "lists", and "trees" for short. These so-called structured types are denoted by $\{T\}$, $\llparenthesis T \rrparenthesis$, $[T]$, and $\langle T \rangle$, respectively, and in the sequel we will sometimes call elements of these types "structures". In accordance with this notation, we will also use the above pairs of brackets to form structures of the respective types. In particular, we use $\{\,\}$, $\llparenthesis\,\rrparenthesis$, $[\,]$, and $\langle\,\rangle$ to denote the empty set, the empty bag, the empty list, and the empty tree. Similarly, $\{a\}$, $\llparenthesis a \rrparenthesis$, $[a]$, and $\langle a \rangle$ denote singleton structures, and, for instance, $\llparenthesis 1, 2, 1 \rrparenthesis$ denotes a three-element bag. In this notation, expressions like $[\{0\}]$ denote both a type and a structure. The role of types and structures is however quite different such that the context usually resolves such ambiguities.

To denote the *size* of a structure, we use the symbol $\#$; in case of lists we also speak of "length" instead of "size". Furthermore, we write $(a\in)$ for the function that tells us whether a *occurs* in a structure. In case T is linearly ordered, we use $\downarrow x$ to denote the *minimum* value occurring in a nonempty structure x, and $\uparrow x$ to denote its *maximum* value. For some T, such as Int, minimum and maximum of the empty structure are defined as well, respectively equal to ∞ and $-\infty$.

So much for the common notations for the structured types. Next we introduce some notations which are specific to each of these types.

For finite sets we use, in addition to standard set notation, $\Downarrow S$ to denote set $S \setminus \{\downarrow S\}$, for nonempty S. As a restricted version of $\cup$ we have $\uplus$ which denotes *disjoint set union*; that is, $S \uplus T$ is defined if and only if $S \cap T = \{\,\}$. Hence, $\uplus$ is a partial function from $\{T\} \times \{T\}$ to $\{T\}$. Similarly, $\oplus$ and $\ominus$ with $S \oplus a = S \cup \{a\}$ and $S \ominus a = S \setminus \{a\}$ are partial set operations which are defined only if $a \notin S$ and $a \in S$, respectively.

For a bag B, $\Downarrow B$ stands for the bag obtained by removing a single occurrence of $\downarrow B$ from B: $\Downarrow B = B \ominus \llparenthesis \downarrow B \rrparenthesis$, for B nonempty. Here, $\ominus$ denotes *bag subtraction*, and *bag summation* is denoted by $\oplus$.

For lists we have a quite comprehensive set of operations. First of all, to construct nonempty lists we have $\vdash$ (*cons*) and $\dashv$ (*snoc*): $a \vdash s$ denotes the list with *head* a and *tail* s, and $s \dashv a$ denotes the list with *front* s and *last element* a. Note that $[a]$ abbreviates both $a \vdash [\,]$ and $[\,] \dashv a$. Furthermore, $s +\!\!+ t$ denotes the *catenation* of lists s and t. To dissect nonempty lists, we use $hd.s$, $tl.s$, $ft.s$, $lt.s$, $s{\uparrow}n$, and $s{\downarrow}n$ $(0{\le}n{\le}\#s)$ to select, respectively, the head, tail, front, last element, *prefix* of length n, and *suffix* of length $\#s{-}n$ of s. (Hence, $s = s{\uparrow}n +\!\!+ s{\downarrow}n$.) Also,

we let $s.n$ stand for the $(n{+}1)$-st element of s $(0{\le}n{<}\#s)$, and $rev.s$ denotes the *reverse* of s.

Finally, for binary trees, we use $\langle t, a, u\rangle$ to denote a nonempty binary tree with *left subtree* t, *root* a, and *right subtree* u. To select the respective components of a nonempty tree, we use functions l, m, and r ("left", "middle", and "right"). Note that $\langle a\rangle$ is short for $\langle\langle\,\rangle, a, \langle\,\rangle\rangle$. In addition to $\#$, which denotes the actual size of a tree, it is often convenient to use the actual size *plus one* in efficiency analyses because it is positive. We denote it by $\sharp$, and it may be defined recursively by $\sharp\langle\,\rangle = 1$ and $\sharp\langle t, a, u\rangle = \sharp t + \sharp u$.

The common use of $\#$, $\in$, $\downarrow$, and $\uparrow$ for the structured types is justified by the following natural correspondence between these types. First of all, the *inorder traversal* of a tree converts a tree into a list:

$$\begin{array}{lcl} \overline{\langle\,\rangle} & = & [\,] \\ \overline{\langle t, a, u\rangle} & = & \overline{t} +\!\!+ [a] +\!\!+ \overline{u}. \end{array}$$

Similarly, a list can be converted into a bag:

$$\begin{array}{lcl} \overline{[\,]} & = & \{\!|\,|\!\} \\ \overline{a \vdash s} & = & \{\!|a|\!\} \oplus \overline{s}. \end{array}$$

And, finally, a bag can be converted into a set:

$$\begin{array}{lcl} \overline{\{\!|\,|\!\}} & = & \{\,\} \\ \overline{\{\!|a|\!\} \oplus B} & = & \{a\} \cup \overline{B}. \end{array}$$

These conversions leave the set of values present in a structure intact, but each application of $\overline{\cdot}$ destroys some structure until that set is obtained. Note that $\overline{\cdot}$ also leaves the size of a structure intact, except that for bags $\#B$ is larger than $\#\overline{B}$ when B contains duplicates.

1.2 Algebras

Independent of a particular programming style, the following notion of algebras combined with a notion of algebra refinement is believed to be of great importance to algorithm design, since it provides a basis for a good separation of concerns.

Definition 1.2
An *algebra* A consists of a sequence of sets τ and a sequence of relations o. It is written

$$\mathsf{A} = (\,\tau \mid o\,).$$

The sets are called A's *data types*, and the relations are called A's *operations*.
□

For the scope of this chapter, the order of the data types and operations is irrelevant; it plays a role in our definition of algebra refinement in the next chapter. By removing data types and/or operations from an algebra and possibly reordering the data types and operations, we obtain, what we call, a *subalgebra*. Algebra (|) is a subalgebra of every algebra.

The following examples give an idea of the aspects of algebras that are considered throughout this thesis.

Example 1.3
One of the most basic algebras is the algebra of booleans:

$$(\,\mathrm{Bool} \mid \mathrm{false}, \neg, \wedge\,).$$

Other boolean operations like $\vee$ and $\equiv$ can be expressed in terms of the operations of this algebra. For example, true may be obtained as $\neg$false, and $a \vee b$ is equivalent to $\neg(\neg a \wedge \neg b)$. So, the operations of this algebra form a basis for the boolean operations.

It is in general a good idea to keep the set of operations of an algebra small and simple but, unfortunately, this advice does not help in all circumstances. For instance, an equally powerful algebra, using the well-known "nand" operator (the composite of $\wedge$ and $\neg$), is:

$$(\,\mathrm{Bool} \mid \mathrm{false}, \neg \circ \wedge\,).$$

Although this algebra has only two operations, it is more complicated to express the other boolean operations in terms of these two operations.
□

In the algebra of booleans, Bool is the only data type that matters[1], but in general the operations involve more than one data type. However, when implementing such a set of operations, we concentrate on a single data type—common to all operations—that has to be implemented. This data type then becomes the data type of the algebra, and the implementation of the remaining types is taken for granted.

Example 1.4
Algebras involving numerical types are available in almost any programming language. Here is a very simple one:

$$(\,\mathrm{Nat} \mid 0, (+1), (-1), (=0)\,).$$

We remark that (-1) is a partial function w.r.t. the data type of this algebra; its domain is Nat$\backslash\{0\}$.

Numbers are usually represented by digit sequences. A *unary* implementation of the above algebra is for example:

[1] Actually, this is not quite true. See Remark 2.24

$$(\,[\{0\}]\ \mid\ [\,], (\dashv 0), ft, (=[\,])\,).$$

In this implementation, natural number n is represented by $(\dashv 0)^n.[\,]$, the list of n zeros. A *binary* implementation might look as follows:

$$(\,[\{0,1\}]\ \mid\ \mathsf{zero}, \mathsf{suc}, \mathsf{pred}, \mathsf{iszero}\,),$$

with

$$\begin{array}{lcl}
\mathsf{zero} & = & [\,] \\
\mathsf{suc}.[\,] & = & [1] \\
\mathsf{suc}.(s \dashv 0) & = & s \dashv 1 \quad , s \neq [\,] \\
\mathsf{suc}.(s \dashv 1) & = & \mathsf{suc}.s \dashv 0 \\
\mathsf{pred}.[1] & = & [\,] \\
\mathsf{pred}.(s \dashv 1) & = & s \dashv 0 \quad , s \neq [\,] \\
\mathsf{pred}.(s \dashv 0) & = & \mathsf{pred}.s \dashv 1 \\
\mathsf{iszero}.s & = & s = [\,].
\end{array}$$

Only binary lists containing at least one 1 are in the domain of pred (hence, $[\,] \notin \mathrm{dom}\,\mathsf{pred}$). The non-recursive alternatives in the definition of pred will therefore terminate any evaluation of pred.

A more useful numerical algebra is for instance:

$$(\,\mathrm{Int}\ \mid\ 1, +, -, *, \mathrm{div}\,, \mathrm{mod}\,, \max, \min, =, <, >\,).$$

A drawback of this algebra is that each integer has to be generated from the integer 1; e.g., 0 may be obtained as $1-1$. In many programming languages numerical values can be generated from their decimal representation. This facility may be provided as follows:

$$(\,\mathrm{Int}\ \mid\ \mathit{Dec2Int}, \mathit{Int2Dec}, +, -, *, \mathrm{div}\,, \mathrm{mod}\,, \max, \min, =\,).$$

Operations *Dec2Int* and *Int2Dec* convert types [[0..10)] and Int into each other.

For reals we also have such conversion operations. In most languages only a subset of the reals can be generated in this way, and operations return approximations of the exact result. Some languages, however, offer exact real-arithmetic. For instance, the following algebra can be available:

$$(\,\mathrm{Real}\ \mid\ e, \pi, \mathit{nth}\,),$$

where $\mathit{nth} \in \mathrm{Nat}\times\mathrm{Real} \to [0..10)$ and $\mathit{nth}.n.\alpha$="$(n+1)$-st digit of the fractional part of the decimal representation of α".

To describe conversions between integers and reals, say, we need an algebra with more than one data type. Given algebras $(\,\mathrm{Int}\ \mid\ \sigma\,)$ and $(\,\mathrm{Real}\ \mid\ \varsigma\,)$, the following algebra is appropriate:

$$(\,\mathrm{Int}, \mathrm{Real}\ \mid\ \sigma, \varsigma, \mathit{Int2Real}, \mathit{Real2Int}\,).$$

The conversion from Real to Int is for instance done by rounding, and the reverse conversion may simply be the identity on Int.
□

Many of the aspects touched upon in the previous example will be ignored in the rest of this thesis. That is, the ins and outs of built-in algebras will not be discussed much further, but we will mainly concentrate on efficient implementations of more abstract algebras.

Example 1.5
The algebra of *stacks* is usually provided as "kernel" for the list operations in functional languages. A possible definition is

$$([T] \mid [], (=[]), \vdash, hd, tl),$$

for arbitrary type T. The operations of this algebra are usually assumed to have $O(1)$ time complexity. Other list algebras like the algebra of *queues*, defined by

$$([T] \mid [], (=[]), \dashv, hd, tl),$$

can be implemented using this kernel. Since $\dashv$ may be expressed in terms of the stack-operations and $\vdash$ may be expressed in terms of the queue-operations, both algebras are equally powerful. Taking efficiency into consideration, however, gives a different picture. Programming $\dashv$ in terms of stack-operations will lead to a linear time program, while the algebra of queues can be implemented such that $\dashv$ takes constant time only (see Section 4.5).

Examples of more advanced list algebras that we shall encounter in the sequel are concatenable queues (in Section 4.5):

$$([T] \mid [], (=[]), \dashv, hd, tl, +\!\!+),$$

and stacks extended with the minimum operation (in Chapter 8):

$$([T] \mid [], (=[]), \vdash, hd, tl, \downarrow),$$

where T is assumed to be linearly ordered so that $\downarrow$ is defined.
□

The above list algebras are all equally powerful in the sense that their operations are rich enough to program any operation on lists. The relative efficiencies of (the implementations of) these algebras, however, make up the difference. For example, operation $\downarrow$ is included as additional operation in the last algebra because it is possible that there exist more efficient ways of implementing $\downarrow$ than by just programming it in terms of the stack-operations. In general, an operation is not only included in an algebra because it cannot be programmed in terms of the other operations, but also because it cannot be programmed *without loss of efficiency* in terms of the other operations!

Algebras involving sets and bags are usually not provided by programming languages, but they are used frequently in (abstract) programs.

Example 1.6
The following algebra is one of the many variations of so-called *priority queues*:

$$(\text{Int} \mid \emptyset, \{\cdot\}, \oplus, (=\emptyset), \downarrow, \Downarrow).$$

Implementations of priority queues often use some kind of trees to represent bags. In Part II we will give several examples thereof.
□

Arrays and pointers are two important data types provided by most imperative languages. As algebras they might look as follows.

Example 1.7
For natural N and nonempty type T, we consider an algebra with data type $[0..N) \to T$. Elements of this data type are called *arrays*. For array a we use $a[i]$ as alias for $a.i$, $0 \leq i < N$, and $a[i{:=}x]$ denotes the array equal to a except that $a[i{:=}x][i] = x$. Furthermore, there is operation ? (cf. Section 1.1.3) to create an arbitrary array which may serve as an initial value. In summary, the algebra of arrays looks like

$$([0..N) \to T \mid ?, \mathit{lookup}, \mathit{update}),$$

with $\mathit{lookup}.a.i = a[i]$ and $\mathit{update}.a.i.x = a[i{:=}x]$. Usually all operations of this algebra are assumed to have $O(1)$ time complexity. Therefore, ? is a relation since if it were to be a function, evaluation of ? should always return the same value (of type $[0..N) \to T$), but this requires $O(N)$ time for the standard implementation of arrays. So, this is an example of an algebra with a relational operation: the outcome of $?[0] = ?[0]$, for example, is indeterminate. Furthermore, evaluation of $a[i{:=}x]$ will in general destroy the representation of a, since the value of $a[i]$ is simply overwritten to achieve $O(1)$ time complexity. The usage of such *destructive* operations will be discussed in Chapter 4.
□

Example 1.8
Pointer structures are similar to arrays in the sense that they can be viewed as functions too. The difference is that arrays are total functions on a finite interval of integers, whereas pointer structures are partial functions on some, usually anonymous, domain. To sketch the idea behind pointer types we consider the following algebra:

$$(\Omega, \Omega \rightharpoonup T \mid \text{nil}, =, \{(\text{nil}, ?_T)\}, \mathit{value}, \mathit{new}, \mathit{assign}, \mathit{dispose}),$$

where

Ω is some set ("addresses of memory cells");

T is some nonempty type ("contents of memory cells");

$\text{nil} \in \Omega$;

$=$ denotes equality on Ω;
$value \in \Omega \times (\Omega \rightharpoonup T) \rightharpoonup T$ is defined by $value.p.s = s.p$, for $p \in \mathrm{dom}\, s$;
$new \subseteq (\Omega \rightharpoonup T) \times (\Omega \times (\Omega \rightharpoonup T))$ is defined by $new.s = (p, s')$ with $p \notin \mathrm{dom}\, s$ and $s' = s \cup \{(p, ?_T)\}$, for $\mathrm{dom}\, s \neq \Omega$;
$assign \in \Omega \times T \times (\Omega \rightharpoonup T) \rightharpoonup (\Omega \rightharpoonup T)$ is defined by $assign.p.x.s = s[p{:=}x]$, for $p \in \mathrm{dom}\, s$;
$dispose \in \Omega \times (\Omega \rightharpoonup T) \rightharpoonup (\Omega \rightharpoonup T)$ is defined by $dispose.p.s = s \backslash \{(p, s.p)\}$, for $p \in \mathrm{dom}\, s \backslash \{\mathrm{nil}\}$.

Elements of Ω are *pointers* and an element of $\Omega \rightharpoonup T$ can be considered as a piece of *memory*. Operation *new* is not a function: the outcome of $new.s$ not only depends on the value of s but also on the way this value has been obtained.

Type T often refers to Ω. For example, $T = \mathrm{Int} \times \Omega$ for singly-linked lists, and in such a case several linked lists may be represented in the same part of memory; think, for instance, of an array of linked-lists. The advantage of a pointer type (over an array type) is that the same part of memory, namely Ω, can be used to represent a number of lists with a total number of $\#\Omega$ elements, where the distribution over the lists evolves dynamically, depending on the input.

Note that $\emptyset \in \Omega \rightharpoonup T$ cannot be constructed by means of the operations of this algebra; in particular, nil is always in the domain. Note that we allow *value* and *assign* to operate on nil.
□

In Section 4.3, we will introduce some Pascal-like notation for pointers in an informal way, and on the whole we will not be very formal about pointers.

1.3 Monoalgebras

Instead of the general notion of algebras as defined in Definition 1.2, we will use the following restricted version as our "working definition" throughout Part I.

Definition 1.9
A *monoalgebra* A is an algebra with a *single, nonempty* data type A and a *finite* number of operations. The operations should be *functional*, and *first-order with respect to A*. An operation is called first-order w.r.t. A when both its domain and its range are (subsets of) cartesian products composed of data type A and data types of other algebras not involving algebra A. As representatives of such operations we take

(a) *creations* of type $T \to A$

(b) *transformations* of type $A \times A \rightharpoonup A$

(c) *inspections* of type $A \to T$,

where T stands for data types of algebras not involving A.
□

Remark 1.10
As is common practice in definitions of programming languages, algebras are identified by the names of their data types, since the data type uniquely determines the set of operations belonging to it. *In general, however, we should use algebras instead of their data types to type operations.* In Definition 1.9, for instance, A and T may be equal when viewed as sets, but they should be data types of different algebras—or else creations and inspections are indistinguishable.
□

Generally, creations and transformations are used to generate values of the data type, whereas inspection operations are used to discriminate values. Characteristic of inspections is that A does not occur in the range. The difference between creations and transformations is that A does not occur in the domain of a creation, whereas it occurs at least once in the domain of a transformation. The representatives in Definition 1.9 have been chosen such that the definitions and proofs involving monoalgebras (particularly, in Chapter 2) can be limited to these cases. The requirement that operations of monoalgebras should be first-order w.r.t. the data type of the algebra means, informally, that the arguments of the operations should be "first-order data": the domains and ranges of first-order operations are cartesian products of A and other types, but a function type like $A \to A$ is not allowed as component of such a cartesian product.

However, operations of type $[A] \to A$, $(\!|A|\!) \to \langle A \rangle$, $\{\bot\} \to [\{A\}]$, etc., are also considered as first-order w.r.t. A: in all these cases, arguments involving type A are exclusively manipulated by the operation itself, not by other arguments of the operation. The following example deals with such an operation.

Example 1.11
A well-known higher-order operation on lists is $\star$ ("map"), defined by $(f \star s).i = f.(s.i)$ for $0 \leq i < \#s$. Such an operation may be added to the algebra of stacks:

$$(\,[T] \mid [\,], (=[\,]), \vdash, hd, tl, \star\,),$$

with $\star$ of type $(T \to T) \times [T] \to [T]$. Although $\star$ takes a function as parameter, implementation of the above algebra is not difficult because the implementation of the function parameter has to be taken care of elsewhere. That is, given a program for f, $(f\star)$ can be programmed in terms of the stack-operations. In our terminology, $\star$ is therefore *not* a higher-order operation w.r.t. type $[T]$. Still, operations like $\star$ will not receive much attention in the sequel.
□

In the previous section some algebras have been exhibited that violate the restrictions imposed in the definition of monoalgebras. In Example 1.7, the algebra of arrays provides an example of an algebra with a nonfunctional operation, though a very simple one. Operation *new* in Example 1.8 is also nonfunctional. At the end of Example 1.4 we have seen an algebra with more than one data type,

and in the next example we encounter an algebra with a higher-order operation w.r.t. its data type. In pathological cases we may also encounter algebras with an empty data type (Example 2.18).

Besides the fact that most algebras in this thesis are monoalgebras, most of them also share the property that the elements of their data types are objects of finite size. Algebra $(\,\mathrm{Real} \mid e, \pi, nth\,)$ from Example 1.4 is an example of a monoalgebra with objects of infinite size. Another illustration of objects of infinite size is presented in the next example, but there the algebra is not a monoalgebra.

Example 1.12
Let L denote the set of *infinite* lists over some anonymous universe. To generate infinite lists we use a fixpoint constructor μ of type $(L{\to}L){\curvearrowright}L$, which returns a solution of equation $s : s = F.s$, where $F \in L \to L$ is such that this equation has a unique solution (see e.g. [17, Section 5.5]). In some functional programming languages the following algebra is then available:

$$(\,L \mid \mu, hd, tl, \vdash\,).$$

Note that μ is a *higher-order* operation w.r.t. L, since it takes a function on L as parameter; therefore, this is not a monoalgebra.

An example of an infinite list created by means of μ is $\mu.(1 \vdash)$, the infinite lists of ones. Another example is the sorted list of naturals, corresponding to the function F given by $F.s = 0 \vdash (+1) \star s$.
□

In the next chapters we concentrate on monoalgebras. As will become apparent in the sequel, the essential point about a monoalgebra is that a number of connected operations involving the same data type are grouped together. Typical properties of monoalgebras are that all elements of its data type can be constructed by means of its operations, and also that distinct elements of the data type can be distinguished by means of (compositions of) the operations of the algebra. Removal or addition of a single operation may affect these properties drastically—as we shall see in the next chapter. In the design of data structures, it is also a well-known phenomenon that one or two extra operations may complicate efficient implementations of an algebra significantly.

Chapter 2

Algebra refinement

The main reason to introduce algebra refinement is that we want to use it as a formal, yet powerful and flexible, mechanism for specifying data structures. A specification of a data structure then asks for a *concrete* refinement of a given *abstract* algebra or, less formally, it asks for an implementation of a given algebra. An obvious characteristic of this approach to the design of data structures is that algebras play the role of specifications as well as the role of implementations.

As the majority of algebras in Part II of this thesis are monoalgebras, we shall define refinement only for this restricted class of algebras (Section 2.4). To prepare for this definition, we first discuss refinement of functions (and functional programs). In Section 2.3 we introduce data refinement as a generalization of algorithmic refinement, and—as stepping-stone towards our definition of algorithmic refinement in Section 2.2—specifications of functions are introduced in the next section.

2.1 Functional programs and specifications

We do not present a definition of a program notation in this chapter, because neither the actual borderline between abstract and concrete nor the efficiency of concrete programs is relevant for the development of a refinement theory—in contrast to the use of such a theory. What is more, the notion of a program is decoupled from the notion of a program notation in our refinement theory.

Definition 2.1
A *program* is a definition of a relation. If the defined relation is functional, then the program is called *functional* as well. The name of a program is the name of the relation defined by it. Relative to a particular program notation, a program is called *concrete* if it is expressed in the program notation; otherwise, it is called *abstract*.
□

In this chapter we deal with functional programs only; therefore we just use "program" instead of "functional program", or even "function", since in many

contexts the actual definition of a function is irrelevant. Furthermore, we will use "abstract data types" and "concrete data types", "abstract algebras" and "concrete algebras", and so on, all relative to a particular program notation. We also use the terms "abstract" and "concrete" to refer to the refined object and its refinement, respectively; with respect to many program notations this is correct, but in some refinements this use of "abstract" and "concrete" does not comply with Definition 2.1. This is believed to cause no major difficulties.

As follows from its definition in Section 1.1.2, a function is completely defined by its domain and a function value for each value in the domain. In a specification we make a similar division:

Definition 2.2
A *specification* is a pair $\langle D, P\rangle$, in which D is a set and P is a predicate (on functions whose domain include D) satisfying

$$(\forall\, f, g : D \subseteq \mathrm{dom}\, f \;\wedge\; D \subseteq \mathrm{dom}\, g \;\wedge\; f{\restriction}D = g{\restriction}D : P(f) \equiv P(g)).$$

□

The restriction is imposed on $\langle D, P\rangle$ to exclude pathological specifications in which $P(f)$ depends on values $f.x$ where $x \notin D$. Usually, this restriction is trivially satisfied:

Property 2.3
For any set D and predicate Q (of the appropriate type),

$$\langle D, (\lambda\, f : D \subseteq \mathrm{dom}\, f \;:\; (\forall\, x : x \in D : Q(x, f.x)))\rangle$$

is a specification.
□

Specifications of this form may be called "pointwise" specifications because each function value is specified independent of the other function values. Note that not every specification can be written in this form. For instance, specification $\langle \mathrm{Bool}, (\lambda\, f : \mathrm{Bool} \subseteq \mathrm{dom}\, f \;:\; f.\mathrm{false} \neq f.\mathrm{true})\rangle$ cannot be written as a pointwise specification.

Definition 2.4
Function f is said to *satisfy* specification $\langle D, P\rangle$ when

$$D \subseteq \mathrm{dom}\, f \;\wedge\; P(f).$$

□

Instead of writing specifications as pairs, we usually formulate them in a less strict way. We write, for example: "Design a program for function $sum \in [\mathrm{Int}]{\to}\mathrm{Int}$ satisfying $(\forall\, s : s \in [\mathrm{Int}] : sum.s = (\Sigma\, i : 0 \leq i < \#s : s.i))$". In this specification *sum* is a dummy, so we are free to use any name we like for our solutions; in particular, we can use different names to distinguish successive refinements of the same specification. Moreover, we tacitly imply by this formulation that a function with a domain *larger* than [Int] is also satisfactory.

2.2 Algorithmic refinement of functions

Function g is called an algorithmic refinement of function f when every specification satisfied by f is satisfied by g as well. More formally:

Definition 2.5
Function f is said to be *algorithmically refined* by function g when

(1) $(\forall\, D, P : f \text{ satisfies } \langle D, P\rangle : g \text{ satisfies } \langle D, P\rangle)$.

□

Using this definition of algorithmic refinement, the verification of a particular refinement is rather cumbersome. Fortunately, we can derive, on account of the restriction in Definition 2.2, the following more useful characterizations of algorithmic refinement:

Property 2.6
Predicate (1) is equivalent to

(2) $\mathrm{dom}\, f \;\subseteq \mathrm{dom}\, g \;\wedge\; (\forall\, x : x \in \mathrm{dom}\, f \;: f.x = g.x)$,

and to

(3) $f \subseteq g$.

Proof To prove that (1) implies (2) for any f and g, we instantiate (1) with specification

$$\langle D, P\rangle = \langle \mathrm{dom}\, f\,, (\lambda\, h : \mathrm{dom}\, f \;\subseteq \mathrm{dom}\, h \;: (\forall\, x : x \in \mathrm{dom}\, f \;: f.x = h.x))\rangle.$$

Then f satisfies $\langle D, P\rangle$, hence so does g, which is equivalent to (2).

Next, assume (2) for some f and g. Then we observe for any specification $\langle D, P\rangle$:

$$\begin{array}{ll} & f \text{ satisfies } \langle D, P\rangle \\ \equiv & \{\ \text{Definition 2.4}\ \} \\ & D \subseteq \mathrm{dom}\, f \;\wedge\; P(f) \\ \Rightarrow & \{\ (2), \text{ hence } D \subseteq \mathrm{dom}\, g \text{ and } f\!\upharpoonright\! D = g\!\upharpoonright\! D;\ \text{Definition 2.2}\ \} \\ & D \subseteq \mathrm{dom}\, g \;\wedge\; P(g) \\ \equiv & \{\ \text{Definition 2.4}\ \} \\ & g \text{ satisfies } \langle D, P\rangle, \end{array}$$

which settles the proof of the equivalence of (1) and (2).

The equivalence of (2) and (3) is obvious.

□

From (2) we infer that an algorithmic refinement is allowed to have a larger domain. Characterization (3) is more appropriate than (2) for deriving properties of algorithmic refinement. For instance, from (3) it follows immediately that algorithmic refinement induces a *partial order* on functions. Moreover, any application of f may be replaced by an application of g when $f \subseteq g$. In particular, composition is *monotonic* w.r.t. algorithmic refinement.

Property 2.7

$f \subseteq f$	reflexivity
$f \subseteq g \ \wedge \ g \subseteq f \ \Rightarrow \ f = g$	anti-symmetry
$f \subseteq g \ \wedge \ g \subseteq h \ \Rightarrow \ f \subseteq h$	transitivity
$f \subseteq g \ \Rightarrow \ f \circ h \subseteq g \circ h \ \wedge \ h \circ f \subseteq h \circ g$	monotonicity of $\circ$.

□

Characterization (2) is useful when one has to show that a particular *recursively* defined program is a correct refinement. For instance, the fact that function $g \in [\text{Int}] \rightarrow \text{Int}$, given by $g.[\,] = 0$ and $g.(a \vdash s) = a + g.s$, is a correct refinement of function f of the same type, defined by $f.s = (\Sigma\, i : 0 \leq i < \#s : s.i)$, is easily proved by establishing the second conjunct of (2) by induction on s.

The notion of algorithmic refinement enables us to specify a programming problem as a request for a concrete refinement of a given function. The summation problem from the previous section, for instance, may be specified by "Design a concrete refinement of program *sum* defined by $\text{dom}\, sum = [\text{Int}]$ and $sum.s = (\Sigma\, i : 0 \leq i < \#s : s.i)$". (Note that in this specification *sum* is *not* a dummy.) Since such specifications employ a notion of refinement of *functions*, they are bound to be deterministic. Since we regard such a specification method too restrictive, we have introduced nondeterministic specifications as well. In the next section we shall see that the notion of data refinement also enables a form of nondeterministic specifications that is especially suited for specifying data structures.

2.3 Data refinement of functions

The notion of algorithmic refinement enables us to relate the successive programs in a program development. The types of these programs are strongly related: the domains form an ascending sequence with respect to set inclusion. The types of the *components* of these programs may, however, be quite different. For example, suppose $f_1 \circ f_0 \in X \rightarrow Y$ is refined by $g_1 \circ g_0 \in X \rightarrow Y$, with $f_0 \in X \rightarrow A$, $f_1 \in A \rightarrow Y$, $g_0 \in X \rightarrow C$, and $g_1 \in C \rightarrow Y$. Then neither f_0 and g_0 nor f_1 and g_1 can be related when types A and C are incompatible. In such a refinement, A is typically the more abstract type and C the more concrete type. To relate such types and functions the notion of *data refinement* is introduced.

We introduce data refinement as a generalization of algorithmic refinement. For this purpose we rewrite characterization (2) of algorithmic refinement as follows:

$$(\forall\, x, y : x \in \mathrm{dom}\, f \;\;\wedge\;\; x = y : y \in \mathrm{dom}\, g \;\;\wedge\;\; f.x = g.y).$$

To relate functions of different types, we replace the equality signs by arbitrary relations (and rename the dummies for convenience later on):

Definition 2.8
Function f is said to be *data-refined* by function g under relations R and S, when

$$(4) \quad (\forall\, a, c : a \in \mathrm{dom}\, f \;\;\wedge\;\; a\, R\, c : c \in \mathrm{dom}\, g \;\;\wedge\;\; f.a\, S\, g.c).$$

□

In the above context, relations R and S are called coupling relations, or just *couplings*. When discussing (data) refinements we will often abuse the terms "abstract" and "concrete" to distinguish the refined objects from their refinements. The refined objects are called abstract and their refinements concrete, although this might give rise to "inconsistencies": e.g., in one refinement a data type is treated as abstract, while it is considered as concrete in another refinement.

Usually, couplings R and S are strongly related or even identical. Furthermore, identities are often used as couplings.

Example 2.9 (see Example 1.4)
In many applications of Definition 2.8, the couplings between the domains and ranges of the functions are described in terms of a relation between an abstract and a concrete type. In the specification below, for example, Nat is the abstract type and C the specified concrete type.

> Design a concrete type C, a relation $\simeq$ between Nat and C, and concrete functions $\mathsf{zero} \in C$, $\mathsf{iszero} \in C \rightharpoonup \mathrm{Bool}$, $\mathsf{suc} \in C \rightharpoonup C$, and $\mathsf{pred} \in C \rightharpoonup C$ satisfying
>
> $$\begin{array}{l} 0 \simeq \mathsf{zero} \\ (\forall\, n, c : n \simeq c : c \in \mathrm{dom}\, \mathsf{iszero} \;\;\wedge\;\; n{=}0 \equiv \mathsf{iszero}.c) \\ (\forall\, n, c : n \simeq c : c \in \mathrm{dom}\, \mathsf{suc} \;\;\wedge\;\; n{+}1 \simeq \mathsf{suc}.c) \\ (\forall\, n, c : n{\neq}0 \;\wedge\; n \simeq c : c \in \mathrm{dom}\, \mathsf{pred} \;\;\wedge\;\; n{-}1 \simeq \mathsf{pred}.c). \end{array}$$

This specification contains four instances of (4). The identity on $\{\bot\}$ couples the domains of 0 and zero, which are, as usual, omitted. The identity on Bool serves as coupling between the ranges of $(=0)$ and iszero. Relation $\simeq$ couples the remaining domains and ranges.

A simple solution to this problem is to take a unary representation for naturals like type $[\{0\}]$. The coupling between Nat and $[\{0\}]$ is then defined by $n \simeq s \;\equiv\; n = \#s$. Under this coupling, we have the following refinements:

$0 \in \mathrm{Nat}$	$\mathsf{zero} \in [\{0\}]$	$\mathsf{zero} = [\,]$
$(=0) \in \mathrm{Nat} \to \mathrm{Bool}$	$\mathsf{iszero} \in [\{0\}] \to \mathrm{Bool}$	$\mathsf{iszero}.s = (s = [\,])$
$(+1) \in \mathrm{Nat} \to \mathrm{Nat}$	$\mathsf{suc} \in [\{0\}] \to [\{0\}]$	$\mathsf{suc}.s = 0 \vdash s$
$(-1) \in \mathrm{Nat} \rightharpoonup \mathrm{Nat}$	$\mathsf{pred} \in [\{0\}] \rightharpoonup [\{0\}]$	$\mathsf{pred}.(0 \vdash s) = s,$

where $\mathrm{dom}\,(-1) = \mathrm{Nat} \setminus \{0\}$ and $\mathrm{dom}\,\mathsf{pred} = [\{0\}] \setminus \{[\,]\}$. The verification of the correctness of this solution is straightforward.
□

In this example, all refinements are described in terms of the same coupling $\simeq$ (and the identities on $\{\bot\}$ and Bool). This is possible because our definition of data refinement does not refer to the types of the functions and the coupling relations. Instead of (4), an alternative definition of data refinement could be:

$$(\forall\, a, c : a\, R\, c : f.a\, S\, g.c)$$

in a context in which $R \subseteq \mathrm{dom}\, f \times \mathrm{dom}\, g$ (the context defines, for instance, $f \in A \to B$, $g \in C \to D$, and $R \subseteq A \times C$). With such a definition, the justification of the refinement of (-1) by pred requires $\simeq \setminus \{([\,], 0)\}$ as instantiation of R instead of $\simeq$, because $\simeq$ is a relation between Nat and $[\{0\}]$, not between $\mathrm{Nat} \setminus \{0\}$ and $[\{0\}] \setminus \{[\,]\}$. And for the refinement of $-$ (subtraction on the naturals) we would need another coupling. In this way we get a tailored coupling for each function that is partial with respect to the domain of the coupling and such an approach would lead to an awkward definition of algebra refinement.

Just as for algorithmic refinement, there exists a more succinct characterization of data refinement, which may be interpreted as a generalization of (3):

Property 2.10
Predicate (4) is equivalent to

(5) $\quad f \circ R \subseteq S \circ g.$

Proof

$$\begin{array}{ll} & (\forall\, a, c : a \in \mathrm{dom}\, f \;\wedge\; aRc : c \in \mathrm{dom}\, g \;\wedge\; f.aSg.c) \\ \equiv & \quad \{ \text{ one-point rule (twice) } \} \\ & (\forall\, a, b, c : b = f.a \;\wedge\; a \in \mathrm{dom}\, f \;\wedge\; aRc : (\exists\, d : d{=}g.c : c{\in}\mathrm{dom}\, g \wedge bSd)) \\ \equiv & \quad \{ \text{ property of functions } \} \\ & (\forall\, a, b, c : bfa \;\wedge\; aRc : (\exists\, d :: bSd \;\wedge\; dgc)) \\ \equiv & \quad \{ \text{ predicate calculus: } (\forall\, x :: P(x) \Rightarrow Q) \equiv (\exists\, x :: P(x)) \Rightarrow Q \; \} \\ & (\forall\, b, c : (\exists\, a :: bfa \;\wedge\; aRc) : (\exists\, d :: bSd \;\wedge\; dgc)) \\ \equiv & \quad \{ \text{ definition of } \circ \text{ (twice) } \} \\ & (\forall\, b, c : b(f \circ R)c : b(S \circ g)c) \\ \equiv & \quad \{ \text{ definition of } \subseteq \; \} \\ & f \circ R \subseteq S \circ g. \end{array}$$

□

Interesting properties of data refinement are easily derived from this characterization. For example, the composition of two successive data refinements is again a data refinement:

Property 2.11 (cf. Property 2.7)

$$f \circ I \subseteq I \circ f \qquad \text{“reflexivity”}$$

$$f \circ R_0 \subseteq S_0 \circ g \ \wedge\ g \circ R_1 \subseteq S_1 \circ h \ \Rightarrow\ f \circ (R_0 \circ R_1) \subseteq (S_0 \circ S_1) \circ h \qquad \text{“transitivity”}$$

$$f_0 \circ R \subseteq S \circ g_0 \ \wedge\ f_1 \circ S \subseteq T \circ g_1 \ \Rightarrow\ (f_1 \circ f_0) \circ R \subseteq T \circ (g_1 \circ g_0) \qquad \text{“monotonicity”},$$

where I is an identity relation with $\operatorname{dom} I \supseteq \operatorname{dom} f \cup \operatorname{rng} f$.

Proof Relying on the associativity of $\circ$ and its monotonicity w.r.t. $\subseteq$ (Property 2.7d), we observe for the last two parts:

$$\begin{array}{ll} & f \circ R_0 \circ R_1 \\ \subseteq & \{\ f \circ R_0 \subseteq S_0 \circ g\ \} \\ & S_0 \circ g \circ R_1 \\ \subseteq & \{\ g \circ R_1 \subseteq S_1 \circ h\ \} \\ & S_0 \circ S_1 \circ h; \end{array} \qquad \begin{array}{ll} & f_1 \circ f_0 \circ R \\ \subseteq & \{\ f_0 \circ R \subseteq S \circ g_0\ \} \\ & f_1 \circ S \circ g_0 \\ \subseteq & \{\ f_1 \circ S \subseteq T \circ g_1\ \} \\ & T \circ g_1 \circ g_0. \end{array}$$

□

2.4 Refinement of monoalgebras

In the development of programs by stepwise refinement, one often encounters a data refinement step which comprises simultaneous substitutions of a type and a number of connected operations by some other type and corresponding operations. In order to separate such substitutions from the overall development, we introduce the notion of *algebra refinement*. In a refinement of a monoalgebra there is only one relevant coupling which we usually denote by $\simeq$.

Definition 2.12
Let A be a monoalgebra with data type A. Let C be a monoalgebra with data type C and as many operations as A. Let $\simeq$ be a relation on C and A. Then algebra A is said to be *(data-)refined* by algebra C under coupling $\simeq$ when each operation of A is data-refined by the corresponding operation of C under coupling $\simeq$, which is defined as follows for each of the three representatives of first-order operations (cf. Definitions 1.9 and 2.8).

(a) Creation $f \in T \to A$ is data-refined by creation $g \in T \to C$ under coupling $\simeq$ if

$$(\forall\, x : x \in T : f.x \simeq g.x).$$

(b) Transformation $f \in A \times A \rightharpoonup A$ is data-refined by transformation $g \in C \times C \rightharpoonup C$ under coupling $\simeq$ if

$$(\forall\, a, b, c, d : (a, b) \in \mathrm{dom}\, f \;\wedge\; a \simeq c \;\wedge\; b \simeq d : (c, d) \in \mathrm{dom}\, g \;\wedge\; f.a.b \simeq g.c.d)\ .$$

(c) Inspection $f \in A \to T$ is data-refined by inspection $g \in C \to T$ under coupling $\simeq$ if

$$(\forall\, a, c : a \simeq c : f.a = g.c).$$

Algebra C is said to *implement* A when there exists a coupling $\simeq$ such that C refines A under $\simeq$.
□

As a corollary of Property 2.11 we then have

Property 2.13

(a) Algebra A is refined by A under the identity coupling on A's data type.

(b) If A is refined by C under coupling $\simeq$ and C is refined by E under coupling $\cong$, then A is refined by E under coupling $\simeq \circ \cong$.

□

The definition of algebra refinement enables us to formulate specifications of data structures very concisely. For instance, the cumbersome problem description in Example 2.9, in which the specifications of the respective operations are all very much alike, may be rendered as follows:

> Design a concrete refinement of algebra $(\,\mathrm{Nat} \mid 0, (=0), (+1), (-1)\,)$ with signature $(\,C \mid \mathsf{zero}, \mathsf{iszero}, \mathsf{suc}, \mathsf{pred}\,)$ under coupling $\simeq$.

In such specifications we use *signatures* to name the respective components of the requested refinement. A definition of an algebra assigns a value to each component of its signature in the same fashion as a definition of a function assigns a value to its name. Instead of prescribing a signature and a name for the coupling, we may also leave this open by saying:

> Design an implementation of algebra $(\,\mathrm{Nat} \mid 0, (=0), (+1), (-1)\,)$.

The specifications of zero, iszero, suc, and pred in Example 2.9 are specific instances of the general pattern implied by Definition 2.4. The following example exhibits some more "typical data refinements" involving one nontrivial coupling $\simeq$, the other couplings being identities.

Example 2.14
Let A, C and T be data types. Let $\simeq$ be a relation on C and A. We say that function f is data-refined by function g under coupling $\simeq$ when

$$\begin{array}{ll} f \simeq g & \text{for constants } f \in A \text{ and } g \in C \\ (\forall\, a, c : a \simeq c : f.a \simeq g.c) & \text{for } f \in A{\to}A \text{ and } g \in C{\to}C \\ (\forall\, a, c : a \in \operatorname{dom} f \ \wedge\ a \simeq c : c \in \operatorname{dom} g \ \wedge\ f.a \simeq g.c) & \text{for } f \in A{\rightharpoonup}A \text{ and } g \in C{\rightharpoonup}C \\ (\forall\, a, c : a \in \operatorname{dom} f \ \wedge\ a \simeq c : c \in \operatorname{dom} g \ \wedge\ f.a = g.c) & \text{for } f \in A{\rightharpoonup}T \text{ and } g \in C{\rightharpoonup}T. \end{array}$$

Now, strictly speaking, (c) and (d) are conflicting: in case $A = C = T$ both are applicable but they yield different results. Sets A, C, and T are however intended to be data types of different algebras, so a possible way to prevent such conflicts is to use the algebras instead of the data types to type functions (see Remark 1.10). In this thesis we do not wish to be so rigorous because in our examples it will be clear which parts have to be refined.

Note that in (b), $a \simeq c$ implies $a \in A$ and $c \in C$, hence (b) is a special case of (c). Furthermore, note that (a) and (b) express that g is data-refined by f under coupling $\simeq^\star$, hence, in these cases data refinement is "symmetric". In general, however, data refinement is "asymmetric" since the refinement may have a larger domain in the sense that its domain contains values to which no values in the domain of the refined function are coupled (cf. algorithmic refinement).
□

Instead of "typical data refinements" we can also use the term "induced couplings":

Example 2.15
A coupling $\simeq$ between A and C induces couplings between for example (a) $T \times A$ and $T \times C$, (b) $[A]$ and $[C]$, and (c) $\langle [A] \rangle$ and $\langle [C] \rangle$. Such induced couplings are denoted by the same name and defined in the obvious way:

(a) $(x, a) \simeq (y, c) \ \equiv\ x = y \ \wedge\ a \simeq c$

(b) $as \simeq cs \ \equiv\ \#as = \#cs \ \wedge\ (\forall\, i : 0 \le i < \#as : as.i \simeq cs.i)$

(c) $\langle\,\rangle \simeq \langle\,\rangle$ and $\langle la, as, ra \rangle \simeq \langle lc, cs, rc \rangle \ \equiv\ la \simeq lc \ \wedge\ as \simeq cs \ \wedge\ ra \simeq rc.$

Note that in the last case, only trees of the same shape are coupled to each other. Together with Definition 2.8, induced couplings provide an alternative way to describe "typical data refinements".

The data refinement relation between $A{\to}A$ and $C{\to}C$ can also be considered as an induced coupling (of a higher order, though). Couplings between for instance $\{A\}$ and $\{C\}$ are induced in a less straightforward way.
□

In many applications, the coupling relation is a *function* from C to A. We call such couplings *abstractions*, and we will frequently use $[\![\cdot]\!]$ instead of $\simeq$ to denote an abstraction. The coupling in Example 2.9, for instance, is an abstraction from $[\{0\}]$ to Nat, viz. $[\![s]\!] = \#s$. In general, the coupling corresponding to abstraction $[\![\cdot]\!]$ is given by $a \simeq c \;\equiv\; a = [\![c]\!]$.

Another important case is that the *dual* of a coupling is a function from A to C. We call such couplings *representations*, and we use $(\!|\cdot|\!)$ instead of $\simeq^\star$ to denote these. For example, we have $(\!|n|\!) = (\dashv 0)^n.[\,]$ as representation in Example 2.9. In general, the coupling corresponding to representation $(\!|\cdot|\!)$ is given by $a \simeq c \;\equiv\; (\!|a|\!) = c$. When describing refinements with representation functions, we will henceforth omit *: we speak of a refinement under $(\!|\cdot|\!)$ instead of a refinement under $(\!|\cdot|\!)^\star$, as for example in the following property.

Property 2.16 (cf. Definition 2.12)
Let C refine A under representation $(\!|\cdot|\!)$, $(\!|\cdot|\!) \in A \to C$ and under abstraction $[\![\cdot]\!]$, $[\![\cdot]\!] \in C \to A$, respectively. Then we have for the corresponding operations of A and C, respectively:

(a) Creations $f \in T \to A$ and $g \in T \to C$ satisfy

$$(\forall\, x : x \in T : (\!|f.x|\!) = g.x),$$

and

$$(\forall\, x : x \in T : f.x = [\![g.x]\!]).$$

(b) Transformations $f \in A{\times}A \rightharpoonup A$ and $g \in C{\times}C \rightharpoonup C$ satisfy

$$(\forall\, a, b : (a, b) \in \operatorname{dom} f \;:\; ((\!|a|\!), (\!|b|\!)) \in \operatorname{dom} g \;\;\wedge\;\; (\!|f.a.b|\!) = g.(\!|a|\!).(\!|b|\!)),$$

and

$$(\forall\, c, d : ([\![c]\!], [\![d]\!]) \in \operatorname{dom} f \;:\; (c, d) \in \operatorname{dom} g \;\;\wedge\;\; f.[\![c]\!].[\![d]\!] = [\![g.c.d]\!]).$$

(c) Inspections $f \in A \to T$ and $g \in C \to T$ satisfy

$$(\forall\, a : a \in A : f.a = g.(\!|a|\!)),$$

and

$$(\forall\, c : c \in C : f.[\![c]\!] = g.c).$$

□

To keep this property simple we have confined it to total abstraction functions and total representation functions. Partial abstraction functions are particularly useful when describing pointer implementations; for example, see Section 4.3 where abstraction $[\![\cdot]\!]$ is defined only for pointers that correspond to a finite list.

Two frequently occurring instantiations of the general pattern implied by Property 2.16 are as follows.

Example 2.17
Let A and C be data types. Let $(\!|\cdot|\!) \in A \to C$. Then f is said to be data-refined by g under representation $(\!|\cdot|\!)$ when

$$\begin{array}{ll} (\!|f|\!) = g & \text{for constants } f \in A \text{ and } g \in C \\ (\forall\, a : a \in \operatorname{dom} f \;:\; (\!|a|\!) \in \operatorname{dom} g \;\wedge\; (\!|f.a|\!) = g.(\!|a|\!)) & \\ & \text{for } f \in A \leadsto A \text{ and } g \in C \leadsto C. \end{array}$$

Let $[\![\cdot]\!] \in C \to A$. Then f is said to be data-refined by g under abstraction $[\![\cdot]\!]$ when

$$\begin{array}{ll} f = [\![g]\!] & \text{for constants } f \in A \text{ and } g \in C \\ (\forall\, c : [\![c]\!] \in \operatorname{dom} f \;:\; c \in \operatorname{dom} g \;\wedge\; f.[\![c]\!] = [\![g.c]\!]) & \\ & \text{for } f \in A \leadsto A \text{ and } g \in C \leadsto C. \end{array}$$

□

Since many data refinements can be expressed conveniently in terms of functional couplings, one may wonder why we have introduced relational couplings at all. Well, a compelling reason is that transitivity cannot be formulated for functional data refinements, when data refinements are used under representations as well as under abstractions. For instance, a data refinement of type A by C under representation $(\!|\cdot|\!) \in A \to C$, followed by a data refinement of type C by E under abstraction $[\![\cdot]\!] \in E \to C$ is a data refinement under coupling $(\!|\cdot|\!)^\star \circ [\![\cdot]\!]$ (Property 2.13b), but $(\!|\cdot|\!)^\star \circ [\![\cdot]\!]$ is in general not a function. By restricting couplings to either representations or abstractions, transitivity is of course retained, but—as we will argue in the next section—this is too restrictive to be useful.

2.5 Surjectivity and injectivity of monoalgebras

A well-known phenomenon in the design of data structures is that a single addition to the repertoire of operations can make all the difference to the implementation. This is demonstrated in the following example.

Example 2.18
In this example Nat plays again the role of abstract type. We will successively refine the following algebras:

$$\begin{array}{lcl} \mathsf{Nuts0} & = & (\,\mathrm{Nat} \mid (+1)\,) \\ \mathsf{Nuts1} & = & (\,\mathrm{Nat} \mid (+1), 0\,) \\ \mathsf{Nuts2} & = & (\,\mathrm{Nat} \mid (+1), 0, (=0)\,) \\ \mathsf{Nats} & = & (\,\mathrm{Nat} \mid (+1), 0, (=0), (-1)\,), \end{array}$$

by a concrete algebra with data type C under coupling $\simeq \subseteq C \times \mathrm{Nat}$, such that the size of C is minimal in each refinement.

We start with Nuts0. An immediate consequence of the definition of data refinement (cf. (2.14b)) is that $\emptyset$ refines $(+1)$ under coupling $\emptyset$. Hence, in this case we may take $C = \emptyset$. This result is to be expected because a data refinement of a single operation of type $A \to A$ is pointless, since there is no way to provide this operation with an argument. For example, we cannot use Nuts0 to refine $(+1).0$, since we have no corresponding data refinement of 0.

Nuts1 provides 0 as operation. To refine 0 we see that we cannot take C empty anymore (cf. (2.14a)). It is, however, still possible to refine $(+1)$ and 0 as follows: take as concrete type $\{0\}$ and as coupling $\{(0, n) \mid n \in \mathrm{Nat}\}$; then the respective refinements are $\{(0, 0)\}$ and 0. Hence, in this case C is a singleton. This result is not strange either because, although it is possible to generate any natural number with operations 0 and $(+1)$, there is no way to distinguish any two distinct naturals. So, it is not surprising that it is possible to represent all natural numbers by the same value.

The situation changes slightly when we consider Nuts2. Again, however, we are able to refine these operations using a finite concrete type. Operationally speaking, $(=0)$ may be used to distinguish 0 from the positive naturals, but there is no way to distinguish distinct positive naturals. We may therefore choose $\{0, 1\}$ as concrete type and $\{(0, 0)\} \cup \{(1, n{+}1) \mid n \in \mathrm{Nat}\}$ as coupling. The operations of Nuts2 are refined by $\{(0, 1), (1, 1)\}$, 0, and $(=0)$, respectively.

Finally, by adding operation (-1) the concrete type cannot be finite anymore: we use a unary representation in which a list of n zeros represents natural n.

The respective refinements are summarized in the following table.

	C	suc	zero	iszero	pred
Nuts0	$\emptyset$	$\mathsf{suc} = \emptyset$			
Nuts1	$\{0\}$	$\mathsf{suc}.0 = 0$	$\mathsf{zero} = 0$		
Nuts2	$\{0, 1\}$	$\mathsf{suc}.0 = 1$ $\mathsf{suc}.1 = 1$	$\mathsf{zero} = 0$	$\mathsf{iszero}.b = b{=}0$	
Nats	$[\{0\}]$	$\mathsf{suc}.s = s \dashv 0$	$\mathsf{zero} = [\,]$	$\mathsf{iszero}.s = s{=}[\,]$	$\mathsf{pred}.(s \dashv 0) = s$

For Nuts0, coupling $\simeq$ is the empty set; for the other refinements, couplings $\simeq$ are given by $n \simeq 0$, $0 \simeq 0$ and $n{+}1 \simeq 1$, and $\#s \simeq s$, respectively.

All these refinements can also be described as refinements under a representation. For Nuts0, representation $(\!|\cdot|\!)$ is the empty set, which is a partial function from Nat to C. For the other refinements, representation $(\!|\cdot|\!)$ is a total function of type $\mathrm{Nat} \to C$, with, respectively, $(\!|n|\!) = 0$, $(\!|0|\!) = 0$ and $(\!|n{+}1|\!) = 1$, and $(\!|n|\!) = (\dashv 0)^n.[\,]$. However, only the refinements of Nuts0 and Nats can be described as refinements under an abstraction $[\![\cdot]\!] \in C \to \mathrm{Nat}$, viz. $[\![\cdot]\!] = \emptyset$ and $[\![s]\!] = \#s$. The other refinements cannot be refinements under an abstraction, since the existence of abstraction $[\![\cdot]\!]$ implies that C has to be infi-

nite: C cannot be empty, for $\mathsf{zero} \in C$; by induction, $[\![\mathsf{suc}^n.\mathsf{zero}]\!] = n$, hence $(\forall\, m, n : m \neq n : \mathsf{suc}^m.\mathsf{zero} \neq \mathsf{suc}^n.\mathsf{zero})$.

By interchanging the roles of abstract and concrete we see that there are also refinements that cannot be described as refinements under a representation.
□

From this example we learn that there are refinements that cannot be described by abstractions or representations. Algebras Nuts1 and Nuts2 are, however, a bit artificial and therefore we will present some more realistic examples shortly; in these examples it is even impossible to use functional couplings. The fact that there exist interesting nonfunctional couplings is another compelling reason to use relational couplings in Definition 2.8.

Yet, many designs of data structures can be modelled as refinements under an abstraction function—often even a *surjective* one. In the sequel we investigate under which circumstances this is the case. An algebra is called *injective* when all its refinements can be described by means of abstractions. To prepare for a formal definition of injectivity we first introduce the notion of *surjectivity*.

Definition 2.19
For monoalgebra A, the set of "reachable" values is defined as the largest subset X of A's data type satisfying

$$(\forall\, a : a \in X : (\exists\, c :: a \simeq c)),$$

for all algebras C and relations $\simeq$, for which C refines A under coupling $\simeq$. We call X the *range* of A and denote it by $\mathrm{rng}\,\mathsf{A}$. A monoalgebra is called *surjective* when its range equals its data type.
□

The values outside the range of an algebra need not be represented in an implementation. Operationally speaking, the range of an algebra contains all values of its data type that can be generated by any composition of its operations.

Example 2.20

$$\begin{array}{lcl}
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0\,) & = & \{0\} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid (+1)\,) & = & \emptyset \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, (+1)\,) & = & \mathrm{Nat} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, (+2)\,) & = & \{2n \mid n \in \mathrm{Nat}\} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, 1, (+2)\,) & = & \mathrm{Nat} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, (*2), (+1) \circ (*2)\,) & = & \mathrm{Nat} \\
\mathrm{rng}\,(\,[T] \mid [\,], \vdash\,) & = & [T].
\end{array}$$

Although the equalities in this example will be intuitively clear, a proof requires some effort. Let us prove the middle one.

Let $\mathsf{N2}$ denote algebra $(\,\mathrm{Nat} \mid 0, (+2)\,)$. We first remark that $\mathsf{N2}$ is refined by $(\,\mathrm{Nat} \mid 0, (+1)\,)$ under coupling $\simeq$ given by $2n \simeq n$. So, there exists a refinement of $\mathsf{N2}$ in which no odd natural is coupled to a concrete value. This implies $\mathrm{rng}\,\mathsf{N2} \subseteq \{2n \mid n \in \mathrm{Nat}\}$.

To prove the other inclusion $\mathrm{rng}\,\mathsf{N2} \supseteq \{2n \mid n \in \mathrm{Nat}\}$, we reason as follows. Suppose $(\,C \mid \mathsf{zero}, \mathsf{suc2}\,)$ refines $\mathsf{N2}$ under coupling $\simeq$. Then we prove $(\forall\, n :: (\exists\, c :: 2n \simeq c))$ by induction on n.

Case $n = 0$. Since zero refines 0, we conclude that $0 \simeq \mathsf{zero}$.

Case $n = m{+}1$. We derive

$$
\begin{array}{ll}
 & (\exists\, c :: 2m \simeq c) \\
\Rightarrow & \{\ \mathsf{suc2} \text{ refines } (+2) \text{ under } \simeq\ \} \\
 & (\exists\, c :: 2m{+}2 \simeq \mathsf{suc2}.c) \\
\Rightarrow & \{\ \text{dummy transformation } c := \mathsf{suc2}.c\ \} \\
 & (\exists\, c :: 2m{+}2 \simeq c) \\
\equiv & \{\ n = m{+}1\ \} \\
 & (\exists\, c :: 2n \simeq c).
\end{array}
$$

□

So, by considering surjective algebras only, some strange ones are eliminated. However, an algebra like $(\,\mathrm{Nat} \mid 0, (+1)\,)$ is still a bit strange. Although all naturals can be generated with 0 and $(+1)$, there is no way to distinguish any two distinct naturals. We introduce the notion of *injectivity* to capture this phenomenon.

Definition 2.21
A monoalgebra A is called *injective* when

$$(\forall\, c :: (\#\, a : a \in \mathrm{rng}\,\mathsf{A} \ : a \simeq c) \leq 1),$$

for all algebras C and relations $\simeq$, for which C refines A under coupling $\simeq$.
□

Note that an algebra is injective when its range is empty. The operational interpretation is that an algebra is injective when any two distinct values in its range can be distinguished by some composition of its operations; in other words, there should exist compositions that produce different results for distinct values.

Example 2.22

$(\,\mathrm{Nat} \mid (+1)\,)$	is injective
$(\,\mathrm{Nat} \mid 0, 1, (=0)\,)$	is injective

$(\,\mathrm{Nat} \mid 0, (+1), (=0)\,)$	is not injective
$(\,\mathrm{Nat} \mid 0, (+2), (-2), (=0)\,)$	is injective
$(\,\mathrm{Nat}\backslash\{0\} \mid 1, (*2), (\mathrm{div}\,2), (=1)\,)$	is injective
$(\,\mathrm{Real} \mid e, \pi, nth\,)$	is injective (see Example 1.4)
$(\,[\{0\}] \mid [\,], (=[\,]), \vdash, tl\,)$	is injective
$(\,[\mathrm{Bool}] \mid [\,], (=[\,]), \vdash, tl\,)$	is not injective
$(\,[T] \mid [\,], (=[\,]), \vdash, hd, tl\,)$	is injective, provided T belongs to an injective algebra.

To give an example of a proof of injectivity we demonstrate that Nats from Example 2.18 is injective.

Let $(\,C \mid \mathsf{suc}, \mathsf{zero}, \mathsf{iszero}, \mathsf{pred}\,)$ refine Nats under coupling $\simeq$. Since Nats is surjective we have to show that $(\forall\, c :: (\#\, n :: n \simeq c) \leq 1)$. To this end, we observe for any c and for any m and n, $m \leq n$:

$$
\begin{array}{ll}
 & m \simeq c \ \wedge\ n \simeq c \\
\Rightarrow & \{\ \mathsf{pred} \text{ refines } (-1) \text{ under } \simeq\ \} \\
 & m{-}m \simeq \mathsf{pred}^m.c \ \wedge\ n{-}m \simeq \mathsf{pred}^m.c \\
\Rightarrow & \{\ \mathsf{iszero} \text{ refines } (=0) \text{ under } \simeq\ \} \\
 & 0 = 0 \ \equiv\ \mathsf{iszero}.(\mathsf{pred}^m.c) \ \wedge\ n{-}m = 0 \ \equiv\ \mathsf{iszero}.(\mathsf{pred}^m.c) \\
\Rightarrow & \{\ \text{predicate calculus}\ \} \\
 & m = n.
\end{array}
$$

□

In the above proof of injectivity of Nats, we have directly applied Definition 2.21. Another way to show this result is by programming = on Nat in terms of the operations of Nats:

Property 2.23 (=-test)
Consider a surjective monoalgebra A with data type A. If = of type $A{\times}A \to$ Bool can be expressed in terms of the operations of A, then algebra A is injective.

Proof Let C refine A under coupling $\simeq$. Define **eq** in terms of the operations of C in the same way as = is expressed in terms of the operations of A. Then **eq** refines = under $\simeq$. That is:

$$(\forall\, a, b, c, d : a \simeq c \ \wedge\ b \simeq d : a = b \ \equiv\ \mathbf{eq}.c.d).$$

To show that A is injective, suppose that c is coupled to both a and b, that is suppose that $a \simeq c$ and $b \simeq c$. Instantiation of the above quantification then yields $a = b \ \equiv\ \mathbf{eq}.c.c$. Since $\mathbf{eq}.c.c$ is, for instance, also equivalent to $a = a$, which is equivalent to true, we conclude that $a = b$.
□

Remark 2.24
The algebras in Example 1.3 are surjective but not injective because they do not provide any inspection operations: all operations involve type Bool (and $\{\bot\}$), but for an inspection we need an extra type. According to our definition of algebra refinement, a possible implementation of $(\,\mathrm{Bool} \mid \mathrm{false}, \neg \circ \wedge\,)$ is therefore $(\,\{0\} \mid 0, \{((0,0),0)\}\,)$. This implementation is however not very useful and to exclude it we have to add an inspection operation to the algebra of booleans, say $(=\mathrm{false}) \in \mathrm{Bool} \to \{F,T\}$. Here, $\{F,T\}$ is the data type of a both surjective and injective algebra, and we simply have to postulate the existence of this algebra. We cannot describe an implementation of such an algebra other than by assuming the existence of another algebra of this type; in the end, we need to postulate the existence of such an algebra (as "hardware").
□

Note that omission of "$a \in \mathrm{rng}\,\mathsf{A}$" from Definition 2.21 yields that algebra $(\,\mathrm{Nat} \mid 0, 1, (=0)\,)$ is not injective. Indeed, omitting $a \in \mathrm{rng}\,\mathsf{A}$ turns about every nonsurjective algebra into a noninjective one, as the reader may verify.

From the results in Examples 2.20 and 2.22 it follows that Nats is bijective:

Definition 2.25
A monoalgebra A is called *bijective* when it is both surjective and injective, that is, when

$$(\forall\, a :: (\#\, c :: a \simeq c) \geq 1) \;\wedge\; (\forall\, c :: (\#\, a :: a \simeq c) \leq 1),$$

for all algebras C and relations $\simeq$, for which C refines A under coupling $\simeq$.
□

The importance of this notion is that any refinement of a bijective algebra can be described as a data refinement under a (surjective) abstraction function. The conclusion of our investigation is that we can safely use abstractions in specifications of injective algebras, without running the risk of excluding efficient refinements that can only be justified using a representation function or a coupling relation (as in Example 2.18).

Unfortunately, it is not always obvious that an algebra is bijective. For example, for positive m and n, algebra $(\,\mathrm{Nat} \mid 0, (+m), (-n), (=0)\,)$ has range $\{k\ \gcd(m,n) \mid k \in \mathrm{Nat}\}$, the multiples of $\gcd(m,n)$. Hence, it is surjective precisely when $\gcd(m,n) = 1$, and it is unconditionally injective.

What is more, the removal of an operation from an algebra can turn it from injective into noninjective, as shown in the following example.

Example 2.26
We add operation $\downarrow$ to the algebra of queues of Example 1.5:

$$(\,[\mathrm{Int}] \mid [\,], (=[\,]),\ \dashv, hd, tl, \downarrow\,).$$

Since the algebra of queues is bijective, this algebra is also bijective, and we can use abstraction functions to describe its refinements. However, in some applications operation *hd* is not required, and then the following algebra will do:

$$(\,[\mathrm{Int}] \mid [\,], (=[\,]), \dashv, tl, \downarrow\,).$$

For this algebra a more efficient implementation is designed in Chapter 8, exploiting the fact that it is not injective (e.g., $[1,0]$ and $[2,0]$ are indistinguishable). Since different abstract values are represented by the same concrete value, the efficient implementation cannot be described by an abstraction function. □

We hope that we have convinced the reader that it is necessary to introduce the notion of data refinement under a coupling relation, although many interesting refinements can be modelled in terms of abstraction functions. For obviously-bijective algebras it is best to use abstractions in a specification of a data structure, but when this is not so clear—or, when one does not want to investigate this—it is better to use a coupling relation.

2.6 A closer look at surjectivity and injectivity

As argued in Example 2.26, noninjective algebras arise naturally as specifications of data structures. Efficient implementations of these data structures, however, exploit the fact that indistinguishable abstract values can be represented by the same concrete value. The following theorem shows that this is always possible—and also that unreachable values need not be represented (in case of nonsurjective algebras).

Theorem 2.27 Any monoalgebra has a bijective refinement. □

This theorem follows from the two lemmas below and the transitivity of algebra refinement (Property 2.13b).

Lemma 2.28 Any monoalgebra has a surjective refinement.

Proof Let A be a monoalgebra with data type A. We define a surjective refinement R of A as follows. The data type of R is $\mathrm{rng}\,\mathsf{A}$, and for each operation f of A, the corresponding operation f' of R is defined as (see Definition 2.12):

(a) $f' = f$, if f is a creation of type $T \to A$.

(b) $f' = f{\restriction}(\mathrm{rng}\,\mathsf{A} \times \mathrm{rng}\,\mathsf{A})$, if f is a transformation of type $A \times A \rightharpoonup A$.

(c) $f' = f{\restriction}(\mathrm{rng}\,\mathsf{A})$, if f is an inspection of type $A \to T$.

The coupling $\simeq$ between these algebras is the identity on $\mathrm{rng}\,\mathsf{A}$.

Now, observe that $\mathrm{rng}\,\mathsf{A}$ is a subset of A, hence $\simeq$ is a relation on $\mathrm{rng}\,\mathsf{A}$ and A. To prove that R refines A under $\simeq$, we show that f' refines f under $\simeq$ in the respective cases.

Case (a). We have to prove, cf. Definition 2.12a:

$$(\forall\, x : x \in T : f.x \simeq f'.x).$$

Since $f' = f$ and $\simeq$ is the identity on $\mathrm{rng}\,\mathsf{A}$, it suffices to prove that $f.x \in \mathrm{rng}\,\mathsf{A}$, for all $x \in T$. On account of the definition of rng this means that in any refinement of A, $f.x$ should be represented by a concrete value. This is indeed the case, since by the definition of data refinement $g.x$ represents $f.x$ for any refinement g of f.

Case (b). We have to prove, cf. Definition 2.12b:

$$\begin{array}{l}(\forall\, a,b,c,d : (a,b) \in \mathrm{dom}\, f \;\wedge\; a \simeq c \;\wedge\; b \simeq d : \\ \qquad\qquad (c,d) \in \mathrm{dom}\, f' \;\wedge\; f.a.b \simeq f'.c.d).\end{array}$$

Since $a \simeq c \;\equiv\; a \in \mathrm{rng}\,\mathsf{A} \;\wedge\; a = c$ and $\mathrm{dom}\, f' = \mathrm{dom}\, f \cap (\mathrm{rng}\,\mathsf{A} \times \mathrm{rng}\,\mathsf{A})$, the interesting part left to prove is that $f.a.b \in \mathrm{rng}\,\mathsf{A}$ for $(a,b) \in \mathrm{dom}\, f'$. That is, we have to show that $f.a.b$ is represented in any refinement of A. So, let g refine f. Since $a \in \mathrm{rng}\,\mathsf{A}$ there is a representation c, say, of a. Similarly, there exists a representation d of b. By the definition of data refinement $(c,d) \in \mathrm{dom}\, g$, hence application of g to these values yields $g.c.d$ as representation of $f.a.b$.

Case (c). We have to prove, cf. Definition 2.12c:

$$(\forall\, a,c : a \simeq c : f.a = f'.c).$$

Since $a \simeq c \;\equiv\; a \in \mathrm{rng}\,\mathsf{A} \;\wedge\; a = c$, this is equivalent to $(\forall\, a : a \in \mathrm{rng}\,\mathsf{A} : f.a = f'.a)$, which follows from the definition of f'.

To show that R is surjective, we prove $\mathrm{rng}\,\mathsf{A} \subseteq \mathrm{rng}\,\mathsf{R}$ as follows. By the transitivity of algebra refinement it follows that any refinement of R under coupling $\cong$, say, is a refinement of A under $\simeq \circ \cong$. The latter coupling equals $\cong$ because $\simeq$ is the identity on $\mathrm{rng}\,\mathsf{A}$. So, any refinement of R is also a refinement of A under the same coupling. Therefore, any value in the range of A is in the range of R.
□

Lemma 2.29 Any surjective monoalgebra has a bijective refinement.

Proof Let A be a surjective monoalgebra with data type A. Binary relation $\sim$ on A is defined by $a \sim b$ if there exists a refinement of A in which a and b are represented by the same value. This relation is symmetric, hence the reflexive-transitive closure of $\sim$, denoted as $\overset{*}{\sim}$, is an equivalence relation.

Using this equivalence relation, we define a bijective refinement E of A as follows. Its data type E consists of the equivalence classes of $\overset{*}{\sim}$. Using $(\!|\cdot|\!)$ to denote the induced mapping from A to E, we define for each operation f of A, the corresponding operation f' of E by:

(a) $f'.x = (\!|f.x|\!)$ for $x \in T$, if f is a creation of type $T \to A$.

(b) $f'.(\!|a|\!).(\!|b|\!) = (\!|f.a.b|\!)$ for $(a, b) \in \operatorname{dom} f$, if f is a transformation of type $A \times A \curvearrowright A$.

(c) $f'.(\!|a|\!) = f.a$ for $a \in A$, if f is an inspection of type $A \to T$.

By Property 2.16, f' refines f under $(\!|\cdot|\!)$, provided f' is a well-defined function. This proviso is indeed fulfilled, as we will now prove for each case.

Case (a). This case is trivial, since $f' = (\!|\cdot|\!) \circ f$.

Case (b). In this case, well-definedness means that $(\!|f.a.b|\!) = (\!|f.a'.b'|\!)$ if $(\!|a|\!) = (\!|a'|\!)$ and $(\!|b|\!) = (\!|b'|\!)$, for (a, b) and (a', b') in $\operatorname{dom} f$. Or, in terms of $\overset{*}{\sim}$: $a \overset{*}{\sim} a' \ \wedge \ b \overset{*}{\sim} b' \ \Rightarrow \ f.a.b \overset{*}{\sim} f.a'.b'$. To this end we first prove that $a \sim a' \ \Rightarrow \ f.a.b \sim f.a'.b$. So, assume $a \sim a'$. Then, by the definition of $\sim$, there exists a refinement in which a and a' are represented by the same value c, say. Since A is surjective, b is also represented, say by d. Let g be the refinement of f, then $(c, d) \in \operatorname{dom} g$ and $g.c.d$ represents both $f.a.b$ and $f.a'.b$, hence $f.a.b \sim f.a'.b$. Interchanging the roles of a and b gives a similar property. Together these properties suffice to complete the proof by an induction argument, which we omit.

Case (c). For an inspection f, we prove $a \overset{*}{\sim} a' \ \Rightarrow \ f.a = f.a'$. This follows by induction from $a \sim a' \ \Rightarrow \ f.a = f.a'$, which is proved as follows. Assumption $a \sim a'$ gives that there exists a refinement in which a and a' are represented by the same value c, say. Let g refine f. Then, by definition of data refinement (2.12c), $g.c$ is equal to both $f.a$ and $f.a'$, hence $f.a = f.a'$.

So much for the proof that E refines A under $(\!|\cdot|\!)$. We now show that E is bijective.

To see that E is surjective, we consider an element $(\!|a|\!)$ of E. Any refinement of E under coupling $\simeq$ also refines A under $(\!|\cdot|\!)^\star \circ \simeq$ (Property 2.13b). This implies, on account of the surjectivity of A, that there exists a representation c of a in such a refinement. But this means that $a \, (\!|\cdot|\!)^\star \circ \simeq c$. Or, $(\!|a|\!) \simeq c$, hence c represents $(\!|a|\!)$.

Finally, to see that E is injective, assume that $(\!|a|\!)$ and $(\!|b|\!)$ are represented by the same value c in some refinement C of E under a coupling $\simeq$. Then, by the transitivity of algebra refinement, $a \, (\!|\cdot|\!)^\star \circ \simeq c$ and $b \, (\!|\cdot|\!)^\star \circ \simeq c$. Hence, C is a refinement of A in which a and b are coupled to the same value c. Therefore $a \sim b$, hence $(\!|a|\!) = (\!|b|\!)$.
□

Remark 2.30
Since E refines A under representation $(\!|\cdot|\!)$, E identifies elements a and b of A whenever $a \overset{*}{\sim} b$. This implies that $\overset{*}{\sim}$ and $\sim$ are actually the same relations on A. We have used $\overset{*}{\sim}$ in the proof because a direct proof of the transitivity of $\sim$ is somewhat awkward.
□

Chapter 3

Amortized complexity

To amortize: to end (a debt) by making regular payments into a special fund.
[Oxford advanced learner's dictionary]

In a traditional worst-case analysis of an algorithm one derives a worst-case bound by adding the worst-case bounds of the constituent parts of the algorithm (which are often operations on data structures). However, bounds obtained in this fashion may be overly pessimistic, possibly orders of magnitude too high. Amortization is a simple principle to analyze the worst-case behaviour of algorithms in a compositional way: to avoid overly pessimistic bounds, so-called *amortized* cost measures are used instead of *actual* cost measures for the components of algorithms.

As is often the case with simple mathematical principles, there are many ways to formulate them and to explain them. Sometimes a principle is so simple that it is often applied without even knowing its name, let alone its existence (e.g., the Pigeonhole Principle). It is however important to expose such principles in a convenient way and to collect instructive applications of them. In imperative programming, for example, the Invariance Theorem is a coding of the Principle of Mathematical Induction particularly suited for the design of repetitions (see, e.g., [5]). In this chapter we will present several views of amortization based on the banker's and physicist's views of Sleator and Tarjan [31]. Of course, simple as such principles can be, their application will still be difficult when the problem is inherently complex. (It is, for instance, an open problem whether pairing heaps [7] are as efficient as Fibonacci heaps [8] in the amortized sense.)

In the next section, we explain the idea of amortization in an imperative setting. In the subsequent section, we present abstract versions of the banker's and physicist's views of amortization, and we show that they are equally powerful. The subject of Chapter 5 is a calculational approach to the (amortized) analysis of functional programs, in particular operations of algebras. The material in this chapter provides a basis for that approach.

3.1 Amortized analysis of imperative programs

As a simple imperative programming language we use Dijkstra's guarded command language (GCL) [5]. A suitable cost measure for GCL programs is the number of times that guards of repetitions are passed. Apart from constant factors, the time complexity of these programs is then determined.

To explain how amortization is applied in the analysis of GCL programs, we consider the following nondeterministic program (with $N \geq 0$):

$m, n := 0, 0$
{ invariant: $0 \leq m \ \wedge \ 0 \leq n \leq N$; bound: $m + 2(N-n)$ }
; **do** $n \neq N \rightarrow m, n := m+1, n+1$
  [] $m \neq 0 \rightarrow m := m-1$
od .

This program occurs as common projection of many programs that employ, say, a stack as auxiliary variable. Variable m then corresponds to the height of such a stack, and, among other steps, these programs repeatedly either *push* a value onto the stack ($m := m+1$), or *pop* a value from the stack ($m := m-1$). Initially and upon termination the stack is empty. Instead of stacks, other data structures that are often used in this fashion are first-in first-out queues (with put and get operations), priority queues (with insertion of an arbitrary value and removal of the minimum value), etc.

Remark 3.1
Usually, those programs consist of a nested repetition, followed by a repetition that empties the data structure. For example:

$m, n := 0, 0$
; **do** $n \neq N \ \rightarrow$
    do $\ldots \ \rightarrow \{m \neq 0\}\ m := m-1$ **od**
  ; $m, n := m+1, n+1$
od
; **do** $m \neq 0 \ \rightarrow m := m-1$ **od**.

The number of steps of the inner repetition per step of the outer repetition depends on more variables than m and n alone. This irregular behaviour is nicely captured by the nondeterminism in the former program, which we prefer for its simplicity.
□

There are many ways to prove that the repetition of the above program is unfolded exactly $2N$ times, or even that both alternatives are unfolded N times each. An informal way to do this is as follows. It is obvious that there are exactly N pushes. Moreover, there corresponds one pop to each push, since the stack is empty initially and upon termination. Therefore, the number of pops is

also exactly N. A more formal way to prove that the repetition is unfolded $2N$ times is by means of a bound function (see program annotation). The bound function $m + 2(N-n)$ is decremented in each step. Since, its initial value is $2N$ and its final value is 0, there are $2N$ steps (hence, the program terminates).

We consider both of the above ways inadequate for the purpose of *analyzing* GCL programs. The first approach because it is informal and ad hoc, the second because it also entails a termination proof which we consider part of the correctness proof of a program. In general, we analyze only correct (hence terminating) programs.

Given that the program terminates, a formal, but simpler way to analyze it is as follows. We introduce a fresh variable t to count the number of unfoldings of the repetition and establish an invariant of the form $t = E$:

$$
\begin{array}{l}
m, n := 0, 0;\ t := 0 \\
\{\ \text{invariant: } t = 2n - m\ \} \\
;\ \textbf{do}\ n \neq N \rightarrow m, n := m{+}1, n{+}1;\ t := t{+}1 \\
\quad [\!]\ \ m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1 \\
\textbf{od}\ .
\end{array}
$$

From the annotation in this program we conclude that $t = 2N$ holds as post-condition. Note that we count one unit of cost for each stack-operation, except for the initialization of the stack (initialization $t := 1$ gives a more realistic measure).

The problem with the last approach is that we have to invent an exact relation for t, viz. $t = 2n - m$. Although this relation is not that complicated, it is in many cases difficult to find an exact relation. Often we must content ourselves with an upper bound for t. As we will explain below, the principle of amortization helps to achieve a better separation of concerns in such analyses. A key point in the following amortized analysis is that we exploit the fact that it is clear that the first alternative of the repetition is executed exactly N times (and that it is not so clear that the second alternative is also executed exactly N times).

In addition to variable t, the *(accumulated) actual* costs, we introduce variable a, the *(accumulated) amortized* costs. We see to it that upon termination $a = t$ by taking 2 as amortized costs for a push and 0 as amortized costs for a pop. This *choice* for the amortized costs is guided by the observation that the accumulated number of pushes is simply equal to n, while an expression for the accumulated number of pops is more complicated (viz. $n - m$). In this way, we obtain a simple relation for the amortized costs, viz. $a = 2 * n + 0 * (n{-}m)$:

$$
\begin{array}{l}
m, n := 0, 0;\ t, a := 0, 0 \\
\{\ \text{invariant: } a = t + m\ \wedge\ a = 2n\ \} \\
;\ \textbf{do}\ n \neq N \rightarrow m, n := m{+}1, n{+}1;\ t, a := t{+}1, a{+}2 \\
\quad [\!]\ \ m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1 \\
\textbf{od}\ .
\end{array}
$$

The analysis can now be divided into two parts, each part corresponding to a conjunct of the above invariant. The first part shows that upon termination $a = t$, that is, the amortized costs are equal to the actual costs in the end. The second part gives a simple relation for the amortized costs. Proving each of the parts is simpler than proving the invariance of $t = 2n-m$ in one go, but taken together the amortized analysis is of course more laborious. However, in more complicated applications of amortization this degree of disentanglement is really fruitful.

Instead of defining the amortized costs of each step of a repetition directly (which corresponds to the *banker's view* of amortization), there is also an indirect way (corresponding to the *physicist's view*). Then the amortized costs are defined in terms of a *potential function*, a function defined on the state space of the program (cf. a bound function in the Invariance Theorem). The potential of a state can be thought of as the accumulated difference between the amortized costs and the actual costs. The potential function, often called Φ, corresponding to the choice of amortized costs in the above analysis is m:

$m, n := 0, 0;\ t := 0$
{ invariant: $t + \Phi = 2n$; potential Φ: m }
; **do** $n \neq N \rightarrow m, n := m{+}1, n{+}1;\ t := t{+}1$
  [] $m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1$
od .

In general, $t + \Phi$ equals the accumulated amortized costs, and the change in this quantity per step equals the amortized cost of a step. Notice that the potential m is nonnegative, which means that there is always a surplus: the accumulated actual costs do not exceed the accumulated amortized costs.

Potential m ("size of stack, queue, etc.") is a natural potential function. However, since the initial value of m equals its final value, any real-valued function of m will do, for example $\Phi = -m$ or $\Phi = 0$. In the first case, the amortized costs of a push are zero and a pop costs two units, so we count the pops. This approach is not so good when the number of pops is possibly less than the number of pushes; then some pushes are never counted. The second case gives an analysis in which the amortized costs equal the actual costs, so this does not help. Some potentials even lead to *negative* amortized costs (like $\Phi = 2m$, which gives -1 as amortized cost for a pop, and 3 for a push), or to *nonconstant* amortized costs (like $\Phi = m^2$).

In general, we choose an invariant of the form $t + \Phi = E$, and we choose potential Φ such that a simple expression E for the amortized costs results. Often it is convenient to have a nonnegative potential, since this guarantees that $t \leq E$ at any point in the program. Sometimes we must content ourselves with an upper bound for the amortized costs. The form of the invariant is then $t + \Phi \leq E$, as in the following program. This program has been obtained from the previous program by substituting N for $m{+}1$.

$m, n := 0, 0;\ t := 0$
{ invariant: $0 \leq m\ \wedge\ t + \Phi \leq n(N{+}1)$; potential Φ: m }
; **do** $n \neq N \rightarrow m, n := N, n{+}1;\ t := t{+}1$
  [] $m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1$
od .

It follows that $t \leq N^2 + N$ upon termination (which is a tight bound).

We conclude this section with a more complicated application of the "potential technique".

Example 3.2
Kaldewaij derives in [20, Section 12.3] the following program for the computation of the length of a shortest segment of array X in which the maximum value on that segment occurs exactly twice:

$n, r := 1, \infty$
; **do** $n \neq N\ \rightarrow$
    $s := n{-}1$
  ; **do** $s \neq 0\ \wedge\ X[s] < X[n]\ \rightarrow s := f[s]$ **od**
  ; $f[n] := s$
  ; **if** $X[s] = X[n]\ \rightarrow r := r \min (n{+}1{-}s)$
    [] $X[s] \neq X[n]\ \rightarrow$ skip
    fi
  ; $n := n{+}1$
od .

Here, integer array X of length N, $N{\geq}1$, is the input array and the output is stored in integer variable r.

The problem with the analysis of this program is that the number of steps of the inner repetition per step of the outer repetition is difficult to determine. Instead, we therefore perform an amortized analysis in which we determine the *total* number of steps of the inner repetition.

To this end we concentrate on the variables n, f, and s. For the analysis, the relevant invariant for the outer repetition reads:

$$\text{P}: \quad 0 < n\ \wedge\ (\forall\, i : 0 < i < n : 0 \leq f[i] < i),$$

hence statement $s := f[s]$ decreases the value of s and proper termination of the inner repetition is guaranteed. To avoid counting the number of steps of this repetition per step of the outer repetition, we choose the potential such that the amortized costs per step of the inner repetition are *zero*. That is, since a step of the inner repetition consists of $s := f[s]$, we take as potential $\Psi = \#s$, with

$$\begin{array}{lcl} \#0 & = & 0 \\ \#i & = & 1 + \#(f[i]) \end{array}$$

for $0 < i < n$. The invariance of $0 \leq s < n \ \wedge$ P guarantees that this potential is well-defined.

The amortized analysis is now completed by showing that the following annotation is correct, using that the invariance of P (and $0 \leq s < n$ for the inner repetition) has already been shown in [20]. We use one potential for each repetition:

$$
\begin{array}{l}
\quad n := 1;\ t := 0 \\
\{\ \text{invariant: P } \wedge\ t + \Phi = n{-}1\text{; potential } \Phi\text{: } \#(n{-}1)\ \} \\
;\ \textbf{do}\ n \neq N\ \rightarrow \\
\qquad s := n{-}1 \\
\qquad \{\ \text{invariant: } 0 \leq s < n\ \ \wedge\ \text{P}\ \ \wedge\ t + \Psi = n{-}1\text{; potential } \Psi\text{: } \#s\ \} \\
\qquad ;\ \textbf{do}\ s \neq 0\ \wedge\ \ldots\ \rightarrow s := f[s];\ t := t{+}1\ \textbf{od} \\
\qquad ;\ f[n] := s \\
\qquad ;\ n := n{+}1 \\
\quad \textbf{od}\ .
\end{array}
$$

The definition of potential Φ has been obtained by applying the substitution rule for $s := n{-}1$ to the definition of Ψ. The invariance of $t + \Psi = n{-}1$ for the inner repetition is trivial, and it is matter of straightforward verification to show the invariance of $t + \Phi = n{-}1$:

$$
\begin{array}{ll}
 & ((t + \#(n{-}1) = n - 1)^{n}_{n+1})^{f}_{f[n:=s]} \\
\equiv & \{\ \text{see definition of } \#'\ \text{below}\ \} \\
 & t + \#'n = n \\
\equiv & \{\ \#'n = 1 + \#'s = 1 + \#s\text{, since } 0 \leq s < n\ \} \\
 & t + \#s = n - 1,
\end{array}
$$

where $\#'$ is given by

$$
\begin{array}{lcl}
\#'0 & = & 0 \\
\#'i & = & 1 + \#'(f[n{:=}s][i])
\end{array}
$$

for $0 < i < n{+}1$. The last line of this derivation may be written as $t + \Psi = n{-}1$, which is a postcondition of the inner repetition.

Since Φ is nonnegative, we have that $t \leq N{-}1$ upon termination. Hence the program is linear in N, since the outer repetition is unfolded exactly $N{-}1$ times.

□

3.2 Abstract views of amortization

In [31, Section 2], Tarjan describes two views of amortization, called the banker's view and the physicist's view, and he states that these views are "entirely equivalent". There is however no precise framework to support the meaning of this

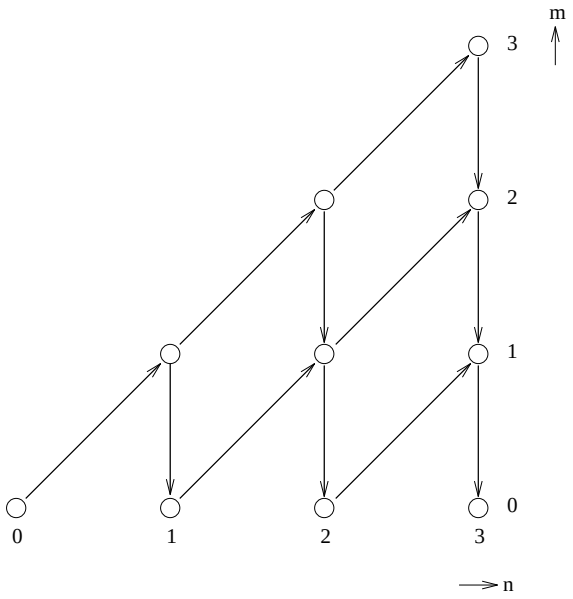

Figure 3.1: An abstract algorithm with $s = (0, 0)$.

statement, let alone to justify it. Based on Tarjan's description, Mehlhorn and Tsakalidis also mention two views of amortization in [24, p.304] but here the equivalence is trivial: they speak of "a function *bal* which maps the possible configurations of the data structure into the real numbers" when describing the banker's view, but this is precisely the idea of a potential function.

What is not pointed out in [24] and what is not clear in Tarjan's explanation is that the essential point about potential functions is that they depend on the *current state* (of the data structure) only, whereas a more general notion of amortization (based on the banker's view) may take the entire *history* (of the data structure) into account. By recording the history of the data structure, it is of course possible to simulate a banker's analysis by a physicist's analysis, but this we consider pointless. In the sequel we will show how a banker's analysis may be simulated by a physicist's analysis *without* extending the state of the data structure.

Before we present an abstract view of amortization, we introduce an abstract view of algorithms.

Definition 3.3
An *abstract algorithm* is a quadruple (V, E, s, t), in which V is a set of *states*, $E \subseteq V \times V$ (*steps*), $s \in V$ is the *initial state*, and $t \in E \to \text{Real}$ (*(actual) costs per step*). Furthermore, all states should be *reachable* from s, where the graph terminology applies to the directed graph (V, E).
□

Note that it is not required that V (or E) is finite. Neither it is required that (V, E) is acyclic, or that vertices in (V, E) should have a finite number

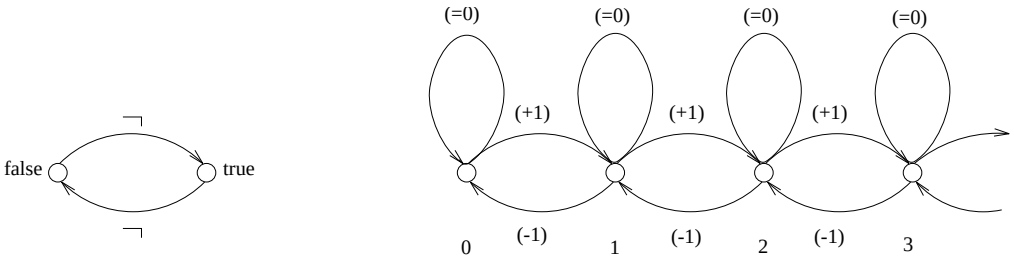

Figure 3.2: Abstract algorithms corresponding to algebras $(\,\mathrm{Bool} \mid \mathrm{false}, \neg\,)$ and $(\,\mathrm{Nat} \mid 0, (=0), (+1), (-1)\,)$.

of predecessors and successors. An example of an abstract algorithm is given in Figure 3.1; it corresponds to all possible executions of the program in the previous section for N=3. The cost function t is the all-one function in this case.

One may ask why we do not exclude cyclic graphs in the above definition, since at first sight this corresponds to non-terminating programs. However, the state of an abstract algorithm does not necessarily correspond to the *entire* state of a program; it may also correspond to a *projection* of the state. For instance, projecting the diagram in Figure 3.1 on variable m gives a cyclic abstract algorithm. What is more, as depicted in Figure 3.2, algebras typically correspond to cyclic abstract algorithms in which the initial state is reachable from all states.

The fact that an abstract algorithm has only one initial state is not really a limitation. For example, algebras with several constants can be viewed as abstract algorithms with initial state $\bot$ and the respective constants as successors of $\bot$. Furthermore, if there are several transitions between two states with different costs, this can be modelled in an abstract algorithm as a single step whose cost is the largest cost of these transitions. (If this is not satisfactory one must distinguish these transitions by distinguishing more states.) Another issue is that algebras often provide binary operators, like addition of natural numbers. In such circumstances one cannot just take a single natural value as state (cf. Figure 3.2) but, for instance, a *bag* of naturals is required. This phenomenon is particularly relevant for implementations of *mergeable* priority queues (Chapter 9) and for implementations of similar algebras that are efficient in the amortized sense only.

Now that we have defined an abstract notion of algorithms, we are ready to introduce an abstract notion of amortization. As shown in the previous section, an amortized analysis of an imperative program consists of a choice of the amortized costs for each step of the program such that for all paths from the initial state to the final state(s), the actual costs are bounded from above by the amortized costs of these paths. Since an abstract algorithm does not necessarily have final states (states without successors), we have to be a little bit more careful:

Definition 3.4
An *abstract amortized analysis* is a quintuple (V, E, s, t, a), in which (V, E, s, t) is an abstract algorithm and $a \in E \to \text{Real}$ (*amortized costs per step*). Furthermore, for each vertex x there should exist a vertex y reachable from x, for which the actual cost of every *finite* path[1] from the initial state s to y is at most its amortized cost.
□

We call $a(x, y) - t(x, y)$ the *surplus* of edge (x, y). The surplus of a path is the sum of the surpluses of the composing edges. A *pluspoint* is a vertex for which the surpluses of all paths from s ending in it are nonnegative. By definition, each vertex has a pluspoint within its reach. Note that a cycle cannot have a negative surplus, for this would lead to unbounded negative surpluses for paths from s to vertices on the cycle; this contradicts with the requirement that every vertex has a pluspoint within its reach. Also note that the initial state is a pluspoint, because the empty path has surplus zero and all cycles ending in the initial state have nonnegative surpluses.

We now distinguish a special kind of amortization, in which the amortized costs are defined in terms of a potential function. This corresponds to the physicist's method of Sleator and Tarjan [31].

Definition 3.5
An abstract amortized analysis (V, E, s, t, a) is called *conservative*, when there exists a function $\Phi \in V \to \text{Real}$ such that $a(x, y) = t(x, y) + \Phi.y - \Phi.x$. Such a function is called a *potential (function)*.
□

Note that the surplus of edge (x, y) equals potential difference $\Phi.y - \Phi.x$. Furthermore, any cycle has surplus zero in a conservative analysis. The pluspoints of a conservative analysis are those states x for which $\Phi.x \geq \Phi.s$.

The physicist's method seems less powerful but we will show that it is as powerful as the banker's method ("as powerful as" in the sense of the theorem below). The problem with the physicist's method is that the amortized costs of a step have to be defined in terms of a function on the *vertices*, whereas in the banker's method amortized costs can be allocated separately to each *edge*.

Theorem 3.6
Let $A = (V, E, s, t, a)$ be an abstract amortized analysis. Then there exists a potential function Φ, for which the conservative analysis $A' = (V, E, s, t, a')$ of the same abstract algorithm, with $a'(x, y) = t(x, y) + \Phi.y - \Phi.x$, satisfies

(a) A' has the same set of pluspoints as A, and

(b) for each step, the amortized costs in A' are at most the amortized costs in A.

[1] Throughout this section, paths are finite.

Proof The definition of Φ is as follows: for each $x \in V$, $\Phi.x$ is defined as the *infimum* of the surpluses of all paths from s to x. Before we show that (a) and (b) hold, we must first convince ourselves that Φ is indeed a function from V to Real, that is $\Phi.x \neq -\infty$ for all $x \in V$. For this we use that A is an amortized analysis, which means according to Definition 3.4 that for any state x, there exists a pluspoint y reachable from x by a path with surplus δ, say. Since y is a pluspoint, all paths from s to y have a nonnegative surplus. Hence all paths from s to x have a surplus that is bounded from below by $-\delta$. So, $\Phi.x \geq -\delta$.

Now, it is easy to show (a). First we observe that in A' the surpluses of the paths from s to a state x are all equal to $\Phi.x$. So the pluspoints in A' are those states for which the potential is nonnegative. But this means, by the definition of Φ, that the surpluses of all paths from s to x in A are nonnegative, hence that x is a pluspoint in A. (Note, in particular, that s is a pluspoint in both A and A'.)

Finally, (b) means that $a'(x, y) \leq a(x, y)$ must hold for each edge (x, y). By the definition of a' above, this may be written as $\Phi.y \leq \Phi.x + a(x, y) - t(x, y)$. This is clearly true because $\Phi.y$ is at most the surplus of any path from s to x followed by (x, y).
□

The intuition behind the definition of Φ is that the *least* surplus determines the potential of a state. It is easy to see that Φ is nonnegative whenever all paths starting in s have nonnegative surpluses. Of course, when all surpluses are bounded from below by a negative constant C, say, then we may also obtain a nonnegative potential by increasing Φ by $-C$. It is however important to note that the infimum of the potentials of all states may be $-\infty$, since the surplus over *all* paths starting in s is not bounded from below. Because of this fact it is not always possible to use a nonnegative potential in an amortized analysis (see Chapter 7).

It should be noted that the proof of the above theorem does not give a nice description of the potential. To obtain the potential of a state we have to determine an infimum over all paths leading to that state. Fortunately, paths with cycles can be ignored, for adding cycles to a path cannot decrease the surplus. For the example program in the previous section, potential m corresponds to the potential defined in the theorem. In practice, the banker's method can be used when defining a potential becomes too complicated.

Chapter 4

Implementation aspects

In this chapter, we introduce the program notation that we will use to describe implementations of algebras. According to Definition 2.1, a program notation defines the borderline between abstract and concrete programs. Usually, the borderline is chosen such that concrete programs can be translated relatively easily into machine code by a compiler program. The compilation of the programs presented in this thesis is, however, of no concern to us. For our purposes it is important that the program notation is a well-chosen compromise between a high-level mathematical notation and a low-level machine language: a mathematically-oriented notation facilitates the correctness proof of programs, whereas a machine-oriented notation facilitates the definition of realistic cost measures for programs. Relying on some common knowledge of functional and imperative languages (and their implementations), we will confine ourselves to an informal description of the notation.

In favour of the mathematical part, we shall start with a *purely-functional* program notation. This goes well with the fact that we are primarily interested in monoalgebras—whose operations are functional, after all. We will use this notation in a restricted way only, so that *eager evaluation* suffices as simple and efficient evaluation method for our programs. For instance, the notation is used in a *first-order* fashion only, which means that there will be a clear distinction between the data on the one hand and the programs operating on the data on the other hand. In this way, realistic cost measures are easily defined for our programs.

In addition to the usual algebras provided by functional languages, we will also use algebras involving arrays and pointers—as provided by most imperative languages. For reasons of efficiency, operations of these algebras are implemented *destructively*. As a consequence, the usage of these algebras must be restricted to what will be called *linear usage* (cf. [36]) so as to guarantee that they behave as specified. To facilitate the use of arrays and pointers, the program notation will be extended with some "imperative features". This is illustrated by a number of pointer implementations for list algebras like stacks and queues.

4.1 Functional program notation

Our functional program notation is based on the notation of [17], which in turn finds its roots in SASL [33]. We confine ourselves to an informal introduction of the notation, which will suffice to understand the programs in the sequel. Program *Mergesort* on page 60 shows (almost) all of the features of the notation described below.

In our notation, a functional program is a function definition of the form $f.x = F$, in which f is the function's name, x is the name of the parameter and F is the defining expression. Here, expression F may refer to x, but also to f; in the latter case, the definition is recursive. The function's domain is defined informally or left implicit.

The expression in the right-hand side of a function definition is built from the operations of a number of predefined algebras. As simple algebras, we use the data types Bool, Nat, and Int, along with the usual operations. Furthermore, the algebra of stacks is assumed to be available which provides a "kernel" for the list operations (cf. Example 1.5). A similar algebra involving binary trees is also assumed to be concrete (cf. Section 1.1.4):

$$(\,\langle T\rangle \;\mid\; \langle\,\rangle, (=\langle\,\rangle), \langle\cdot,\cdot,\cdot\rangle, l, m, r\,),$$

where operation $\langle\cdot,\cdot,\cdot\rangle \in \langle T\rangle \times T \times \langle T\rangle \to \langle T\rangle$ is used to construct nonempty trees; it satisfies $\langle l.x, m.x, r.x\rangle = x$ for nonempty x. (The standard pointer implementations of these algebras are presented in Section 4.3.)

To support case analysis, we use *conditional expressions*. The *alternatives* of a conditional expression are separated by $[]$'s. For example, a frequently occurring form is $(E, B \,[]\, F, C)$, in which B and C are boolean expressions, called the *guards*. Usually, B and C exclude each other because they are complementary, but we will sometimes write programs that violate this rule in order to retain symmetry. For example, we prefer to leave a program like

$$\begin{array}{lcll} f.\langle x, a, y\rangle & = & x & , a \le 0 \\ & [] & y & , a \ge 0 \end{array}$$

nondeterministic; strengthening one of its guards with $a \neq 0$ is "left to the reader".

To support modular design, expressions may contain so-called *where-clauses*. A where-clause consists of one or more function definitions enclosed by the pair $[\![$ and $]\!]$. Where-clauses are often used to abbreviate subexpressions that occur more than once; for instance, we write $x * x[\![x = a * b - a/b]\!]$ instead of $(a * b - a/b) * (a * b - a/b)$.

In the left-hand side of a function definition it is allowed to use *patterns* instead of names as parameters. Patterns are restricted forms of expressions, which are useful in shortening a function definition. For instance, using the patterns $2{*}a$ and $2{*}a{+}1$, integer division by 2 may be defined by $d.(2{*}a) = a$

and $d.(2{*}a{+}1) = a$. Patterns are particularly useful in definitions of operations on structures like lists and trees (see also Chapter 6). Instead of using the pattern (x, y) when the domain of a function is a cartesian product of the form $X \times Y$, we usually write $f.x.y$ as abbreviation for $f.(x, y)$, and so on.

Merely for the sake of brevity, we will also use $\circ$ (*function composition*) in our programs. For example, a definition like $h.x = g.(f.x)$ is abbreviated to $h = g \circ f$. Because of this restricted use, it is no problem that $\circ$ is a higher-order function.

The remaining features of our program notation are supposed to be well-understood. In Section 4.3 the program notation will be extended with some imperative features. Programs written in the notation introduced in the present section will be called *purely-functional* programs.

4.2 Eager evaluation

In this section we consider a number of aspects related to the execution of functional programs. A basic understanding of these aspects is assumed in the performance analyses in later chapters. We emphasize that many other aspects, such as "lazy evaluation", "infinite lists", and "input/output processing", are ignored, simply because they do not play a role in the programs considered in this thesis.

Central to the execution of functional programs is the notion of *reduction* (or *evaluation*) of expressions. In fact, the execution of a program f, given an input value x from the domain of f, boils down to the reduction of expression $f.x$. More generally, execution of the programs considered in this thesis gives rise to the reduction of expressions of the form $f.E$. For these programs the following simple reduction method suffices to reduce $f.E$: first E is reduced, and subsequently f is applied to the *value* of E. This reduction method is known as *eager evaluation* (see, e.g., [27]), and it is applied until the expression is free of function applications. The result of the reduction is then the *value* of the expression—where we assume that the reduction indeed terminates. Here is a simple example of a reduction (cf. Example 1.4):

$$
\begin{aligned}
& \mathsf{suc}.(\mathsf{suc}.\mathsf{zero}) \\
= \quad & \{ \text{ unfold definition of } \mathsf{zero} \ \} \\
& \mathsf{suc}.(\mathsf{suc}.[\,]) \\
= \quad & \{ \text{ unfold definition of } \mathsf{suc} \ \} \\
& \mathsf{suc}.[1] \\
= \quad & \{ \text{ unfold definition of } \mathsf{suc}, \text{ using } [1]{=}[\,] \dashv 1 \ \} \\
& \mathsf{suc}.[\,] \dashv 0 \\
= \quad & \{ \text{ unfold definition of } \mathsf{suc}, \text{ using } [1] \dashv 0{=}[1, 0] \ \} \\
& [1, 0],
\end{aligned}
$$

in which we have focused on the *unfoldings* of (the definitions of) zero and suc. The reduction of suc.(suc.zero) takes place in the context which defines zero and suc. This is common practice, and in the sequel an expression is always understood to have a context that defines its constituents.

In the next chapter, we discuss the analysis of the time complexity of functional programs. Since we will concentrate on cost measures that count unfoldings of user-defined functions, we have to motivate that those cost measures are indeed realistic. Therefore, we discuss the execution of our functional programs in more detail.

For instance, an important fact is that all operations of the predefined algebras mentioned in Section 4.1 have $O(1)$ time complexity. Furthermore, we rely on the fact that one unfolding of a user-defined function definition $f.x = F$ takes $O(1)$ time, since it merely amounts to the substitution of the value for x in expression F; the result then has to be reduced further, which may give rise to more unfoldings. For conditional expressions, we assume that only the relevant parts of the expression are evaluated. The evaluation of $(E, B \,[]\, F, \neg B)$, for instance, comprises one evaluation of B, followed by the evaluation of either E or F, depending on the value of B.

In the presence of where-clauses of the form $|[x = E]|$, it is important that multiple evaluation of E is avoided. The value of E should be *shared* by all occurrences of x in the expression to which the where-clause is attached. So, although $x = E$ may be written in the general form $x.\bot = E$ of a function definition, evaluation of these definitions proceeds differently for reasons of efficiency. For example, evaluation of $x + x|[x = E]|$ consists of a single evaluation of E, followed by an unfolding of $+$. However, a where-clause of the form $|[f.x = E]|$ attached to an expression F gives rise to as many unfoldings of f as there are applications of f in F, since these applications will in general be quite different.

So much for the time complexity of our programs. As for space complexity, we assume the existence of an ideal *garbage collector*, which recycles memory cells as soon as they become "garbage", thereby minimizing the maximal number of cells ever required during the execution of a program. Space complexity is particularly relevant for algebras with structured data types. In the next section we present a pointer implementation for stacks, in which each application of $\vdash$ occupies an extra cell, and an application of tl possibly releases a cell, namely when the application removes the last reference to the cell. For instance, evaluation of $tl.(a \vdash s)$ will first occupy an extra cell and then the application of tl will release this cell again because it removes the only reference to this cell.

4.3 Pointer implementation of stacks

In Example 1.8 we presented an algebra for pointers. In that algebra, the memory is explicitly represented by an element of type $\Omega \curvearrowright T$. In pointer notations of imperative languages, however, the memory is usually not explicitly mentioned.

In the example below, we will introduce a Pascal-like notation for pointers in which the memory is also left implicit.

The example concerns the standard representation for finite lists used by implementations of functional languages. For fixed type T, we consider the algebra of stacks:

$$\mathsf{S} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl\,),$$

for which we give a refinement at pointer level with signature:

$$(\,L \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{cons}, \mathsf{hd}, \mathsf{tl}\,).$$

To allow for an efficient implementation of this algebra, a list of type $[T]$ is represented by a so-called *singly-linked list*. That is, type L is defined by

$$L = {}^{\wedge}\langle a{:}T, r{:}L\rangle.$$

This definition of L expresses that elements of type $L\backslash\{\text{nil}\}$ are pointers to pairs of type $T{\times}L$. Moreover, it implies that $p^{\wedge} = \langle p^{\wedge}.a, p^{\wedge}.r\rangle$, for $p \in L\backslash\{\text{nil}\}$. The abstraction function is given by

$$\begin{array}{lcl} [\![\text{nil}]\!] & = & [\,] \\ [\![p]\!] & = & p^{\wedge}.a \vdash [\![p^{\wedge}.r]\!] \quad , p \neq \text{nil}, \end{array}$$

and to guarantee that the range of $[\![\cdot]\!]$ consists of *finite* lists only, the domain of $[\![\cdot]\!]$ is defined by

$$\text{dom}\,[\![\cdot]\!] = \{p \mid p \in L \ \wedge\ (\exists\, i : 0 \leq i : p({}^{\wedge}.r)^i = \text{nil})\}.$$

Remark 4.1
The above is an example of the use of partial abstraction functions. In this case we have defined the domain of $[\![\cdot]\!]$ explicitly, but often the domain is clear from the context: since $[\![\cdot]\!]$ is a partial function with range $[T]$, it follows that, for instance, a pointer p satisfying $p = p^{\wedge}.r$ cannot be in the domain of $[\![\cdot]\!]$; the domain of $[\![\cdot]\!]$ is the *largest* subset of L for which the above definition of $[\![\cdot]\!]$ defines values in $[T]$.
□

To facilitate the manipulations of pointers in functional programs, we extend the program notation as follows. Instead of an expression, the right-hand side of a function definition is allowed to be an imperative *statement*. This statement defines the function value by means of a call to procedure *return*; each execution of such a statement should give rise to exactly one call to return. The notation used for statements is supposed to be self-evident; procedures *new* and *dispose* are used in a Pascal-like manner in these statements.

Using this notation, the stack operations may be programmed as follows:

$$\begin{array}{lcl}
\mathsf{empty} & = & \text{nil} \\
\mathsf{isempty}.p & = & p = \text{nil} \\
\mathsf{cons}.a.p & = & |[\text{var } h{:}L;\ \text{new}(h);\ h^{\wedge} := \langle a, p\rangle;\ \text{return}(h)\ ]| \\
\mathsf{hd}.p & = & p^{\wedge}.a \\
\mathsf{tl}.p & = & p^{\wedge}.r.
\end{array}$$

Note that the program for cons is not functional: the value of $\mathsf{cons}.a.p$ is not determined by the values of a and p, because operation new is not a function. However, the value of $(\mathsf{cons}.a.p)^{\wedge}$ is a function of a and p, and this is the value that matters: $[\![p]\!]$ depends on the value of $p^{\wedge}$ only, for $p \neq \text{nil}$.

In Section 4.1 we have assumed that the following algebra involving binary trees is also part of the functional program notation:

$$\mathsf{T} = (\ \langle T\rangle\ \mid\ \langle\ \rangle, (=\langle\ \rangle), \langle\cdot,\cdot,\cdot\rangle, l, m, r\ ).$$

This algebra is in essence the same as algebra S, and therefore we confine ourselves to a brief description of its implementation. The appropriate pointer type is

$$B = {}^{\wedge}\langle l{:}B, a{:}T, r{:}B\rangle,$$

for which we have as corresponding abstraction:

$$\begin{array}{lcll}
[\![\text{nil}]\!] & = & \langle\ \rangle & \\
[\![p]\!] & = & \langle [\![p^{\wedge}.l]\!], p^{\wedge}.a, [\![p^{\wedge}.r]\!]\rangle & , p \neq \text{nil}.
\end{array}$$

With this representation each of the operations of T can be supported in $O(1)$ time.

4.4 Destructivity

Mainly for two reasons, purely-functional programming languages do not provide arrays and pointers. The first and minor reason is that the underlying algebras for arrays and pointers involve nonfunctional operations, viz. operation ? for arrays (cf. Example 1.7) and operation *new* for pointers (cf. Example 1.8).

The second and major reason is, however, that *efficient* implementations of array and pointer operations are *destructive*. For instance, evaluation of $a[i{:=}x]$ in general destroys the representation of a because the value of $a[i]$ is simply overwritten so as to achieve $O(1)$ time complexity. To guarantee that these operations behave in accordance with their specifications, the use of these operations must be restricted. For instance, an expression like (a[0:=11],a[0:=13]) should be avoided because evaluation of this expression results either in (a[0:=11],a[0:=11]) or in (a[0:=13],a[0:=13]), depending on which component of the pair is evaluated first.

Destructivity is related to the fact that parameters of structured types such as arrays and lists are passed *by reference* rather than *by value*. A pass-by-reference mechanism has to be used for these types to achieve the desired time complexity. For instance, to achieve $O(1)$ time complexity for an operation that operates on structures of size N, a pass-by-value mechanism cannot be used, for this already requires $O(N)$ time to copy the parameter.

Yet, destructivity is not a necessary consequence of the use of a pass-by-reference mechanism. For instance, in the implementation of stacks in the previous section none of the operations is destructive, although all list parameters are passed by reference ("reference p is passed instead of value $p^{\wedge}$"). Phrased differently, all parameters can be thought of as being passed by value, although they are actually passed by reference. Therefore, this implementation of stacks can be used in implementations of purely-functional programming languages. Only after addition of dispose(p) to the program for $\mathsf{tl}.p$ is a destructive implementation obtained:

$$\mathsf{tl}.p \quad = \quad \text{return}(p^{\wedge}.r);\ \text{dispose}(p).$$

This particular implementation of stacks has the advantage that cells are recycled explicitly so that no garbage collection is required. The price to be paid is that, for instance, $(q, q)\ ||[q = \mathsf{tl}.p]||$ and $(\mathsf{tl}.p, \mathsf{tl}.p)$ are not equivalent anymore: evaluation of the former expression yields two references to $\mathsf{tl}.p$, whereas evaluation of the latter one fails because evaluation of either of the two occurrences of $\mathsf{tl}.p$ disposes of the cell to which p points, and this blocks the evaluation of the remaining occurrence. To prevent such problems, this implementation may be used in a *linear* fashion only, as will be explained in Section 4.6.

4.5 Queues and concatenable deques

For fixed type T, we now present destructive implementations for two more algebras operating on lists of type $[T]$. In these implementations we shall use well-known techniques such as circularly-linked lists and doubly-linked lists. The operations of these implementations all have $O(1)$ time complexity.

First, we consider the algebra of queues:

$$\mathsf{Q} = (\ [T]\ \mid\ [\,], (=[\,]), hd, tl, \dashv\ ),$$

for which we give two destructive refinements at pointer level. We use the following signature:

$$(\ Q\ \mid\ \mathsf{empty}, \mathsf{isempty}, \mathsf{hd}, \mathsf{tl}, \mathsf{snoc}\ ).$$

In our first refinement, two pointers are used, one of them pointing to the head of a singly-linked list and the other one to the last cell in that list (if present). More precisely, we take $Q = L \times L$, with $L = {}^{\wedge}\langle a{:}T, r{:}L\rangle$ (as in Section 4.3), and we take as coupling

$$s \simeq \langle p, q\rangle \;\equiv\; s = list.p \;\wedge\; (s \neq [\,] \Rightarrow q = p(\hat{}.r)^{\#s-1}),$$

where

$$\begin{array}{lcl} list.\mathrm{nil} & = & [\,] \\ list.p & = & p\hat{}.a \vdash list.(p\hat{}.r) \quad , p \neq \mathrm{nil}. \end{array}$$

Note that $s = list.p$ implies $p(\hat{}.r)^{\#s} = \mathrm{nil}$. This coupling leads to the following programs:

$$\begin{array}{lcl} \mathsf{empty} & = & \langle \mathrm{nil}, \mathrm{nil}\rangle \\ \mathsf{isempty}.\langle p, q\rangle & = & p = \mathrm{nil} \\ \mathsf{hd}.\langle p, q\rangle & = & p\hat{}.a \\ \mathsf{tl}.\langle p, q\rangle & = & \mathrm{return}(\langle p\hat{}.r, q\rangle);\ \mathrm{dispose}(p) \\ \mathsf{snoc}.\langle p, q\rangle.a & = & |[\mathrm{var}\ h{:}L;\ \mathrm{new}(h);\ h\hat{} := \langle a, \mathrm{nil}\rangle \\ & & \quad ;\ (\ \mathrm{return}(\langle h, h\rangle) \qquad\qquad ,\ p = \mathrm{nil} \\ & & \quad\ \ []\ q\hat{}.r := h;\ \mathrm{return}(\langle p, h\rangle)\ \ ,\ p \neq \mathrm{nil} \\ & & \quad\ \ ) \\ & & \quad]|. \end{array}$$

Note that removal of dispose(p) from the program for tl does *not* turn this implementation into a nondestructive one, because evaluation of $\mathsf{snoc}.\langle p, q\rangle.a$ mutilates the list represented by $\langle p, q\rangle$.

The above implementation requires $2+\#s$ pointers to represent queue s. Another implementation that uses $1+\#s$ pointers is obtained by using so-called *circularly-linked lists*. The idea is to exploit the fact that, according to the above coupling, pointer $q\hat{}.r$ is equal to nil when $p \neq \mathrm{nil}$. To that end, the value of p is stored in $q\hat{}.r$, that is, we take $Q = L$ and we define the coupling by $[\,] \simeq \mathrm{nil}$ and for nonempty s:

$$s \simeq q \;\equiv\; \#s = (\mathrm{Min}\, i : 1 \leq i : q(\hat{}.r)^i{=}q) \;\wedge\; s = list.(q\hat{}.r)\uparrow\#s.$$

Hence, q points to the representation of the last element of s. The programs are:

$$\begin{array}{lcl} \mathsf{empty} & = & \mathrm{nil} \\ \mathsf{isempty}.q & = & q = \mathrm{nil} \\ \mathsf{hd}.q & = & q\hat{}.r\hat{}.a \\ \mathsf{tl}.q & = & |[\mathrm{var}\ p{:}L;\ p := q\hat{}.r \\ & & \quad ;\ (\ q := \mathrm{nil} \qquad ,\ p = q \\ & & \quad\ \ []\ q\hat{}.r := p\hat{}.r\ \ ,\ p \neq q \\ & & \quad\ \ ) \\ & & \quad ;\ \mathrm{return}(q);\ \mathrm{dispose}(p) \\ & & \quad]| \end{array}$$

$$
\begin{array}{l}
\mathsf{snoc}.q.a = |[\text{var } h{:}L;\ \text{new}(h) \\
\quad ;\ (\ \ h^\wedge := \langle a, h\rangle \qquad\qquad\qquad\quad ,\ q = \text{nil} \\
\quad\ \ [\!]\ \ h^\wedge := \langle a, q^\wedge.r\rangle;\ q^\wedge.r := h\ \ ,\ q \neq \text{nil} \\
\quad\ \ ) \\
\quad ;\ \text{return}(h) \\
\quad]|.
\end{array}
$$

The use of circularly-linked lists pays off when, for example, N queues with a total length of N are to be represented; then circularly-linked lists require only $2N$ pointers, whereas singly-linked lists require $3N$ pointers.

It is left to the reader to verify that both pointer representations for queues allow $O(1)$ implementations for operations $+\!\!+$ and lt ("last"). However, extending the repertoire of operations with operation ft ("front") requires a change of representation to achieve the desired degree of efficiency. The algebra that supports these additional operations is called the algebra of *concatenable deques* ("concatenable double-ended queues"):

$$\mathsf{CDQ} = (\ [T]\ \mid\ [\,], (=[\,]), [\cdot],\ +\!\!+\,, hd, tl, lt, ft\ ).$$

As signature for the refinement of CDQ we use

$$(\ D\ \mid\ \mathsf{empty}, \mathsf{isempty}, \mathsf{single}, \mathsf{cat}, \mathsf{hd}, \mathsf{tl}, \mathsf{lt}, \mathsf{ft}\ ).$$

To accommodate manipulations at both ends of lists of type $[T]$, we now use *doubly-linked lists* to represent these. That is, pointer type D is defined by

$$D = {}^\wedge\langle l{:}D, a{:}T, r{:}D\rangle.$$

Like a pointer of type L, a pointer p of type D represents list $list.p$. To enable an efficient implementation of ft, the coupling is defined as:

$$s \simeq p\ \equiv\ s = list.p\ \wedge\ (\forall\, i : 0 \leq i < \#s : p(^\wedge.r)^i = p(^\wedge.l)^{\#s-i}).$$

Note that $p(^\wedge.l)^{\#s} = p$ for $p \neq \text{nil}$, hence the l-links form a circular list and $p^\wedge.l$ points to the cell corresponding to the last element of $list.p$. Note also that $p(^\wedge.r)^{\#s} = \text{nil}$, hence the representation is not symmetric in l and r. Of course, we could have chosen for a representation symmetric in l and r but, since we want to use only one pointer that either points to the first or to the last cell in the doubly-linked list, the symmetry is broken anyway. The corresponding programs are also asymmetric:

$$
\begin{array}{lcll}
\mathsf{empty} & = & \text{nil} & \\
\mathsf{isempty}.p & = & p = \text{nil} & \\
\mathsf{single}.a & = & |[\text{var } h{:}D;\ \text{new}(h);\ h^\wedge := \langle h, a, \text{nil}\rangle;\ \text{return}(h)\ ]| & \\
\mathsf{cat}.p.q & = & q & ,p = \text{nil} \\
 & [\!] & p & ,q = \text{nil} \\
 & [\!] & p^\wedge.l^\wedge.r := q;\ p^\wedge.l, q^\wedge.l := q^\wedge.l, p^\wedge.l;\ \text{return}(p) & \\
 & & & ,p \neq \text{nil}\ \wedge\ q \neq \text{nil} \\
\mathsf{hd}.p & = & p^\wedge.a &
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{tl}.p & = & \text{nil} & , p^{\wedge}.r = \text{nil} \\
 & [] & p^{\wedge}.r^{\wedge}.l := p^{\wedge}.l;\ \text{return}(p^{\wedge}.r);\ \text{dispose}(p) & , p^{\wedge}.r \neq \text{nil} \\
\mathsf{lt}.p & = & p^{\wedge}.l^{\wedge}.a & \\
\mathsf{ft}.p & = & \text{nil} & , p^{\wedge}.r = \text{nil} \\
 & [] & |[\text{var } h{:}D;\ h := p^{\wedge}.l;\ p^{\wedge}.l := h^{\wedge}.l & \\
 & & ;\ h^{\wedge}.l^{\wedge}.r := \text{nil};\ \text{return}(p);\ \text{dispose}(h) & \\
 & &]| & , p^{\wedge}.r \neq \text{nil}\ .
\end{array}
$$

Clearly, this implementation is destructive.

4.6 Linear usage of destructive monoalgebras

We call an algebra destructive if at least one of its operations is destructive. To guarantee the correctness of programs in which destructive algebras are used, certain conditions must be satisfied. Such restrictions are formally described by P. Wadler in [36]. Using his terminology, the use of destructive algebras must be *linear*. Roughly speaking, this means that in a program no multiple references ("pointers") are created to variables that are subject to destructive operations.

In this thesis, the idea of linearity is explained informally and illustrated by a number of examples. For a more precise description we refer to [36]. We discuss restrictions on the use of x in function definitions of the form $f.x = F$, where the data type of x belongs to a destructive monoalgebra. As a first approximation, linear usage of x means in this case that

> each evaluation of F should give rise to at most one application of a *transformation* on x.

Hence, the use of creations and inspections is not restricted by this rule. Such a transformation is either a transformation of the algebra to which x belongs or a user-defined one. For instance, the identity on the data type of x is a user-defined transformation, which implies that definition $dup.x = (x, x)$ is not allowed. We say that this definition contains a *fork in* x, since (x, x) contains *two* applications of the identity function. Clearly, when x is used linearly there are no forks in x, and vice versa. Note that conditional expressions may contain more than one transformation on x. For instance, $(x, B\ [\!]\ x, \neg B)$ contains two transformations, but its evaluation gives rise to only one of these.

We illustrate the notion of linearity by some examples of linear and nonlinear usage of the destructive implementation of stacks:

$$
\begin{array}{lcl}
\mathsf{empty} & = & \text{nil} \\
\mathsf{isempty}.p & = & p = \text{nil} \\
\mathsf{cons}.a.p & = & |[\text{var } h{:}L;\ \text{new}(h);\ h^{\wedge} := \langle a, p\rangle;\ \text{return}(h)\ ]| \\
\mathsf{hd}.p & = & p^{\wedge}.a \\
\mathsf{tl}.p & = & \text{return}(p^{\wedge}.r);\ \text{dispose}(p).
\end{array}
$$

Clearly, the use of tl must be restricted because it is destructive. The use of the other transformation cons is also restricted because evaluation of cons.a.p creates an extra reference to p (and to a). There are no restrictions for creation empty and inspections isempty and hd.

To show how these operations can be used in a correct way, we consider the program:

$$f_1.p = \mathsf{cons}.(\mathsf{hd}.p).(\mathsf{tl}.p).$$

Exploiting the fact that eager evaluation does not prescribe the order in which the arguments of cons are to be evaluated, hd.p can be evaluated before tl.p. Hence, there exists a *safe* evaluation order so that execution of this program indeed establishes $[\![f_1.p]\!] = [\![p]\!]$—as desired. Since tl.$p$ destroys p, function f_1 is also destructive.

A safe evaluation order does not exist when two or more destructive operations are to be performed on the same variable, as in

$$f_2.p = \mathsf{cons}.(\mathsf{hd}.p).(\mathsf{cons}.(\mathsf{hd}.(\mathsf{tl}.p)).(\mathsf{tl}.(\mathsf{tl}.p))).$$

Evaluation of either the first or the second occurrence of tl.p destroys p, hence the remaining occurrence cannot be evaluated anymore; this definition contains a fork in p. Since, in this case, the same destructive operation is applied to p, the fork can be removed using a where-clause:

$$f_3.p = \mathsf{cons}.(\mathsf{hd}.p).(\mathsf{cons}.(\mathsf{hd}.q).(\mathsf{tl}.q)) \ |[\, q = \mathsf{tl}.p \,]|.$$

To show that it is necessary to bound the number of transformations in the right-hand side of a function definition rather than the number of destructive operations, we consider:

$$f_4.p = (\mathsf{tl}.p, \mathsf{cons}.a.p).$$

In this case, evaluation of tl.p destroys p, hence also cons.a.p is mutilated. The problem is that transformation cons.a.p creates an extra reference to p, and therefore its use is also restricted.

The necessity of restricting transformations other than tl and cons as well, follows from the following refinement of f_4:

$$f_5.p = (\mathsf{tl}.q, \mathsf{cons}.a.r) \ |[\ (q, r) = (p, p)\ ]|.$$

In this definition no stack-operations are applied to p. Nevertheless, evaluation of $f_5.p$ does not yield the desired result. The problem is that p is duplicated in the where-clause by means of two applications of the identity function. Another example of this phenomenon is

$$f_6.p = (\mathsf{tl}.q, \mathsf{cons}.a.p) \ |[\ q = p\ ]|.$$

This definition also contains a fork in p.

Remark 4.2
To obtain programs without forks for the above functions, duplication can be programmed as follows:

$$\begin{array}{lcll} \mathsf{dup}.p & = & (\mathsf{empty}, \mathsf{empty}) & , \mathsf{isempty}.p \\ & [] & (\mathsf{cons}.(\mathsf{hd}.p).q, \mathsf{cons}.(\mathsf{hd}.p).r) & \\ & & \|[\ (q,r) = \mathsf{dup}.(\mathsf{tl}.p)\ ]\| & , \neg\mathsf{isempty}.p. \end{array}$$

This operation is a destructive transformation, hence its usage must be restricted as well.

Another approach is to define operation copy which refines the identity on $[T]$:

$$\begin{array}{lcll} \mathsf{copy}.p & = & \mathrm{nil} & , p = \mathrm{nil} \\ & [] & \|[\mathrm{var}\ h{:}L; & \\ & & ;\ \mathrm{new}(h) & \\ & & ;\ h^{\wedge} := \langle p^{\wedge}.a, \mathsf{copy}.(p^{\wedge}.r)\rangle;\ \mathrm{return}(h) & \\ & &]\| & , p \neq \mathrm{nil}. \end{array}$$

The use of copy need not be restricted because the value of p is merely inspected, not altered. In other words, although the type of this operation is $L{\curvearrowright}L$, copy should not be considered as a transformation, but as an inspection followed by a creation. Note that copy cannot be programmed nondestructively in terms of stack operations because tl is destructive.
□

Finally, we present two examples involving conditional expressions. As a rule, transformations should not be used in guards of conditional expressions, but guarded expressions may contain one transformation per variable each.

Example 4.3
The following program for list reversal is nondestructive when the nondestructive implementation of stacks is used:

$$\begin{array}{lllcll} \mathsf{rev}.x = & h.x.[\,] & & & & \\ & \|[\ h.x.y & = & y & & , \mathsf{isempty}.x \\ & & [] & h.(\mathsf{tl}.x).(\mathsf{cons}.(\mathsf{hd}.x).y) & & , \neg\mathsf{isempty}.x \\ &]\|. & & & & \end{array}$$

However, since both x and y are used in a linear fashion in the definition of h, this program also allows for the use of a destructive implementation of stacks. This yields a destructive implementation of list reversal, which is in fact an *in-situ* implementation, since the cell disposed by tl can be recycled by the operation new invoked by cons.
□

Example 4.4
Let $\bowtie$ denote the operator that merges two ascending lists of integers into a single ascending list. With this operator at our disposal we can write a version of mergesort that uses linear recursion only:

$$
\begin{array}{lll}
\mathit{Mergesort} = g \circ f & & \\
\quad |[\ f.[\,] & = & \mathsf{empty} \\
\quad\ \ f.(a \vdash s) & = & \mathsf{snoc}.(f.s).[a] \\
\quad\ \ g.x & = & [\,] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , \mathsf{isempty}.x \\
\quad\qquad & [] & (\ \mathsf{hd}.x \qquad\qquad\qquad\qquad\qquad\qquad , \mathsf{isempty}.y \\
\quad\qquad & & [] \ g.(\mathsf{snoc}.(\mathsf{tl}.y).(\mathsf{hd}.x \bowtie \mathsf{hd}.y)) \ , \neg\mathsf{isempty}.y \\
\quad\qquad & &)\ |[\ y = \mathsf{tl}.x\ ]| \qquad\qquad\qquad\qquad , \neg\mathsf{isempty}.x \\
\quad]|. & &
\end{array}
$$

To allow for the use of a destructive implementation of queues, variable y has been introduced to see to it that x is used in a linear way. Note that function f creates a queue and that g destructively inspects a queue.
□

The last example shows that our rule for linear usage is not adequate in general, since it does not restrict the use of inspection g, although it is destructive. A better rule for a definition $f.x = F$ seems to be that evaluation of F gives rise to at most one application of a transformation or a destructive inspection to x. This is however a little bit too restrictive, as we shall see in the next section.

4.7 Benevolent side-effects

Crucial to the efficiency of some data structures is that inspections "rearrange" their argument so that later operations take less time. These rearrangements are harmless in the sense that the abstract value represented by the argument remains the same. Such inspections are said to have *benevolent side-effects* [16].

As a somewhat contrived example of such an inspection, we consider the following version of stack-operation hd:

$$\mathsf{hd}.p = \mathrm{return}(p^{\wedge}.a);\ |[\mathrm{var}\ h{:}L;\ \mathrm{new}(h);\ h^{\wedge} := p^{\wedge};\ \mathrm{dispose}(p);\ p := h\ ]|.$$

Note that the value of p is altered by $\mathsf{hd}.p$, but that the value of $[\![p]\!]$ remains intact. A possible advantage of this version of hd is that by selection of an appropriate cell by new(h), the cells in use can be kept in a more contiguous part of memory. In Part II we will encounter more appealing examples of inspections with benevolent side-effects (e.g., operation member in Section 11.2.1 on splay trees).

To discuss the correctness of inspections with side-effects, we associate with each inspection $g \in C \to T$ a function $g' \in C \to T \times C$ such that $g'.c = (g.c, d)$, where d equals the value of c after evaluation of $g.c$. With the corresponding abstract inspection $f \in A \to T$ we associate function $f' \in A \to T \times A$ with $f'.a = (f.a, a)$. Then f is said to be data-refined by g under coupling $\simeq$ when f' is data-refined by g' under coupling $\simeq$. Hence, an inspection with benevolent side-effects is a mixture of an inspection and a transformation.

Since inspections with benevolent side-effects do not alter the abstract value of their argument, they can be treated as nondestructive inspections. In summary, we thus have the following restriction on the usage of a parameter that belongs to a destructive algebra.

> For a function definition of the form $f.x = F$, x is said to be used in a linear fashion when each evaluation of F gives rise to at most one application of a transformation or a destructive inspection to x, not counting inspections with benevolent side-effects.

As before, this rule refers to transformations and inspections that are either operations of the algebra or user-defined functions.

Chapter 5

Analysis of functional programs and algebras

As programs are designed in a modular way, we wish to analyze them in a modular way as well. In particular, we want to determine the complexity of a program from the complexities of the operations of the algebras used in the program. For instance, given the complexities of zero and suc, we want to determine the complexity of $\mathsf{suc}^n.\mathsf{zero}$ as a function of n. We call this a *compositional* way of analyzing programs, which—in essence—means that the complexity of a composition $g \circ f$, say, can be expressed in terms of the complexities of f and g.

Since it is often difficult—or even infeasible—to determine the exact complexity of a program, it is customary to work with approximations. Traditionally, worst-case bounds are used for this purpose, but this approach does not give satisfactory results in all circumstances: the worst-case complexity of $g \circ f$ may be considerably "better" than the sum of the worst-case complexities of f and g. In other words, worst-case complexity is not compositional. As explained in Chapter 3, we will therefore use amortized costs instead of actual costs to describe the efficiency of implementations of algebras.

To guarantee that the amortized cost of a program is the sum of the amortized costs of the operations used by it, the corresponding algebras should be used linearly—in the same way as destructive algebras should be used linearly to guarantee the correctness of a program. This connection between destructivity and amortization leads to interesting trade-offs, as will be shown at the end of this chapter.

5.1 Cost measures

As explained in Section 4.2, execution of a functional program f on an input x boils down to the reduction of expression $f.x$. Such a reduction requires time as well as space and it is the goal of an *analysis* to quantify the use of these resources. More concretely, the goal of an analysis of a program f is to

express, as a function of x, the *cost* of evaluating $f.x$. Here, the cost depends on the choice of a *cost measure*, which is chosen beforehand. Since execution of functional programs amounts to the reduction of expressions, our cost measures will be mappings from expressions to the real numbers.

There are two—usually conflicting—demands that determine the suitability of a cost measure: it should be *realistic* for the intended purpose, and it should be *manageable*, i.e., not overly complicated. For instance, cost measure $\mathcal{N}$, defined by $\mathcal{N}(E)$="the total number of unfoldings needed to evaluate E", is in general a realistic measure but often much too complicated. For a sorting program, a measure like "the number of comparisons needed to evaluate E" is probably more suitable; this measure counts the number of unfoldings of the predefined operation $<$, say.

Once a suitable cost measure has been defined, the program can be analyzed. We want to do so by analyzing the auxiliary functions, which constitute the program, one at a time. To support such a modular analysis, we introduce the "cost of a function".

Definition 5.1
A *cost measure* is a mapping from expressions to the real numbers. For a functional program f, the *cost of* f with respect to cost measure $\mathcal{T}$ is the mapping $\mathcal{T}[f]$ defined by

$$\mathcal{T}[f](E) = \mathcal{T}(f.E) - \mathcal{T}(E),$$

for $E \in \operatorname{dom} f$.
□

The important property of mapping $\mathcal{T}[\cdot]$ is that it satisfies the *composition rule*, which enables us to decompose the analysis of compositions like $g \circ f$.

Property 5.2 (composition rule for $\mathcal{T}[\cdot]$)
If $\mathcal{T}((g \circ f).E) = \mathcal{T}(g.(f.E))$, then

$$\mathcal{T}[g \circ f](E) = \mathcal{T}[f](E) + \mathcal{T}[g](f.E).$$

Proof

$$\begin{aligned}
& \mathcal{T}[g \circ f](E) \\
= & \quad \{ \text{ Definition 5.1 } \} \\
& \mathcal{T}((g \circ f).E) - \mathcal{T}(E) \\
= & \quad \{ \text{ above proviso } \} \\
& \mathcal{T}(g.(f.E)) - \mathcal{T}(E) \\
= & \quad \{ \text{ arithmetic } \} \\
& \mathcal{T}(g.(f.E)) - \mathcal{T}(f.E) + \mathcal{T}(f.E) - \mathcal{T}(E) \\
= & \quad \{ \text{ Definition 5.1 } \} \\
& \mathcal{T}[g](f.E) + \mathcal{T}[f](E).
\end{aligned}$$

□

The proviso in this property is rather weak; it is satisfied by all cost measures that we use for our programs. This has to do with the fact that we use $\circ$ as an abbreviation mechanism only, and therefore we do not want that expressions $(g \circ f).E$ and $g.(f.E)$ are distinguished by our cost measures. Fancy cost measures that do not satisfy this proviso, such as "the length of expression E" or "the number of occurrences of $\circ$ in E", will not be used.

At first sight one might think that another property of $\mathcal{T}[f]$ is that $\mathcal{T}[f](E)$ depends on the *value* of E only, not on the *shape* of E, but this is not true in general. Take, for instance, cost measure $\mathcal{S}$ defined by $\mathcal{S}(E)$="the maximal amount of storage space in use during the evaluation of E". Since the maximal amount of storage space in use during the evaluation of $f.E$ is either in use before f is unfolded, in which case $\mathcal{S}(f.E) = \mathcal{S}(E)$, or only after f has been unfolded, in which case $\mathcal{S}(f.E) > \mathcal{S}(E)$, we observe that $\mathcal{S}[f](E)$ may be zero as well as positive, depending on the shape of E.

In the sequel, however, we will concentrate on cost measures that are related to the amount of time required for the evaluation of expressions. We call these *distributive cost measures*. The class of distributive cost measures will not be defined explicitly: we only postulate some *distribution rules* which are satisfied by such cost measures.

One of the most important distribution rules for a cost measure $\mathcal{T}$ is that for a function definition of the form $f.x = F$:

$$\mathcal{T}(f.E) = \mathcal{T}(E) + C_f(val(E)) + \mathcal{T}(F^x_{val(E)}),$$

where $val(E)$ denotes the value of E, and $C_f \in \mathrm{dom}\, f \rightarrow$Real describes the actual cost of unfolding f's definition for each possible value. Since eager evaluation of $f.E$ amounts to the evaluation of E, followed by the unfolding of f's definition, which in turn gives rise to the evaluation of F with x replaced by the value of E (cf. Section 4.2), we see that the cost of $f.E$ equals the sum of the costs of these three phases for distributive cost measures. Hence, the value of E is computed only once and *shared* by all occurrences of x in F. A consequence of this distribution rule is that

$$\mathcal{T}[f](E) = \mathcal{T}[f](val(E)),$$

since $val(val(E)) = val(E)$.

In the applications in Part II, all cost measures count unfoldings of (usually recursive) user-defined functions and/or unfoldings of predefined operations. Since we assume that our programs are executed according to the eager evaluation scheme (Section 4.2), these cost measures enjoy some nice distributivity properties. Below, we will present these properties for an extreme member of this class, viz. the previously introduced cost measure $\mathcal{N}$, which counts *all* unfoldings of predefined operations and user-defined functions.

Distribution rules for $\mathcal{N}$

1. The first rule is an instance of the above distribution rule for $\mathcal{T}(f.E)$, where f is defined by $f.x = F$:

$$\mathcal{N}(f.E) = \mathcal{N}(E) + 1 + \mathcal{N}(F^x_{val(E)}).$$

2. With respect to $\circ$, we have (cf. Property 5.2):

$$\mathcal{N}((g \circ f).E) = \mathcal{N}(g.(f.E)).$$

3. For conditional expressions the following rule is appropriate:

$$\mathcal{N}((E, B \,[]\, F, \neg B)) = \mathcal{N}(B) + \begin{cases} \mathcal{N}(E) & , B \\ \mathcal{N}(F) & , \neg B, \end{cases}$$

   which expresses that in case of complementary guards, only one of these is evaluated, and that, subsequently, only the relevant expression is evaluated.

4. As for where-clauses, we consider expressions of the form $F|[x = E]|$. These are reduced in the same way as $f.E$ is reduced. Hence:

$$\mathcal{N}(F|[x = E]|) = \mathcal{N}(E) + 1 + \mathcal{N}(F^x_{val(E)}).$$

5. Another simple rule is that for a predefined binary operator $\oplus$, say, we have

$$\mathcal{N}(E \oplus F) = \mathcal{N}(E) + \mathcal{N}(F) + 1,$$

   which reflects the fact that $\mathcal{N}$ counts unfoldings of predefined operations.

6. In connection with the use of patterns in the left-hand sides of function definitions, we consider a definition of the form

$$\begin{array}{lcl} f.[\,] & = & F \\ f.(a \vdash s) & = & G. \end{array}$$

   For this definition we have the property that

$$\mathcal{N}(f.E) = \mathcal{N}(E){+}1{+}1{+} \begin{cases} \mathcal{N}(F) & , E = [\,] \\ 1{+}1{+}\mathcal{N}(G^{a,s}_{val(hd.E),val(tl.E)}) & , E \neq [\,]. \end{cases}$$

   It may be interpreted as follows: first E is evaluated and it is determined whether E is empty or not, which requires one unfolding of $(=[\,])$. Subsequently f's definition is unfolded, which gives rise to evaluation of F *or* to evaluation of G with a and s replaced by the values of $hd.E$ and $tl.E$. The latter case requires one unfolding of hd and one of tl.

7. Finally, there is the trivial but useful property that evaluation of an expression terminates as soon as it has been reduced to its value, which is reflected by the rule

$$\mathcal{N}(val(E)) = 0.$$

These rules hold for all cost measures $\mathcal{T}$ that count unfoldings of user-defined and/or predefined functions, except that the "+1 terms" in the equations for $\mathcal{T}(f.E)$, $\mathcal{T}(E \oplus F)$, and $\mathcal{T}(F|[x = E]|)$ are absent for those unfoldings that are not counted by $\mathcal{T}$.

To facilitate the definition of distributive cost measures, we use *dots* on top of the =-signs in function definitions to mark the unfoldings that are counted by the measure. In case a function definition consists of several alternatives the dot marks *all* the alternatives. For example, the cost measure $\mathcal{T}$ defined by

$$\begin{array}{lcll} bubble.[\,] & = & [\,] & \\ bubble.[a] & = & [a] & \\ bubble.(a \vdash b \vdash s) & \doteq & a \vdash bubble.(b \vdash s) & , a \leq b \\ & \dot{[]} & b \vdash a \vdash bubble.s & , a > b \end{array}$$

satisfies $\mathcal{T}(E)$="the number of unfoldings of the recursive alternative of *bubble* needed to evaluate E", and also $\mathcal{T}(E)$="the number of unfoldings of $\leq$ (or $>$) needed to evaluate E".

The distribution rules for $\mathcal{T}$ enable us to derive relations for $\mathcal{T}[f]$ that follow the structure of the definition of f. In case f's definition is recursive, this gives rise to recurrence relations for $\mathcal{T}[f]$. For example, using Definition 5.1, we have as recurrence relation for $\mathcal{T}[bubble]$:

$$\begin{array}{lcl} \mathcal{T}[bubble]([\,]) & = & 0 \\ \mathcal{T}[bubble]([a]) & = & 0 \\ \mathcal{T}[bubble](a \vdash b \vdash s) & = & 1 + \begin{cases} \mathcal{T}[bubble](b \vdash s) & , a \leq b \\ \mathcal{T}[bubble](s) & , a > b. \end{cases} \end{array}$$

Using the distribution rules, this recurrence relation may actually be derived, but this is too laborious and therefore omitted. Note that it suffices to investigate $\mathcal{T}[bubble](s)$ for all *values* s because $\mathcal{T}$ is distributive.

5.2 Worst-case analysis

It seems that we have already reached our goal now that we have introduced $\mathcal{T}[\cdot]$, since it satisfies some nice composition and distribution properties when $\mathcal{T}$ is distributive. Unfortunately, however, an explicit formula for $\mathcal{T}[f]$ cannot always be given, and in such a case we must content ourselves with approximations of $\mathcal{T}[f]$. A common method of approximating the cost of a function f is to use upper bounds for $\mathcal{T}[f](x)$ that depend on the "size" of x only. In these so-called

worst-case analyses, the input values are partitioned into classes of equal-sized inputs, and the *worst-case complexity* of a function f is then determined for each possible size N. That is, with $\#x$ denoting the size of x,

$$(\mathrm{Max}\, x : x \in \mathrm{dom}\, f \ \wedge\ \#x = N : \mathcal{T}[f](x))$$

is determined as a function of N. For program *bubble*, for example, the worst case occurs when s is ascending; the solution of the recurrence relation then is $\mathcal{T}[bubble](s) = (\#s{-}1)\max 0$.

In many cases it is even too complicated to find an explicit formula for the worst-case complexity. Instead of an exact formula we then give an upper bound like $\#s$ for $\mathcal{T}[bubble](s)$, or we simply say that $\mathcal{T}[bubble](s)$ is $O(\#s)$ when we are only interested in the *asymptotic cost*. The problem with such approximations is that they are *not compositional* in the sense that a tight upper bound for the worst-case complexity of $\mathcal{T}[g \circ f]$ cannot be obtained from tight upper bounds for the worst-case complexities of $\mathcal{T}[f]$ and $\mathcal{T}[g]$. This is illustrated by the next example.

Example 5.3
We analyze a binary implementation of $(\,\mathrm{Nat} \mid 0, (+1)\,)$. The concrete data type is $[\{0,1\}]$ and the concrete operations are

$$\begin{array}{lcl} \mathsf{zero} & \doteq & [\,] \\ \mathsf{suc}.[\,] & \doteq & [1] \\ \mathsf{suc}.(0 \vdash s) & \doteq & 1 \vdash s \\ \mathsf{suc}.(1 \vdash s) & \doteq & 0 \vdash \mathsf{suc}.s. \end{array}$$

The cost measure defined by the dots is called $\mathcal{T}$, which could also be done in words by $\mathcal{T}(E)$="the number of unfoldings of zero and suc needed to evaluate E". Our goal is to derive a tight bound for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ as a function of n.

To this end we first analyze zero and suc in isolation. Clearly, $\mathcal{T}[\mathsf{zero}] = 1$, and for $\mathcal{T}[\mathsf{suc}]$ we have the following recurrence relation:

$$\begin{array}{lcl} \mathcal{T}[\mathsf{suc}]([\,]) & = & 1 \\ \mathcal{T}[\mathsf{suc}](0 \vdash s) & = & 1 \\ \mathcal{T}[\mathsf{suc}](1 \vdash s) & = & 1 + \mathcal{T}[\mathsf{suc}](s). \end{array}$$

A tight upper bound for $\mathcal{T}[\mathsf{suc}](s)$ in terms of $\#s$ thus is $1{+}\#s$.

Repeatedly applying the composition rule for $\mathcal{T}[\cdot]$ yields

$$\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = \mathcal{T}[\mathsf{zero}] + (\Sigma\, i : 0 \le i < n : \mathcal{T}[\mathsf{suc}](\mathsf{suc}^i.\mathsf{zero})).$$

In order to use the upper bound for $\mathcal{T}[\mathsf{suc}]$, we observe that list $\mathsf{suc}^i.\mathsf{zero}$ has length at most $1{+}\log_2(i{+}1)$. Therefore $\mathcal{T}[\mathsf{suc}](\mathsf{suc}^i.\mathsf{zero})$ is at most $2{+}\log_2(i{+}1)$, and we obtain an $O(n \log n)$ bound for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$.

However, $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ is $O(n)$, as will be shown in Example 5.7; the fact that $\mathcal{T}[\mathsf{suc}](s) \le \#s$ (and that this bound is tight) cannot be used to prove this.
□

5.3 Amortized cost of functions

Suppose that our goal is to determine the worst-case complexity of f^n as a function of n. As demonstrated in Example 5.3, the worst-case complexity of f does not, in general, provide a clue to the worst-case complexity of f^n. Instead, we will use *amortized costs* for f and f^n, which are defined in terms of a *potential function*. In the functional setting, a potential function is—like a cost measure—a mapping on expressions. Unlike a cost measure, however, the potential of an expression depends on its value only, not on the way in which this value is expressed. This corresponds to the idea behind potential functions in Chapter 3: in the imperative setting, a potential function depends on the value of the state only, not on how the state has been reached.

Definition 5.4
A *potential function* Φ is a mapping from expressions to the real numbers satisfying

$$\Phi.E = \Phi.val(E).$$

The *amortized cost of* f with respect to cost measure $\mathcal{T}$ and potential Φ is the mapping $\mathcal{A}[f]$ defined by

$$\mathcal{A}[f](E) = \mathcal{T}[f](E) + \Phi.(f.E) - \Phi.E,$$

for $E \in \operatorname{dom} f$.
□

In an amortized analysis, the potential function is defined for the relevant values only, e.g., for the data type of an algebra. For other values, the definition of the potential is extended in a straightforward way. In connection with the use of tuples, for example, the potential of the empty tuple $\perp$ is usually defined equal to 0. For constant f, Definition 5.4 then yields

$$\mathcal{A}[f] = \mathcal{T}[f] + \Phi.f.$$

Also, in connection with the use of pairs, $\Phi.(E, F)$ is usually defined equal to the sum of $\Phi.E$ and $\Phi.F$, so that

$$\mathcal{A}[f]((E, F)) = \mathcal{T}[f]((E, F)) + \Phi.(f.E.F) - \Phi.E - \Phi.F$$

for functions f on pairs.

Since $\Phi.E = \Phi.val(E)$, $\mathcal{A}[\cdot]$ inherits a number of properties of $\mathcal{T}[\cdot]$. For instance, the composition rule is inherited.

Property 5.5 (composition rule for $\mathcal{A}[\cdot]$)
If

$$\mathcal{T}[g \circ f](E) = \mathcal{T}[f](E) + \mathcal{T}[g](f.E),$$

then also

$$\mathcal{A}[g \circ f](E) = \mathcal{A}[f](E) + \mathcal{A}[g](f.E).$$

Proof

$$\begin{array}{ll} & \mathcal{A}[g \circ f](E) \\ = & \quad \{ \text{ Definition 5.4 } \} \\ & \mathcal{T}[g \circ f](E) + \Phi.((g \circ f).E) - \Phi.E \\ = & \quad \{ \text{ above proviso and restriction on } \Phi \text{ in Definition 5.4 } \} \\ & \mathcal{T}[f](E) + \mathcal{T}[g](f.E) + \Phi.(g.(f.E)) - \Phi.E \\ = & \quad \{ \text{ arithmetic } \} \\ & \mathcal{T}[g](f.E) + \Phi.(g.(f.E)) - \Phi.(f.E) + \mathcal{T}[f](E) + \Phi.(f.E) - \Phi.E \\ = & \quad \{ \text{ Definition 5.4 } \} \\ & \mathcal{A}[g](f.E) + \mathcal{A}[f](E). \end{array}$$

□

Also "independence of the shape of E" is inherited by $\mathcal{A}[f](E)$.

Property 5.6
If $\mathcal{T}[f](E) = \mathcal{T}[f](val(E))$, then $\mathcal{A}[f](E) = \mathcal{A}[f](val(E))$ as well.
□

There is however a crucial difference between $\mathcal{A}[\cdot]$ and $\mathcal{T}[\cdot]$. To discuss this difference, we introduce cost measure $\mathcal{A}$:

$$\mathcal{A}(E) = \mathcal{T}(E) + \Phi.E.$$

Then the amortized cost of f w.r.t. $\mathcal{T}$ and Φ (cf. Definition 5.4) equals the cost of f w.r.t. $\mathcal{A}$ (cf. Definition 5.1). Since $\mathcal{A}$ inherits the common properties of $\mathcal{T}$ and Φ, Properties 5.5 and 5.6 are readily seen to be valid. However, distributivity of $\mathcal{T}$ does not imply distributivity of $\mathcal{A}$ because Φ is not distributive!

Now, since $\mathcal{T}$ is usually distributive, it is not difficult to obtain a relation for $\mathcal{T}[f]$ based on the structure of f's definition. But $\mathcal{A}$ is not distributive, and therefore a relation for $\mathcal{A}[f]$ cannot be obtained in general by substituting $\mathcal{A}$ for $\mathcal{T}$ in the relation for $\mathcal{T}[f]$. Consider, for instance, function *dup* defined by

$$dup.x \doteq (x, x).$$

Assuming that $\mathcal{T}$ is distributive and that Φ distributes over pairs, hence that $\mathcal{A}$ distributes over pairs (i.e., $\mathcal{A}((x, y)) = \mathcal{A}(x) + \mathcal{A}(y)$), we observe for value x:

$$
\begin{array}{ll}
& \mathcal{T}[dup](x) \\
= & \quad \{ \text{ Definition 5.1 } \} \\
& \mathcal{T}(dup.x) - \mathcal{T}(x) \\
= & \quad \{ \text{ definition } dup \} \\
& 1 + \mathcal{T}((x,x)) - \mathcal{T}(x) \\
= & \quad \{ \mathcal{T} \text{ is distributive } \} \\
& 1 + \mathcal{T}(x) + \mathcal{T}(x) - \mathcal{T}(x) \\
= & \quad \{ \mathcal{T}(x) = 0 \} \\
& 1,
\end{array}
\qquad
\begin{array}{ll}
& \mathcal{A}[dup](x) \\
= & \quad \{ \text{ Definition 5.1 } \} \\
& \mathcal{A}(dup.x) - \mathcal{A}(x) \\
= & \quad \{ \text{ definition } dup \} \\
& 1 + \mathcal{A}((x,x)) - \mathcal{A}(x) \\
= & \quad \{ \mathcal{A} \text{ distributes over pairs } \} \\
& 1 + \mathcal{A}(x) + \mathcal{A}(x) - \mathcal{A}(x) \\
= & \quad \{ \mathcal{T}(x) = 0, \text{ hence } \mathcal{A}(x) = \Phi.x \} \\
& 1 + \Phi.x.
\end{array}
$$

Hence, $\mathcal{T}[dup](x) = 1$ but $\mathcal{A}[dup](x) = 1 + \Phi.x$. Similarly, when x is tripled, the difference becomes $2\,\Phi.x$, and so on. On the other hand, for function *sink*, with $sink.x \doteq \bot$, we have that $\mathcal{T}[sink](x) = 1$ and $\mathcal{A}[sink](x) = 1 - \Phi.x$.

In general, for a definition of the form $f.x = F$, there will be no difference between the relations for $\mathcal{A}[f]$ and $\mathcal{T}[f]$ when x occurs exactly once in F (and $\mathcal{T}$ is distributive). This is reflected by the difference between

$$\mathcal{T}[f](E) = C_f(val(E)) + \mathcal{T}(F^x_{val(E)}),$$

and

$$\mathcal{A}[f](E) = C_f(val(E)) + \mathcal{A}(F^x_{val(E)}) - \Phi.E.$$

(Note that the equality for $\mathcal{T}[f](E)$ implies the equality for $\mathcal{A}[f](E)$.) To rewrite $\mathcal{T}(F^x_{val(E)})$ into a formula in terms of $\mathcal{T}[\cdot]$, as many terms $\mathcal{T}(val(E))$ can be used as necessary (since $\mathcal{T}(val(E)) = 0$), but to rewrite $\mathcal{A}(F^x_{val(E)})$ into a formula in terms of $\mathcal{A}[\cdot]$, only a single term $\Phi.E$ is available.

In Section 5.5 we will present a simple method to ensure that the relations for $\mathcal{T}[\cdot]$ are inherited by $\mathcal{A}[\cdot]$.

5.4 Amortized analysis

Consider a function f for which $\operatorname{rng} f \subseteq \operatorname{dom} f$. Suppose that the worst-case complexity of f^n (as a function of n) cannot be determined from the worst-case complexity of f. Then we perform an *amortized analysis*, which typically proceeds as follows. For the sake of convenience we assume that $\#(f.x) = \#x$.

Having defined cost measure $\mathcal{T}$, our aim is to bound $\mathcal{T}[f^n](x)$ in terms of $\#x$ and n. To this end we repeatedly apply the composition rule for $\mathcal{A}[\cdot]$ (Property 5.5) to $\mathcal{A}[f^n]$, where

$$\mathcal{A}[f^n](x) = \mathcal{T}[f^n](x) + \Phi.(f^n.x) - \Phi.x.$$

This yields the following equation:

$$(1) \quad \mathcal{T}[f^n](x) = (\Sigma\, i : 0 \leq i < n : \mathcal{A}[f](f^i.x)) + \Phi.x - \Phi.(f^n.x),$$

from which we obtain a tight bound for $\mathcal{T}[f^n](x)$ by choosing a suitable potential Φ. But, just as it is difficult to find an explicit formula for $\mathcal{T}[f](x)$, it is, in general, difficult to find an explicit formula for $\mathcal{A}[f](x)$. Therefore, we use the *worst-case* approximation of $\mathcal{A}[f]$ by considering the following bound for $\mathcal{T}[f^n]$ derived from (1), using that $\#(f.x) = \#x$:

$$\mathcal{T}[f^n](x) \leq n * (\mathrm{Max}\, y : \#y = \#x : \mathcal{A}[f](y)) + \Phi.x - \Phi.(f^n.x).$$

The object is now to choose Φ such that $(\mathrm{Max}\, y : \#y{=}\#x : \mathcal{A}[f](y))$ is minimized, thereby keeping in mind that $\Phi.x - \Phi.(f^n.x)$ may not be too large as well.

As for the latter requirement on Φ, it is often convenient to choose a *nonnegative* function for Φ. The above upper bound then simplifies to

$$\mathcal{T}[f^n](x) \leq n * (\mathrm{Max}\, y : \#y = \#x : \mathcal{A}[f](y)) + \Phi.x.$$

By keeping $\Phi.x$ small with respect to $\#x$, a tight upper bound for $\mathcal{T}[f^n](x)$ in terms of $\#x$ is obtained.

This above method of amortized analysis is applied in the next example to improve the analysis of $\mathsf{suc}^n.\mathsf{zero}$ in Example 5.3.

Example 5.7 (see Example 5.3)
The amortized costs for zero and suc are

$$\begin{aligned} \mathcal{A}[\mathsf{zero}] &= \mathcal{T}[\mathsf{zero}] + \Phi.\mathsf{zero} \\ \mathcal{A}[\mathsf{suc}](s) &= \mathcal{T}[\mathsf{suc}](s) + \Phi.(\mathsf{suc}.s) - \Phi.s, \end{aligned}$$

where $\Phi \in [\{0,1\}]{\rightarrow}\mathrm{Real}$ is the potential function. Since

$$\mathcal{A}[\mathsf{suc}^n.\mathsf{zero}] = \mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] + \Phi.(\mathsf{suc}^n.\mathsf{zero}),$$

the composition rule for $\mathcal{A}[\cdot]$ yields

$$\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = \mathcal{A}[\mathsf{zero}] + (\Sigma\, i : 0{\leq}i{<}n : \mathcal{A}[\mathsf{suc}](\mathsf{suc}^i.\mathsf{zero})) - \Phi.(\mathsf{suc}^n.\mathsf{zero}).$$

This equality yields an $O(n)$ bound for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ provided Φ is nonnegative, and $\mathcal{A}[\mathsf{zero}]$ and $\mathcal{A}[\mathsf{suc}](s)$ are both constant.

By definition, we have that $\mathcal{A}[\mathsf{zero}] = 1 + \Phi.[\,]$, hence constant. To keep $\mathcal{A}[\mathsf{zero}]$ small and Φ nonnegative, we take $\Phi.[\,] = 0$. Then $\mathcal{A}[\mathsf{zero}] = 1$.

Similarly, we have that $\mathcal{A}[\mathsf{suc}]([\,])$ is $O(1)$, independent of the definition of Φ. To *derive* a suitable definition for Φ, we first observe for the second alternative of suc:

$$
\begin{array}{ll}
 & \mathcal{A}[\mathsf{suc}](0 \vdash s) \\
= & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{suc}] \ \} \\
 & \mathcal{T}[\mathsf{suc}](0 \vdash s) + \Phi.(\mathsf{suc}.(0 \vdash s)) - \Phi.(0 \vdash s) \\
= & \quad \{ \text{ recurrence relation for } \mathcal{T}[\mathsf{suc}], \text{ definition of } \mathsf{suc} \ \} \\
 & 1 + \Phi.(1 \vdash s) - \Phi.(0 \vdash s) \\
= & \quad \{ \text{ simplify by choosing } \Phi.(b \vdash s) = \varphi.b + \Phi.s \ \} \\
 & 1 + \varphi.1 - \varphi.0.
\end{array}
$$

By the introduction of φ, $\mathcal{A}[\mathsf{suc}](0 \vdash s)$ becomes independent of s, hence $O(1)$. To obtain a suitable definition for φ we proceed with the last alternative of suc:

$$
\begin{array}{ll}
 & \mathcal{A}[\mathsf{suc}](1 \vdash s) \\
= & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{suc}] \ \} \\
 & \mathcal{T}[\mathsf{suc}](1 \vdash s) + \Phi.(\mathsf{suc}.(1 \vdash s)) - \Phi.(1 \vdash s) \\
= & \quad \{ \text{ recurrence relation for } \mathcal{T}[\mathsf{suc}], \text{ definition of } \mathsf{suc} \ \} \\
 & 1 + \mathcal{T}[\mathsf{suc}](s) + \Phi.(0 \vdash \mathsf{suc}.s) - \Phi.(1 \vdash s) \\
= & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{suc}] \ \} \\
 & \mathcal{A}[\mathsf{suc}](s) + 1 + \Phi.(0 \vdash \mathsf{suc}.s) - \Phi.(\mathsf{suc}.s) - (\Phi.(1 \vdash s) - \Phi.s) \\
= & \quad \{ \ \Phi.(b \vdash s) = \varphi.b + \Phi.s, \text{ see previous derivation } \} \\
 & \mathcal{A}[\mathsf{suc}](s) + 1 + \varphi.0 - \varphi.1.
\end{array}
$$

The term $1 + \varphi.0 - \varphi.1$ may be interpreted as the amortized cost of unfolding $\mathsf{suc}.(1 \vdash s)$.

To ensure that $\mathcal{A}[\mathsf{suc}]$ is $O(1)$, φ should be chosen such that $1+\varphi.0-\varphi.1 \leq 0$. To obtain a nonnegative and small Φ, we take $\varphi.0 = 0$ and $\varphi.1 = 1$ (hence, $1 + \varphi.0 - \varphi.1 = 0$). As potential we thus obtain:

$$
\begin{array}{lcl}
\Phi.[\,] & = & 0 \\
\Phi.(b \vdash s) & = & b + \Phi.s,
\end{array}
$$

hence $\Phi.s$ equals the number of 1's in s. From this definition for Φ, it follows by induction that $\mathcal{A}[\mathsf{suc}](s) = 2$. The above equation for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ then yields

$$\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = 1 + 2n - \Phi.(\mathsf{suc}^n.\mathsf{zero}),$$

from which we conclude that $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ is $O(n)$, since Φ is nonnegative. We remark that the upper bound $1+2n$ is as good as tight ($\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = 2n$ when n is a power of two).
□

In this example, the potential has been derived in a calculational way. In particular, the fact that $\varphi.b = b$ is a suitable definition directly follows from the derivation. The systematic derivation of potential functions is an important topic in the case-studies of Part II.

5.5 Amortization and linearity

At this point it should be clear that worst-case *amortized* complexities should be used instead of worst-case *actual* complexities as a general way to describe the efficiency of implementations of algebras. But, as explained at the end of Section 5.3, a problem with the use of amortized costs is that for a function definition of the form $f.c = F$, say, the relations for $\mathcal{T}[f]$ are not necessarily inherited by $\mathcal{A}[f]$. Recall, for example, that $\mathcal{A}[dup](c) = 1 + \Phi.c$ whereas $\mathcal{T}[dup](c) = 1$ for $dup.c \doteq (c, c)$. To achieve that the relations for $\mathcal{T}[f]$ are indeed inherited by $\mathcal{A}[f]$ some restrictions must be imposed on expression F. Below we will describe these restrictions under the assumption that $\mathcal{A}[\cdot]$ is defined according to the following scheme.

Let C be a (concrete) monoalgebra with data type C. Let $\mathcal{T}$ denote a cost measure for the programs of C, and let Φ be a potential function defined on C. In accordance with Definition 5.4, the amortized costs of creations and transformations are defined as follows.

(a) For a creation g of type $T \to C$:

$$\mathcal{A}[g](x) = \mathcal{T}[g](x) + \Phi.(g.x).$$

(b) For a transformation g of type $C \times C \curvearrowright C$:

$$\mathcal{A}[g](c, d) = \mathcal{T}[g](c, d) + \Phi.(g.c.d) - \Phi.c - \Phi.d.$$

To obtain (a) as instantiation of Definition 5.4, we take $\Phi.x = 0$ for $x \in T$. Doing the same for inspections, however, yields a not so useful definition. Instead, we distinguish two types of inspections.

(c) For an inspection g of type $C \to T$:

(i) $\mathcal{A}[g](c) = \mathcal{T}[g](c)$, or
(ii) $\mathcal{A}[g](c) = \mathcal{T}[g](c) - \Phi.c$.

Here, (ii) can be obtained as instantiation of Definition 5.4, again with $\Phi.x = 0$ for $x \in T$. In most cases, however, (i) is the appropriate definition for $\mathcal{A}[g]$: defining $\mathcal{A}[g](c) = \mathcal{T}[g](c)$ means that the costs of inspections are *not* amortized.

If $\mathcal{A}[\cdot]$ is defined according to the above scheme, the restriction on $f.c = F$ is that each evaluation of F should give rise to at most one application to c of a transformation or an inspection of type (ii). In that case, $\mathcal{A}[f]$ inherits the relations for $\mathcal{T}[f]$. Note that the use of inspections of type (i) is not restricted by this rule. This is essential because inspections are typically used in combination with a transformation, and this would be impossible when the potential played a role in the amortized costs of both.

Inspections are allowed to have benevolent side-effects. In that case the potential of c may be altered by the evaluation of $g.c$, and this change in potential

may be necessary to amortize the cost of g. To analyze such an inspection, we consider function $g' \in C \to T \times C$ with $g'.c = (g.c, d)$, where d equals the value of c after evaluation of $g.c$ (cf. Section 4.7). The amortized costs of g' are defined as follows:

$$\mathcal{A}[g'](c) = \mathcal{T}[g](c) + \Phi.d - \Phi.c.$$

In Part II we will see some examples of inspections with benevolent side-effects that are efficient in the amortized sense only (e.g., operation member in Section 11.2.1 on splay trees). In the above rule, such an inspection may be treated as an inspection of type (i).

Calling an inspection of type (i) a nondestructive inspection and an inspection of type (ii) a destructive one, the restriction on $f.c = F$ coincides with the notion of linearity defined at the end of Chapter 4. Therefore we shall say that an algebra which is efficient in the amortized sense only must be restricted to *linear usage* in order to ensure that the amortized costs add up in the same way as the actual costs. An important observation is now that an algebra which is either destructive or efficient in the amortized sense only might as well be both. This leads to interesting trade-offs, as we will show in the next section.

5.6 Purely-functional deques

In Chapter 7 of his Ph.D. thesis [17] Hoogerwoord derives an efficient implementation for a symmetric set of list operations. The idea behind his design is based on an efficient implementation of queues in LISP (see e.g. [13, pp. 250–251]). In this section, we first cast this design in our style, after which we go on to experiment with the amortized analysis of more advanced operations programmed in terms of the symmetric list operations. In this way we gain some experience with the computation of the amortized costs of composite functions from the amortized costs of the constituent parts.

5.6.1 Specification and implementation

Using the algebra of stacks

$$\mathsf{S} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl\,),$$

whose operations all have $O(1)$ actual cost, we present an implementation of the algebra of *deques* ("double-ended queues")

$$\mathsf{DQ} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl, \dashv, lt, ft, rev\,),$$

with signature

$$\mathsf{S2} = (\,\mathrm{X} \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{cons}, \mathsf{hd}, \mathsf{tl}, \mathsf{snoc}, \mathsf{lt}, \mathsf{ft}, \mathsf{rev}\,),$$

such that all operations of S2 have $O(1)$ amortized cost. Apart from the symmetrical counterparts of the last three operations of S, we have included operation *rev*, because any program for *rev* in terms of the other operations of DQ is bound to be linear, while it is a simple $O(1)$ operation in the representation used for $[T]$ below. Hence, *rev* cannot be programmed without loss of efficiency in terms of the other operations of DQ.

The definition of S2 is as follows (see [17] for a solid derivation). Set X is a subset of $[T]\times[T]$ and S2 refines DQ under abstraction $[\![\cdot]\!]$ with $[\![\langle s,t\rangle]\!] = s +\!\!+ rev.t$. To allow for an efficient implementation of hd and lt, set X is defined as follows:

$$\mathrm{X} = \{\langle s,t\rangle \mid (s=[\,] \;\Rightarrow\; \#t \leq 1) \;\wedge\; (t=[\,] \;\Rightarrow\; \#s \leq 1)\}.$$

Using the operations of S, the operations of S2 are programmed as follows:

$$\mathsf{empty} \;=\; \langle [\,],[\,]\rangle$$

$$\mathsf{isempty}.\langle s,t\rangle \;=\; s=[\,] \;\wedge\; t=[\,]$$

$$\begin{array}{lcll} \mathsf{cons}.a.\langle s,t\rangle & = & \langle a \vdash s,t\rangle & ,t\neq[\,] \\ & & [\!] \;\; \langle [a],s\rangle & ,t=[\,] \end{array}$$

$$\begin{array}{lcll} \mathsf{hd}.\langle s,t\rangle & = & hd.s & ,s\neq[\,] \\ & & [\!] \;\; hd.t & ,s=[\,] \end{array}$$

$$\begin{array}{lcll} \mathsf{tl}.\langle s,t\rangle & = & \langle [\,],[\,]\rangle & ,\#s=0 \\ & & [\!] \;\; \langle rev.(t{\downarrow}k),t{\uparrow}k\rangle \quad |[\; k=\#t \operatorname{div} 2 \;]| & ,\#s=1 \\ & & [\!] \;\; \langle tl.s,t\rangle & ,\#s\geq 2 \end{array}$$

$$\begin{array}{lcll} \mathsf{snoc}.\langle s,t\rangle.a & = & \langle s,a\vdash t\rangle & ,s\neq[\,] \\ & & [\!] \;\; \langle t,[a]\rangle & ,s=[\,] \end{array}$$

$$\begin{array}{lcll} \mathsf{lt}.\langle s,t\rangle & = & hd.t & ,t\neq[\,] \\ & & [\!] \;\; hd.s & ,t=[\,] \end{array}$$

$$\begin{array}{lcll} \mathsf{ft}.\langle s,t\rangle & = & \langle [\,],[\,]\rangle & ,\#t=0 \\ & & [\!] \;\; \langle s{\uparrow}k,rev.(s{\downarrow}k)\rangle \quad |[\; k=\#s \operatorname{div} 2 \;]| & ,\#t=1 \\ & & [\!] \;\; \langle s,tl.t\rangle & ,\#t\geq 2 \end{array}$$

$$\mathsf{rev}.\langle s,t\rangle \;=\; \langle t,s\rangle \;.$$

In the programs for tl and ft some list operations are used which are not provided by algebra S. We assume that these operations are implemented in terms of the operations of S such that $rev.s$ and $\#s$ take $O(\#s)$ time, and, $s{\uparrow}n$ and $s{\downarrow}n$ take $O(n)$ time—as usual. Furthermore, an expression like $\#s{\geq}2$ is just short for $s{\neq}\,[\,]$ cand $tl.s{\neq}[\,]$, so it takes only $O(1)$ time to evaluate it.

Now, note that all operations are $O(1)$, except that evaluation of $\mathsf{tl}.\langle s,t\rangle$ takes $O(\#t)$ time in case $\#s{=}1$ (and similarly for ft). Therefore we count $1{+}\#t$ units for such an unfolding of tl, $1{+}\#s$ time units for the corresponding unfolding of ft, and in all other cases we just count one time unit per unfolding of an operation of $\mathsf{S2}$. This defines cost measure $\mathcal{T}$. The potential function is defined as $\Phi.\langle s,t\rangle = |\#s - \#t|$, and the amortized costs of the operations are all $O(1)$, with (cf. scheme in Section 5.5):

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{isempty}](x) & = & \mathcal{T}[\mathsf{isempty}](x) \\
\mathcal{A}[\mathsf{cons}](a,x) & = & \mathcal{T}[\mathsf{cons}](a,x) + \Phi.(\mathsf{cons}.a.x) - \Phi.x \\
\mathcal{A}[\mathsf{hd}](x) & = & \mathcal{T}[\mathsf{hd}](x) \\
\mathcal{A}[\mathsf{tl}](x) & = & \mathcal{T}[\mathsf{tl}](x) + \Phi.(\mathsf{tl}.x) - \Phi.x \\
\mathcal{A}[\mathsf{snoc}](x,a) & = & \mathcal{T}[\mathsf{snoc}](x,a) + \Phi.(\mathsf{snoc}.x.a) - \Phi.x \\
\mathcal{A}[\mathsf{lt}](x) & = & \mathcal{T}[\mathsf{lt}](x) \\
\mathcal{A}[\mathsf{ft}](x) & = & \mathcal{T}[\mathsf{ft}](x) + \Phi.(\mathsf{ft}.x) - \Phi.x \\
\mathcal{A}[\mathsf{rev}](x) & = & \mathcal{T}[\mathsf{rev}](x) + \Phi.(\mathsf{rev}.x) - \Phi.x.
\end{array}$$

In the analyses in the next section, we shall use that Φ satisfies $0{\leq}\Phi.x{\leq}\#[\![x]\!]$, for all $x{\in}X$.

To give an idea of how the correctness of the above results can be proved, we end this section with a treatment of case $\#s = 1$ for $\mathsf{tl}.\langle s,t\rangle$. The correctness of the program for this case follows from $\langle rev.(t{\downarrow}k), t{\uparrow}k\rangle \in X$ (since $k = \#t \operatorname{div} 2$), and

$$\begin{array}{ll}
 & [\![\mathsf{tl}.\langle s,t\rangle]\!] \\
= & \quad \{ \text{ definition of } \mathsf{tl} \ \} \\
 & [\![\langle rev.(t{\downarrow}k), t{\uparrow}k\rangle]\!] \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & rev.(t{\downarrow}k) +\!\!+ rev.(t{\uparrow}k) \\
= & \quad \{ \text{ property of } rev;\ t{\uparrow}k +\!\!+ t{\downarrow}k = t \ \} \\
 & rev.t \\
= & \quad \{ \ \#s = 1 \ \} \\
 & tl.(s +\!\!+ rev.t) \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & tl.[\![\langle s,t\rangle]\!].
\end{array}$$

The $O(1)$ bound for the amortized cost follows from (using $\mathcal{T}[\mathsf{tl}](\langle s,t\rangle) = 1{+}\#t$):

$$\begin{array}{ll}
 & \mathcal{A}[\mathsf{tl}](\langle s,t\rangle) \\
= & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{tl}] \ \} \\
 & \mathcal{T}[\mathsf{tl}](\langle s,t\rangle) + \Phi.(\mathsf{tl}.\langle s,t\rangle) - \Phi.\langle s,t\rangle
\end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ definitions of } \mathcal{T} \text{ and } \mathsf{tl} \ \} \\
& 1 + \#t + |(\#t - k) - k| - |1 - \#t| \\
\leq & \{ \text{ arithmetic } (k = \#t \operatorname{div} 2) \ \} \\
& 1 + \#t + \#t \bmod 2 + 1 - \#t \\
\leq & \{ \text{ arithmetic } \} \\
& 3.
\end{array}
$$

5.6.2 Some applications and their analyses

In this section we perform a series of amortized analyses of more complicated list operations. Each example emphasizes a particular issue of amortized analysis. Given the choice for X in the previous section, all these operations are implemented without loss of efficiency, so there is no need to add any of these to the algebra of deques.

Catenation — *a simple amortized analysis*

For $+\!\!+$ we have as obvious refinement:

$$
\begin{array}{lcll}
\mathsf{cat}.x.y & = & y & , \ \mathsf{isempty}.x \\
& \square & \mathsf{cat}.(\mathsf{ft}.x).(\mathsf{cons}.(\mathsf{lt}.x).y) & , \neg \mathsf{isempty}.x.
\end{array}
$$

Since the costs of the operations of S2 dominate, we charge the cost of each unfolding of cat to the applications of these operations. Thus, we leave the definition of $\mathcal{T}$ unchanged and define

$$\mathcal{A}[\mathsf{cat}](x, y) = \mathcal{T}[\mathsf{cat}](x, y) + \Phi.(\mathsf{cat}.x.y) - \Phi.x - \Phi.y,$$

since $+\!\!+$ is a transformation. Then the definition of cat is linear, so we have as recurrence relation for $\mathcal{A}[\mathsf{cat}]$:

$$
\mathcal{A}[\mathsf{cat}](x, y) = \begin{cases}
\mathcal{A}[\mathsf{isempty}](x) & , [\![x]\!] = [\,] \\
\mathcal{A}[\mathsf{cat}](\mathsf{ft}.x, \mathsf{cons}.(\mathsf{lt}.x).y) + \mathcal{A}[\mathsf{ft}](x) + & \\
\mathcal{A}[\mathsf{cons}](\mathsf{lt}.x, y) + \mathcal{A}[\mathsf{lt}](x) + \mathcal{A}[\mathsf{isempty}](x) & , [\![x]\!] \neq [\,],
\end{cases}
$$

from which we infer that $\mathcal{A}[\mathsf{cat}](x, y)$ is $O(\#[\![x]\!])$.

Of course, one can also design a program for $\mathsf{cat}.x.y$ with cost $O(\#[\![y]\!])$. By choosing the cheapest program, we obtain an implementation of cat for which $\mathsf{cat}.x.y$ takes $O(\#[\![x]\!] \min \#[\![y]\!])$ time, provided that we see to it that $\#[\![x]\!]$ and $\#[\![y]\!]$ can be determined in $O(1)$ time.

Length — *introducing an auxiliary definition*

Operation # may be refined by the following function:

$$
\begin{array}{lcll}
\mathsf{len}.x & = & 0 & , \ \mathsf{isempty}.x \\
& \square & 1 + \mathsf{len}.(\mathsf{tl}.x) & , \neg \mathsf{isempty}.x.
\end{array}
$$

Since # is an inspection operation, we define:

$$\mathcal{A}[\mathsf{len}](x) = \mathcal{T}[\mathsf{len}](x).$$

However, in order to analyze len, we introduce the following auxiliary definition:

$$\mathcal{A}'[\mathsf{len}](x) = \mathcal{T}[\mathsf{len}](x) - \Phi.x,$$

for which we have the following recurrence relation:

$$\mathcal{A}'[\mathsf{len}](x) = \begin{cases} \mathcal{A}[\mathsf{isempty}](x) - \Phi.x & , [\![x]\!] = [\,] \\ \mathcal{A}'[\mathsf{len}](\mathsf{tl}.x) + \mathcal{A}[\mathsf{tl}](x) + \mathcal{A}[\mathsf{isempty}](x) & , [\![x]\!] \neq [\,]. \end{cases}$$

Since $\Phi.x \geq 0$, it follows that $\mathcal{A}'[\mathsf{len}](x)$ is $O(\#[\![x]\!])$, and, since $\Phi.x \leq \#[\![x]\!]$, we have that $\mathcal{A}[\mathsf{len}](x)$ is $O(\#[\![x]\!])$ as well.

From the above analysis we conclude that it is sometimes necessary to introduce a different definition for the amortized cost of an operation in order to perform the analysis of its program. In this case we need the term $-\Phi.x$ in the definition of $\mathcal{A}'[\mathsf{len}](x)$ to amortize the cost of the applications of tl in the recursive alternative of len. Subsequently, the bounds for Φ enabled us to derive a bound for $\mathcal{A}[\mathsf{len}](x)$.

Take and drop — *extending the definition of* $\mathcal{T}$

For $\uparrow$ ("take") and $\downarrow$ ("drop") we have as refinements

$$\begin{array}{lcl} \mathsf{take}.x.0 & = & \mathsf{empty} \\ \mathsf{take}.x.(n{+}1) & = & \mathsf{cons}.(\mathsf{hd}.x).(\mathsf{take}.(\mathsf{tl}.x).n), \end{array}$$

and

$$\begin{array}{lcl} \mathsf{drop}.x.0 & = & x \\ \mathsf{drop}.x.(n{+}1) & = & \mathsf{drop}.(\mathsf{tl}.x).n. \end{array}$$

With $\mathcal{T}$ as defined so far we have $\mathcal{T}[\mathsf{drop}](x, 0) = 0$, which is evidently not realistic. Therefore, we extend the definition of $\mathcal{T}$ as follows: we count one unit per unfolding of the nonrecursive alternative of drop. The amortized costs of these transformations are

$$\begin{array}{lcl} \mathcal{A}[\mathsf{take}](x, n) & = & \mathcal{T}[\mathsf{take}](x, n) + \Phi.(\mathsf{take}.x.n) - \Phi.x \\ \mathcal{A}[\mathsf{drop}](x, n) & = & \mathcal{T}[\mathsf{drop}](x, n) + \Phi.(\mathsf{drop}.x.n) - \Phi.x. \end{array}$$

A straightforward analysis gives $O(n)$ bounds for the amortized costs of $\mathsf{take}.x.n$ and $\mathsf{drop}.x.n$.

Element selection — *less efficient than expected?*

We have $\mathsf{elt}.x.n$ as refinement for $[\![x]\!].n$, where

$$\begin{array}{lcl} \mathsf{elt}.x.0 & = & \mathsf{hd}.x \\ \mathsf{elt}.x.(n{+}1) & = & \mathsf{elt}.(\mathsf{tl}.x).n. \end{array}$$

Operation elt is an inspection, hence

$$\mathcal{A}[\mathsf{elt}](x,n) = \mathcal{T}[\mathsf{elt}](x,n).$$

In order to amortize the cost of the applications of tl, we introduce the following auxiliary definition:

$$\mathcal{A}'[\mathsf{elt}](x,n) = \mathcal{T}[\mathsf{elt}](x,n) - \Phi.x,$$

for which we have as recurrence relation:

$$\begin{array}{lcl} \mathcal{A}'[\mathsf{elt}](x,0) & = & \mathcal{A}[\mathsf{hd}](x) - \Phi.x \\ \mathcal{A}'[\mathsf{elt}](x,n{+}1) & = & \mathcal{A}'[\mathsf{elt}](\mathsf{tl}.x,n) + \mathcal{A}[\mathsf{tl}](x). \end{array}$$

Since $\Phi.x \geq 0$, $\mathcal{A}'[\mathsf{elt}](x,n)$ is $O(n)$, but for $\mathcal{A}[\mathsf{elt}](x,n)$ we can only conclude that it is $O(\#[\![x]\!])$, using that $\Phi.x \leq \#[\![x]\!]$. Now, given the choice for X in the previous section, this bound for $\mathcal{A}[\mathsf{elt}](x,n)$ is tight; the worst case occurs when the second element of $[\![\langle s,t\rangle]\!]$ is selected and $\#s = 1$, or when the last but one element of $[\![\langle s,t\rangle]\!]$ is selected and $\#t = 1$. So, element selection for deques is not as efficient as for stacks.

Splitting lists — *an avoidable fork*

As a kind of inverse of cat we consider operation split, which splits a list into a prefix of specified length and the remaining suffix:

$$\mathsf{split}.x.n = \langle \mathsf{take}.x.n, \mathsf{drop}.x.n \rangle.$$

Since split is a transformation with two outputs, we have

$$\mathcal{A}[\mathsf{split}](x,n) = \mathcal{T}[\mathsf{split}](x,n) + \Phi.(\mathsf{split}.x.n.0) + \Phi.(\mathsf{split}.x.n.1) - \Phi.x.$$

Now recall that the definitions of $\mathcal{A}[\mathsf{take}](x,n)$ and $\mathcal{A}[\mathsf{drop}](x,n)$ both contain term $-\Phi.x$, and observe that this term occurs only once in the definition of $\mathcal{A}[\mathsf{split}](x,n)$. This gives rise to a fork in x:

$$\mathcal{A}[\mathsf{split}](x,n) = \mathcal{A}[\mathsf{take}](x,n) + \mathcal{A}[\mathsf{drop}](x,n) + \Phi.x.$$

Since $\Phi.x \leq \#[\![x]\!]$, the best we can conclude from this relation is that $\mathcal{A}[\mathsf{split}](x,n)$ is $O(\#[\![x]\!])$, using that $\mathcal{A}[\mathsf{take}](x,n)$ and $\mathcal{A}[\mathsf{drop}](x,n)$ are $O(n)$. However, by

tupling the programs for take and drop, we obtain an algorithmic refinement of split for which $\mathcal{A}[\mathsf{split}](x,n)$ can be shown to be $O(n)$:

$$\begin{array}{lcl} \mathsf{split}.x.0 & = & \langle \mathsf{empty}, x\rangle \\ \mathsf{split}.x.(n{+}1) & = & \langle \mathsf{cons}.(\mathsf{hd}.x).y, z\rangle \quad |[\ \langle y,z\rangle = \mathsf{split}.(\mathsf{tl}.x).n\ ]|. \end{array}$$

As in the previous program, x occurs twice in the right-hand side of the second alternative, but now only one transformation (viz. tl) is applied to it. Therefore,

$$\begin{array}{lcl} \mathcal{A}[\mathsf{split}](x,0) & = & \mathcal{A}[\mathsf{empty}] \\ \mathcal{A}[\mathsf{split}](x,n{+}1) & = & \mathcal{A}[\mathsf{split}](\mathsf{tl}.x,n) + \mathcal{A}[\mathsf{hd}](x) + \mathcal{A}[\mathsf{tl}](x) \\ & & +\mathcal{A}[\mathsf{cons}](\mathsf{hd}.x, \mathsf{split}.(\mathsf{tl}.x).n.0), \end{array}$$

which yields an $O(n)$ bound for the amortized costs of this program for split.

From this example we draw the important conclusion that it may be necessary to consider a more detailed refinement of a program in order to avoid forks. In this case, we were able to avoid a fork, but in the next example we encounter a situation in which we cannot.

List of all prefixes — *unavoidable forks*

Let function f, $f \in \mathrm{X} \to [\mathrm{X}]$, be defined by

$$\begin{array}{lcll} f.x & = & [x] & , \ \mathsf{isempty}.x \\ & & [] \quad x \vdash f.(\mathsf{ft}.x) & , \neg \mathsf{isempty}.x. \end{array}$$

Then $f.x$ is the list of all prefixes of x (in decreasing order of length). Since f transforms a deque into a list (more precisely, a stack) of deques, the amortized costs of f are defined as follows:

$$\mathcal{A}[f](x) = \mathcal{T}[f](x) + \Phi^*.(f.x) - \Phi.x,$$

with $\Phi^*.[\,] = 0$ and $\Phi^*.(x \vdash xs) = \Phi.x + \Phi^*.xs$. Just as in the previous section there is a fork in the recursive alternative of f, which gives rise to the term $\Phi.x$ in the following recurrence relation:

$$\mathcal{A}[f](x) = \begin{cases} \mathcal{A}[\mathsf{isempty}](x) & , [\![x]\!] = [\,] \\ \mathcal{A}[f](\mathsf{ft}.x) + \mathcal{A}[\mathsf{ft}](x) + \mathcal{A}[\mathsf{isempty}](x) + \Phi.x & , [\![x]\!] \neq [\,]. \end{cases}$$

Since $\Phi.x$ is at most $\#[\![x]\!]$, it follows that $\mathcal{A}[f](x)$ is $O((\#[\![x]\!])^2)$. Now, in contrast with the previous example, we cannot give an algorithmic refinement of f such that $\mathcal{A}[f]$ is linear; we even conjecture that it is impossible to refine f such that $\mathcal{A}[f]$ is linear[1]. This result seems somewhat disappointing when one realizes

[1] As a motivation for this conjecture, we briefly describe a failing attempt of ours: we assume that X and Φ are defined as in Hoogerwoord's refinement. To achieve linearity of $\mathcal{A}[f](x)$, we aim at linearity of $\mathcal{T}[f](x)$ and $\Phi^*.(f.x)$. This suffices because Φ is nonnegative. A simple way to establish linearity of $\Phi^*.(f.x)$ is to keep the Φ-values $O(1)$ for all elements of $f.x$. This means that the two components of all elements of $f.x$ should be roughly of equal length. Hence, $f.x$ should be a list of $\#[\![x]\!]{+}1$ "balanced pairs". From a study of this list, we conclude that it is impossible to generate it in linear time.

that $\mathcal{T}[f]$ itself is linear: with $\mathcal{A}'[f](x) = \mathcal{T}[f](x) - \Phi.x$, we have that $\mathcal{A}'[f](x)$ is $O(\#[\![x]\!])$, and, consequently, $\mathcal{T}[f](x)$ is $O(\#[\![x]\!])$ (using $\Phi.x \leq \#[\![x]\!]$). However, as we will argue in Section 5.6.3, it is not so bad that $\mathcal{A}[f]$ is quadratic.

Deque of all prefixes — *nested deques*

The following data refinement of f is also an instructive example:

$$\begin{array}{lcll} g.x & = & \mathsf{cons}.x.\mathsf{empty} & , \ \mathsf{isempty}.x \\ & \square & \mathsf{cons}.x.(g.(\mathsf{ft}.x)) & , \neg\mathsf{isempty}.x. \end{array}$$

Hence, $g.x$ is a deque of deques. In such a case we use the abstraction function in the definition of the amortized costs:

$$\mathcal{A}[g](x) = \mathcal{T}[g](x) + \Phi.(g.x) + \Phi^*.[\![g.x]\!] - \Phi.x.$$

Compared to f the additional amortized costs are linear and, therefore, we have that $\mathcal{A}[g](x)$ is also $O((\#[\![x]\!])^2)$, while $\mathcal{T}[g](x)$ is $O(\#[\![x]\!])$.

5.6.3 Comparison with doubly-linked list representation

In this section we compare the "pair of stacks" representation (Section 5.6.1) with the doubly-linked list representation (Section 4.5), which we call $S2$ and D, respectively.

First, some important differences:

(a) Algebras using $S2$ are efficient in the amortized sense only, whereas algebras using D are efficient in the worst-case sense.

(b) Algebras using D are destructive, whereas algebras using $S2$ are destructive only if the destructive implementation of stacks is used.

(c) Representation D uses about twice as much pointers as representation $S2$.

(d) Representation D supports an $O(1)$ implementation for $+\!\!+$, whereas representation $S2$ does not. Furthermore, selection of the n-th element of a list of length N takes $O(n)$ time for representation D, but it takes $O(N)$ amortized time for representation $S2$.[2]

Now, focusing on (a) and (c), we observe an interesting *trade-off* between time and space: representation $S2$ requires less space than representation D, but algebras using $S2$ are efficient in the amortized sense only whereas algebras using D are efficient in the worst-case sense. Thus, one can save space by

[2]Representation $S2$ supports a simple $O(1)$ implementation for rev. Representation D does not, but it can be easily extended to do so because of the symmetry in the l-links and r-links: by interchanging the role of these links the reverse is obtained; the present meaning of the links can be recorded by a boolean.

contenting oneself with amortized costs, which is as good as actual costs when one is interested in linear usage only.

In the previous section we have seen that the fact that S2's operations are efficient in the amortized sense only may lead to strange results for the amortized costs of programs with forks. For instance, the amortized costs of program f, which computes the list of all prefixes of its input list, are higher than its actual costs: $\mathcal{A}[f](x)$ is quadratic in $\#[\![x]\!]$ while $\mathcal{T}[f](x)$ is linear. The fact that $\mathcal{A}[f](x)$ is quadratic is however not so bad when we consider the cost of an implementation of f using doubly-linked lists. In that case, correct evaluation of f requires that the value of x is copied in each unfolding of $f.x$, which takes time quadratic in $\#[\![x]\!]$ (cf. Remark 4.2). Similarly, the amortized costs for the nonlinear program for split.$x.n$ are higher than desired, viz. $O(\#[\![x]\!])$ instead of $O(n)$. But, again, when doubly-linked lists are used, execution of this nonlinear program requires that the value of x is copied one time to supply it once to take and once to drop. Hence, evaluation of split.$x.n$ takes $O(\#[\![x]\!])$ time, which equals the bound for $\mathcal{A}[\mathsf{split}](x, n)$. So, the results for the amortized costs of these programs are not so strange after all.

Chapter 6

More on lists and trees

In many data structure designs, lists and trees play a vital role as interface between specifications in terms of sets or bags on the one hand and implementations in terms of arrays or pointers on the other. The use of lists and trees in these designs is often explained by means of pictorial descriptions, but—for our purposes—this approach has two major drawbacks.

A first problem is that correctness proofs of data structures based on pictures of lists and trees tend to be rather sloppy. We regard this as a consequence of the informal status of the pictures in such proofs, their meaning being vague or ambiguous. Moreover, pictorial descriptions are often incomplete because the pictures do not cover all cases. In this way, only an intuitive explanation of the implementation (in terms of arrays or pointers) is provided.

A more serious drawback is that the use of pictures does not support the calculational way of amortized analysis we have in mind. To be able to derive potential functions in a systematic way, we will use *patterns*—as formal counterpart of the use of pictures—to program operations of data structures. Using the appropriate patterns, the essential effect of a program on the *structure* of its arguments can be expressed formally and concisely. For operations on lists, operators like $\vdash$, $\dashv$, and $+\!\!+$ can be used to build patterns. For example, an operation may transform pattern $s +\!\!+ [a] +\!\!+ t$ into $[a] +\!\!+ s +\!\!+ t$.

To build patterns for trees, however, we do not have such an adequate set of operators. In formal definitions of operations on trees of type $\langle T \rangle$, for instance, a tree is either $\langle\,\rangle$ or of the form $\langle t, a, u \rangle$. This may be compared to a restricted view of lists in which a list is either $[\,]$ or of the form $a \vdash s$ (like lists in a functional language). To program more advanced operations on trees of type $\langle T \rangle$, we see that the counterparts of $\dashv$ and $+\!\!+$ are missing. Therefore, we will introduce alternative tree types such that, for instance, swapping the subtrees of node a can be expressed as the transformation of pattern $v \dashv \langle t, a, u \rangle$ into pattern $v \dashv \langle u, a, t \rangle$.

The goal of this chapter is to lay a foundation for the use of list and tree patterns in Part II by presenting (pointer) implementations for the relevant list and tree algebras.

6.1 Lists

In the list algebras considered thus far, an important class of operations has been ignored, viz. operations that modify a list around a specified element of the list. Typical examples are insertion after a specified element and deletion of an element, which may defined as follows for lists without duplicates:

$$\begin{aligned} ins.b.a.(s +\!\!+ [a] +\!\!+ t) &= s +\!\!+ [a, b] +\!\!+ t \\ del.a.(s +\!\!+ [a] +\!\!+ t) &= s +\!\!+ t. \end{aligned}$$

A more primitive operation of this kind is *split*, where

$$split.a.(s +\!\!+ [a] +\!\!+ t) = (s, t),$$

in terms of which *ins* and *del* can be defined without the use of parameter patterns:

$$\begin{aligned} ins.b.a.x &= s +\!\!+ [a, b] +\!\!+ t \;\; |[\ (s, t) = split.a.x\ ]| \\ del.a.x &= s +\!\!+ t \;\; |[\ (s, t) = split.a.x\ ]|. \end{aligned}$$

Note that the effect of these operations on the structure of their arguments is made explicit by the use of parameter patterns; for instance, the first definition of $del.a.s$ not only shows that a is removed from s but also that the order of the remaining elements is not altered.

Although the definitions of these operations employ patterns built from $[\cdot]$ and $+\!\!+$—instead of patterns built from $[\,]$ and $\vdash$, as is common for functional languages—we want to regard these definitions as (functional) programs. What is more, we will assume in Part II that these programs have constant time complexity. This "advanced" use of list patterns, however, requires some restrictions on the use of lists.

First of all, to be able to determine the position of an element a in list s in constant time, we will assume that

- list elements are of type $[0..N)$, for some fixed natural number N, and
- lists do not contain duplicates.

As will be shown in Section 6.1.1, these assumptions enable us to keep track of the position of elements of $[0..N)$ in the linked structures representing the lists (by means of an array of pointers with domain $[0..N)$).

Furthermore, in case an algebra provides operations for combining and splitting lists, such as $+\!\!+$ and *split*, we must guarantee that

- each element of $[0..N)$ occurs in at most one list.

This enables the use of *one* pointer array with domain $[0..N)$ for a collection of lists (see Section 6.1.2). Because of the usage of arrays, these implementations are all destructive, hence these list algebras may be used in a linear fashion only.

6.1.1 Stacks with deletion

For fixed N, $N \geq 0$, we consider the algebra of stacks with $\vdash$ replaced by $\dot{\vdash}$, and tl replaced by the above defined operation del:

$$([\,[0..N)\,] \mid [\,], (=[\,]), \dot{\vdash}, hd, del\,).$$

Operation $\dot{\vdash}$ is a restricted version of $\vdash$: $a \dot{\vdash} s$ is defined (equal to $a \vdash s$) only for $a \notin s$. Likewise, $del.a.s$ is defined only if $a \in s$. Consequently, the lists in the range of this algebra are free of duplicates.

To obtain an $O(1)$ implementation of del, we represent a list of type $[\,[0..N)\,]$ by a pair consisting of a pointer to a doubly-linked list and an array. More precisely, the concrete type is

$$D \;\times\; ([0..N) \to D),$$

where $D = {}^\wedge\langle l{:}D, n{:}[0..N), r{:}D\rangle$. The coupling is then defined by $[\,] \simeq \langle \mathrm{nil}, f\rangle$ for all f, and (cf. Section 4.5):

$$\begin{aligned} s \simeq \langle p, f\rangle \quad \equiv \quad & (\forall\, i : 0 \leq i < \#s : f[s.i]^\wedge.n = s.i) \\ & \wedge\ \#s = (\mathrm{Min}\, i : 1 \leq i : p({}^\wedge.r)^i{=}p) \\ & \wedge\ s = list.p{\uparrow}\#s \\ & \wedge\ (\forall\, i : 0 \leq i \leq \#s : p({}^\wedge.r)^i = p({}^\wedge.l)^{\#s-i}). \end{aligned}$$

for all nonempty s. That is, for $a \in s$, $f[a]$ points to the cell containing a, and both the l-links and the r-links form circular lists.

The following programs show how array f is used:

$$\begin{aligned} \mathsf{empty} \quad &= \quad \langle \mathrm{nil}, ?\rangle \\ \mathsf{isempty}.\langle p, f\rangle \quad &= \quad p = \mathrm{nil} \\ \mathsf{cons}.a.\langle p, f\rangle \quad &= \quad |[\mathrm{var}\ h{:}D;\ \mathrm{new}(h) \\ &\qquad ;\ (\ h^\wedge := \langle h, a, h\rangle \qquad\qquad , p = \mathrm{nil} \\ &\qquad\ \ [\!]\ h^\wedge := \langle p^\wedge.l, a, p^\wedge.r\rangle \\ &\qquad\qquad ;\ p^\wedge.l^\wedge.r, p^\wedge.r^\wedge.l := h, h \quad , p \neq \mathrm{nil} \\ &\qquad\ \ ) \\ &\qquad ;\ \mathrm{return}(\langle h, f[a{:=}h]\rangle) \\ &\qquad]| \\ \mathsf{hd}.\langle p, f\rangle \quad &= \quad p^\wedge.n \\ \mathsf{delete}.a.\langle p, f\rangle \quad &= \quad |[\mathrm{var}\ h{:}D;\ h := f[a] \\ &\qquad ;\ h^\wedge.l^\wedge.r, h^\wedge.r^\wedge.l := h^\wedge.r, h^\wedge.l \\ &\qquad ;\ (\mathrm{skip}\ , p \neq h\ \ [\!]\ \ p := p^\wedge.r\ , p = h) \\ &\qquad ;\ (\mathrm{skip}\ , p \neq h\ \ [\!]\ \ p := \mathrm{nil}\ , p = h) \\ &\qquad ;\ \mathrm{dispose}(h);\ \mathrm{return}(\langle p, f\rangle) \\ &\qquad]|. \end{aligned}$$

This implementation is clearly destructive. Note that $tl.s$ can be expressed as $del.(hd.s).s$ for nonempty s. Hence, tl is refined by tl, with $\mathsf{tl}.\langle p, f\rangle = \mathsf{delete}.(p^{\wedge}.n).\langle p, f\rangle$.

6.1.2 Partitions

The representation used in the previous section also allows $O(1)$ implementations of operations like $\dashv$ and lt, and even an operation like ins can be implemented in constant time (with $ins.b.a.s$ defined only if $a \in s$ and $b \notin s$). However, addition of operations that combine or split lists give rise to complications.

For example, suppose we want to use type $D \times ([0..N) \to D)$ as concrete type in a refinement of algebra

$$(\,[\,[0..N)\,]\;\mid\;[\,],(=[\,]),[\cdot],+\!\!+\,,hd,lt,del\,),$$

where use of $s +\!\!+ t$ is restricted to disjoint s and t so that the lists remain free of duplicates. To achieve $O(1)$ cost for $+\!\!+$, we have to combine two representations $\langle p, f\rangle$ and $\langle q, g\rangle$, say, in constant time. But combining f and g into one array cannot be done in constant time, unless f and g denote the same array! To support an efficient implementation of $+\!\!+$, we therefore use a *global* array f for a collection of mutually disjoint lists.

To describe implementations of algebras of this kind formally, we may consider algebras with $\lfloor\,[\,[0..N)\,]\,\rfloor$ instead of $[\,[0..N)\,]$ as data type. Accordingly, the operations on $[\,[0..N)\,]$ are then lifted to operations on $\lfloor\,[\,[0..N)\,]\,\rfloor$. For example, $s +\!\!+ t$ is lifted to the operation which transforms $B \oplus \lfloor s \rfloor \oplus \lfloor t \rfloor$ into $B \oplus \lfloor s +\!\!+ t \rfloor$, thus keeping the bag elements mutually disjoint. To represent such a bag of mutually disjoint lists (a "partition of $[0..N)$"), one array with domain $[0..N)$ can be used plus a pointer for each bag element.

In this thesis we will however not pursue such a formal approach to the implementation of partitions. As a final remark on this subject, we mention that the use of $[\cdot]$ must be restricted as well: creation of singleton $[a]$ is only allowed if a does not occur in any of the lists in use.

6.2 Binary trees

6.2.1 Top-down views

Formal definitions of binary trees are often limited to, what we will call, *top-down* views. An example is type $\langle T\rangle$, which has been introduced in Section 1.1.4. Trees of this type are either $\langle\,\rangle$ or of the form $\langle t, a, u\rangle$ with $t \in \langle T\rangle$, $a \in T$, and $u \in \langle T\rangle$. That is, type $\langle T\rangle$ is the smallest solution of

$$X: \quad X = \{\langle\,\rangle\} \;\cup\; X \times T \times X.$$

Another common top-down type of binary trees is the smallest solution of

$$X: \quad X = T \ \cup \ X \times X.$$

Trees of this type are of the form a or $\langle t, u \rangle$, hence the values reside in the leaves of the tree. (This tree type corresponds to the type of LISP objects, which are either atoms or pairs of LISP objects.) A top-down type with both internal and external nodes may be defined as the smallest solution of

$$X: \quad X = T_e \ \cup \ X \times T_i \times X,$$

where T_e corresponds to external nodes (leaves) and T_i to internal nodes. Of course there are many more top-down views around, but we will focus on type $\langle T \rangle$ in the sequel.

Although these top-down views suffice—in principle—to define any operation on binary trees, many operations cannot be described in a neat way. For instance, an operation that increments the value of the node that is last in the inorder traversal of a tree may be defined as follows in terms of type $\langle T \rangle$:

$$\begin{array}{lcll} inc.\langle t, a, \langle\,\rangle\rangle & = & \langle t, a{+}1, \langle\,\rangle\rangle & \\ inc.\langle t, a, u\rangle & = & \langle t, a, inc.u\rangle & , u \neq \langle\,\rangle. \end{array}$$

Formally, this definition is correct, but as starting-point for an efficiency analysis it has two drawbacks. One drawback is that it is not immediate from this definition that $inc.t$ leaves the structure of t intact. In general, the effect of an operation on the structure of its tree arguments is essential for its analysis. Moreover, suppose that we have a pointer representation for $\langle T \rangle$ in mind for which operation inc can be implemented by a simple $O(1)$ program. In that case, a recursive definition of inc is not a suitable starting-point for an efficiency analysis, because it does not look like an $O(1)$ program.

What we want is a description of inc that shows the relevant changes in the structure, just as the definition of list operation del shows that $del.a$ transforms pattern $s +\!\!+ [a] +\!\!+ t$ into pattern $s +\!\!+ t$. To enable such a description of inc, we will introduce a special tree type in the next section, where we will also present a pointer implementation of an algebra involving this tree type. Subsequently, we consider two tree types which are particularly suited for the definition of operations on subtrees of specified nodes.

6.2.2 A skewed view

Suppose we wish to perform a number of operations on binary trees that manipulate the rightmost path. To provide quick access to this path, we define type $\langle T \rangle_1$ as the smallest solution of

$$X: \quad X = [X \times T].$$

Just as we interpret elements of $\langle T \rangle$ as binary trees although they are just tuples, we interpret elements of $\langle T \rangle_1$ as binary trees although they are just lists:

$[\,]$ stands for the empty tree and $\langle t, a\rangle \vdash u$ for a nonempty tree with left subtree t, root a, and right subtree u. Formally, this type is "isomorphic" to type $\langle T\rangle$ because, on the one hand, we have as abstraction function from $\langle T\rangle_1$ to $\langle T\rangle$:

$$\begin{array}{lcl} [\![[\,]]\!] & = & \langle\,\rangle \\ [\![\langle t, a\rangle \vdash u]\!] & = & \langle [\![t]\!], a, [\![u]\!]\rangle, \end{array}$$

and, on the other hand, we have as representation function from $\langle T\rangle$ to $\langle T\rangle_1$:

$$\begin{array}{lcl} (\!|\langle\,\rangle|\!) & = & [\,] \\ (\!|\langle t, a, u\rangle|\!) & = & \langle (\!|t|\!), a\rangle \vdash (\!|u|\!). \end{array}$$

In addition, however, nonempty elements of $\langle T\rangle_1$ can be written in the form $u \dashv \langle t, a\rangle$, in which a is the last element of the inorder traversal of this tree. In terms of patterns of type $\langle T\rangle_1$, operation *inc* can thus be programmed as

$$inc.(u \dashv \langle t, a\rangle) = u \dashv \langle t, a{+}1\rangle.$$

From this program it is immediate that the structure of the tree remains the same. Moreover, it looks like a program with constant time complexity—which is indeed the case, as will be shown shortly.

Another nice property of type $\langle T\rangle_1$ is that its elements can be catenated: if $t \in \langle T\rangle_1$ and $u \in \langle T\rangle_1$, then also $t +\!\!+ u \in \langle T\rangle_1$. This property can, for instance, be used to generate the inorder traversal in a simple way:

$$\begin{array}{lcl} io.[\,] & = & [\,] \\ io.(u \dashv \langle t, a\rangle) & = & io.(u +\!\!+ t) \dashv a, \end{array}$$

using that $\overline{[\![t +\!\!+ u]\!]} = \overline{[\![t]\!]} +\!\!+ \overline{[\![u]\!]}$. This program is as good as tail recursive—in contrast with the definition of $\bar{\cdot}$ for type $\langle T\rangle$, which is not even linearly recursive.

To support the above-mentioned operations we implement algebra

$$\mathsf{T1} = (\,\langle T\rangle_1 \mid [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, lt, ft\,).$$

Since $\langle T\rangle_1 = [\langle T\rangle_1 \times T]$, this algebra may be considered as an instance of algebra CDQ (see Section 4.5):

$$\mathsf{CDQ} = (\,[U] \mid [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, lt, ft\,).$$

However, instantiation of the refinement of CDQ with $U := \langle T\rangle_1 \times T$ does not give a concrete refinement of T1 because type $\langle T\rangle_1$ is not concrete. To get around this problem, we have to instantiate the refinement of CDQ with $U := D \times T$, where D is the concrete type refining $[U]$. After renaming we thus obtain the following pointer type:

$$R = {}^{\wedge}\langle l{:}R, a{:}R{\times}T, r{:}R\rangle.$$

The programs for the operations of T1 can now be obtained from the programs for CDQ. This yields a destructive implementation of T1, for which all operations have constant time complexity. The price to be paid is an extra pointer per node, viz. three pointers instead of two for the standard implementation of $\langle T\rangle$ as described in Section 4.3.

In Chapter 9 we will use algebra T1 to program bottom-up skew heaps.

6.2.3 Root-path views

Another example of an operation that is often supported in constant time is removal of a specified subtree. Assuming that a occurs exactly once, removal of the subtree with root a may be defined as follows in terms of type $\langle T\rangle$ (using $\in$ to determine the location of a):

$$\begin{array}{lcll} rem.a.\langle t,b,u\rangle & = & \langle\,\rangle & ,a=b \\ & & [] \quad \langle rem.a.t,b,u\rangle & ,a\in t \\ & & [] \quad \langle t,b,rem.a.u\rangle & ,a\in u. \end{array}$$

As before with the definition of *inc*, this definition of *rem* in terms of $\langle T\rangle$ is not what we want when *rem* corresponds to a simple $O(1)$ program at pointer level.

We obtain a more appropriate definition for *rem* by defining it in terms of type $\langle T\rangle_2$, which is the smallest solution of

$$X: \quad X = [X\times T \cup T\times X].$$

Although elements of this type are actually lists, they can be interpreted as binary trees according to the following abstraction function:

$$\begin{array}{lcl} [\![[\,]]\!] & = & \langle\,\rangle \\ [\![\langle t,a\rangle \vdash u]\!] & = & \langle [\![t]\!],a,[\![u]\!]\rangle \\ [\![\langle a,u\rangle \vdash t]\!] & = & \langle [\![t]\!],a,[\![u]\!]\rangle. \end{array}$$

Since $[\![\langle t,a\rangle \vdash u]\!] = [\![\langle a,u\rangle \vdash t]\!]$, the same tree in $\langle T\rangle$ is represented by many different "trees" in $\langle T\rangle_2$. Consequently, a function definition in terms of type $\langle T\rangle_2$ does not necessarily induce a function on $\langle T\rangle$. A sufficient condition to ensure that a function f on $\langle T\rangle_2$ induces a function on $\langle T\rangle$ is that $[\![f.t]\!] = [\![f.u]\!]$ should hold whenever $[\![t]\!] = [\![u]\!]$.

In terms of type $\langle T\rangle_2$, removal of the subtree rooted in a can now be described as

$$rem.a.(v +\!\!+ \langle t,a\rangle \vdash u) = v.$$

A similar, but more primitive operation is

$$split.a.(v +\!\!+ \langle t,a\rangle \vdash u) = (t,v,u).$$

In these definitions v represents more than a tree: it also records the location where the subtree with root a resided prior to removal. Or, formally: if $split.a.x = (t,v,u)$ then $[\![x]\!] = [\![v +\!\!+ \langle t,a\rangle \vdash u]\!]$, which means that the tree represented by x can be reconstructed from a and $split.a.x$.

As counterpart of *rem* we have "catenation" of trees: for t and u in $\langle T\rangle_2$, also $t +\!\!+ u$ denotes a tree in $\langle T\rangle_2$. In terms of type $\langle T\rangle$, catenation of t and u means that an empty subtree of a node in t is replaced by u; which one is replaced is determined by t. For example, catenation of $[\langle t_0,a_0\rangle, \langle a_1,t_1\rangle, \langle a_2,t_2\rangle]$

and u corresponds to replacing the left subtree of node a_2—which is empty—by u.

Note that rem is not defined for all elements of $\langle T\rangle_2$; $rem.a.(v +\!\!+ \langle a, u\rangle \vdash t)$, for example, is not defined. This is no problem as long as a function value is defined for at least one representation of each tree in $\langle T\rangle$. For a particular function we choose the elements of $\langle T\rangle_2$ that are best suited to define it.

To define operations that manipulate specified subtrees, often a type like

$$\langle T\rangle_3 = [\langle T\rangle \times T \cup T \times \langle T\rangle] \times \langle T\rangle$$

suffices instead of the more intricate type $\langle T\rangle_2$. An element of this type represents a tree according to the following abstraction:

$$\begin{array}{lcl} [\![([\,], x)]\!] & = & x \\ [\![(\langle t, a\rangle \vdash u, x)]\!] & = & \langle t, a, [\![(u, x)]\!]\rangle \\ [\![(\langle a, u\rangle \vdash t, x)]\!] & = & \langle [\![(t, x)]\!], a, u\rangle. \end{array}$$

In terms of type $\langle T\rangle_3$, a tree x in which a occurs can be decomposed in exactly one way as $x = (v, \langle t, a, u\rangle)$. That is, this representation allows us to dissect tree x in the subtree rooted in a with left subtree t and right subtree u, and the remainder of x, called v, which encodes the *root-path* of a in x. This contrasts with type $\langle T\rangle_2$, for which these components are not uniquely determined. As programs for rem and $split$ we now have

$$\begin{array}{lcl} rem.a.(v, \langle t, a, u\rangle) & = & (v, \langle\,\rangle) \\ split.a.(v, \langle t, a, u\rangle) & = & (([\,], t), (v, \langle\,\rangle), ([\,], u)). \end{array}$$

In Part II we will often abbreviate $([\,], x)$ to x, and $(v, \langle\,\rangle)$ to v.

Pointer representations

We briefly discuss a pointer representation of $\langle T\rangle$ that supports the kind of decomposition required for patterns of type $\langle T\rangle_2$ and $\langle T\rangle_3$. The pointer type is

$$P = {}^{\wedge}\langle l{:}P, a{:}T, u{:}P, r{:}P\rangle.$$

The coupling $\simeq$ between $\langle T\rangle$ and P is defined as follows

$$x \simeq p \;\equiv\; x = tree.p \;\wedge\; DL.p \;\wedge\; p^{\wedge}.u = \text{nil},$$

where

$$\begin{array}{lcl} tree.\text{nil} & = & \langle\,\rangle \\ tree.p & = & \langle tree.(p^{\wedge}.l), p^{\wedge}.a, tree.(p^{\wedge}.r)\rangle \quad , p \neq \text{nil}, \end{array}$$

and $DL.p$ expresses that p points to a "doubly-linked tree":

$$\begin{array}{lcl} DL.\text{nil} & \equiv & \text{true} \\ DL.p & \equiv & (p^{\wedge}.l \neq \text{nil} \;\Rightarrow\; p^{\wedge}.l^{\wedge}.u = p) \;\wedge\; DL.(p^{\wedge}.l) \\ & & \quad \wedge\; (p^{\wedge}.r \neq \text{nil} \;\Rightarrow\; p^{\wedge}.r^{\wedge}.u = p) \;\wedge\; DL.(p^{\wedge}.r). \end{array}$$

As remarked before, an element v of $[\langle T\rangle \times T \cup T \times \langle T\rangle]$ represents more than a tree in $\langle T\rangle$. To represent v we use in addition to a pointer p to the root of v, a pointer q pointing to the last node of v (if $v = [\,]$, then $q =$ nil). In this way, four pointers are used to represent triple (t, v, u), which is the result of *split.a.x*: two for v, one for t, and one for u. To obtain these pointers in constant time, we assume that type T is of the form $[0..N)$, and that trees are free of duplicates. As in Section 6.1, this enables us to use a global array f of type $[0..N) \to P$ such that, given a, the pointers to the roots of t and u are $f[a]^{\wedge}.l$ and $f[a]^{\wedge}.r$, respectively, and $f[a]^{\wedge}.u$ points to the last node of v. Hence, given a and a pointer p to the representation of x, the representation of *split.a.x* can be obtained from array f in constant time.

6.3 Trees and forests

Instead of binary trees in which each node has two subtrees, many data structure designs employ a view of trees in which each node has an unbounded (yet finite) number of subtrees. In the literature, a collection of trees is called a *forest*; in this terminology, a *tree* consists of a root node and a forest of subtrees. In the next section, we present some common types of trees and forests. Subsequently, we generalize the root-path views of binary trees defined in Section 6.2.3 to root-path views of trees and forests.

6.3.1 Top-down views

There are many top-down views of trees and forests. An important distinction between these views is whether the order of the subtrees is taken into account or not. In formal definitions of tree types, this corresponds to either a list or a bag/set of subtrees. Two such tree types are $\langle T\rangle_4$ and $\langle T\rangle_5$, defined as smallest solutions of equations

$$X: \quad X = T \times [X],$$

and

$$X: \quad X = T \times \text{(}X\text{)},$$

respectively. An element $\langle a, t\rangle$ of $\langle T\rangle_4$, with $a \in T$ and $t \in [\langle T\rangle_4]$, stands for a nonempty tree with root a and the trees in *list* t as subtrees. Similarly, an element $\langle a, t\rangle$ of $\langle T\rangle_5$ denotes a nonempty tree with root a and the trees in *bag* t as subtrees. Elements of type $[\langle T\rangle_4]$ and $\text{(}\langle T\rangle_5\text{)}$ are forests.

In many data structures, trees are required to be free of duplicates, and also trees in a forest are required to be disjoint. Instead of type $\langle T\rangle_5$, a type like $\langle T\rangle_6$ may then be used, which is defined as the smallest solution of

$$X: \quad X = T \times \{X\}.$$

The advantage over type $\langle T\rangle_5$ is that standardized set notations can be used to define operations on trees of type $\langle T\rangle_6$.

6.3.2 Root-path views

In a similar way to that in Section 6.2.3, we now introduce root-path views of types $\langle T\rangle_4$ and $\langle T\rangle_6$ to facilitate the definition of operations that modify specified subtrees.

We first consider the simpler type $\langle T\rangle_6$. To program the removal of a specified subtree for this type, it suffices to introduce type

$$\langle T\rangle_7 = [\langle T\rangle_6].$$

A *nonempty* list of this type represents a tree in $\langle T\rangle_6$ according to the following abstraction:

$$\begin{array}{lcl} [\![[x]]\!] & = & x \\ [\![\langle a,t\rangle \vdash v]\!] & = & \langle a, t \cup \{[\![v]\!]\}\rangle \quad , v \neq [\,]. \end{array}$$

(The empty list (of type $\langle T\rangle_7$) may be viewed as an empty tree.) In terms of this type, a tree in which a occurs exactly once can be written in the form $v \dashv \langle a,t\rangle$ in precisely one way. The program for removal is thus:

$$rem.a.(v \dashv \langle a,t\rangle) = v,$$

in which v corresponds to the root-path of a.

Next we consider type $\langle T\rangle_4$. In this case, a more intricate type is required to define *rem* because the order of the subtrees must be taken into account. The following type is appropriate:

$$\langle T\rangle_8 = [\,[\langle T\rangle_4] \times T \times [\langle T\rangle_4]\,] \times \langle T\rangle_4.$$

The abstraction function is

$$\begin{array}{lcl} [\![([\,],x)]\!] & = & x \\ [\![(\langle t,a,u\rangle \vdash v,x)]\!] & = & \langle a, t +\!\!+ [[\![(v,x)]\!]] +\!\!+ u\rangle. \end{array}$$

Hence, (v,x) represents the $(\#t+1)$-st subtree of a, located between those subtrees in t and those in u. In terms of type $\langle T\rangle_8$, a tree in which a occurs exactly once can be written in the form $(v,\langle a,t\rangle)$ in precisely one way, which leads to the following program for removal:

$$rem.a.(v,\langle a,t\rangle) = (v,\langle\,\rangle).$$

Again, v encodes the root-path of a.

Pointer representations

Pointer representations for types like $\langle T\rangle_4$ and $\langle T\rangle_6$ are in essence much the same as those for binary tree types. An appropriate pointer type for $\langle T\rangle_4$ that also supports root-path view $\langle T\rangle_8$ is, for instance (cf. type P in Section 6.2.3):

$$Q = {}^\wedge\langle a{:}T, u{:}Q, s{:}[Q]\rangle.$$

For a pointer p of this type, $p^\wedge.u$ provides access to the "parent" node and $p^\wedge.s$ represents the forest of subtrees of node a. To support efficient deletion of arbitrary subtrees, a doubly-linked list representation can be used for type $[Q]$. This amounts to four pointers per node.

In case $T = [0..N)$ and access to subtrees must be supported in constant time, an array of type $[0..N) \to Q$ is required in addition (see Section 6.2.3). Then a total of five pointers per node is used.

6.4 Overview of tree types

Recall that $(\mu x : P : E)$ denotes the smallest solution of $x : \ P \wedge x = E$. Throughout this chapter we have defined the following types of binary trees:

$$\begin{aligned}
\langle T\rangle &= (\mu X : \langle\,\rangle \in X : X \times T \times X)\\
\langle T\rangle_1 &= (\mu X :: [X \times T])\\
\langle T\rangle_2 &= (\mu X :: [X \times T \cup T \times X])\\
\langle T\rangle_3 &= [\langle T\rangle \times T \cup T \times \langle T\rangle] \times \langle T\rangle.
\end{aligned}$$

Each binary tree of type $\langle T\rangle$ is represented by exactly one binary tree in $\langle T\rangle_1$ but by many different "binary trees" in $\langle T\rangle_2$ and $\langle T\rangle_3$.

Furthermore, we have defined the following types of nonempty trees:

$$\begin{aligned}
\langle T\rangle_4 &= (\mu X :: T \times [X])\\
\langle T\rangle_5 &= (\mu X :: T \times (X))\\
\langle T\rangle_6 &= (\mu X :: T \times \{X\})\\
\langle T\rangle_7 &= [\langle T\rangle_6]\\
\langle T\rangle_8 &= [\,[\langle T\rangle_4] \times T \times [\langle T\rangle_4]\,] \times \langle T\rangle_4,
\end{aligned}$$

where $\langle T\rangle_4$ and $\langle T\rangle_8$ as well as $\langle T\rangle_6$ and $\langle T\rangle_7 \setminus \{[\,]\}$ are isomorphic.

In Part II we will use these tree types without bothering about their pointer implementations. We claim, however, that all basic tree operations used in Part II can be supported in constant time, using the techniques of this chapter.

Part II

Case Studies

Chapter 7

Tricky representations of sets and arrays

As remarked at the end of Chapter 3, there are amortized analyses in which a potential function that is bounded from below by a constant cannot be used. In this chapter we provide a simple example of such a case (Section 7.4). Furthermore, we show in this chapter how arrays can be combined with a functional programming style. An important aspect of this combination is that the algebra of arrays (as defined in Example 1.7) is destructive, hence that arrays must be restricted to linear usage. This restriction causes no problems, but the implemented algebras will be destructive too, so these algebras must be restricted to linear usage as well.

The material in this chapter is inspired by the problem stated in Exercise 2.12 from [1]:

> Develop a technique to initialize an entry of a matrix to zero the first time it is accessed, thereby eliminating the $O((\#V)^2)$ time to initialize an adjacency matrix. [*Hint:* Maintain a pointer in each initialized entry to a back pointer on a stack. Each time an entry is accessed, verify that the contents are not random by making sure the pointer in that entry points to the active region on the stack and that the back pointer points to the entry.]

Although the practical significance of this "trick" for avoiding initialization of large arrays may be small, it has been applied many times throughout the computing literature ([19] and [14] are just two arbitrary papers that refer to Exercise 2.12). Apparently, Mehlhorn recognized that this trick deserves more than the status of an exercise, for he presents a solution in his book on data structures and algorithms [23, pp. 289–290]. In this chapter we present two more solutions and compare them with Mehlhorn's solution.

At first we interpreted Exercise 2.12 as a problem of set representation. Realizing that the adjacency matrix in the exercise represents the *set* of arcs of a directed graph—whose vertices are numbered consecutively—we considered the

problem of implementing an algebra with data type $\{[0..N)\}$, for a fixed natural number N. Sets of this type can be generated from the empty set by means of operation $\oplus$, a restricted version of insertion (as defined in Section 1.1.4). An element can be deleted from such a set by means of operation $\ominus$. To solve the exercise we implement the following algebra of "bounded sets":

$$\mathsf{BS} = (\, \{[0..N)\} \mid \{\,\}, \#, \in, \oplus, \ominus \,),$$

such that each operation is supported in $O(1)$ time. Operation # has been included instead of $(=\{\,\})$ because it can be implemented at no extra cost in the implementation of BS in Section 7.1.

On second thoughts, however, another way to solve the exercise is to implement the following algebra of arrays efficiently:

$$\mathsf{AK} = (\, [0..N) \to T \mid K, \mathit{lookup}, \mathit{update} \,).$$

The difference with the algebra of arrays defined in Example 1.7 is that an arbitrary (but fixed) function $K \in [0..N) \to T$ is provided instead of relation ?. Function K serves as initial array value: for example, $K.i = \text{false}$ is often an appropriate initial value when $T = \text{Bool}$. The problem is to support each operation of AK in constant time. In Section 7.2 we show how this can be done.

7.1 A tricky representation of sets

The problem is to implement BS by a concrete algebra with signature

$$\mathsf{C} = (\, C \mid \mathsf{empty}, \mathsf{size}, \mathsf{member}, \mathsf{insert}, \mathsf{delete} \,),$$

say, such that all operations of C have $O(1)$ time complexity. Although algebra BS is bijective—which enables us to use abstraction functions to describe its refinements—we will use a coupling $\simeq$, say, because this shortens the descriptions somewhat.

As stepping-stone towards a solution for C, we first present two well-known refinements of BS. We do not provide correctness proofs for the operations; this amounts to instantiation of Definition 2.12, where it should be noted that $\oplus$ and $\ominus$ are partial functions: $S \oplus i$ and $S \ominus i$ are defined only if $i \notin S$ and $i \in S$, respectively.

Bit-vector

A standard solution is to represent a subset of $[0..N)$ by an array of type B, where

$$B = [0..N) \to \text{Bool}.$$

In order to obtain an $O(1)$ program for operation size, we take $C = B \times [0..N]$, and define the coupling relation by

$$S \simeq \langle b, n\rangle \;\equiv\; S = \{i \mid b[i]\} \;\wedge\; \#S = n.$$

The operations are implemented as follows:

$$\begin{array}{lcl}
\mathsf{empty} & = & \langle g.?.0, 0\rangle \\
 & & |[\ g.b.i = (g.b[i{:=}\mathrm{false}].(i{+}1)\ , i \neq N\ \ []\ \ b\ , i = N)\ ]| \\
\mathsf{size}.\langle b, n\rangle & = & n \\
\mathsf{member}.i.\langle b, n\rangle & = & b[i] \\
\mathsf{insert}.\langle b, n\rangle.i & = & \langle b[i{:=}\mathrm{true}], n{+}1\rangle \\
\mathsf{delete}.\langle b, n\rangle.i & = & \langle b[i{:=}\mathrm{false}], n{-}1\rangle.
\end{array}$$

The only problem with this refinement is that operation empty takes $O(N)$ time.

Enumeration

Another well-known way to represent a set is by enumeration of its elements. For this purpose lists are adequate, but we use an array type instead to prepare for the next refinement. We introduce array type

$$E = [0..N) \to [0..N),$$

and we take $C = E \times [0..N]$. The coupling relation is defined by

$$S \simeq \langle e, n\rangle \;\equiv\; S = \{e[i] \mid 0 \leq i < n\} \;\wedge\; \#S = n,$$

and the operations are defined as

$$\begin{array}{lcl}
\mathsf{empty} & = & \langle ?, 0\rangle \\
\mathsf{size}.\langle e, n\rangle & = & n \\
\mathsf{member}.i.\langle e, n\rangle & = & g.0 \neq n \\
 & & |[\ g.j \;=\; (j\ , i = e[j]\ \ []\ \ g.(j{+}1)\ , i \neq e[j]) \quad , j \neq n \\
 & & \qquad\quad [] \quad n \qquad\qquad\qquad\qquad\qquad\quad , j = n \\
 & &]| \\
\mathsf{insert}.\langle e, n\rangle.i & = & \langle e[n{:=}i], n{+}1\rangle \\
\mathsf{delete}.\langle e, n\rangle.i & = & \langle e[g.0{:=}e[n{-}1]], n{-}1\rangle \\
 & & |[\ g.j = (j\ , i = e[j]\ \ []\ \ g.(j{+}1)\ , i \neq e[j])\ ]|.
\end{array}$$

This time, we see that empty takes $O(1)$ time, but member and delete take $O(N)$ time in the worst case.

Tricky representation

From the previous refinement, we now construct an efficient solution for C. We first concentrate on an $O(1)$ program for member. In the previous refinement we have that $\mathsf{member}.i.\langle e, n\rangle$ is equivalent to $(\exists\, j : 0 \leq j < n : i = e[j])$. To

determine the value of this predicate in constant time, we observe that segment $e[0..n)$ represents an injective function from $[0..n)$ to $[0..N)$. So, we can introduce an additional array f, say, of type E as "inverse" of $e[0..n)$ such that predicate $(\exists\, j : 0 \le j < n : i = e[j])$ is equivalent to $0 \le f[i] < n \ \wedge\ i = e[f[i]]$; this may be paraphrased as "$f[i]$ equals the location of i in $e[0..n)$ (if present)".

Thus, we take $C = E \times [0..N] \times E$, and define the coupling relation by

$$\begin{array}{lcl} S \simeq \langle e, n, f\rangle & \equiv & S = \{e[i] \mid 0 \le i < n\} \ \wedge\ \#S = n \\ & & \wedge\ (\forall\, i : 0 \le i < n : f[e[i]] = i). \end{array}$$

In this definition the conjunct $\#S = n$ is actually redundant, but its presence reminds us that this coupling is an extension of the previous one. To prove that $i \in S$ is indeed equivalent to $0 \le f[i] < n \ \wedge\ i = e[f[i]]$ when $S \simeq \langle e, n, f\rangle$, we observe:

$$\begin{array}{ll} & i \in S \\ \equiv & \quad \{\ \text{definition of } S\ \} \\ & (\exists\, j : 0 \le j < n : i = e[j]) \\ \equiv & \quad \{\ \text{condition } (\forall\, i : 0 \le i < n : f[e[i]] = i)\ \} \\ & (\exists\, j : 0 \le j < n : i = e[j] \ \wedge\ f[e[j]] = j) \\ \equiv & \quad \{\ f \text{ is a function}\ \} \\ & (\exists\, j : f[i] = j : 0 \le j < n \ \wedge\ i = e[j]) \\ \equiv & \quad \{\ \text{one-point rule}\ \} \\ & 0 \le f[i] < n \ \wedge\ i = e[f[i]]. \end{array}$$

The operations of BS are now refined by the following $O(1)$ programs, which should be compared with those in the previous refinement:

$$\begin{array}{lcl} \mathsf{empty} & = & \langle ?, 0, ?\rangle \\ \mathsf{size}.\langle e, n, f\rangle & = & n \\ \mathsf{member}.i.\langle e, n, f\rangle & = & 0 \le f[i] < n \ \wedge\ i = e[f[i]] \\ \mathsf{insert}.\langle e, n, f\rangle.i & = & \langle e[n{:=}i], n{+}1, f[i{:=}n]\rangle \\ \mathsf{delete}.\langle e, n, f\rangle.i & = & \langle e[f[i]{:=}e[n{-}1]], n{-}1, f[e[n{-}1]{:=}f[i]]\rangle. \end{array}$$

In the program for member we have exploited the fact that f is of type E: this means that $0 \le f[i] < N$, hence $f[i]$ is in the domain of e.

The tricky thing about this implementation is that member always returns the desired value, despite the fact that evaluation of $\mathsf{member}.i.\langle e, n, f\rangle$ may give rise to inspection of "uninitialized" elements of f (and e).

7.2 A tricky representation of arrays

Starting from the tricky representation of BS, we now design a similar implementation of AK, for which we use as signature:

$$\mathsf{D} = (\ D \mid \mathsf{K}, \mathsf{lookup}, \mathsf{update}\ ).$$

The idea is to represent an array of type $[0..N) \to T$ by a quadruple $\langle c, e, m, f\rangle$ in which c is the actual representation of the array and $\langle e, m, f\rangle$ represents the set of "initialized" elements of c, as in the tricky representation of sets in the previous section. To implement this idea we define data type D as

$$D = ([0..N) \to T) \times E \times [0..N] \times E,$$

and coupling $\simeq$ as

$$\begin{array}{lcl} a \simeq \langle c, e, m, f\rangle & \equiv & (\forall\, i : i \in S : a.i = c[i]) \ \wedge\ (\forall\, i : i \notin S : a.i = K.i) \\ & & \wedge\ \#S = m \ \wedge\ (\forall\, i : 0 \le i < m : f[e[i]] = i), \end{array}$$

where $S = \{e[i] \mid 0 \le i < m\}$. The following programs are now easily found (note that P is equivalent to $i \in S$):

$$\mathsf{K} = \langle ?, ?, 0, ?\rangle$$

$$\begin{array}{lcl} \mathsf{lookup}.\langle c, e, m, f\rangle.i & = & c[i] \quad , P \\ & & [] \quad K.i \quad , \neg P \\ & & \quad |[\ P = 0 \le f[i] < m \ \wedge\ i = e[f[i]]\ ]| \end{array}$$

$$\begin{array}{lcl} \mathsf{update}.\langle c, e, m, f\rangle.i.x & = & \langle c[i{:=}x], e, m, f\rangle \quad , P \\ & & [] \quad \langle c[i{:=}x], e[m{:=}i], m{+}1, f[i{:=}m]\rangle \quad , \neg P \\ & & \quad |[\ P = 0 \le f[i] < m \ \wedge\ i = e[f[i]]\ ]|. \end{array}$$

In order that the program for lookup be concrete, a concrete program for K should be available. Provided this program for K has $O(1)$ time complexity, all operations of D have $O(1)$ time complexity as well. For example, if $T = \mathrm{Nat}$, K could be defined as $K.i = i^2$. As for space utilization we remark that this implementation of arrays has an overhead of $O(N \log N)$ bits, and whether or not this overhead is negligible depends on T.

7.3 Mehlhorn's tricky representation of sets

On pages 289–290 in [23], we encountered a somewhat different solution than ours. On second reading of the hint in Exercise 2.12 we discovered that we did not follow it as closely as Mehlhorn did. Mehlhorn's solution is more of a mixture of the bit-vector refinement and enumeration. Indeed, his solution arises when we use algebra D to implement array b in the bit-vector refinement.

To describe his implementation of bounded sets (algebra BS), we take as concrete type $C = B \times [0..N] \times E \times [0..N] \times E$ and as coupling relation:

$$\begin{array}{lcl} S \simeq \langle b, n, e, m, f\rangle & \equiv & S = \{i \mid b[i] \ \wedge\ 0 \le f[i] < m \ \wedge\ i = e[f[i]]\} \\ & & \wedge\ \#S = n \ \wedge\ \#\{e[i] \mid 0 \le i < m\} = m. \end{array}$$

The operations are implemented as follows:

$$
\begin{array}{lcl}
\mathsf{empty} & = & \langle ?, 0, ?, 0, ? \rangle \\
\mathsf{size}.\langle b, n, e, m, f \rangle & = & n \\
\mathsf{member}.i.\langle b, n, e, m, f \rangle & = & b[i] \ \wedge \ 0 \leq f[i] < m \ \wedge \ i = e[f[i]] \\
\mathsf{insert}.\langle b, n, e, m, f \rangle.i & = & \langle b[i{:=}\mathrm{true}], n{+}1, e[m{:=}i], m{+}1, f[i{:=}m] \rangle \ , \neg P \\
 & [] & \langle b[i{:=}\mathrm{true}], n{+}1, e, m, f \rangle \qquad\qquad\qquad\qquad\quad , P \\
 & & |[\ P = 0 \leq f[i] < m \ \wedge \ i = e[f[i]] \ ]| \\
\mathsf{delete}.\langle b, n, e, m, f \rangle.i & = & \langle b[i{:=}\mathrm{false}], n{-}1, e, m, f \rangle .
\end{array}
$$

Note that m is never decreased in these programs and that $m{=}N$ implies P. Hence, when m has reached its maximal value N, Mehlhorn's solution reduces to the bit-vector implementation. Unfortunately, this is of no use because refinement C is only required in case the total number of set operations is considerably less than $O(N)$ (see next section). Hence, in typical applications m will not reach value N.

7.4 An unboundedly small potential

Let us compare the bit-vector refinement with the tricky representations of sets by considering program $f \in [\,[0..N)\,] \rightarrow C$, which converts a list into (a representation of) a set:

$$
\begin{array}{lcl}
f.[\,] & = & \mathsf{empty} \\
f.(i \vdash s) & = & \mathsf{insert}.c.i \quad , \neg \mathsf{member}.c.i \\
 & [] & c \qquad\qquad\quad , \mathsf{member}.c.i \\
 & & |[\ c = f.s \ ]| .
\end{array}
$$

Note that algebra C is used in a linear fashion, as required. Using a tricky representation of sets, computation of $f.s$ clearly takes $O(\#s)$ time. For the bit-vector refinement this computation takes $O(N + \#s)$ time, which is $O(\#s)$ when $\#s$ is $\Omega(N)$, and $O(N)$ otherwise. Phrased differently, when a sufficient number of set operations has been performed, the $O(N)$ cost of empty is negligible, but otherwise it is better to use a tricky representation of sets.

It is interesting to perform an amortized analysis of program f in case the bit-vector refinement is used. A realistic cost measure $\mathcal{T}$ for this implementation of bounded sets is defined by placing the dots as follows:

$$
\begin{array}{lcl}
\mathsf{empty} & = & \langle g.?.0, 0 \rangle \\
 & & |[\ g.b.i \doteq (g.b[i{:=}\mathrm{false}].(i{+}1) \ , i \neq N \ \ [] \ \ b \ , i = N) \ ]| \\
\mathsf{size}.\langle b, n \rangle & \doteq & n \\
\mathsf{member}.i.\langle b, n \rangle & \doteq & b[i] \\
\mathsf{insert}.\langle b, n \rangle.i & \doteq & \langle b[i{:=}\mathrm{true}], n{+}1 \rangle \\
\mathsf{delete}.\langle b, n \rangle.i & \doteq & \langle b[i{:=}\mathrm{false}], n{-}1 \rangle .
\end{array}
$$

Then $\mathcal{T}[\mathsf{empty}] = N + 1$. We introduce amortized costs for algebra C according to the general scheme of Section 5.5:

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{size}](c) & = & \mathcal{T}[\mathsf{size}](c) \\
\mathcal{A}[\mathsf{member}](i, c) & = & \mathcal{T}[\mathsf{member}](i, c) \\
\mathcal{A}[\mathsf{insert}](c, i) & = & \mathcal{T}[\mathsf{insert}](c, i) + \Phi.(\mathsf{insert}.c.i) - \Phi.c \\
\mathcal{A}[\mathsf{delete}](c, i) & = & \mathcal{T}[\mathsf{delete}](c, i) + \Phi.(\mathsf{delete}.c.i) - \Phi.c,
\end{array}$$

where $c \in B \times [0..N]$. Now, to obtain constant amortized costs for empty we have to choose Φ such that $N+1+\Phi.\mathsf{empty}$ is $O(1)$, and to obtain constant amortized costs for the remaining operations it suffices to take a constant function for Φ. Therefore, we take $\Phi.c = -N$, as a consequence of which the amortized costs of each operation of C is exactly one. To determine $\mathcal{T}[f]$, we now define

$$\mathcal{A}[f](s) = \mathcal{T}[f](s) + \Phi.(f.s).$$

Then $\mathcal{A}[f]([\,]) = 1$ and $\mathcal{A}[f](i \vdash s) \leq 2 + \mathcal{A}[f](s)$, and

$$\mathcal{T}[f](s) = \mathcal{A}[f](s) - \Phi.(f.s) \leq 1 + 2\#s + N.$$

This formally proves that the computation of $f.s$ costs $O(N + \#s)$.

So, in this analysis a negative potential function naturally arises. For fixed N this potential is however bounded from below by $-N$, which enables us to apply a trick to change this into a nonnegative potential: view empty as a transformation from $\{\bot\}$ to C, and define $\Phi.\bot = N$ and $\Phi.c = 0$ for $c \in C$. Furthermore, define

$$\mathcal{A}[\mathsf{empty}] = \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} - \Phi.\bot.$$

Then the potential is nonnegative and the amortized costs are all $O(1)$. (A drawback of this trick of defining $\Phi.\bot$ unequal to 0 is that the analysis of f must be adapted too: its amortized costs must be defined as $\mathcal{A}[f](s) = \mathcal{T}[f](s) + \Phi.(f.s) - \Phi.\bot$, which gives the same bound for $\mathcal{T}[f](s)$.)

By "promoting" N to a parameter of algebra BS, however, this trick does not work anymore. By this we mean that data type $\{[0..N)\}$ is changed into

$$\{(n, \{[0..n)\}) \mid n \in \mathrm{Nat}\}.$$

Then empty gets a natural number as parameter such that $\mathsf{empty}.n$ creates a representation for the empty set of type $\{[0..n)\}$. The appropriate potential is now $\Phi.(n, c) = -n$, and since n ranges over all natural numbers, this potential is not bounded from below. From this we conclude that the desire to achieve $O(1)$ amortized costs for this modified algebra, a negative potential is really needed.

7.5 Comparisons

Algebras BS and AK are strongly related in the sense that an efficient implementation for BS is easily designed given an efficient implementation for AK, and vice versa. Comparing the two tricky representations of sets, we see, on the one hand, that our programs for member and insert are simpler than Mehlhorn's programs, and, on the other hand, that his program for delete is simpler than ours. A possible drawback of Mehlhorn's solution is that it requires a boolean array.

In summary, the results for the respective refinements of BS are:

	bit-vector	enumeration	tricky representation	Mehlhorn's solution
$\{\}$	$O(N)$	$O(1)$	$O(1)$	$O(1)$
$\#S$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$i \in S$	$O(1)$	$O(\#S)$	$O(1)$	$O(1)$
$S \oplus i$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$S \ominus i$	$O(1)$	$O(\#S)$	$O(1)$	$O(1)$
space (bits)	N	$N\log_2 N$	$2N\log_2 N$	$2N\log_2 N + N$

This table suggests that the second refinement is hardly useful. However, compared to the bit-vector refinement, enumeration has the advantage that the elements of set S can be enumerated in $O(\#S)$ time, while this takes $O(N)$ time in the bit-vector refinement, regardless of the size of the enumerated set S.

From the bottom line of this table, we conclude that the price to be paid for the $O(1)$ initialization of bounded sets is the increase in space utilization from $O(N)$ to $O(N \log N)$ bits. As explained in Section 7.4, the $O(N)$ initialization should be avoided only when the total number of applications of the other operations in the rest of the program is significantly smaller than $O(N)$, e.g. $O(\sqrt{N})$. An example of such a program is a simple recognition algorithm for so-called series-parallel graphs. The first part of this algorithm computes the adjacency matrix for the input graph, which normally takes $O((\#V)^2)$ time (cf. Exercise 2.12 at the beginning of this chapter). The number of accesses to this matrix in the second part of the algorithm is however linear in $\#V$. Therefore it is worth while to use the tricky representation for the arc set of the input graph, since it is given that the number of arcs is $O(\#V)$, thereby reducing the time complexity of the entire algorithm to $O(\#V)$.

In practice, however, the implicit constant in the $O(N)$ bound for the initialization of arrays is very small, and, consequently, a tricky representation of sets is not worth the trouble for small values of N.

Chapter 8

Maintaining the minimum of a list

In this chapter we show how inspection operation $\downarrow$ can be added to the list algebras from Chapter 4 in such a way that each operation still has constant—possibly amortized—time complexity. Using essentially the same method, we obtain a destructive solution for concatenable queues and purely-functional solutions for stacks, queues, and deques. An efficient implementation of concatenable deques is however beyond the scope of the method described in this chapter. We have found similar implementations of these list algebras in [10].

Throughout this chapter, T denotes a linearly ordered type. In Chapter 1, $\downarrow s$ has been defined for nonempty s only. Here we extend it with $\downarrow[\,] = \infty$, where ∞ is larger than any element of T. As announced in Example 2.26, we shall also discuss noninjective algebras which arise when inspections hd and lt are omitted from the above-mentioned list algebras. This give rise to simpler and more efficient implementations, as will be shown in Section 8.4.

8.1 Stacks

In this section we design an efficient purely-functional implementation for the algebra of *minstacks*:

$$\mathsf{MS} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl, \downarrow).$$

Since the algebra of stacks is the basic list algebra for functional languages (cf. Chapter 4), this shows that inspection $\downarrow$ can be provided efficiently for any functional program.

Informally, the problem is to maintain the minimum of a list under the transformations $\vdash$ and tl. Since $\downarrow(a \vdash s) = a \min \downarrow s$, this can easily be done for $\vdash$. However, $\downarrow(tl.s)$ cannot be determined from $\downarrow s$ in each case: if $hd.s = \downarrow s$, there is no relation between the minima of $tl.s$ and s. A moment's reflection leads to the conclusion that besides the minimum of the stack also the minimum of

the tail part after the minimum must be maintained. Repeating this observation leads to the following definitions.

As concrete data type we use a subset of type $[[T] \times T]$. Lists of this type represent lists in $[T]$ according to abstraction $[\![\cdot]\!]$:

$$\begin{array}{lcl} [\![\,[\,]\,]\!] & = & [\,] \\ [\![(s,a) \vdash x]\!] & = & s +\!\!+ [a] +\!\!+ [\![x]\!]. \end{array}$$

For a list $x = [(s_0, a_0), (s_1, a_1), \ldots]$ of this type, we call x the *outer list* and the s_i's *inner lists*. List x is a kind of "list partition" of $[\![x]\!]$.

To expedite the retrieval of the minimum value, these partitions are required to satisfy condition P defined by

$$\begin{array}{lcl} P.[\,] & \equiv & \text{true} \\ P.((s,a) \vdash x) & \equiv & a \leq \downarrow s \ \wedge\ a < \downarrow [\![x]\!] \ \wedge\ P.x. \end{array}$$

The concrete type thus consists of those lists in $[[T] \times T]$ satisfying P. The corresponding programs of $O(1)$ time complexity are:

$$\begin{array}{lcll} \mathsf{empty} & = & [\,] & \\ \mathsf{isempty}.x & = & x = [\,] & \\ \mathsf{cons}.b.x & = & ([\,], b) \vdash x & , b < \mathsf{min}.x \\ \mathsf{cons}.b.((s,a) \vdash x) & = & (b \vdash s, a) \vdash x & , b \geq a \\ \mathsf{hd}.((s,a) \vdash x) & = & a & , s = [\,] \\ & \square & hd.s & , s \neq [\,] \\ \mathsf{tl}.((s,a) \vdash x) & = & x & , s = [\,] \\ & \square & (tl.s, a) \vdash x & , s \neq [\,] \\ \mathsf{min}.[\,] & = & \infty & \\ \mathsf{min}.((s,a) \vdash x) & = & a. & \end{array}$$

Since the outer list as well as the inner lists are used as stacks, these programs are purely-functional. In the worst case (when all inner lists are empty), about $2\#x$ pointers are required for the representation of $[\![x]\!]$ (instead of $\#x$ for an ordinary stack, cf. Section 4.3).

Note that each list of type $[T]$ can be partioned in only one way, as we have the following representation function, which maps $[T]$ onto $[[T] \times T]$:

$$\begin{array}{lcl} (\!|\,[\,]\,|\!) & = & [\,] \\ (\!|s +\!\!+ [a] +\!\!+ t|\!) & = & (s,a) \vdash (\!|t|\!) \quad , a \leq \downarrow s \ \wedge\ a < \downarrow t\,. \end{array}$$

By writing $a \leq \downarrow s \ \wedge\ a < \downarrow t$ instead of $a < \downarrow s \ \wedge\ a \leq \downarrow t$, a is the rightmost occurrence of the minimum of $s +\!\!+ [a] +\!\!+ t$ instead of the leftmost occurrence; in this way, the length of $(\!|s|\!)$ is minimized. Computation of $(\!|s|\!)$ is easily done in linear time, using one application of empty and $\#s$ applications of cons.

Remark 8.1
The above partioning may be applied to the inner lists as well. Instead of $[[T] \times T]$ we then get the smallest solution of

$$X : X = [X \times T]$$

as concrete type. This tree type has already been introduced in Chapter 6, where it was called $\langle T \rangle_1$. The corresponding representation function now yields a binary tree:

$$\begin{array}{lcl} (\!|\, [\,] \,|\!) & = & [\,] \\ (\!|\, s +\!\!+ [a] +\!\!+ t \,|\!) & = & ((\!|\, s \,|\!), a) \vdash (\!|\, t \,|\!) \quad , a \leq \downarrow s \ \wedge \ a < \downarrow t \,. \end{array}$$

In terms of type $\langle T \rangle$, this a tree satisfying the following heap condition:

$$\begin{array}{lcl} H.\langle\,\rangle & \equiv & \text{true} \\ H.\langle t, a, u \rangle & \equiv & H.t \ \wedge \ a \leq \downarrow t \ \wedge \ a < \downarrow u \ \wedge \ H.u, \end{array}$$

and heap x represents list $\overline{x}$, the inorder traversal of x. These binary heaps are known as *Cartesian trees* [35] and have many applications (see e.g. [3, 9]).
□

8.2 Deques

In Section 5.6.1 we have implemented an algebra of deques using a pair of stacks. From the observations in the previous section we conclude that algebra DQ extended with operation $\downarrow$:

$$\mathsf{MDQ} = (\, [T] \mid [\,], (=[\,]), \vdash, hd, tl, \dashv, lt, ft, rev, \downarrow\,),$$

can thus be implemented by means of a pair of minstacks. The program for min is

$$\mathsf{min}.\langle s, t \rangle = \downarrow s \ \min \downarrow t \,,$$

and the programs for the other operations can simply be copied from Section 5.6.1. This yields an efficient purely-functional solution which uses at most two pointers per list element.

8.3 Concatenable queues

As far as we know there is no implementation for algebra MDQ extended with $+\!\!+$ such that all operations have $O(1)$ time complexity. An efficient implementation of *concatenable minqueues*, however, is possible:

$$\mathsf{CMQ} = (\, [T] \mid [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, \downarrow\,).$$

As stepping-stone to an implementation of this algebra, we first give an alternative implementation of *minqueues*:

$$\mathsf{MQ} = (\, [T] \mid [\,], (=[\,]), \dashv, hd, tl, \downarrow\,).$$

This implementation is identical to the implementation of MS in Section 8.1, except that operation $\dashv$ is refined by

$$\begin{array}{lcll} \mathsf{snoc}.x.b & = & f.x.([\,],b) & \\ & & |[\ f.[\,].(t,b) & = & [(t,b)] \\ & & \quad f.(x \dashv (s,a)).(t,b) & = & x \dashv (s,a) \dashv (t,b) & ,a < b \\ & & & [] & f.x.(s +\!\!+ [a] +\!\!+ t,b) & ,a \geq b \\ & &]|. & & & \end{array}$$

In these programs the inner lists are used as concatenable queues. Since these lists are used in a linear fashion, the destructive implementation of concatenable queues described in Section 4.5 can be used. The outer list is used as a deque. Taking $\Phi.x = \#x$ as potential, the amortized costs of snoc (and the other operations) are all $O(1)$.

As implementation for $+\!\!+$ we have

$$\begin{array}{lcll} \mathsf{cat}.x.y = g.x.y & & & \\ \quad |[\ g.[\,].y & = & y & \\ \quad\ g.x.[\,] & = & x & \\ \quad\ g.(x \dashv (s,a)).((t,b) \vdash y) & = & (x \dashv (s,a)) +\!\!+ ((t,b) \vdash y) & ,a < b \\ & [] & g.x.((s +\!\!+ [a] +\!\!+ t,b) \vdash y) & ,a \geq b \\ \quad]|. & & & \end{array}$$

With the same potential as above, this program also has $O(1)$ amortized cost. Note that the outer list is now used as a concatenable deque.

8.4 Noninjective algebras

There are applications of minqueues in which operation *hd* is not required. In those cases it suffices to implement the following algebra:

$$\mathsf{MQ}' = (\,[T] \mid [\,], (=[\,]), \dashv, tl, \downarrow\,).$$

As explained in Example 2.26, this algebra is not injective because lists $[1,0]$ and $[2,0]$, for example, cannot be distinguished. To describe an efficient implementation of MQ′ we use a representation function $(\!|\cdot|\!)$ that maps $[T]$ onto concrete type $[\mathrm{Nat} \times T]$:

$$\begin{array}{lcl} (\!|\,[\,]\,|\!) & = & [\,] \\ (\!|s +\!\!+ [a] +\!\!+ t|\!) & = & (\#s,a) \vdash (\!|t|\!) \quad , a \leq \downarrow s \ \wedge\ a < \downarrow t\,. \end{array}$$

Hence, instead of a list x of type $[[T] \times T]$, it now suffices to record the *length* of the inner lists.

Accordingly, the programs are obtained by replacing the inner lists in the programs for minqueues by their length:

$$\mathsf{empty} \quad = \quad [\,]$$

$$
\begin{array}{lcll}
\mathsf{isempty}.x & = & x = [\,] & \\
\mathsf{snoc}.x.b & = & f.x.(0,b) & \\
 & & [\![\ f.[\,].(n,b) & = \ [(n,b)] \\
 & & \quad f.(x \dashv (m,a)).(n,b) & = \ x \dashv (m,a) \dashv (n,b) \quad , a < b \\
 & & & [\!] \ f.x.(m{+}1{+}n,b) \quad , a \geq b \\
 & &]\!] & \\
\mathsf{tl}.((m,a) \vdash x) & = & x & , m = 0 \\
 & [\!] & (m{-}1,a) \vdash x & , m \neq 0 \\
\mathsf{min}.[\,] & = & \infty & \\
\mathsf{min}.((m,a) \vdash x) & = & a. &
\end{array}
$$

As the outer list is used as a deque, it follows that there exists a purely-functional implementation of MQ'.

Example 8.2
As an application of algebra MQ', we consider an abstract version of the pagination algorithm used in the pagination strategy MS in [22]. The problem is to determine, for fixed positive integer K, a pair of elements of a given sequence that are no more than K positions apart and that have minimal sum. More formally, the problem is to compute

$$(\mathrm{Min}\, i, j : 0 \leq i < j < N \ \wedge \ j{-}K \leq i : X[i] + X[j])$$

for given integer array X of positive length N.

Our solution relies on the fact that for $1 \leq n < N$:

$$
\begin{array}{ll}
 & (\mathrm{Min}\, i, j : 0 \leq i < j < n{+}1 \ \wedge \ j{-}K \leq i : X[i] + X[j]) \\
= & \quad \{ \text{ split off } j = n \ \} \\
 & (\mathrm{Min}\, i, j : 0 \leq i < j < n \ \wedge \ j{-}K \leq i : X[i] + X[j]) \\
 & \mathrm{min}\, ((\mathrm{Min}\, i : 0 \leq i < n \ \wedge \ n{-}K \leq i : X[i]) + X[n]).
\end{array}
$$

To compute the minimum expressed on the bottom line we use a minqueue q satisfying $[\![q]\!] = [\, i : 0 \leq i < n \ \wedge \ n{-}K \leq i : X[i] \,]$. This yields the following $O(N)$ program for the original problem:

$$
\begin{array}{l}
\quad m, q, n := \infty, \mathsf{snoc.empty}.X[0], 1 \\
; \ \mathbf{do}\ n \neq N \ \rightarrow \\
\qquad m := m \,\mathrm{min}\, (\mathsf{min}.q + X[n]) \\
\quad ; \ q := \mathsf{snoc}.q.X[n] \\
\quad ; \ \mathbf{if}\ n \geq K \ \rightarrow q := \mathsf{tl}.q \\
\qquad [\!]\ n < K \ \rightarrow \mathrm{skip} \\
\qquad \mathbf{fi} \\
\quad ; \ n := n{+}1 \\
\mathbf{od}\ .
\end{array}
$$

Clearly, operation hd is not used in this application, and therefore the above described implementation of minqueues suffices.
□

The algebra of minstacks without operation hd is also noninjective, and type [Nat $\times$ T] can be used to implement this algebra as well. The following algebra of mindeques without operations hd and lt is however injective:

$$([T] \mid [], (=[]), \vdash, tl, \dashv, ft, \downarrow).$$

If two lists of equal length differ in one position, both lists can be reduced by means of tl and ft to singleton lists containing these elements and operation $\downarrow$ will then return different results for these singleton lists. Implementation of these mindeques using a pair of minstacks therefore requires the full set of operations of algebra MS.

Chapter 9

Skew heaps

In this chapter we design and analyze functional programs for a number of priority queue operations. At the specification level (Section 9.1) priority queues are considered as bags, whereas they are represented by pointer-structures at the implementation level (Section 9.5). In the major part of this chapter, however, priority queues will be described in terms of trees so as to achieve a suitable separation of concerns (Sections 9.2 and 9.4).

The implementations of priority queues treated in this chapter are based upon the skew heaps of Sleator and Tarjan [30]. In terms of the appropriate tree types we are able to give concise programs for the various priority queue operations, and these programs form the starting-point for a formal performance analysis of skew heaps. Our systematic approach results in improvements for the programs as well as tighter bounds for the amortized costs thereof. The required potential functions will be derived as much as possible.

9.1 Priority queues

The problem under consideration is the design of efficient implementations of an abstract algebra of priority queues PQ to be defined below. The data type of PQ is $(\!|\mathrm{Int}|\!)$, and PQ should comprise all operations that pertain to bags in programs like $sort0$ presented below. This program puts a list of integers into ascending order by means of a number of bag operations ($sort0 \in [\mathrm{Int}] \to [\mathrm{Int}]$):

$$
\begin{array}{lcl}
sort0 & = & g \circ f \\
 & & |[\ f.[\,] \quad = \ (\!|\,|\!) \\
 & & \quad f.(a \vdash s) \ = \ (\!|a|\!) \oplus f.s \\
 & & \quad g.B \ = \ [\,] \qquad\qquad\quad , B = (\!|\,|\!) \\
 & & \qquad\ \ [] \ \downarrow B \vdash g.(\Downarrow B) \ , B \neq (\!|\,|\!) \\
 & &]|.
\end{array}
$$

As a possible definition for PQ we propose an algebra with six operations:

$$\mathsf{PQ} = (\,(\!|\mathrm{Int}|\!) \mid (\!|\,|\!), (\!|\cdot|\!), \oplus, (=(\!|\,|\!)), \downarrow, \Downarrow\,),$$

where the last two operations both have domain $(|\mathrm{Int}|)\backslash\{(|\,|)\}$. Since any bag can be constructed by means of the first three operations, and since any pair of distinct bags can be distinguished by means of the last three operations, algebra PQ is bijective. As explained in Section 2.5, any refinement of PQ may therefore be described by means of a (surjective) abstraction function. The problem is thus to design a concrete algebra with signature

$$\mathsf{SH} = (\, S \mid \mathsf{empty}, \mathsf{single}, \mathsf{union}, \mathsf{isempty}, \mathsf{min}, \mathsf{delmin} \,),$$

say, such that SH refines PQ under abstraction $[\![\cdot]\!]$.

In comparison with [30], we have included in PQ the operation $(|\cdot|)$ instead of the abstract counterpart of operation $\mathsf{insert} \in \mathrm{Int}\times S \to S$, which satisfies

$$[\![\mathsf{insert}.a.p]\!] = (|a|) \oplus [\![p]\!].$$

Of course, there is no essential difference between our notion of priority queues and theirs, because, on the one hand operation insert can be implemented as

$$\mathsf{insert}.a.p = \mathsf{union}.(\mathsf{single}.a).p,$$

and on the other hand single can be defined as

$$\mathsf{single}.a = \mathsf{insert}.a.\mathsf{empty}.$$

We prefer to include single as a priority queue operation because it is simpler than insert. Note, however, that inclusion of insert instead of single as operation in PQ would make operation union superfluous for the refinement of $sort0$. This operation is nevertheless included because it is an important operation in other applications (e.g., computation of minimum-cost spanning trees); some authors emphasize its presence by using the term "*mergeable* priority queue".

With algebra SH at our disposal, we readily obtain the following concrete program as refinement of $sort0$:

$$\begin{array}{lcl}
sort1 & = & g \circ f \\
 & & |[\ f.[\,] \quad = \quad \mathsf{empty} \\
 & & \quad f.(a \vdash s) \ = \ \mathsf{union}.(\mathsf{single}.a).(f.s) \\
 & & \quad g.p \ = \ [\,] \qquad\qquad\qquad\qquad , \ \mathsf{isempty}.p \\
 & & \qquad\quad [] \ \ \mathsf{min}.p \vdash g.(\mathsf{delmin}.p) \ \ , \neg\mathsf{isempty}.p \\
 & &]|.
\end{array}$$

Of course, we are only interested in *efficient* refinements of PQ. Therefore, we end this section with a number of efficiency requirements.

To begin with, we aim at a representation for priority queues that takes linear space, i.e., $O(\#[\![p]\!])$ space to represent priority queue p. Next, we formulate some requirements for the time complexity of the operations, for which the amortized

costs are defined according to the scheme of Section 5.5:

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{single}](a) & = & \mathcal{T}[\mathsf{single}](a) + \Phi.(\mathsf{single}.a) \\
\mathcal{A}[\mathsf{union}](p,q) & = & \mathcal{T}[\mathsf{union}](p,q) + \Phi.(\mathsf{union}.p.q) - \Phi.p - \Phi.q \\
\mathcal{A}[\mathsf{isempty}](p) & = & \mathcal{T}[\mathsf{isempty}](p) \\
\mathcal{A}[\mathsf{min}](p) & = & \mathcal{T}[\mathsf{min}](p) \\
\mathcal{A}[\mathsf{delmin}](p) & = & \mathcal{T}[\mathsf{delmin}](p) + \Phi.(\mathsf{delmin}.p) - \Phi.p.
\end{array}$$

We would like to have $O(1)$ bounds for the amortized costs of all priority queue operations. But, since we are able to sort with these operations (e.g., by means of program $sort1$), we have to allow that some operations have logarithmic bounds for their amortized costs. In any case, however, we want the computation of $sort1.s$ to take $O(\#s \log \#s)$ time. In addition we want the potential function to be nonnegative and small (e.g., $0 \leq \Phi.p \leq \#[\![p]\!]$).

9.2 Top-down skew heaps

It goes without saying that we try to keep the implementation of an algebra as simple as possible. We hope to achieve this by taking as our principal working hypothesis that—ideally—any design decision should be justly motivated by a phrase like "The simplest solution we can think of is now to ...".

Under this hypothesis, we shall derive a very simple implementation of priority queues. Despite its simplicity, however, it is not easy to show that the efficiency requirements are met. In order to show that the amortized costs of the priority queue operations are sufficiently low, we try to *derive*—as much as possible—a suitable potential function.

So, let us now see whether these intentions come to anything good. In order to get started, we first have to make a decision regarding data type S. In the previous section we have already concluded that abstraction $[\![\cdot]\!]$ must be surjective, hence S should satisfy:

(1) $\quad (\forall\, B : B \in (\!|\mathrm{Int}|\!) : (\exists\, p : p \in S : B = [\![p]\!]))$.

In view of this requirement the simplest concrete type for S we can think of is [Int], or a subset thereof. Unfortunately, as the reader may verify, it is not at all clear how the efficiency requirements can be met in this way.

The next candidate for S is $\langle \mathrm{Int} \rangle$. We decide to let S be a subset of $\langle \mathrm{Int} \rangle$, that is, we require S to satisfy:

(2) $\quad S \subseteq \langle \mathrm{Int} \rangle$.

This restriction on S enables us to define $[\![\cdot]\!]$ already at this point, namely as

$$\begin{array}{lcl}
[\![\langle\,\rangle]\!] & = & (\!|\,|\!) \\
[\![\langle t, a, u \rangle]\!] & = & [\![t]\!] \oplus (\!|a|\!) \oplus [\![u]\!].
\end{array}$$

This representation takes linear space.

9.2.1 Introduction of $\bowtie$

We now turn our attention to the operations of SH. "On the fly" we will derive additional restrictions on S and a suitable definition for S will be presented at the end of this section.

First of all, we deal with the operations empty, isempty, and single, since their definitions are fixed by our choice for $[\![\cdot]\!]$. More precisely, $\langle\,\rangle$ is the only element of $\langle \mathrm{Int}\rangle$ whose $[\![\cdot]\!]$-value equals $\{\!|\,|\!\}$, so we have to define

$$\begin{array}{lcl} \mathsf{empty} & = & \langle\,\rangle \\ \mathsf{isempty}.z & = & z = \langle\,\rangle. \end{array}$$

Similarly, since $\langle a\rangle$ is the only element in $\langle \mathrm{Int}\rangle$ whose $[\![\cdot]\!]$-value equals $\{\!|a|\!\}$, we define

$$\mathsf{single}.a = \langle a\rangle.$$

These three programs all have constant time complexity.

Next, we consider operation min. For this operation we propose the simplest $O(1)$ program we can think of, viz.

$$\mathsf{min}.\langle t, a, u\rangle = a.$$

This program is correct if S satisfies the additional restriction

$$(3) \quad \langle t, a, u\rangle \in S \;\Rightarrow\; a = \downarrow\langle t, a, u\rangle\,,$$

which expresses that the root should contain the minimum value.

Of the remaining two operations, we first consider delmin. We derive, for $\langle t, a, u\rangle \in S$:

$$\begin{array}{ll} & [\![\mathsf{delmin}.\langle t, a, u\rangle]\!] \\ = & \quad \{ \text{ specification of } \mathsf{delmin} \text{ } \} \\ & \Downarrow[\![\langle t, a, u\rangle]\!] \\ = & \quad \{ \text{ definition of } \Downarrow \} \\ & [\![\langle t, a, u\rangle]\!] \ominus \{\!|\downarrow\langle t, a, u\rangle|\!\} \\ = & \quad \{ \text{ definition of } [\![\cdot]\!]\text{; restriction (3) on } S \text{ } \} \\ & ([\![t]\!] \oplus \{\!|a|\!\} \oplus [\![u]\!]) \ominus \{\!|a|\!\} \\ = & \quad \{ \text{ bag calculus } \} \\ & [\![t]\!] \oplus [\![u]\!]. \end{array}$$

Since union is required to satisfy $[\![\mathsf{union}.x.y]\!] = [\![x]\!] \oplus [\![y]\!]$ as well, we can deal with delmin and union in a uniform way by introducing operator $\bowtie$ ("meld") satisfying

$$(4) \quad \bowtie \in S \times S \to S$$

$$(5) \quad [\![x \bowtie y]\!] = [\![x]\!] \oplus [\![y]\!].$$

The resulting programs are

$$\begin{array}{lcl} \mathsf{union}.x.y & = & x \bowtie y \\ \mathsf{delmin}.\langle t, a, u\rangle & = & t \bowtie u, \end{array}$$

provided that S satisfies

(6) $\quad \langle t, a, u\rangle \in S \;\Rightarrow\; t \in S \;\wedge\; u \in S.$

In order not to impose stronger conditions on the priority queue operations than necessary, we choose for S the *largest* set satisfying requirements (1), (2), (3), and (6). That is, we define

$$S = \{z \mid z \in \langle \mathrm{Int}\rangle \;\wedge\; H.z\},$$

where H is the *heap condition* for trees, defined by

$$\begin{array}{lcl} H.\langle\,\rangle & \equiv & \mathrm{true} \\ H.\langle t, a, u\rangle & \equiv & H.t \;\wedge\; a \le \downarrow t \;\wedge\; a \le \downarrow u \;\wedge\; H.u. \end{array}$$

Notice that the above derivation hinges on the decision to impose restriction (3) on S; from this decision it follows in a calculational manner that S becomes the set of (binary) heaps.

From $H.\langle\,\rangle$ and $H.\langle a\rangle$ we infer that empty and single indeed satisfy their specifications, which leaves us with the problem of designing an efficient program for $\bowtie$.

9.2.2 Implementation of $\bowtie$

It is hardly surprising that a program for $\bowtie$ will be recursive. In the inductive derivation of candidates for $x \bowtie y$ below, we first focus on x, for which we distinguish the cases $x = \langle\,\rangle$ and $x \neq \langle\,\rangle$. Since the specification of $x \bowtie y$ is symmetric in x and y, the corresponding cases for y can then be settled by interchanging the role of x and y.

Case $x = \langle\,\rangle$. The obvious candidate for $\langle\,\rangle \bowtie y$ is y.

Case $x \neq \langle\,\rangle$. In this case we dissect x, say $x = \langle t, a, u\rangle$, but we refrain from dissecting y (just to keep it simple). Since part (4) of the specification of $\bowtie$ is too weak on its own, we have to consider part (5) first:

$$\begin{array}{ll} & [\![\langle t, a, u\rangle \bowtie y]\!] \\ = & \quad \{\ (5);\ \text{definition of } [\![\cdot]\!];\ \oplus \text{ is associative }\ \} \\ & [\![t]\!] \oplus \llparenthesis a \rrparenthesis \oplus [\![u]\!] \oplus [\![y]\!] \\ = & \quad \{\ \oplus \text{ is symmetric (choosing 1 out of 3 options: } y, u \text{ or } t \text{ separate})\ \} \\ & ([\![t]\!] \oplus [\![u]\!]) \oplus \llparenthesis a \rrparenthesis \oplus [\![y]\!] \\ = & \quad \{\ \text{ind. hypothesis (5) (choosing 1 out of 2 options: } t \bowtie u \text{ or } u \bowtie t)\ \} \\ & [\![t \bowtie u]\!] \oplus \llparenthesis a \rrparenthesis \oplus [\![y]\!] \\ = & \quad \{\ \text{definition of } [\![\cdot]\!] \text{ (choosing 1 out of 2 options: } \bowtie \text{ left or } \bowtie \text{ right})\ \} \\ & [\![\langle t \bowtie u, a, y\rangle]\!]. \end{array}$$

Thus, for the time being, we take $\langle t \bowtie u, a, y\rangle$ as one out of twelve candidates for $x \bowtie y$. For this candidate, we now consider part (4) of the specification of $\bowtie$ (the heap condition). Assuming that $x \in S \wedge y \in S$, hence that $H.t \ \wedge \ a \leq \downarrow t \ \wedge \ a \leq \downarrow u \ \wedge \ H.u \ \wedge \ H.y$, we derive inductively:

$$
\begin{array}{ll}
 & H.\langle t \bowtie u, a, y\rangle \\
\equiv & \quad \{ \text{ definition of } H \ \} \\
 & H.(t \bowtie u) \ \wedge \ a \leq \downarrow(t \bowtie u) \ \wedge \ a \leq \downarrow y \ \wedge \ H.y \\
\equiv & \quad \{ \text{ ind. hypothesis (4), hence } H.t \wedge H.u \ \Rightarrow \ H.(t \bowtie u) \text{; assumption } \} \\
 & a \leq \downarrow(t \bowtie u) \ \wedge \ a \leq \downarrow y \\
\equiv & \quad \{ \text{ ind. hypothesis (5), hence } [\![t \bowtie u]\!] = [\![t]\!] \oplus [\![u]\!] \text{; bag calculus } \} \\
 & a \leq \downarrow t \ \wedge \ a \leq \downarrow u \ \wedge \ a \leq \downarrow y \\
\equiv & \quad \{ \text{ assumption; } a = \downarrow x \ \} \\
 & \downarrow x \leq \downarrow y \,.
\end{array}
$$

For the other candidates we obtain the same condition to ensure that (4) is satisfied in all cases.

So, we can solve the cases $x = \langle\,\rangle$ and $x \neq \langle\,\rangle \ \wedge \ \downarrow x \leq \downarrow y$. By interchanging x and y in the above derivation, we also obtain solutions for the cases $y = \langle\,\rangle$ and $y \neq \langle\,\rangle \ \wedge \ \downarrow y \leq \downarrow x$. Hence, we are done because the case analysis is now exhaustive. There is even some overlap in this case analysis and we can limit the number of cases to three by removing some of this overlap: instead of the two cases $x = \langle\,\rangle$ and $y = \langle\,\rangle$ we take case $x = \langle\,\rangle \ \wedge \ y = \langle\,\rangle$. (In Section 9.2.6 we will consider a slightly more complicated alternative.) In order to turn the above conditions into concrete guards, we introduce function m, satisfying $H.z \Rightarrow m.z{=}\downarrow z$, defined by

$$
\begin{array}{lcl}
m.\langle\,\rangle & = & \infty \\
m.\langle t, a, u\rangle & = & a.
\end{array}
$$

As a result, we obtain linearly recursive programs of the form

$$
\begin{array}{rcll}
\langle\,\rangle \ \bowtie \ \langle\,\rangle & = & \langle\,\rangle & \\
\langle t, a, u\rangle \ \bowtie \ y & \doteq & \mathrm{B} & , a \leq m.y \\
x \ \bowtie \ \langle t, a, u\rangle & \doteq & \mathrm{B}' & , a \leq m.x,
\end{array}
$$

in which B is one of the twelve candidates for $x \bowtie y$ mentioned above and B′ is the corresponding candidate with y replaced by x. As for the possibilities for B, however, we can limit our attention to the following three candidates:

$$
\begin{array}{ll}
\text{(Y)} & \langle t \bowtie u, a, y\rangle \\
\text{(U)} & \langle t \bowtie y, a, u\rangle \\
\text{(T)} & \langle u \bowtie y, a, t\rangle,
\end{array}
$$

because, firstly, swapping the operands of $\bowtie$ is of no use, since our tentative program for $\bowtie$ is symmetric, and, secondly, swapping the subtrees only exchanges the role of left and right, which cannot induce essentially different results.

The cost measure defined by the dots in the above program scheme for $\bowtie$ will be called $\mathcal{N}$. Notice that $\mathcal{N}(E)$ is approximately equal to the number of integer comparisons performed in the evaluation of E (counting one comparison for each unfolding of the second or third alternative of $\bowtie$, also counting the comparisons with ∞).

The choice between (Y), (U), and (T) is motivated by the following observations. Consider program h (cf. program f in $sort1$, Section 9.1):

$$\begin{array}{lcl} h.[\,] & = & \langle\,\rangle \\ h.(a \vdash s) & = & \langle a \rangle \bowtie h.s. \end{array}$$

Then—as the reader may verify—$\mathcal{N}[h](s) \approx \frac{1}{4}(\#s)^2$, if we take (Y) for B and s decreasing. Similarly, we have that $\mathcal{N}[h](s) \approx \frac{1}{2}(\#s)^2$, if we take (U) for B and s decreasing. It is even the case that the last alternative of the program for $\bowtie$ is never applied in these computations of h. Hence, irrespective of our choice for B′ candidates (Y) and (U) drop out. Fortunately, candidate (T) cannot be eliminated in this way. The remaining program for $\bowtie$ reads:

$$\begin{array}{rcll} \langle\,\rangle & \bowtie & \langle\,\rangle & = \langle\,\rangle \\ \langle t, a, u \rangle & \bowtie & y & \doteq \langle u \bowtie y, a, t \rangle \quad , a \leq m.y \\ x & \bowtie & \langle t, a, u \rangle & \doteq \langle u \bowtie x, a, t \rangle \quad , a \leq m.x. \end{array}$$

This program is almost the same as the program for top-down melding of skew heaps in [30], and Sleator and Tarjan have shown—informally—that their program is efficient in the amortized sense. The fact that t and u are "swapped" in the last two alternatives is crucial for the efficiency. In the next section we perform a formal amortized analysis of the above program for $\bowtie$. Note that $\mathcal{N}[\mathsf{delmin}](h.s)$ is approximately $\#(h.s)$ if s is increasing, hence some applications of delmin may take $\Omega(\#s)$ time during the evaluation of $sort1.s$; therefore an amortized analysis is really necessary.

9.2.3 Analysis of $\bowtie$

The amortized costs of $\bowtie$ are defined by

$$(7) \quad \mathcal{A}[\bowtie](x, y) = \mathcal{N}[\bowtie](x, y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y.$$

In order to facilitate computations with potential Φ, we introduce function φ and we choose the following recursive definition for Φ (cf. the recursive structure of $\langle \mathrm{Int} \rangle$).

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u \rangle & = & \Phi.t + \varphi.t.u + \Phi.u. \end{array}$$

By writing $\varphi.t.u$ instead of $\varphi.t.a.u$, say, we make $\Phi.\langle t, a, u\rangle$ independent of the value of a. This shortens the calculations involving φ somewhat, and—as we shall see later on— this is a harmless design decision. (As a matter of fact, $\Phi.z$ will be independent of the values in z altogether because $\varphi.t.u$ will be independent of the values in t and u.)

Recall that we want Φ to be nonnegative and small. In order to fulfill these requirements we have chosen $\Phi.\langle\,\rangle = 0$. Moreover, we require φ to be nonnegative and small. Our main goal is to define φ in such a way that $\mathcal{A}[\bowtie](x, y)$ can be shown to be $O(\log(\sharp x + \sharp y))$. Here, $\sharp z$ denotes $1 + \#z$, hence it is positive and it makes sense to write things like $\log_2 \sharp z$ and $\sharp x/\sharp y$—also when empty trees are not excluded. In the sequel we will often use that $\sharp$ satisfies the following recurrence relation:

$$\begin{array}{lcl} \sharp\langle\,\rangle & = & 1 \\ \sharp\langle t, a, u\rangle & = & \sharp t + \sharp u. \end{array}$$

As a first step, we derive a recurrence relation for $\mathcal{A}[\bowtie]$.

Case $x = \langle\,\rangle \;\wedge\; y = \langle\,\rangle$. Then $\mathcal{N}[\bowtie](\langle\,\rangle, \langle\,\rangle) = 0$, and

$$\begin{array}{ll} & \mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) \\ = & \quad \{\ (7)\ \} \\ & \mathcal{N}[\bowtie](\langle\,\rangle, \langle\,\rangle) + \Phi.(\langle\,\rangle \bowtie \langle\,\rangle) - \Phi.\langle\,\rangle - \Phi.\langle\,\rangle \\ = & \quad \{\ \text{definition of } \bowtie;\ \Phi.\langle\,\rangle = 0\ \} \\ & 0. \end{array}$$

Case $x = \langle t, a, u\rangle \;\wedge\; a \leq m.y$. Then $\mathcal{N}[\bowtie](x, y) = 1 + \mathcal{N}[\bowtie](u, y)$, and $x \bowtie y = \langle u \bowtie y, a, t\rangle$, and

$$\begin{array}{ll} & \mathcal{A}[\bowtie](x, y) \\ = & \quad \{\ (7)\ \} \\ & \mathcal{N}[\bowtie](x, y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y \\ = & \quad \{\ \text{definition of } \bowtie;\ \text{definition of } \Phi\ \} \\ & 1 + \mathcal{N}[\bowtie](u, y) + \Phi.(u \bowtie y) + \varphi.(u \bowtie y).t + \Phi.t - \Phi.t - \varphi.t.u - \Phi.u - \Phi.y \\ = & \quad \{\ \text{definition of } \mathcal{A}\ \} \\ & \mathcal{A}[\bowtie](u, y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u. \end{array}$$

By symmetry, we thus have:

$$\begin{array}{lll} \mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) & = & 0 \\ \mathcal{A}[\bowtie](\langle t, a, u\rangle, y) & = & \mathcal{A}[\bowtie](u, y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \quad , a \leq m.y \\ \mathcal{A}[\bowtie](x, \langle t, a, u\rangle) & = & \mathcal{A}[\bowtie](u, x) + \varphi.(u \bowtie x).t + 1 - \varphi.t.u \quad , a \leq m.x. \end{array} \tag{8}$$

Unfolding this recurrence relation several times and realizing that $\mathcal{A}[\bowtie](x, y)$ is defined in terms of x, y, and $x \bowtie y$ (cf. (7)), we aim at a result of the following form (symmetric in x and y):

(9) $\mathcal{A}[\bowtie](x,y) = \Gamma.(x \bowtie y) + \Delta.x + \Delta.y.$

Functions Γ and Δ are now derived inductively. Since $\mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) = 0$, we take $\Gamma.\langle\,\rangle = 0$ and $\Delta.\langle\,\rangle = 0$. With case $x = \langle t, a, u\rangle \wedge a \leq m.y$ we deal as follows, using (9) as induction hypothesis:

$$
\begin{array}{ll}
 & \mathcal{A}[\bowtie](x,y) \\
= & \quad \{\ (8)\ \} \\
 & \mathcal{A}[\bowtie](u,y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \\
= & \quad \{\ \text{induction hypothesis (9)}\ \} \\
 & \Gamma.(u \bowtie y) + \Delta.u + \Delta.y + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \\
= & \quad \{\ \text{definitions of } \Gamma \text{ and } \Delta\text{, see below}\ \} \\
 & \Gamma.\langle u \bowtie y, a, t\rangle + \Delta.\langle t, a, u\rangle + \Delta.y \\
= & \quad \{\ \text{definition of } \bowtie\ \} \\
 & \Gamma.(x \bowtie y) + \Delta.x + \Delta.y.
\end{array}
$$

The remaining case follows by symmetry, hence (9) holds if we define

$$
\begin{array}{lcl}
\Gamma.\langle\,\rangle & = & 0 \\
\Gamma.\langle t, a, u\rangle & = & \Gamma.t + \varphi.t.u + \gamma \\
\Delta.\langle\,\rangle & = & 0 \\
\Delta.\langle t, a, u\rangle & = & \Delta.u - \varphi.t.u + \delta,
\end{array}
$$

with γ and δ constants to be chosen later on such that $\gamma + \delta = 1$.

A logarithmic bound for $\mathcal{A}[\bowtie](x,y)$ follows from (9) if we can define φ and choose γ and δ such that

(10) $\Gamma.z \leq \log_\alpha \sharp z \ \wedge\ \Delta.z \leq \log_\alpha \sharp z,$

for some constant α, $\alpha > 1$. To satisfy (10), we derive requirements on φ by induction on z.

Case $z = \langle\,\rangle$. Trivial, since $\sharp\langle\,\rangle = 1$.

Case $z = \langle t, a, u\rangle$. We derive for Γ and Δ, respectively:

$$
\begin{array}{ll}
 & \Gamma.\langle t, a, u\rangle \\
= & \quad \{\ \text{definition of } \Gamma\ \} \\
 & \Gamma.t + \varphi.t.u + \gamma \\
\leq & \quad \{\ \text{ind. hypothesis (10)}\ \} \\
 & \log_\alpha \sharp t + \varphi.t.u + \gamma \\
\leq & \quad \{\ \text{upper bound } \varphi \text{ in (11)}\ \} \\
 & \log_\alpha \sharp\langle t, a, u\rangle
\end{array}
\qquad
\begin{array}{ll}
 & \Delta.\langle t, a, u\rangle \\
= & \quad \{\ \text{definition of } \Delta\ \} \\
 & \Delta.u - \varphi.t.u + \delta \\
\leq & \quad \{\ \text{ind. hypothesis (10)}\ \} \\
 & \log_\alpha \sharp u - \varphi.t.u + \delta \\
\leq & \quad \{\ \text{lower bound } \varphi \text{ in (11)}\ \} \\
 & \log_\alpha \sharp\langle t, a, u\rangle.
\end{array}
$$

Hence, we require that φ satisfies for any t and u, using $\sharp\langle t, a, u\rangle = \sharp t + \sharp u$:

$$(11)\quad \log_\alpha \frac{\alpha^\delta \sharp u}{\sharp t + \sharp u} \le \varphi.t.u \le \log_\alpha \frac{\sharp t + \sharp u}{\alpha^\gamma \sharp t}.$$

The existence of such a φ is guaranteed if the lower bound never exceeds the upper bound. Therefore, we calculate (m and n correspond to $\sharp t$ and $\sharp u$, respectively; hence, m and n are positive integers):

$$\begin{array}{ll} & (\forall\, m,n :: \log_\alpha \frac{\alpha^\delta n}{m+n} \le \log_\alpha \frac{m+n}{\alpha^\gamma m}) \\ \equiv & \quad \{ \text{ monotonicity of } \log_\alpha \ (\alpha > 1) \ \} \\ & (\forall\, m,n :: \frac{\alpha^\delta n}{m+n} \le \frac{m+n}{\alpha^\gamma m}) \\ \equiv & \quad \{ \ \gamma + \delta = 1 \ \} \\ & (\forall\, m,n :: \alpha mn \le (m+n)^2) \\ \equiv & \quad \{ \ \} \\ & (\forall\, m,n :: \alpha \le 4 + \frac{(m-n)^2}{mn}) \\ \equiv & \quad \{ \ \} \\ & \alpha \le 4. \end{array}$$

Hence, for any α, $1 < \alpha \le 4$, it follows from (9) and (10) that there exists a potential function for which

$$\mathcal{A}[\bowtie](x,y) \le \log_\alpha(\sharp x + \sharp y) + \log_\alpha \sharp x + \log_\alpha \sharp y.$$

Thus, we have shown that $\mathcal{A}[\bowtie](x,y)$ is $O(\log(\sharp x + \sharp y))$.

We are, however, not only interested in the asymptotic behaviour of $\mathcal{A}[\bowtie]$. We obtain a low upper bound for $\mathcal{A}[\bowtie](x,y)$ by choosing α as large as possible, viz. $\alpha = 4$. Then

$$\mathcal{A}[\bowtie](x,y) \le \tfrac{3}{2}\ \log_2(\sharp x + \sharp y).$$

This bound improves the bound of [30] by a factor 2. (In the next section we show that the bound can be reduced even further.)

So much for the amortized costs of $\bowtie$. Next we try to define φ such that it is nonnegative. Evidently, this requires that the upper bound for φ in (11) is nonnegative. We observe:

$$\begin{array}{ll} & (\forall\, m,n :: 0 \le \log_4 \frac{m+n}{4^\gamma m}) \\ \equiv & \quad \{ \text{ monotonicity of } \log_4 \ \} \\ & (\forall\, m,n :: 1 \le \frac{m+n}{4^\gamma m}) \\ \equiv & \quad \{ \ \} \\ & (\forall\, m,n :: 4^\gamma \le \frac{m+n}{m}) \\ \equiv & \quad \{ \ \} \\ & 4^\gamma \le 1 \\ \equiv & \quad \{ \text{ monotonicity of exponentiation } \} \\ & \gamma \le 0. \end{array}$$

So, we have to take $\gamma \leq 0$ to ensure that φ can be defined nonnegative. Furthermore, to obtain a small potential, we minimize the lower bound of φ in (11) by choosing δ as small as possible under the constraints $\gamma + \delta = 1$ and $\gamma \leq 0$. This yields $\gamma = 0$ and $\delta = 1$. The smallest nonnegative choice for φ thus is:

$$\varphi.t.u = \log_4 \frac{4\sharp u}{\sharp t + \sharp u} \max 0$$

Then $0 \leq \varphi.t.u \leq 1$, and we have proved the following lemma.

Lemma 9.1
Let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t \;+\; \log_4 \frac{4\sharp u}{\sharp t+\sharp u} \max 0 \;+\; \Phi.u. \end{array}$$

Then $0 \leq \Phi.z \leq \#z$, and the amortized costs of top-down melding satisfy

$$\mathcal{A}[\bowtie](x,y) \leq \tfrac{3}{2}\, \log_2(\sharp x + \sharp y).$$

□

9.2.4 Refined analysis of $\bowtie$

As pointed out to us by D.D. Sleator, it is possible to reduce the constant in the bound for $\mathcal{A}[\bowtie]$ from $\frac{3}{2}$ to approximately 1.44. To achieve this improvement, we refine the analysis of $\bowtie$ by introducing independent constants in the bounds for Γ and Δ. That is, instead of (10) we take

(12) $\Gamma.z \leq \log_\alpha \sharp z \;\wedge\; \Delta.z \leq \log_\beta \sharp z,$

with $\alpha > 1$ and $\beta > 1$. As may be gathered from the derivations in the previous section, these inequalities are satisfied if φ satisfies (cf. (11))

(13) $\log_\beta \frac{\beta^\delta \sharp u}{\sharp t + \sharp u} \leq \varphi.t.u \leq \log_\alpha \frac{\sharp t + \sharp u}{\alpha^\gamma \sharp t}.$

The existence of such a φ is now guaranteed if for any positive m and n (writing $\varepsilon = \log_\alpha \beta$):

$$\begin{array}{ll} & \log_\beta \frac{\beta^\delta n}{m+n} \leq \log_\alpha \frac{m+n}{\alpha^\gamma m} \\ \equiv & \{\ \log_\alpha x = \log_\beta x^\varepsilon;\ \text{monotonicity of } \log_\beta\ (\beta > 1)\ \} \\ & \frac{\beta^\delta n}{m+n} \leq (\frac{m+n}{\alpha^\gamma m})^\varepsilon \\ \equiv & \{\ \alpha^\varepsilon = \beta \text{ and } \gamma + \delta = 1\ \} \\ & \beta \leq \frac{(m+n)^{\varepsilon+1}}{m^\varepsilon n} \\ \equiv & \{\ \} \\ & \beta \leq \frac{(1+\frac{m}{n})^{\varepsilon+1}}{(\frac{m}{n})^\varepsilon}. \end{array}$$

So, proceeding in terms of β and ε ($\alpha = \beta^{1/\varepsilon}$), it suffices to choose β and ε such that $\beta > 1$, $\varepsilon > 0$, and

$$(14)\quad \beta \leq \frac{(\varepsilon+1)^{\varepsilon+1}}{\varepsilon^{\varepsilon}},$$

since function $\frac{(1+x)^{\varepsilon+1}}{x^{\varepsilon}}$ is minimal at $x = \varepsilon$.

It is easily seen that such β and ε exist, but we want to choose them such that we obtain a small bound for $\mathcal{A}[\bowtie]$. We have

$$\begin{array}{ll} & \mathcal{A}[\bowtie](x,y) \\ = & \quad \{\ (9)\ \} \\ & \Gamma.(x \bowtie y) + \Delta.x + \Delta.y \\ \leq & \quad \{\ (12)\ \} \\ & \log_\alpha \sharp(x \bowtie y) + \log_\beta \sharp x + \log_\beta \sharp y \\ \leq & \quad \{\ \text{definition of } \sharp\ \} \\ & \log_\alpha(\sharp x + \sharp y) + 2\log_\beta(\sharp x + \sharp y) \\ = & \quad \{\ \log_2 \alpha = (\log_2 \beta)/\varepsilon\ \} \\ & \frac{\varepsilon+2}{\log_2 \beta}\ \log_2(\sharp x + \sharp y), \end{array}$$

and therefore we want to minimize $\frac{\varepsilon+2}{\log_2 \beta}$ under constraint (14) over all $\beta > 1$ and $\varepsilon > 0$. For this purpose we may take equality in (14), since, for fixed ε, $\frac{\varepsilon+2}{\log_2 \beta}$ decreases as β increases. Using this equality gives

$$\frac{\varepsilon+2}{(\varepsilon+1)\log_2(\varepsilon+1) - \varepsilon\log_2 \varepsilon}$$

as quantity to minimize over all positive ε. This gives $1/\log_2 \phi$ (≈ 1.44) as minimal value at $\varepsilon = \phi$ (≈ 1.618), where ϕ is the golden ratio. From the equality in (14) we get $\beta = \phi^{\phi+2}$ (≈ 5.70) and the definition of ε gives $\alpha = \phi^{2\phi-1}$ (≈ 2.93). In the same fashion as in the previous section, the following lemma now follows.

Lemma 9.2
Let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t \ +\ \log_\beta \frac{\beta\sharp u}{\sharp t+\sharp u} \max 0 \ +\ \Phi.u, \end{array}$$

with $\beta = \phi^{\phi+2}$. Then $0 \leq \Phi.z \leq \#z$, and the amortized costs of top-down melding satisfy

$$\mathcal{A}[\bowtie](x,y) \leq \log_\phi(\sharp x + \sharp y).$$

□

Since the above analysis does not leave much room for improvement, we conjecture that our bound for the amortized costs of $\bowtie$ is tight; that is, we conjecture $(\forall\, \eta : \eta < 1/\log_2 \frac{\sqrt{5}+1}{2} : (\exists\, x,y :: \mathcal{A}[\bowtie](x,y) > \eta\ \log_2(\sharp x + \sharp y)))$, for any potential Φ.

9.2.5 Results for priority queue operations

We have derived the following programs for the priority queue operations:

$$\begin{array}{lcl}
\mathsf{empty} & \doteq & \langle\,\rangle \\
\mathsf{isempty}.z & \doteq & z = \langle\,\rangle \\
\mathsf{single}.a & \doteq & \langle a \rangle \\
\mathsf{min}.\langle t, a, u\rangle & \doteq & a \\
\mathsf{union}.x.y & \doteq & x \bowtie y \\
\mathsf{delmin}.\langle t, a, u\rangle & \doteq & t \bowtie u.
\end{array}$$

The dots in these programs together with those in the program for $\bowtie$ define cost measure $\mathcal{T}$. As a result, we have that the amortized costs of empty, isempty, single, and min are $O(1)$. Furthermore, the amortized costs of union are logarithmic because $\mathcal{A}[\mathsf{union}](x, y) = 1 + \mathcal{A}[\bowtie](x, y)$, and for delmin we have:

$$\begin{array}{ll}
 & \mathcal{A}[\mathsf{delmin}](\langle t, a, u\rangle) \\
= & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
 & \mathcal{T}[\mathsf{delmin}](\langle t, a, u\rangle) + \Phi.(\mathsf{delmin}.\langle t, a, u\rangle) - \Phi.\langle t, a, u\rangle \\
= & \quad \{ \text{ definitions of } \mathsf{delmin},\ \mathcal{T}, \text{ and } \Phi \ \} \\
 & 1 + \mathcal{N}[\bowtie](t, u) + \Phi.(t \bowtie u) - \Phi.t - \varphi.t.u - \Phi.u \\
= & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
 & 1 + \mathcal{A}[\bowtie](t, u) - \varphi.t.u \\
\leq & \quad \{ \ \varphi \text{ is nonnegative } \} \\
 & 1 + \mathcal{A}[\bowtie](t, u).
\end{array}$$

Hence, $\mathcal{A}[\mathsf{delmin}](\langle t, a, u\rangle)$ is $O(\log \sharp\langle t, a, u\rangle)$. Note that such a conclusion can only be drawn if φ is bounded from below by a constant (e.g., if φ is nonnegative).

Thus—hiding the particular implementation of priority queue operations—we have as results for the amortized costs (cf. Lemma 9.2):

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{single}](a) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{union}](p, q) & \leq & 1.44 \log_2(2 + \#[\![p]\!] + \#[\![q]\!]) \text{ comparisons} \\
\mathcal{A}[\mathsf{isempty}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{min}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{delmin}](p) & \leq & 1.44 \log_2(1 + \#[\![p]\!]) \text{ comparisons.}
\end{array}$$

Moreover, there exists a potential function satisfying $0 \leq \Phi.p \leq \#[\![p]\!]$.

9.2.6 Comparison with Sleator and Tarjan's results

On page 54 in [30], Sleator and Tarjan describe top-down melding as follows: "we meld by merging the right paths of the two trees and then swapping the

left and right children of every node on the merge path *except the lowest*". Our program for $\bowtie$ is simpler because it does not have such a strange exception.

The potential function of Sleator and Tarjan for top-down melding is roughly given by

$$\varphi.t.u = \begin{cases} 0 & , \ \sharp u \leq \sharp t \\ 1 & , \ \sharp t < \sharp u \end{cases} \qquad (= \left\lceil \log_2 \frac{2\sharp u}{\sharp t + \sharp u} \right\rceil \max 0).$$

This potential satisfies bounds (11) if we take $\alpha = 2$, $\gamma = 0$, and $\delta = 1$. This potential is simpler, but the corresponding bound for $\mathcal{A}[\bowtie]$ is more than twice as large:

$$\mathcal{A}[\bowtie](x, y) \leq 3 \log_2(\sharp x + \sharp y).$$

Sleator and Tarjan also discussed a "lazy" version of top-down melding that terminates as soon as one of the heaps is empty. This corresponds to the following version for $\bowtie$, which also follows from our derivation in Section 9.2.2:

$$\begin{array}{rcll} \langle\,\rangle \bowtie y & = & y & \\ x \bowtie \langle\,\rangle & = & x & \\ \langle t, a, u\rangle \bowtie y & \doteq & \langle u \bowtie y, a, t\rangle & , \ y \neq \langle\,\rangle \ \wedge \ a \leq m.y \\ x \bowtie \langle t, a, u\rangle & \doteq & \langle u \bowtie x, a, t\rangle & , \ x \neq \langle\,\rangle \ \wedge \ a \leq m.x, \end{array}$$

where m is only used for nonempty trees.

For this implementation of $\bowtie$ one can derive that $\mathcal{A}[\bowtie](\langle\,\rangle, y) = 0 \leq \Gamma.(\langle\,\rangle \bowtie y) + \Delta.\langle\,\rangle + \Delta.y$, provided that Γ and Δ are nonnegative. Since $\gamma = 0$, $\delta = 1$, and $0 \leq \varphi.t.u \leq 1$, functions Γ and Δ are indeed nonnegative. By induction it now follows that (cf. (9))

$$\mathcal{A}[\bowtie](x, y) \leq \Gamma.(x \bowtie y) + \Delta.x + \Delta.y,$$

from which we infer that this alternative program for $\bowtie$ is at least as good the previous one. However, for a decreasing list s we have that $\mathcal{N}[h](s)$ is $\Omega(\#s \log \#s)$ for function h defined in Section 9.2.2, so the amortized cost of $\bowtie$ is still logarithmic. We conjecture that $1.44 \log_2(\sharp x + \sharp y)$ is also a tight bound for $\mathcal{A}[\bowtie](x, y)$ in this case .

9.3 Intermezzo on sorting

As illustration of the usage of skew heaps, we analyze to what extent sorting can be done efficiently using skew heaps. One way is to use top-down skew heaps in program *sort*1 (see Section 9.1); then computation of *sort*1.s takes $2.88\#s \log_2 \#s + O(\#s)$ comparisons in the worst-case. However, with a more sophisticated implementation of function f in *sort*1 we obtain a sorting program that uses at most $1.44\#s \log_2 \#s + O(\#s)$ comparisons. The idea is to see to it that in each application of union only priority queues of about equal size are united, so that computation of $f.s$ takes $O(\#s)$ time only.

$$
\begin{array}{lcl}
PQsort & = & g \circ f \\
 & & |[f = f_1 \circ f_0 \\
 & & \quad |[\ f_0.[\,] \quad = \quad [\,] \\
 & & \quad\ \ f_0.(a \vdash r) \ = \ f_0.r \dashv \mathsf{single}.a \\
 & & \quad\ \ f_1.[\,] \quad = \quad \mathsf{empty} \\
 & & \quad\ \ f_1.[p] \quad = \quad p \\
 & & \quad\ \ f_1.(p \vdash q \vdash ps) \ = \ f_1.(ps \dashv \mathsf{union}.p.q) \\
 & & \quad]| \\
 & & g.p \ = \ [\,] \quad\quad\quad\quad\quad\quad\quad , \ \mathsf{isempty}.p \\
 & & \quad\ \ [] \ \ \mathsf{min}.p \vdash g.(\mathsf{delmin}.p) \ , \neg \mathsf{isempty}.p \\
 & &]|.
\end{array}
$$

Remark 9.3
Of course, *PQsort* is not intended as a practical application of mergeable priority queues. If it is only for the above program, it is better to take the set of ascending lists of integers as definition for S in Section 9.2.1. Implementing $\mathsf{union}.p.q$ as the merge of lists p and q and implementing the other operations in the obvious way, the above program then reduces to *Mergesort* (see Example 4.4), which takes at most $\#s \log_2 \#s$ comparisons. (By the way, merging ascending lists may be considered as a special case of melding heaps. This is reflected by the following program for merging:

$$
\begin{array}{rcll}
[\,] \bowtie [\,] & = & [\,] & \\
(a \vdash u) \bowtie y & = & a \vdash u \bowtie y & , a \leq m.y \\
x \bowtie (a \vdash u) & = & a \vdash u \bowtie x & , a \leq m.x,
\end{array}
$$

where $m.[\,] = \infty$ and $m.(a \vdash u) = a$. To compute $x \bowtie y$, however, this program takes $O(\#x + \#y)$ time instead of logarithmic time for top-down skew heaps.) □

It is not trivial that $\mathcal{A}[f](s)$, which equals $\mathcal{T}[f](s) + \Phi.(f.s)$, is linear in $\#s$. To show this formally, we introduce amortized costs for the components of f:

$$
\begin{array}{lcl}
\mathcal{A}[f_0](s) & = & \mathcal{T}[f_0](s) + \Phi^*.(f_0.s) \\
\mathcal{A}[f_1](ps) & = & \mathcal{T}[f_1](ps) + \Phi.(f_1.ps) - \Phi^*.ps,
\end{array}
$$

such that $\mathcal{A}[f](s) = \mathcal{A}[f_0](s) + \mathcal{A}[f_1](f_0.s)$. Here, potential Φ^* is defined for lists of priority queues by $\Phi^*.[\,] = 0$ and $\Phi^*.(p \vdash ps) = \Phi.p + \Phi^*.ps$.

Since it is easy to show that $\mathcal{A}[f_0](s)$ is linear in $\#s$, we focus our attention on $\mathcal{A}[f_1]$, for which we have the following recurrence relation:

$$
\begin{array}{lcl}
\mathcal{A}[f_1]([\,]) & = & \mathcal{A}[\mathsf{empty}] \\
\mathcal{A}[f_1]([p]) & = & 0 \\
\mathcal{A}[f_1](p \vdash q \vdash ps) & = & \mathcal{A}[\mathsf{union}](p, q) + \mathcal{A}[f_1](ps \dashv \mathsf{union}.p.q).
\end{array}
$$

In order that $\mathcal{A}[f_1](s)$ is linear, it is important that in each application of union only priority queues of about equal sizes are united (more precisely, they differ at most a factor of two in size). This fact is exploited in the following lemma.

Lemma 9.4
For all natural k and l,

$$\mathcal{A}[f_1](ps) \leq \mathcal{A}[\mathsf{empty}] + 1.44\ 2^k\ (\Sigma\, i : 1 \leq i \leq k : (l+1+i)/2^i),$$

provided ps satisfies:

(i) $\#ps \leq 2^k$
(ii) $(\forall\, i : 0 \leq i < \#ps : \#[\![ps.i]\!] \leq 2^l)$.

Proof By induction on k.

Case $k = 0$. Then $\mathcal{A}[f_1](ps)$ is either $\mathcal{A}[\mathsf{empty}]$ or 0, hence at most $\mathcal{A}[\mathsf{empty}]$.

Case $k = m{+}1$. We consider $\lfloor \#ps/2 \rfloor$ successive unfoldings of f_1. The result of these unfoldings is called qs, and may be defined as follows. In case $\#ps$ is even, list qs satisfies $\#qs = \#ps/2$, and $qs.i = \mathsf{union}.(ps.(2i)).(ps.(2i{+}1))$ for $0{\leq}i{<}\#qs$. In case $\#ps$ is odd, $\#qs = (\#ps{+}1)/2$, $qs.0 = ps.(\#ps{-}1)$, and $qs.(i{+}1) = \mathsf{union}.(ps.(2i)).(ps.(2i{+}1))$ for $0{\leq}i{<}\#qs{-}1$.

Now we may apply the induction hypothesis to qs, since $\#qs \leq 2^m$ (cf. (i)) and $\#[\![qs.i]\!] \leq 2^{l+1}$ (cf. (ii)):

$$\begin{array}{ll}
& \mathcal{A}[f_1](ps) \\
= & \quad \{\ \text{unfold recurrence relation for } \mathcal{A}[f_1]\ \lfloor \#ps/2 \rfloor \text{ times}\ \} \\
& (\Sigma\, i : 0 \leq i < \lfloor \#ps/2 \rfloor : \mathcal{A}[\mathsf{union}](ps.(2i), ps.(2i{+}1))) + \mathcal{A}[f_1](qs) \\
\leq & \quad \{\ \text{bound for } \mathcal{A}[\mathsf{union}], \text{ see Section 9.2.5}\ \} \\
& (\Sigma\, i : 0 \leq i < \lfloor \#ps/2 \rfloor : 1.44\log_2(2{+}\#[\![ps.(2i)]\!]{+}\#[\![ps.(2i{+}1]\!])){+}\mathcal{A}[f_1](qs) \\
\leq & \quad \{\ \text{(ii) and } l \geq 0;\ \lfloor \#ps/2 \rfloor \leq 2^m\ \} \\
& 1.44\ 2^m(l{+}2) + \mathcal{A}[f_1](qs) \\
\leq & \quad \{\ \text{induction hypothesis}\ \} \\
& 1.44\ 2^m(l{+}2) + \mathcal{A}[\mathsf{empty}] + 1.44\ 2^m\ (\Sigma\, i : 1 \leq i \leq m : (l{+}2{+}i)/2^i) \\
= & \quad \{\ \text{calculus}\ \} \\
& \mathcal{A}[\mathsf{empty}] + 1.44\ 2^{m+1}\ (\Sigma\, i : 0 \leq i \leq m : (l{+}1{+}i{+}1)/2^{i+1}) \\
= & \quad \{\ k = m{+}1\ \} \\
& \mathcal{A}[\mathsf{empty}] + 1.44\ 2^k\ (\Sigma\, i : 1 \leq i \leq k : (l{+}1{+}i)/2^i).
\end{array}$$

□

Instantiation of this lemma with $k = \lceil \log_2(\#s{+}1) \rceil$, $l = 0$, and $ps = f_0.s$ yields, using $\#(f_0.s) = \#s$:

$$\mathcal{A}[f_1](f_0.s) \leq \mathcal{A}[\mathsf{empty}] + 2.88(\#s{+}1)\ (\Sigma\, i : 1 \leq i \leq \lceil \log_2(\#s{+}1) \rceil : \frac{i{+}1}{2^i}).$$

This implies that $\mathcal{A}[f_1](f_0.s)$ is $O(\#s)$, since the summation has the same order of magnitude as $\int_0^\infty \frac{x+1}{2^x} dx$, which is $O(1)$. Hence, we have shown that $\mathcal{A}[f](s)$ is linear in s (and also that $\mathcal{T}[f](s)$ is linear).

9.4 Bottom-up skew heaps

Reconsidering the derivation of the top-down skew heaps we conclude that the decision to use binary trees as representations for bags is in fact the main design decision. Keeping matters as simple as possible, the introduction of the heap condition and the operator $\bowtie$ followed in a natural way. The implementations derived for $\bowtie$ both have logarithmic amortized costs and there seems to be no way to improve this. In [30], however, Sleator and Tarjan present the bottom-up skew heaps as alternative implementation of priority queues. For bottom-up skew heaps they achieve a logarithmic bound for the amortized costs of delmin and $O(1)$ bounds for the remaining operations; hence, the amortized costs of union are reduced to $O(1)$. How come?

Well, Sleator and Tarjan still use binary trees to represent bags, but—formally speaking—they take a different view of binary trees. In the previous section we have introduced set $\langle \mathrm{Int} \rangle$ in the common way, viz. as the smallest solution of

$$X: \quad X = \{\langle\,\rangle\} \;\cup\; X \times \mathrm{Int} \times X.$$

This "top-down" view of binary trees forces us to dissect a nonempty tree into its left subtree, its root, and its right subtree, and in this way we are almost inescapably led to the programs found for $\bowtie$. It appears that the way one introduces binary trees constitutes an important design decision, for we have to take a special view of binary trees to arrive at the bottom-up skew heaps.

In case of bottom-up skew heaps the following definition is appropriate. We define set $\langle \mathrm{Int} \rangle_1$ as the smallest solution of

$$X: \quad X = [X \times \mathrm{Int}].$$

We have already described this tree type in Section 6.2.2, from which we recall that elements of $\langle \mathrm{Int} \rangle_1$ are interpreted as binary trees in the following way: $[\,]$ stands for the empty tree and $\langle t, a \rangle \vdash u$ for a nonempty tree with left subtree t, root a, and right subtree u. In addition, we will use that nonempty elements of $\langle \mathrm{Int} \rangle_1$ can be dissected as $u \dashv \langle t, a \rangle$ (where a is the last element of the inorder traversal of this tree). For the programs in the sequel the following algebra is appropriate:

$$(\, \langle \mathrm{Int} \rangle_1 \;\mid\; [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, lt, ft \,).$$

As shown in Section 6.2.2, there exists a destructive implementation for this algebra that supports all operations in $O(1)$ time.

Since there is a clear correspondence between $\langle \mathrm{Int} \rangle$ and $\langle \mathrm{Int} \rangle_1$, we omit the definitions of functions like $[\![\cdot]\!]$, m, and H for the latter type. However, doing the same for $\#$ would make $\#z$ ambiguous (for $z \in \langle \mathrm{Int} \rangle_1$). We resolve this ambiguity by interpreting $\#z$ as the length of *list* z. To denote the size of the tree associated with z we write $\#[\![z]\!]$.

To obtain a definition of SH, we simply write the definitions for top-down skew heaps in terms of type $\langle \mathrm{Int} \rangle_1$. This gives for S:

$$S = \{z \mid z \in \langle \text{Int} \rangle_1 \ \wedge \ H.z\},$$

and for the priority queue operations:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & [\,] \\ \mathsf{isempty}.z & \doteq & z = [\,] \\ \mathsf{single}.a & \doteq & [\langle [\,], a\rangle] \\ \mathsf{min}.(\langle t, a\rangle \vdash u) & \doteq & a \\ \mathsf{union}.x.y & \doteq & x \between y \\ \mathsf{delmin}.(\langle t, a\rangle \vdash u) & \doteq & t \between u, \end{array}$$

where $\between$ ("bottom-up meld") is specified in the same way as $\bowtie$: $\between \in S \times S \to S$ and $[\![x \between y]\!] = [\![x]\!] \oplus [\![y]\!]$. We have introduced $\between$ to distinguish top-down and bottom-up implementations of melding in the sequel.

9.4.1 Implementation of $\between$

In this section we construct—in two steps—a bottom-up version of melding which is almost the same as the algorithm described in [30, p.62].

In the first step we turn the (top-down) program for $\bowtie$ into a tail-recursive (bottom-up) program. This transformation bears resemblance to the transformation prescribed by the following variation on well-known "tail-recursion theorems" from functional programming (see, e.g., [17, Theorem 6.2.0]).

Theorem 9.5
If function g is of the form

$$\begin{array}{lcl} g.[\,] & = & X \\ g.(a \vdash s) & = & G.a.(g.s) \end{array}$$

and function h is defined as

$$\begin{array}{lcl} h.[\,].x & = & x \\ h.(s \dashv a).x & = & h.s.(G.a.x), \end{array}$$

then

$$h.s.X = g.s$$

for all lists s.
□

In this theorem, function g is linearly recursive whereas function h is tail recursive.

In terms of type $\langle \text{Int} \rangle_1$, the program for $\bowtie$ is:

$$\begin{array}{rcll} [\,] \bowtie [\,] & = & [\,] & \\ (\langle t, a\rangle \vdash u) \bowtie y & \doteq & \langle u \bowtie y, a\rangle \vdash t & , a \le m.y \\ x \bowtie (\langle t, a\rangle \vdash u) & \doteq & \langle u \bowtie x, a\rangle \vdash t & , a \le m.x. \end{array}$$

As in Theorem 9.5, we introduce a function F, say, with one more parameter than $\bowtie$, and we use a recursion pattern based on $\dashv$ instead of $\vdash$ to define F. Then $x \bowtie y = F.x.y.[\,]$ (see after Lemma 9.6), where

$$\begin{array}{lcll} F.[\,].[\,].z & = & z & \\ F.(u \dashv \langle t, a\rangle).y.z & \doteq & F.u.y.(\langle z, a\rangle \vdash t) & , a \ge M.y \\ F.x.(u \dashv \langle t, a\rangle).z & \doteq & F.u.x.(\langle z, a\rangle \vdash t) & , a \ge M.x, \end{array}$$

with M defined by

$$\begin{array}{lcl} M.[\,] & = & -\infty \\ M.(u \dashv \langle t, a\rangle) & = & a. \end{array}$$

(Note that, in general, $M.z$ differs from the maximum of $[\![z]\!]$, for heap z.) The relation between F and $\bowtie$ is expressed in Lemma 9.6, in which we use the following characterization of H for $\langle \mathrm{Int}\rangle_1$-trees:

$$(15) \quad \begin{array}{lcl} H.[\,] & \equiv & \mathrm{true} \\ H.(u \dashv \langle t, a\rangle) & \equiv & H.u \ \wedge \ M.u \le a \ \wedge \ a \le m.t \ \wedge \ H.t. \end{array}$$

Lemma 9.6
For all *heaps* x, y, v, and w satisfying $M.x \max M.y \le m.v \min m.w$:

$$F.x.y.(v \bowtie w) = (x +\!\!+ v) \bowtie (y +\!\!+ w).$$

Proof Note that the premiss implies that $x +\!\!+ v$ and $y +\!\!+ w$ are heaps. The proof proceeds by induction on x and y, using the following exhaustive case analysis.

Case $x = [\,] \ \wedge \ y = [\,]$. By the definition of F, $F.[\,].[\,].(v \bowtie w) = v \bowtie w$.

Case $x = u \dashv \langle t, a\rangle \ \wedge \ a \ge M.y$. Then $M.x \max M.y = a$, and for all heaps v and w satisfying $a \le m.v \ \wedge \ a \le m.w$ we have:

$$\begin{array}{ll} & F.(u \dashv \langle t, a\rangle).y.(v \bowtie w) \\ = & \quad \{ \text{ definition of } F \ (a \ge M.y) \ \} \\ & F.u.y.(\langle v \bowtie w, a\rangle \vdash t) \\ = & \quad \{ \text{ definition of } \bowtie \ (a \le m.w) \ \} \\ & F.u.y.((\langle t, a\rangle \vdash v) \bowtie w) \\ = & \quad \{ \text{ induction hypothesis } (M.u \max M.y \le a \min m.w, \text{ see below}) \ \} \\ & (u +\!\!+ (\langle t, a\rangle \vdash v)) \bowtie (y +\!\!+ w) \\ = & \quad \{ \text{ list calculus } \} \\ & ((u \dashv \langle t, a\rangle) +\!\!+ v) \bowtie (y +\!\!+ w). \end{array}$$

Since x is a heap, we have $H.u \ \wedge \ M.u \leq a \ \wedge \ a \leq m.t \ \wedge \ H.t$ (cf. (15)). In conjunction with $a \leq m.v \ \wedge \ H.v$, this implies $H.(\langle t, a\rangle \vdash v)$.

Case $y = u \dashv \langle t, a\rangle \ \wedge \ a \geq M.x$. This case follows by symmetry.
□

Instantiation of this lemma with $v = [\,]$ and $w = [\,]$ gives $F.x.y.[\,] = x \bowtie y$. Furthermore, we have $\mathcal{N}[F](x, y, [\,]) = \mathcal{N}[\bowtie](x, y)$, where $\mathcal{N}$ denotes the cost measure for the above programs. So, the introduction of F does not give an improvement. However, we are now ready for our second step in which we extend the base case for $F.x.y.z$ from $x = [\,] \ \wedge \ y = [\,]$ to $x = [\,] \ \vee \ y = [\,]$. As a result, we will obtain a program for $\between$ with $O(1)$ amortized costs, whereas the same extension applied to the program for $\bowtie$ gives no essential reduction (see Section 9.2.6).

We observe that $M.x \max M.y \leq m.z$ holds as precondition for all (recursive) applications $F.x.y.z$ generated during the evaluation of $F.x.y.[\,]$. For $x = [\,]$, this is equivalent to $M.y \leq m.z$, and consequently, $y +\!\!+ z$ is a heap. Since we assume that $+\!\!+$ takes $O(1)$ time for $\langle \mathrm{Int}\rangle_1$-trees, we simply take this for $F.[\,].y.z$. By symmetry, we take $F.x.[\,].z = x +\!\!+ z$, and we obtain as program for $\between$:

$$
\begin{array}{l}
x \between y = F.x.y.[\,] \\
\qquad |[\ \begin{array}[t]{lcll}
F.[\,].y.z & = & y +\!\!+ z & \\
F.(u \dashv \langle t, a\rangle).y.z & \doteq & F.u.y.(\langle z, a\rangle \vdash t) & , a \geq M.y \ \wedge \ y \neq [\,] \\
F.x.[\,].z & = & x +\!\!+ z & \\
F.x.(u \dashv \langle t, a\rangle).z & \doteq & F.u.x.(\langle z, a\rangle \vdash t) & , a \geq M.x \ \wedge \ x \neq [\,]
\end{array} \\
\qquad]|.
\end{array}
$$

In this program, M is used for nonempty trees only.

Compared to the program for $\bowtie$, the program for $\between$ is more intricate and so will be its analysis. In a first, exploratory analysis we will strive for an $O(1)$ bound for the amortized costs of $\between$. Unfortunately, this results in a linear bound for delmin, which is not what we are after. In Section 9.4.3 we will therefore combine the result of the next section with our approach to top-down skew heaps so as to obtain a logarithmic bound for delmin.

9.4.2 Bottom-up analysis of $\between$

As before, we use $\mathcal{N}$ to denote the cost measure defined by the dots in the above program. This measure exactly counts the number of comparisons. The amortized costs for $\between$ are defined as usual.

$$\mathcal{A}[\between](x, y) = \mathcal{N}[\between](x, y) + \Phi.(x \between y) - \Phi.x - \Phi.y.$$

However, the pattern of the definition of the amortized costs for F, given by

$$\mathcal{A}[F](x, y, z) = \mathcal{N}[F](x, y, z) + \Phi.(F.x.y.z) - \Phi.x - \Phi.y,$$

deviates from the general pattern used thus far. We have omitted "$-\Phi.z$", for otherwise it is impossible—as far as we know—to define Φ such that $\mathcal{A}[F]$ is $O(1)$. As a possible explanation, we remark that we use F in such a way that z is merely an accumulation parameter, whereas x and y are input parameters and $F.x.y.z$ is the result.

From the program for $\mathbb{X}$, it is immediate that $\mathcal{A}[\mathbb{X}](x,y) = \mathcal{A}[F](x,y,[\,])$. Our goal is now to define Φ such that $\mathcal{A}[F]$ is $O(1)$. To achieve this, we change the above program for F a little bit because an analysis of this program along the same lines as below leads to a potential function for which $\mathcal{A}[F]$ is not $O(1)$. We shall therefore study the following equivalent program for F, which we have obtained by singling out the cases in which one of the first two arguments is a singleton list:

$$\begin{array}{lcll} F.[\,].y.[\,] & = & y & \\ F.[\langle t,a\rangle].y.z & \doteq & y +\!\!+ (\langle z,a\rangle \vdash t) & ,a \geq M.y \ \wedge \ y \neq [\,] \\ F.(u \dashv \langle t,a\rangle).y.z & \doteq & F.u.y.(\langle z,a\rangle \vdash t) & ,a \geq M.y \ \wedge \ y \neq [\,] \ \wedge \ u \neq [\,]. \end{array}$$

The symmetric counterparts of these alternatives are omitted.

We first consider the recursive case for F, since this puts the strongest requirements on Φ ($u \neq [\,]$):

$$\begin{array}{ll} & \mathcal{A}[F](u \dashv \langle t,a\rangle, y, z) \\ = & \quad \{ \text{ definition of } \mathcal{A} \ \} \\ & \mathcal{N}[F](u \dashv \langle t,a\rangle, y, z) + \Phi.(F.(u \dashv \langle t,a\rangle).y.z) - \Phi.(u \dashv \langle t,a\rangle) - \Phi.y \\ = & \quad \{ \text{ definitions of } F \text{ and } \mathcal{N} \ \} \\ & 1 + \mathcal{N}[F](u, y, \langle z,a\rangle \vdash t) + \Phi.(F.u.y.(\langle z,a\rangle \vdash t)) - \Phi.(u \dashv \langle t,a\rangle) - \Phi.y \\ = & \quad \{ \text{ definition of } \mathcal{A} \ \} \\ & \mathcal{A}[F](u, y, \langle z,a\rangle \vdash t) + 1 + \Phi.u - \Phi.(u \dashv \langle t,a\rangle). \end{array}$$

In order to obtain an $O(1)$ bound for $\mathcal{A}[F]$ we define Φ such that $1 + \Phi.u - \Phi.(u \dashv \langle t,a\rangle) \leq 0$. To keep Φ as small as possible, we even take

$$\Phi.(u \dashv \langle t,a\rangle) = \Phi.u + 1, \tag{16}$$

for $u \neq [\,]$.

Taking $\Phi.[\,] = 0$ gives $\mathcal{A}[F]([\,], y, [\,]) = 0$, and for the remaining nonrecursive case we derive ($y \neq [\,]$):

$$\begin{array}{ll} & \mathcal{A}[F]([\langle t,a\rangle], y, z) \\ = & \quad \{ \text{ definition of } \mathcal{A} \ \} \\ & \mathcal{N}[F]([\langle t,a\rangle], y, z) + \Phi.(F.[\langle t,a\rangle].y.z) - \Phi.[\langle t,a\rangle] - \Phi.y \\ = & \quad \{ \text{ definition of } F \text{ and } \mathcal{N} \ \} \\ & 1 + \Phi.(y +\!\!+ (\langle z,a\rangle \vdash t)) - \Phi.[\langle t,a\rangle] - \Phi.y \\ = & \quad \{ \ (16) \text{ implies } \Phi.(y +\!\!+ (\langle z,a\rangle \vdash t)) = \Phi.y + 1 + \#t, \text{ for } y \neq [\,] \ \} \end{array}$$

$$
\begin{array}{ll}
 & 1 + 1 + \#t - \Phi.[\langle t, a\rangle] \\
= & \quad \{ \text{ take } \Phi.[\langle t, a\rangle] = \#t + 1 \text{ (see definition } \Phi \text{ below) } \} \\
 & 1.
\end{array}
$$

Hence, $\mathcal{A}[F](x, y, z) \leq 1$, if we define Φ as follows:

$$
\begin{array}{lcl}
\Phi.[\,] & = & 0 \\
\Phi.(\langle t, a\rangle \vdash u) & = & \#t + 1 + \#u,
\end{array}
$$

and, consequently, $\mathcal{A}[\,\between\,](x, y) \leq 1$. (We leave it to the reader to ascertain that a similar analysis of the program derived in Section 9.4.1 leads to potential $\Phi.z = \#z$, for which $\mathcal{A}[F]$ is not $O(1)$.)

Remark 9.7
The above derived potential Φ is reminiscent of the potential used by Sleator and Tarjan in their analysis of bottom-up skew heaps. In their terminology, $\Phi.x$ equals the sum of the lengths of the major path and the minor path of x. (The *major path* of x is the rightmost path of x, and the *minor path* of x is the rightmost path of x's left subtree)
□

With $\mathcal{T}$ as defined for top-down skew heaps, we now have $O(1)$ bounds for all priority queue operations, except that for delmin:

$$
\begin{array}{ll}
 & \mathcal{A}[\mathsf{delmin}](\langle t, a\rangle \vdash u) \\
= & \quad \{ \text{ definition of } \mathcal{A} \, \} \\
 & \mathcal{T}[\mathsf{delmin}](\langle t, a\rangle \vdash u) + \Phi.(\mathsf{delmin}.(\langle t, a\rangle \vdash u)) - \Phi.(\langle t, a\rangle \vdash u) \\
= & \quad \{ \text{ definitions of } \mathsf{delmin} \text{ and } \Phi \, \} \\
 & 1 + \mathcal{N}[\,\between\,](t, u) + \Phi.(t \between u) - \#t - 1 - \#u \\
= & \quad \{ \text{ definition of } \mathcal{A} \, \} \\
 & \mathcal{A}[\,\between\,](t, u) + \Phi.t - \#t + \Phi.u - \#u \\
\leq & \quad \{ \text{ bound for } \mathcal{A}[\,\between\,] \, \} \\
 & 1 + \Phi.t - \#t + \Phi.u - \#u.
\end{array}
$$

Unfortunately, $\Phi.z - \#z$ may be linear in the size of z, and therefore we can only conclude that $\mathcal{A}[\mathsf{delmin}](z)$ is linear in the size of z.

9.4.3 First top-down analysis of $\between$

To obtain a logarithmic bound for the amortized costs of delmin we analyze yet another (equivalent) program for $\between$. Inspired by the potential function of Sleator and Tarjan for bottom-up skew heaps, we take a potential function of the following form:

$$
\begin{array}{lcl}
\Phi.\langle\,\rangle & = & 0 \\
\Phi.\langle t, a, u\rangle & = & \Psi.\langle t, a, u\rangle + \Pi.t + \Pi.\langle t, a, u\rangle,
\end{array}
$$

with

$$\begin{aligned}\Psi.\langle\,\rangle &= 0\\ \Psi.\langle t,a,u\rangle &= \Psi.t+\psi.t.u+\Psi.u,\end{aligned}$$

and

$$\begin{aligned}\Pi.\langle\,\rangle &= 0\\ \Pi.\langle t,a,u\rangle &= \pi.t.u+\Pi.u.\end{aligned}$$

The form of Φ may be considered as a mixture of the form of the potential function for top-down skew heaps (Ψ) and the form of the potential function found in the previous section (Π, twice).

Remark 9.8 (see Remark 9.7)
In the above form for Φ, the nodes on the major path and those on the minor path are treated as special—as is also done in [30]. The term $\Pi.\langle t,a,u\rangle$ corresponds to the major path and $\Pi.t$ to the minor path.
□

Our analysis of top-down skew heaps proceeded smoothly because the recursion pattern in the program of $\bowtie$ matched the recursive definition of the potential so well. In view of the above definition of Φ we therefore decide to construct a top-down *simulation* (read "algorithmic refinement") of $\between$. This simulation is based on the observation that $x \between y = t +\!\!+ (u \bowtie y)$ if $m.x \leq m.y$, where t and u satisfy $t +\!\!+ u = x$ and $M.t \leq m.y \leq m.u$. Phrased differently, $x \between y$ can be computed in two stages: first compute t and then compute $u \bowtie y$. This is encoded in the following program:

$$\begin{aligned}\langle\,\rangle \between \langle\,\rangle &= \langle\,\rangle\\ \langle t,a,u\rangle \between y &= \langle t,a,u \lhd y\rangle &&, a \leq m.y\\ x \between \langle t,a,u\rangle &= \langle t,a,u \lhd x\rangle &&, a \leq m.x,\end{aligned}$$

where

$$\begin{aligned}\langle\,\rangle \lhd \langle\,\rangle &= \langle\,\rangle\\ \langle t,a,u\rangle \lhd y &= \langle t,a,u \lhd y\rangle &&, a \leq m.y\\ x \lhd \langle t,a,u\rangle &\doteq \langle u \bowtie x,a,t\rangle &&, a \leq m.x,\end{aligned}$$

and where

$$\begin{aligned}\langle\,\rangle \bowtie \langle\,\rangle &= \langle\,\rangle\\ \langle t,a,u\rangle \bowtie y &\doteq \langle u \bowtie y,a,t\rangle &&, a \leq m.y\\ x \bowtie \langle t,a,u\rangle &\doteq \langle u \bowtie x,a,t\rangle &&, a \leq m.x.\end{aligned}$$

Note that some of the above definitions are longer than necessary (e.g., the definition of $\between$). The uniformity of the above definitions, however, turns out to be the basis for a smooth amortized analysis. The cost measure $\mathcal{N}$ for the above

program has been chosen such that it coincides with the cost measure for the bottom-up program for $\between$.

To facilitate the calculations below, we introduce functions Θ and θ, defined by $\Theta.z = \Psi.z + \Pi.z$ and $\theta.t.u = \psi.t.u + \pi.t.u$. Then

$$\begin{array}{lcl} \Theta.\langle\,\rangle & = & 0 \\ \Theta.\langle t, a, u\rangle & = & \Psi.t + \theta.t.u + \Theta.u, \end{array}$$

and

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Theta.t + \theta.t.u + \Theta.u, \end{array}$$

as a consequence of which Π and π become superfluous. Recall that the amortized costs of $\between$ are defined as

$$\mathcal{A}[\between](x, y) = \mathcal{N}[\between](x, y) + \Phi.(x \between y) - \Phi.x - \Phi.y.$$

In what follows, we will derive suitable definitions for $\mathcal{A}[\lhd]$ and $\mathcal{A}[\bowtie]$.

Six requirements on θ and ψ

In order that Φ is nonnegative and small, we require that θ and ψ satisfy

$$\begin{array}{ll} \text{(i)} & 0 \le \theta.t.u \\ \text{(ii)} & 0 \le \psi.t.u, \end{array}$$

and also that θ and ψ are small. In the sequel we will accumulate further requirements on θ and ψ that ensure that $\mathcal{A}[\between]$ is $O(1)$ and $\mathcal{A}[\mathsf{delmin}]$ is logarithmic.

For $\mathcal{A}[\between]$ we have $\mathcal{A}[\between](\langle\,\rangle, \langle\,\rangle) = 0$, and in case $x \between y = \langle t, a, u \lhd y\rangle$, we derive:

$$\begin{array}{ll} & \mathcal{A}[\between](x, y) \\ = & \quad \{ \text{ definition of } \mathcal{A}[\between] \ \} \\ & \mathcal{N}[\between](x, y) + \Phi.(x \between y) - \Phi.x - \Phi.y \\ = & \quad \{ \text{ definitions of } \mathcal{N} \text{ and } \Phi \ (x = \langle t, a, u\rangle) \ \} \\ & \mathcal{N}[\lhd](u, y) + \Theta.t + \theta.t.(u \lhd y) + \Theta.(u \lhd y) - \Theta.t - \theta.t.u - \Theta.u - \Phi.y \\ = & \quad \{ \text{ definition of } \mathcal{A}[\lhd] \text{ (see below) } \} \\ & \mathcal{A}[\lhd](u, y) + \theta.t.(u \lhd y) - \theta.t.u. \end{array}$$

The appropriate definition for the amortized cost of $\lhd$ is thus given by

$$\mathcal{A}[\lhd](x, y) = \mathcal{N}[\lhd](x, y) + \Theta.(x \lhd y) - \Theta.x - \Phi.y.$$

Simply turning the handle gives $\mathcal{A}[\lhd](\langle\,\rangle, \langle\,\rangle) = 0$, and for the case $x \lhd y = \langle t, a, u \lhd y\rangle$:

$$\mathcal{A}[\lhd](x,y) = \mathcal{A}[\lhd](u,y) + \theta.t.(u \lhd y) - \theta.t.u.$$

In case $x \lhd y = \langle u \bowtie x, a, t\rangle$, we derive the definition of $\mathcal{A}[\bowtie]$ as follows:

$$\begin{array}{ll} & \mathcal{A}[\lhd](x,y) \\ = & \quad \{ \text{ definition of } \mathcal{A}[\lhd] \ \} \\ & \mathcal{N}[\lhd](x,y) + \Theta.(x \lhd y) - \Theta.x - \Phi.y \\ = & \quad \{ \text{ definitions of } \mathcal{N} \text{ and } \Theta \ (y = \langle t,a,u\rangle) \ \} \\ & 1 + \mathcal{N}[\bowtie](u,x) + \Psi.(u \bowtie x) + \theta.(u \bowtie x).t + \Theta.t - \Theta.x - \Theta.t - \theta.t.u - \Theta.u \\ = & \quad \{ \text{ definition of } \mathcal{A}[\bowtie] \text{ (see below) } \} \\ & \mathcal{A}[\bowtie](u,x) + 1 + \theta.(u \bowtie x).t - \theta.t.u, \end{array}$$

where the amortized cost of $\bowtie$ is defined by

$$\mathcal{A}[\bowtie](x,y) = \mathcal{N}[\bowtie](x,y) + \Psi.(x \bowtie y) - \Theta.x - \Theta.y.$$

Omitting the calculations for $\mathcal{A}[\bowtie]$, we thus obtain as recurrence relations:

$$\begin{array}{lcll} \mathcal{A}[\,\barwedge\,](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\ \mathcal{A}[\,\barwedge\,](\langle t,a,u\rangle, y) & = & \mathcal{A}[\lhd](u,y) + \theta.t.(u \lhd y) - \theta.t.u & , a \le m.y \\ \mathcal{A}[\,\barwedge\,](x, \langle t,a,u\rangle) & = & \mathcal{A}[\lhd](u,x) + \theta.t.(u \lhd x) - \theta.t.u & , a \le m.x \\ & & & \\ \mathcal{A}[\lhd](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\ \mathcal{A}[\lhd](\langle t,a,u\rangle, y) & = & \mathcal{A}[\lhd](u,y) + \theta.t.(u \lhd y) - \theta.t.u & , a \le m.y \\ \mathcal{A}[\lhd](x, \langle t,a,u\rangle) & = & \mathcal{A}[\bowtie](u,x) + 1 + \theta.(u \bowtie x).t - \theta.t.u & , a \le m.x \\ & & & \\ \mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\ \mathcal{A}[\bowtie](\langle t,a,u\rangle, y) & = & \mathcal{A}[\bowtie](u,y) + 1 + \psi.(u \bowtie y).t - \theta.t.u & , a \le m.y \\ \mathcal{A}[\bowtie](x, \langle t,a,u\rangle) & = & \mathcal{A}[\bowtie](u,x) + 1 + \psi.(u \bowtie x).t - \theta.t.u & , a \le m.x. \end{array}$$

Solving $\mathcal{A}[\,\barwedge\,]$ from this set of recurrence relations leads to at most one unfolding of the last alternative of $\mathcal{A}[\lhd]$. In order that $\mathcal{A}[\,\barwedge\,]$ is $O(1)$, it is therefore sufficient to require that the term $1 + \theta.(u \bowtie x).t - \theta.t.u$ is bounded, while the other terms are required to be nonpositive. Thus, we require that, for some constant γ and for all t, u, x and y:

(iii) $\theta.t.(u \lhd y) \le \theta.t.u$

(iv) $\theta.(u \bowtie x).t \le \theta.t.u + \gamma$

(v) $1 + \psi.(u \bowtie y).t \le \theta.t.u.$

If we succeed in fulfilling these five requirements, we may conclude from the above set of recurrence relations that $\mathcal{A}[\,\barwedge\,](x,y) \le \gamma + 1$. These requirements are, of course, easily met when we choose $\psi.t.u = 0$ and $\theta.t.u = 1$; then Φ coincides with the potential function derived in Section 9.4.2. There is, however, an additional requirement to ensure a logarithmic bound for delmin:

$$
\begin{array}{ll}
& \mathcal{A}[\mathsf{delmin}](\langle t,a,u\rangle) \\
= & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
& \mathcal{T}[\mathsf{delmin}](\langle t,a,u\rangle) + \Phi.(\mathsf{delmin}.\langle t,a,u\rangle) - \Phi.\langle t,a,u\rangle \\
= & \quad \{ \text{ definitions of } \mathcal{T}, \mathsf{delmin}, \text{ and } \Phi \ \} \\
& 1 + \mathcal{N}[\,\barwedge\,](t,u) + \Phi.(t \barwedge u) - \Theta.t - \theta.t.u - \Theta.u \\
= & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
& 1 + \mathcal{A}[\,\barwedge\,](t,u) + \Phi.t + \Phi.u - \Theta.t - \theta.t.u - \Theta.u \\
\leq & \quad \{ \text{ above bound for } \mathcal{A}[\,\barwedge\,]; \ \theta \text{ nonnegative (requirement (i))} \ \} \\
& \gamma + 2 + \Phi.t - \Theta.t + \Phi.u - \Theta.u \\
\leq & \quad \{ \ \Phi.z - \Theta.z \leq \log_\alpha \sharp z, \text{ see below} \ \} \\
& \gamma + 2 + \log_\alpha \sharp t + \log_\alpha \sharp u \\
\leq & \quad \{ \ 4mn \leq (m+n)^2 \ \} \\
& \gamma + 2 + 2\log_\alpha(\sharp t + \sharp u) - \log_\alpha 4.
\end{array}
$$

We choose ψ and θ such that $\Phi.z - \Theta.z \leq \log_\alpha \sharp z$, for some constant α, $\alpha > 1$. Evidently, this is true for $z = \langle\,\rangle$. For $z = \langle t,a,u\rangle$, we have that $\Phi.z - \Theta.z = \Pi.t$, and therefore we want Π to satisfy

(17) $\Pi.z \leq \log_\alpha \sharp z$,

for all z. To obtain a requirement in terms of ψ and θ, we first translate this requirement on Π into a requirement on π. We apply induction on z.

Case $z = \langle\,\rangle$. Trivial, since $\sharp\langle\,\rangle = 1$.

Case $z = \langle t,a,u\rangle$.

$$
\begin{array}{ll}
& \Pi.\langle t,a,u\rangle \\
= & \quad \{ \text{ definition of } \Pi \ \} \\
& \pi.t.u + \Pi.u \\
= & \quad \{ \text{ induction hypothesis (17)} \ \} \\
& \pi.t.u + \log_\alpha \sharp u \\
\leq & \quad \{ \text{ see upper bound for } \pi \text{ below} \ \} \\
& \log_\alpha \sharp\langle t,a,u\rangle.
\end{array}
$$

Hence, we require that $\pi.t.u$ is bounded from above by $\log_\alpha \frac{\sharp t + \sharp u}{\sharp u}$, or, in terms of θ and ψ:

(vi) $\theta.t.u - \psi.t.u \leq \log_\alpha \frac{\sharp t + \sharp u}{\sharp u}$.

This concludes the derivation of six requirements on ψ and θ, which are sufficient to ensure that the amortized cost of delmin is logarithmic and that the amortized costs of the other priority queue operations are $O(1)$. In addition, we obtain a nonnegative and small potential Φ, provided ψ and θ are small.

Construction of θ and ψ

First of all, we remark that (i) is redundant, since it follows from (ii) and (v). Next, we do away with $\log_\alpha$ in requirement (vi) by introducing functions f and g and defining

$$\begin{aligned} \theta.t.u &= \log_\alpha f.\sharp t.\sharp u \\ \psi.t.u &= \log_\alpha g.\sharp t.\sharp u. \end{aligned}$$

Apart from the removal of $\log_\alpha$, the introduction of f and g embodies the decision to let $\theta.t.u$ and $\psi.t.u$ depend on the sizes of t and u only—this simplifies the formulation of the requirements and for top-down skew heaps we also found that $\varphi.t.u$ depends on the sizes of t and u only. The corresponding requirements on f and g then read ((i) is redundant; constant β replaces α^γ):

$$\begin{aligned} &\text{(i)} && 1 \le f.k.l \\ &\text{(ii)} && 1 \le g.k.l \\ &\text{(iii)} && f.k.m \le f.k.l \\ &\text{(iv)} && f.m.k \le \beta\ f.k.l \\ &\text{(v)} && \alpha\ g.m.k \le f.k.l \\ &\text{(vi)} && \frac{f.k.l}{g.k.l} \le \frac{k+l}{l}, \end{aligned}$$

for all positive k, l, and m such that $l \le m$ ($\alpha > 1$ and $\beta > 0$).

In our next step, we get rid of m in requirements (iv) and (v). Firstly, we may replace (iv) by (iv$'$), with

$$\text{(iv}') \quad f.l.k \le \beta\ f.k.l\ ,$$

because (iv$'$) is not weaker than (iv) on account of (iii):

$$\begin{aligned} & f.m.k \\ \le\ & \{\ \text{(iv}')\ \} \\ & \beta\ f.k.m \\ \le\ & \{\ \text{(iii)}\ \} \\ & \beta\ f.k.l, \end{aligned}$$

for any k, l, and m with $l \le m$. On similar grounds, (v) may be replaced by (v$'$), where

$$\text{(v}') \quad \alpha\ g.l.k \le f.k.l.$$

Now we observe that it is necessary that $\alpha \le \beta$, since it is required that for all k and l:

$$
\begin{array}{ll}
 & \alpha \\
\leq & \{\ (\mathrm{v}')\ \} \\
 & \dfrac{f.k.l}{g.l.k} \\
\leq & \{\ (\mathrm{iv}')\ \} \\
 & \beta\ \dfrac{f.l.k}{g.l.k} \\
\leq & \{\ (\mathrm{vi})\ \} \\
 & \beta\ \dfrac{l+k}{k}.
\end{array}
$$

Since we want α to be as large as possible and β as small as possible, we take $\alpha = \beta$.

As a further simplification we decide to eliminate g from the set of requirements. To this end we define

$$g.k.l = \frac{f.l.k}{\alpha},$$

i.e., we choose equality in (v′). This choice for g does not strengthen the requirements on f and α: requirement (vi) becomes $\alpha\ \dfrac{f.k.l}{f.l.k} \leq \dfrac{k+l}{l}$, but this already follows from (v′) and (vi). Taking $k = l$ in (vi) then gives $\alpha \leq 2$, so we take $\alpha = 2$ (as large as possible). As a result, we obtain the following set of requirements for f, where we have swapped k and l in (III) to let it resemble (IV):

$$
\begin{array}{ll}
\text{(I)} & 2 \leq f.k.l \leq 4 \\
\text{(II)} & f.k.m \leq f.k.l \\
\text{(III)} & f.k.l \leq 2\ f.l.k \\
\text{(IV)} & f.k.l \leq \dfrac{k+l}{2l} f.l.k,
\end{array}
$$

for all positive k, l, and m such that $l \leq m$. In (I) we have added an absolute upper bound for f to make explicit how small we want f to be; we have chosen 4 because $(\forall\, k, l :: 2 \leq f.k.l \leq \delta)$ implies $(\forall\, k, l :: 4l \leq \delta(k+l))$ on account of (IV), hence it is necessary that $\delta \geq 4$.

Requirements (I) through (III) are satisfied if we take $f.k.l = 2$, but then (IV) does not hold if $k < 3l$. To arrive at a solution for f, we distinguish the cases $k \leq l$ and $k \geq l$ in the definition of $f.k.l$. By simply taking $f.k.l = 2$ in case $k \leq l$, we see that (IV) reduces to $2 \leq \dfrac{k+l}{2l} f.l.k$. To satisfy this requirement we define f such that we have equality in (IV). The resulting f is then given by

$$f.k.l = 2\ \frac{k + k \max l}{k+l}.$$

The reader may now verify that this definition meets all requirements on f.

Remark 9.9
By first taking $f.k.l = 2$ in case $k \geq l$ and subsequently taking equality in (IV) as above, we find:

$$f.k.l = 4\ \frac{k+l}{k \max l + l}.$$

The choice for f in the text is preferred because it results in a smaller potential. □

Thus we find (since $g.k.l = f.l.k/2$):

$$\psi.t.u = \log_2 \frac{\sharp t \max \sharp u + \sharp u}{\sharp t + \sharp u} \qquad (= \log_2 \frac{2\sharp u}{\sharp t + \sharp u} \max 0),$$

and $\theta.t.u = 1 + \psi.u.t$. Since $0 \leq \psi.t.u \leq 1$, potential Φ satisfies $0 \leq \Phi.z \leq 2\#z$.

With this potential, each priority queue operation except delmin has $O(1)$ amortized costs. The amortized costs of the latter operation satisfy

$$\begin{array}{ll} & \mathcal{A}[\mathsf{delmin}](\langle t, a, u\rangle) \\ \leq & \quad \{ \text{ see page 135 } \} \\ & \gamma + 2 + 2\log_\alpha(\sharp t + \sharp u) - \log_\alpha 4 \\ = & \quad \{\ \alpha = 2,\ \beta = 2, \text{ and } \gamma = \log_\alpha \beta = 1\ \} \\ & 1 + 2\log_2 \sharp\langle t, a, u\rangle, \end{array}$$

which completes the proof of the following lemma.

Lemma 9.10
Let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Theta.t \ +\ \log_2 \frac{4\sharp t}{\sharp t + \sharp u} \max 1 \ +\ \Theta.u \end{array}$$

with

$$\begin{array}{lcl} \Theta.\langle\,\rangle & = & 0 \\ \Theta.\langle t, a, u\rangle & = & \Psi.t \ +\ \log_2 \frac{4\sharp t}{\sharp t + \sharp u} \max 1 \ +\ \Theta.u \\ \Psi.\langle\,\rangle & = & 0 \\ \Psi.\langle t, a, u\rangle & = & \Psi.t \ +\ \log_2 \frac{2\sharp u}{\sharp t + \sharp u} \max 0 \ +\ \Psi.u. \end{array}$$

Then $0 \leq \Phi.z \leq 2\#z$, and the amortized costs of union and delmin satisfy

$$\begin{array}{lcl} \mathcal{A}[\mathsf{union}](x, y) & \leq & 3 \\ \mathcal{A}[\mathsf{delmin}](z) & \leq & 1 + 2\ \log_2 \sharp z. \end{array}$$

□

Remark 9.11 (see Remark 9.7)
In Sleator and Tarjan's analysis of bottom-up skew heaps, a node with subtrees t and u contributes $\lceil \psi.t.u \rceil + 2\lceil \psi.u.t \rceil$ if it is on the major or minor path, and $\lceil \psi.t.u \rceil$ otherwise. Since $\lceil \psi.t.u \rceil + 2\lceil \psi.u.t \rceil \approx 1 + \psi.u.t$, this may be compared to our potential that counts $1 + \psi.u.t$ for nodes on the major and minor paths and $\psi.t.u$ for the remaining nodes.
□

9.4.4 Second top-down analysis of ⧖

Although the $O(1)$ bound for union is a nice result, it is somewhat disappointing that the bound for delmin.p is as high as $2\log_2(\#[\![p]\!] + 1)$. These bounds imply that at most $2N\log_2 N$ comparisons are needed to sort N numbers by means of bottom-up skew heaps (since this requires N applications of delmin). As shown in Section 9.3, however, we have $1.44N\log_2 N$ as bound when top-down skew heaps are used, and we would like to have a similar bound for bottom-up skew heaps.

To achieve this, we allow logarithmic costs for ⧖. For $\alpha > 1$, we try as bound:

(18) $\mathcal{A}[\,⧖\,](x, y) \leq 1 + \log_\alpha(\sharp x + \sharp y)$.

To take advantage of the work done to analyze top-down skew heaps, we take

$$\theta.t.u = 1,$$

and we derive several requirements for the remaining function ψ. This choice for θ considerably simplifies the recurrence relations for the amortized costs in the analysis of ⧖ in the previous section:

$$\begin{array}{lcll}
\mathcal{A}[\,⧖\,](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\
\mathcal{A}[\,⧖\,](\langle t, a, u\rangle, y) & = & \mathcal{A}[\triangleleft](u, y) & , a \leq m.y \\
\mathcal{A}[\,⧖\,](x, \langle t, a, u\rangle) & = & \mathcal{A}[\triangleleft](u, x) & , a \leq m.x \\
 & & & \\
\mathcal{A}[\triangleleft](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\
\mathcal{A}[\triangleleft](\langle t, a, u\rangle, y) & = & \mathcal{A}[\triangleleft](u, y) & , a \leq m.y \\
\mathcal{A}[\triangleleft](x, \langle t, a, u\rangle) & = & \mathcal{A}[\bowtie](u, x) + 1 & , a \leq m.x \\
 & & & \\
\mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\
\mathcal{A}[\bowtie](\langle t, a, u\rangle, y) & = & \mathcal{A}[\bowtie](u, y) + \psi.(u \bowtie y).t & , a \leq m.y \\
\mathcal{A}[\bowtie](x, \langle t, a, u\rangle) & = & \mathcal{A}[\bowtie](u, x) + \psi.(u \bowtie x).t & , a \leq m.x.
\end{array}$$

To solve (18), we first observe—as in our analysis of top-down skew heaps (cf. (9))—that

$$\mathcal{A}[\bowtie](x,y) = \Gamma.(x \bowtie y),$$

where

$$\begin{array}{lcl} \Gamma.\langle\,\rangle & = & 0 \\ \Gamma.\langle t,a,u\rangle & = & \Gamma.t + \psi.t.u. \end{array}$$

Since $\mathcal{A}[\,\barwedge\,](x,y) \leq 1 + \mathcal{A}[\bowtie](x,y)$, we see that (18) is satisfied when

$$\Gamma.z \leq \log_\alpha \sharp z,$$

which is in turn satisfied when

(i) $\psi.t.u \leq \log_\alpha \frac{\sharp t + \sharp u}{\sharp t}$.

To obtain logarithmic amortized costs for delmin, we want in addition—cf. (17)—that

(19) $\Pi.z \leq \log_\beta \sharp z$,

for some $\beta > 1$, which leads to the following requirement on ψ:

(ii) $1 - \psi.t.u \leq \log_\beta \frac{\sharp t + \sharp u}{\sharp u}$.

As shown in Section 9.2.4, the existence of a function ψ satisfying (i) and (ii) (cf. (13)) is guaranteed provided (cf. (14))

$$\beta \leq \frac{(\varepsilon+1)^{\varepsilon+1}}{\varepsilon^\varepsilon},$$

where $\varepsilon = \log_\alpha \beta$. Moreover, we have shown that it is possible to define ψ such that $0 \leq \psi.t.u \leq 1$. Since $\theta.t.u = 1$, we thus obtain a potential Φ satisfying $0 \leq \Phi.z \leq \#z$.

To obtain a small bound for delmin, we derive

$$\begin{array}{ll} & \mathcal{A}[\mathsf{delmin}](\langle t,a,u\rangle) \\ = & \quad \{\ \text{see page 135}\ \} \\ & 1 + \mathcal{A}[\,\barwedge\,](t,u) + \Phi.t + \Phi.u - \Theta.t - \theta.t.u - \Theta.u \\ = & \quad \{\ \theta.t.u = 1\ \} \\ & \mathcal{A}[\,\barwedge\,](t,u) + \Phi.t - \Theta.t + \Phi.u - \Theta.u \\ \leq & \quad \{\ (18);\ (19),\ \text{using that } \Phi.z - \Theta.z = \Pi.(l.z) \text{ for nonempty } z\ \} \\ & 1 + \log_\alpha(\sharp t + \sharp u) + \log_\beta \sharp t + \log_\beta \sharp u \\ \leq & \quad \{\ \log_\alpha x = \varepsilon \log_\beta x\ \} \\ & 1 + (\varepsilon+2)\log_\beta(\sharp t + \sharp u). \end{array}$$

This proves the following lemma.

Lemma 9.12
Let $\varepsilon > 0$ and let potential Φ be defined by

$$\begin{aligned} \Phi.\langle\,\rangle &= 0 \\ \Phi.\langle t, a, u\rangle &= \Theta.t + 1 + \Theta.u \end{aligned}$$

with

$$\begin{aligned} \Theta.\langle\,\rangle &= 0 \\ \Theta.\langle t, a, u\rangle &= \Psi.t + 1 + \Theta.u \\ \Psi.\langle\,\rangle &= 0 \\ \Psi.\langle t, a, u\rangle &= \Psi.t \;+\; \log_\beta \frac{\beta\sharp u}{\sharp t + \sharp u} \max 0 \;+\; \Psi.u. \end{aligned}$$

Then $0 \leq \Phi.z \leq \#z$, and the amortized costs of union and delmin satisfy

$$\begin{aligned} \mathcal{A}[\mathsf{union}](x, y) &\leq \log_\alpha(\sharp x + \sharp y) \;+\; O(1) \\ \mathcal{A}[\mathsf{delmin}](z) &\leq (\varepsilon + 2)\, \log_\beta \sharp z \;+\; O(1), \end{aligned}$$

where $\alpha = \beta^{1/\varepsilon}$ and $\beta = \frac{(\varepsilon+1)^{\varepsilon+1}}{\varepsilon^\varepsilon}$.
□

For each ratio of the number of union's to the number of delmin's, a suitable choice for ε in Lemma 9.12 can be made. For example, if this ratio is 1, we minimize $\frac{2\varepsilon+2}{\log_2 \beta}$ over all $\varepsilon > 0$. This yields 2 as minimal value at $\varepsilon = 1$. In this way, we get $2N \log_2 N$ as bound on the number of comparisons needed by program *sort*1 to sort N numbers. Note that this bound for *sort*1 also follows from Lemma 9.10.

To obtain a good bound for *PQsort*, we minimize the bound for delmin. As shown in Section 9.2.4, minimizing $\frac{\varepsilon+2}{\log_2 \beta}$ yields

$$\mathcal{A}[\mathsf{delmin}](z) \leq \log_\phi \sharp z \;+\; O(1)$$

for $\varepsilon = \phi$ (≈ 1.618) and $\beta = \phi^{\phi+2}$ (≈ 5.70). The corresponding bound for union is

$$\mathcal{A}[\mathsf{union}](x, y) \leq \log_\alpha(\sharp x + \sharp y) \;+\; O(1),$$

with $\alpha = \phi^{2\phi-1}$ (≈ 2.93). Compared to the bounds for top-down skew heaps, this bound for union is lower whereas the bounds for delmin are equal. So, according to these bounds, bottom-up skew heaps are at least as good as top-down skew heaps.

9.4.5 Results for priority queue operations

With $\mathcal{T}$ as defined for top-down skew heaps, we have derived in Section 9.4.3 the following bounds for bottom-up skew heaps (cf. Lemma 9.10):

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{single}](a) & = & 1 \text{ comparison} \\
\mathcal{A}[\mathsf{union}](p,q) & \leq & 2 \text{ comparisons} \\
\mathcal{A}[\mathsf{isempty}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{min}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{delmin}](p) & \leq & 2\log_2(1+\#[\![p]\!]) \text{ comparisons,}
\end{array}$$

for a potential function satisfying $0 \leq \Phi.p \leq 2\#[\![p]\!]$. These bounds for union and delmin improve the bounds obtained by Sleator and Tarjan by a factor 2.

The analysis in Section 9.4.4 yields as bounds (instantiate Lemma 9.12 with $\varepsilon = \phi$):

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{single}](a) & = & 1 \text{ comparison} \\
\mathcal{A}[\mathsf{union}](p,q) & \leq & 0.64\log_2(2+\#[\![p]\!]+\#[\![q]\!]) \text{ comparisons} \\
\mathcal{A}[\mathsf{isempty}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{min}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{delmin}](p) & \leq & 1.44\log_2(1+\#[\![p]\!]) \text{ comparisons,}
\end{array}$$

for a potential function satisfying $0 \leq \Phi.p \leq \#[\![p]\!]$. These bounds for union and delmin cannot be compared with the bounds obtained by Sleator and Tarjan, since the bound for union is not $O(1)$ but the bound for delmin is better.

9.5 Pointer implementations

Top-down skew heaps

In the implementation of top-down skew heaps we have used algebra

$$(\,\langle \text{Int} \rangle \mid \langle\,\rangle, (=\langle\,\rangle), \langle\cdot,\cdot,\cdot\rangle, l, m, r\,).$$

A purely-functional implementation of this algebra, which supports all operations in $O(1)$ time, is described in Section 4.3. The only disadvantage of this implementation is that the recycling of cells is not trivial. This can be made trivial by disposing cells explicitly, like we did in the destructive implementation of stacks in Chapter 4. This yields a destructive implementation of top-down skew heaps.

A next step is to recycle cells in-line to obtain an *in-situ* version of top-down skew heaps. That is, we avoid a call to new by using the cell which is

disposed by a nearby call to dispose; e.g., for the destructive implementation of stacks, $\mathsf{cons}.a.(\mathsf{tl}.p)$ is equivalent to $p^\wedge.a := a;\ \mathrm{return}(p)$. This gives the following destructive implementation of top-down skew heaps at pointer level:

$$\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.p & = & p = \mathrm{nil} \\
\mathsf{single}.a & = & |[\ h{:}B;\ \mathrm{new}(h);\ h^\wedge := \langle \mathrm{nil}, a, \mathrm{nil}\rangle;\ \mathrm{return}(h)\ ]| \\
\mathsf{min}.p & = & p^\wedge.a \\
\mathsf{union}.p.q & = & p \bowtie q \\
\mathsf{delmin}.p & = & \mathrm{return}(p^\wedge.l \bowtie p^\wedge.r);\ \mathrm{dispose}(p),
\end{array}$$

where

$$\begin{array}{lcll}
p \bowtie q & = & q & ,p = \mathrm{nil} \\
 & [] & p & ,q = \mathrm{nil} \\
 & [] & (\ \mathrm{return}(p);\ p^\wedge.l, p^\wedge.r := p^\wedge.r \bowtie q, p^\wedge.l & ,p^\wedge.a \le q^\wedge.a \\
 & & [] \ \mathrm{return}(q);\ q^\wedge.l, q^\wedge.r := q^\wedge.r \bowtie p, q^\wedge.l & ,q^\wedge.a \le p^\wedge.a \\
 & &)\ & ,p \neq \mathrm{nil} \wedge q \neq \mathrm{nil}.
\end{array}$$

Strictly speaking, the program for $\bowtie$ is not in-situ yet because it is recursive. It is however a simple form of recursion, and therefore it is not difficult to transform it into an iterative program. We leave this to the interested reader (see [30] for more details).

Bottom-up skew heaps

The algebra used in the programs for bottom-up skew heaps on pages 127 and 129 is algebra $\mathsf{T1}$ from Section 6.2.2:

$$\mathsf{T1} = (\ \langle \mathrm{Int}\rangle_1\ \mid\ [\,], (=[\,]), [\cdot],\ +\!\!+\ , hd, tl, lt, ft\ ).$$

Operations $\vdash$ and $\dashv$ have been omitted because they are redundant in the presence of operations $[\cdot]$ and $+\!\!+$ (e.g., $\langle t, a\rangle \vdash u = [\langle t, a\rangle] +\!\!+ u$). It turns out that we must—as far as we know—content ourselves with a destructive refinement of $\mathsf{T1}$ to achieve that all operations have $O(1)$ time complexity. As a consequence, the bottom-up skew heap implementation of PQ will also be destructive. (See Section 6.2.2.)

The programs are:

$$\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.p & = & p = \mathrm{nil} \\
\mathsf{single}.a & = & |[\ h{:}R;\ \mathrm{new}(h);\ h^\wedge := \langle h, \langle \mathrm{nil}, a\rangle, \mathrm{nil}\rangle;\ \mathrm{return}(h)\ ]| \\
\mathsf{min}.p & = & p^\wedge.a.1 \\
\mathsf{union}.p.q & = & p \between q \\
\mathsf{delmin}.p & = & (\mathrm{skip}\ \ ,p^\wedge.r = \mathrm{nil}\ []\ p^\wedge.r^\wedge.l := p^\wedge.l\ \ ,p^\wedge.r \neq \mathrm{nil}) \\
 & & ;\ \mathrm{return}(p^\wedge.a.0 \between p^\wedge.r);\ \mathrm{dispose}(p).
\end{array}$$

We do not present a refinement of the program for ⧖ (page 129) at pointer level. A destructive implementation for ⧖ is readily obtained from the implementation of T1 described in Section 6.2.2. For further—more interesting—refinements to an in-situ version of ⧖ we refer to [30]; the above described representation of $\langle \mathrm{Int} \rangle_1$ is called the *ring representation* by Sleator and Tarjan.

9.6 Concluding remarks

In a systematic and formal way we have designed and analyzed various implementations of priority queues. We have improved upon the results of Sleator and Tarjan by deriving somewhat simpler programs, and, more importantly, by deriving roughly twice as low upper bounds for the amortized costs of top-down melding and bottom-up melding. Moreover, we have shown that bottom-up skew heaps are at least as good as top-down skew heaps. The following table summarizes these new results.

	top-down skew heaps	bottom-up skew heaps	
		1st result	2nd result
$\mathcal{A}[\mathsf{union}](p,q)$	$1.44 \log_2(m+n)$	$O(1)$	$0.64 \log_2(m+n)$
$\mathcal{A}[\mathsf{delmin}](p)$	$1.44 \log_2 m$	$2 \log_2 m$	$1.44 \log_2 m$

where $m = \#[\![p]\!] + 1$ and $n = \#[\![q]\!] + 1$. An interesting fact is that $1.44 \log_2 m$ is an approximation of $\log_\phi m$ with $\phi = (1 + \sqrt{5})/2$. So, instead of a base-2 logarithm we get a base-ϕ logarithm.

As for the top-down skew heaps, we are very satisfied with the derivation of the program for ⋈ as well as with its analysis; we consider them to be highly calculational. Our program for ⋈ precisely captures the essence of this operation, and we have shown that it may be refined either by a nondestructive or a destructive program at pointer level. The treatment of the bottom-up skew heaps is less calculational, but still rather systematic. We obtained the program for ⧖ by first transforming a version of ⋈ in terms of type $\langle \mathrm{Int} \rangle_1$ into a tail-recursive one, and subsequently modifying this tail-recursive version a little bit. Applying well-known implementation techniques for lists, we finally constructed a destructive program for bottom-up melding at pointer level. In the analysis of bottom-up melding, however, we used a "top-down simulation" of the derived program for ⧖ in order to amortize the costs of the steps of bottom-up melding in a suitable way.

An interesting observation is that φ can be defined equal to its upper bound in (11) in the analysis of top-down skew heaps. This yields as potential (for $\alpha = 4$ and $\gamma = 0$):

$$\begin{aligned} \Phi.\langle\,\rangle &= 0 \\ \Phi.\langle t, a, u \rangle &= \Phi.t + \log_4 \frac{\sharp t + \sharp u}{\sharp t} + \Phi.u. \end{aligned}$$

That is, $\Phi.z$ is approximately equal to the sum of the logarithms of the sizes of all *right* subtrees of z. This potential can thus be seen as a variation of the

well-known potential for splay trees (see [29] and Chapter 11), which is the sum of the logarithms of the sizes of *all* subtrees.

Finally we would like to remark that deletion of an arbitrary value from a priority queue can be supported as well. To delete a from a skew heap, subtree $\langle t, a, u\rangle$, say, is replaced by $t \bowtie u$, resp. $t \between u$. This replacement not only affects the potential of the nodes in t and u, but also the potential of the nodes on the root path of a. In all cases, however, the amortized costs of deletion can be shown to be logarithmic (see also [30]).

Chapter 10

Fibonacci heaps

The following algebra of priority queues plays a central role in important graph algorithms such as Prim's minimum spanning tree algorithm and Dijkstra's shortest path algorithm:

$$\mathsf{PQ}' = (\,\{\mathrm{Int}\} \mid \{\,\}, (=\{\,\}), \{\cdot\}, \uplus, \mathit{remin}, \mathit{deckey}\,).$$

Compared to algebra PQ of Chapter 9, algebra PQ' involves sets instead of bags. Note that union of sets is restricted to *disjoint* set union by means of operation $\uplus$. Another difference between these algebras is that operations $\downarrow$ and $\Downarrow$ have been combined into one operation *remin*: $\mathit{remin}.S = (\downarrow S\,, \Downarrow S\,)$ for nonempty S. In this way, the implementation of algebra PQ' is somewhat simplified, whereas its applicability is only slightly reduced. Moreover, it is not difficult to adapt the implementation presented in this chapter in such a way that $\downarrow$ is available as inspection operation.

The important difference with algebra PQ is, however, that algebra PQ' provides *deckey* ("decrease key") of type $\mathrm{Int}\times\mathrm{Nat}\times\{\mathrm{Int}\}\rightharpoonup\{\mathrm{Int}\}$ as additional operation, which is defined by

$$\mathit{deckey}.a.k.S = S\backslash\{a\} \uplus \{a-k\}, \quad \text{provided } a \in S \text{ and } a-k \notin S\backslash\{a\}.$$

Using skew heaps, an implementation of *deckey* of logarithmic amortized complexity is easily constructed as a deletion followed by an insertion. The problem is that an implementation with $O(1)$ (amortized) time complexity is required to make the above-mentioned graph algorithms efficient.

In this chapter we present an implementation of PQ' based upon the *Fibonacci heaps* invented by Fredman and Tarjan [8]. As signature for this implementation we take

$$\mathsf{FH} = (\,F \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{single}, \mathsf{union}, \mathsf{remin}, \mathsf{deckey}\,),$$

and we will use an abstraction function $[\![\cdot]\!] \in F \to \{\mathrm{Int}\}$ to couple this algebra with PQ'. To enhance the clarity of exposition we first construct an implementation in which operation deckey is ignored. The concrete algebra without

operation deckey is called LBQ ("lazy binomial queues"). Our solution for LBQ is based upon *binomial queues*, a data structure invented by Vuillemin [34].

The aim is to achieve $O(\log \#[\![p]\!])$ amortized costs for remin.p and $O(1)$ amortized costs for the other operations. The amortized costs are defined according to the general scheme of Section 5.5:

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{isempty}](p) & = & \mathcal{T}[\mathsf{isempty}](p) \\
\mathcal{A}[\mathsf{single}](a) & = & \mathcal{T}[\mathsf{single}](a) + \Phi.(\mathsf{single}.a) \\
\mathcal{A}[\mathsf{union}](p,q) & = & \mathcal{T}[\mathsf{union}](p,q) + \Phi.(\mathsf{union}.p.q) - \Phi.p - \Phi.q \\
\mathcal{A}[\mathsf{remin}](p) & = & \mathcal{T}[\mathsf{remin}](p) + \Phi.(\mathsf{remin}.p.1) - \Phi.p \\
\mathcal{A}[\mathsf{deckey}](a,k,p) & = & \mathcal{T}[\mathsf{deckey}](a,k,p) + \Phi.(\mathsf{deckey}.a.k.p) - \Phi.p.
\end{array}$$

Deletion can then be implemented with logarithmic amortized cost as well, since $S \setminus \{a\} = \Downarrow(deckey.a.\infty.S)$ —using the extreme value ∞ as a trick to achieve this.

10.1 Lazy binomial queues

The starting-point for the definition of LBQ is that data type F is a set of *forests* over Int. In contrast with skew heaps, the trees in these forests are not restricted to binary trees. Also the order of the trees in a forest is immaterial, so we have

(1) $F \subseteq \{\langle \mathrm{Int} \rangle_6\}$,

where tree type $\langle \mathrm{Int} \rangle_6$ has been defined in Section 6.3.1 as the smallest solution of

$$X: \quad X = \mathrm{Int} \times \{X\}.$$

Recall from Chapter 6 that this means that an element $\langle a, t \rangle$ of this type denotes a nonempty tree with value $a \in \mathrm{Int}$ attached to the root and the trees in forest $t \in \{\langle \mathrm{Int} \rangle_6\}$ as subtrees of the root.

A forest represents a set in $\{\mathrm{Int}\}$ according to abstraction $[\![\cdot]\!]$ defined on $\{\langle \mathrm{Int} \rangle_6\}$ by

$$\begin{array}{lcl}
[\![\{\,\}]\!] & = & \{\,\} \\
[\![\{\langle a, t \rangle\} \uplus f]\!] & = & \{a\} \cup [\![t]\!] \cup [\![f]\!].
\end{array}$$

To allow for efficient implementation of the operations, two restrictions are imposed on F. The first one is that

(2) $(\forall\, f : f \in F : (\forall\, x : x \in f : x \text{ is a heap}))$,

where

$$\langle a, t\rangle \text{ is a heap } \equiv\ a < \downarrow[\![t]\!]\ \wedge\ (\forall\, x : x \in t : x \text{ is a heap}).$$

In connection with operation remin, a second restriction on F follows later on, but first we consider the other operations of LBQ for which programs with $O(1)$ cost are now easily obtained.

For the first three operations, there is no choice:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & \{\,\} \\ \mathsf{isempty}.f & \doteq & f = \{\,\} \\ \mathsf{single}.a & \doteq & \{\langle a, \{\,\}\rangle\}, \end{array}$$

and for union we take the following simple program:

$$\mathsf{union}.f.g \doteq f \cup g.$$

The fact that—in contrast with the binomial queues in [34]—forests f and g are simply united in this program for union, is the reason why we speak of "*lazy* binomial queues". As a consequence, the number of trees in f may be as large as $\#[\![f]\!]$ for f in rng LBQ because f may consist of $\#f$ singleton trees.

Remark 10.1
As union refines $\cup$, it follows that $[\![f]\!]$ and $[\![g]\!]$ in the program for $\mathsf{union}.f.g$ are disjoint. Consequently, we have that for all f in the range of LBQ (and FH):

$$\#[\![f]\!] = (\Sigma\, x : x \in f : \#x),$$

which means that these forests do not contain duplicates. Moreover, this implies that f and g in the program for $\mathsf{union}.f.g$ are disjoint as well. In the sequel we will rely on these facts, for example when we use that $\#(f \cup g) = \#f + \#g$.
□

The computation of $\mathsf{remin}.f$ involves the computation of $\downarrow[\![f]\!]$ and the computation of a representation for $\Downarrow[\![f]\!]$. As the trees in f are heaps, a search through the roots of f suffices to determine $\downarrow[\![f]\!]$, which takes time proportional to $\#f$. Because of the lazy implementation of union, the worst-case complexity of remin is thus linear.

To obtain logarithmic amortized costs for remin, the idea is to define $\mathsf{remin}.f$ such that the number of trees in the forest returned by this operation is small compared to $\#[\![f]\!]$. In this way, future applications of remin will be cheaper. To implement this idea, we take a program of the following form:

$$\mathsf{remin}.f \doteq (a, t \cup g) \quad |[\ \{\langle a, t\rangle\} \cup g = \mathit{condense}.f \quad , a < \downarrow[\![g]\!]\ ]|,$$

where *condense* should satisfy $[\![\mathit{condense}.f]\!] = [\![f]\!]$. The sole purpose of function *condense* is to reduce the number of trees in f. It will be implemented such that the actual cost of $\mathsf{remin}.f$ is proportional to $\#f$, and therefore we will define $\mathcal{T}$

such that $\mathcal{T}[\mathsf{remin}](f) = \#f$. With potential Φ defined by $\Phi.f = \#f$, we then have

$$\begin{array}{lcl} \mathcal{A}[\mathsf{remin}](f) & = & \mathcal{T}[\mathsf{remin}](f) + \Phi.(\mathsf{remin}.f.1) - \Phi.f \\ & = & \#f + \#(t \cup g) - \#f \\ & = & \#t + \#g. \end{array}$$

Note that with this potential, the amortized costs of the other operations are indeed $O(1)$; in particular, $\mathcal{A}[\mathsf{union}](f, g) = 1$ because $\#(f \cup g) = \#f + \#g$. To achieve that $\mathcal{A}[\mathsf{remin}]$ is logarithmic, we implement *condense* such that both $\#t$ and $\#g$ are $O(\log \#[\![f]\!])$.

Remark 10.2
An alternative program for remin is:

$$\mathsf{remin}.f = (a, condense.(t \cup g)) \quad |[\ \{\langle a, t\rangle\} \cup g = f \quad , a < \downarrow[\![g]\!]\ ]|.$$

That is, first the tree with minimum root is determined and its root is removed, and, subsequently, the remaining forest is "condensed". As actual costs for this program we take $1 + \#t + \#g$. With the same potential Φ, we now have

$$\begin{array}{lcl} \mathcal{A}[\mathsf{remin}](f) & = & \mathcal{T}[\mathsf{remin}](f) + \Phi.(\mathsf{remin}.f.1) - \Phi.f \\ & = & (1 + \#t + \#g) + \#(condense.(t \cup g)) - (1 + \#g) \\ & = & \#t + \#(condense.(t \cup g)). \end{array}$$

To guarantee that $\mathcal{A}[\mathsf{remin}]$ is logarithmic, *condense* should be designed such that both $\#t$ and $\#(condense.(t \cup g))$ are $O(\log \#[\![f]\!])$.

In this way, virtually the same requirements for *condense* and bounds for $\mathcal{A}[\mathsf{remin}]$ are obtained. The actual costs of the first program for remin are however smaller and therefore it is preferred; in other words, the first program is "lazier". □

Application of function *condense* will combine trees of the forest to which it is applied. For this purpose we use operator $\bowtie$ ("link"), which combines two trees into one under invariance of the heap order. For trees of type $\langle \mathrm{Int} \rangle_6$, this can be done by a simple nonrecursive program:

$$\begin{array}{lcll} \langle a, t\rangle \bowtie \langle b, u\rangle & \doteq & \langle a, t \cup \{\langle b, u\rangle\}\rangle & , a < b \\ & [] & \langle b, u \cup \{\langle a, t\rangle\}\rangle & , b < a. \end{array}$$

Since a forest f can be reduced to a singleton forest $\{\langle a, t\rangle\}$, say, by means of $\#f - 1$ links, we could now define $condense.f = \{\langle a, t\rangle\}$. Then $g = \{\,\}$ in the program for $\mathsf{remin}.f$, hence $\#g$ is evidently logarithmic in $\#[\![f]\!]$. The problem is that $\#t$ may be linear in $\#[\![f]\!]$, when the trees of f are linked in an arbitrary order. Therefore, a more sophisticated method of linking is required.

To achieve that $\#t$ is logarithmic in $\#[\![f]\!]$, the solution of Vuillemin [34] is to constrain the applications of $\bowtie$ to trees of equal size. That is, $\#x = \#y$ is taken as a precondition for $x \bowtie y$. As a result, in a forest constructed by means of the operations of LBQ, any tree x satisfies

(3) $\#x = 2^{\dagger x}$,

where $\dagger x$ denotes the *rank of* x defined by $\dagger\langle a, t\rangle = \#t$, the number of subtrees of a. Indeed (3) holds for singleton trees, and with respect to linking we reason as follows. Let x and y satisfy (3) and assume that $\#x = \#y$. Then $\dagger x = \dagger y$ as well on account of (3), and we observe:

$$\begin{array}{ll} & \#(x \bowtie y) \\ = & \quad \{ \text{ definitions of } \bowtie \text{ and } \# \ \} \\ & \#x + \#y \\ = & \quad \{ \ x \text{ and } y \text{ satisfy (3) } \} \\ & 2^{\dagger x} + 2^{\dagger y} \\ = & \quad \{ \ \dagger(x \bowtie y) = \dagger x + 1 = \dagger y + 1 \ \} \\ & 2^{\dagger(x \bowtie y)}. \end{array}$$

What is more, this property is preserved by $\bowtie$ for all subtrees of x and y as well. As a consequence, all trees in forests $f \in \mathrm{rng}\,\mathsf{LBQ}$ are *binomial*, where

$$\langle a, t\rangle \text{ is binomial } \equiv \#\langle a, t\rangle = 2^{\#t} \ \wedge \ (\forall\, x : x \in t : x \text{ is binomial}).$$

The third and last restriction on F now is

(4) $(\forall\, f : f \in F : (\forall\, x : x \in f : x \text{ is binomial}))$,

and type F is defined as the largest set satisfying (1), (2), and (4). This set may be paraphrased as the set of forests of binomial heaps (without duplicates).

Remark 10.3
Binomial trees enjoy numerous nice properties. For instance, the height of a binomial tree is equal to its rank, and the number of nodes at depth d in a binomial tree of rank r—hence of height r as well—equals $\binom{r}{d}$, for $0 \le d \le r$. The latter property explains the name of these trees and its straightforward proof relies on the identity $\binom{r+1}{d+1} = \binom{r}{d} + \binom{r}{d+1}$.
□

We can now complete the program for remin. Due to the definition of F, tree $\langle a, t\rangle$ is binomial, hence $\#t$, which is equal to $\log_2 \#\langle a, t\rangle$, is at most $\log_2 \#[\![f]\!]$. To bound the size of g, *condense* is programmed as follows:

$$\begin{array}{lcll} \mathit{condense}.f & = & f & , S = \{\,\} \\ & \Box & \mathit{condense}.(f \backslash \{x, y\} \cup \{x \bowtie y\}) \ \ |[\ (x, y) \in S\ ]| & , S \neq \{\,\} \\ & & |[\ S = \{(x, y) \mid x, y \in f \ \wedge \ x \neq y \ \wedge \ \dagger x = \dagger y\}\ ]|. & \end{array}$$

The idea is that *condense* combines trees until all ranks are different, so that forest *condense*.f contains no trees of equal rank. Since the rank of any tree in

condense.f is at most $\log_2 \#[\![f]\!]$, it then follows that $\#(\mathit{condense}.f)$ is at most $1 + \log_2 \#[\![f]\!]$, which implies $\#g \leq \log_2 \#[\![f]\!]$.

To implement this program such that *condense.f* takes $O(\#f)$ time, we use the method described in [8]. This method uses an array with domain $[0..\log_2 \#[\![f]\!]]$ and range Bool $\times$ $\langle \mathrm{Int} \rangle_6$. Initially, all boolean components are false, indicating that the tree components are "free". Then the trees of f are inserted one by one at the position indexed by their rank (which is at most $\log_2 \#[\![f]\!]$ for binomial trees). In case the tree component is free, the tree is inserted. Otherwise, this tree and the tree already occupying the position can be linked because they have the same rank, thereby emptying the position. The result is a tree whose rank is one larger, which must now be inserted at the next position, and so on.

When all trees have been inserted, forest *condense.f* can be extracted from the array, in the process determining the tree with minimum root. To avoid the $O(\log \#[\![f]\!])$ initialization of the array, the tricky representation of arrays (Section 7.2) can be used, which reduces the cost of initialization to $O(1)$. The actual cost of $\mathsf{remin}.f$ is then proportional to $\#f$—as assumed in the above. Another possibility is, however, to use $O(\log \#[\![f]\!])$ time for the initialization of the array. The actual cost is then $O(\#f \ \max \ \log \#[\![f]\!])$, hence the logarithmic bound for $\mathcal{A}[\mathsf{remin}]$ is not affected.

This completes the description of an implementation of $\mathsf{remin}.f$ whose amortized costs are bounded by $2 \log_2 \#[\![f]\!]$. Observing that cost measure $\mathcal{T}$ approximately counts the number of comparisons used by $\mathsf{remin}.f$, we see that the amortized number of comparisons used by $\mathsf{remin}.f$ is also at most $2 \log_2 \#[\![f]\!]$. (The actual number of comparisons made by $\mathsf{remin}.f$ equals $\#f - 1$, namely $\#f - \#(\mathit{condense}.f)$ comparisons by links and $\#(\mathit{condense}.f) - 1$ comparisons to determine the minimum tree.) From this observation we conclude that $2N \log_2 N$ is an asymptotic bound on the number of comparisons required to sort N numbers by means of lazy binomial queues (since this requires N applications of remin). In Section 10.3 we shall see that addition of operation *deckey* slightly increases this bound.

10.2 Intermezzo on the precondition of linking

In the previous section $\bowtie$ is defined as

$$\begin{array}{lcll} \langle a, t \rangle \bowtie \langle b, u \rangle & \doteq & \langle a, t \cup \{\langle b, u \rangle\} \rangle & , a < b \\ & \square & \langle b, u \cup \{\langle a, t \rangle\} \rangle & , b < a, \end{array}$$

and application of $\bowtie$ is restricted to trees of equal size (or rank) so that (3) holds. Instead of (3), however, a more general approach is to bound the ranks by

(5) $\quad \alpha^{\dagger x} \leq \#x,$

for some constant α larger than 1. Evidently, this condition is satisfied by singleton trees, and for $x \bowtie y$, with $x = \langle a, t\rangle$ and $y = \langle b, u\rangle$, we observe for the case $a < b$:

$$\begin{array}{ll} & \alpha^{\dagger(x \bowtie y)} \leq \#(x \bowtie y) \\ \equiv & \quad \{ \text{ definition of } \bowtie;\ \#(x \bowtie y) = \#x + \#y \ \} \\ & \alpha^{\dagger\langle a,\, t \cup \{\langle b, u\rangle\}\rangle} \leq \#x + \#y \\ \equiv & \quad \{ \text{ definition of } \dagger \ \} \\ & \alpha^{1+\dagger x} \leq \#x + \#y. \end{array}$$

Under the assumption that x and y satisfy (5), there are several ways to simplify this condition. For instance, we can head for a condition in terms of $\#x$ and $\#y$:

$$\begin{array}{ll} & \alpha^{1+\dagger x} \leq \#x + \#y \\ \Leftarrow & \quad \{ \text{ use } \alpha^{\dagger x} \leq \#x \text{ to eliminate } \dagger x \ \} \\ & \alpha\#x \leq \#x + \#y \\ \equiv & \quad \{\ \} \\ & (\alpha - 1)\#x \leq \#y. \end{array}$$

By symmetry, we obtain $(\alpha - 1)\#y \leq \#x$ as condition for the case $b < a$. As precondition for $x \bowtie y$ in terms of sizes we thus get

$$(6) \quad \alpha - 1 \leq \frac{\#x}{\#y} \leq \frac{1}{\alpha - 1}.$$

Another possibility is to look for a condition in terms of $\dagger x$ and $\dagger y$:

$$\begin{array}{ll} & \alpha^{1+\dagger x} \leq \#x + \#y \\ \Leftarrow & \quad \{ \text{ use } \alpha^{\dagger x} \leq \#x \text{ to eliminate } \#x \text{ and similarly for } y \ \} \\ & \alpha^{1+\dagger x} \leq \alpha^{\dagger x} + \alpha^{\dagger y} \\ \equiv & \quad \{\ \} \\ & (\alpha - 1)\alpha^{\dagger x} \leq \alpha^{\dagger y}. \end{array}$$

Combined with the other case this yields as precondition for $x \bowtie y$ in terms of ranks:

$$(7) \quad \log_\alpha(\alpha{-}1) \leq \dagger x - \dagger y \leq \log_\alpha \tfrac{1}{\alpha - 1},$$

using that $\log_\alpha$ is monotonic for $\alpha > 1$.

These preconditions make sense for $\alpha \leq 2$ only. Hence, at this point, we may decide to take $\alpha = 2$, since this gives the best bound for $\#t$ in the program for $\mathsf{remin}.f$, viz. $\#t \leq \log_2 \#\langle a, t\rangle$, using that $\#t = \dagger\langle a, t\rangle$. However, taking α smaller than 2 possibly reduces $\#(\mathit{condense}.f)$. For instance, condition (7) may be reformulated as

$$|\dagger x - \dagger y| \leq K,$$

with $K = -\log_\alpha(\alpha - 1)$. In the program for *condense*, trees can now be linked until all ranks are more than K apart. In this way, at most $(\log_\alpha \#[\![f]\!])/(K{+}1)$ trees remain in the end. As upper bound on $\#t + 1 + \#g$ we then have

$$\begin{array}{ll} & \log_\alpha \#[\![f]\!] + (\log_\alpha \#[\![f]\!])/(K{+}1) \\ = & \quad \{\ K{+}1 = \log_\alpha \frac{\alpha}{\alpha-1}\ \} \\ & (\frac{1}{\log_2 \alpha} + \frac{1}{\log_2 \frac{\alpha}{\alpha-1}})\log_2 \#[\![f]\!]. \end{array}$$

However, minimizing this bound over all $\alpha > 1$ yields $2\log_2 \#[\![f]\!]$ as minimal bound at $\alpha = 2$, from which we conclude that $\alpha = 2$ is indeed an appropriate choice. It corresponds to precondition $\dagger x = \dagger y$ on account of (7). Note that (6) simplifies to $\#x = \#y$ if $\alpha = 2$.

10.3 Fibonacci heaps

Leaving the implementation of the other operations virtually the same, we will now define operation deckey. To allow for a concise definition, we use root-path view $\langle \mathrm{Int} \rangle_7$ which has been discussed in Section 6.3.2:

$$\langle \mathrm{Int} \rangle_7 = [\langle \mathrm{Int} \rangle_6].$$

We recall that a nonempty list of this type represents a tree in $\langle \mathrm{Int} \rangle_6$ according to the following abstraction:

$$\begin{array}{lcll} [\![[x]]\!] & = & x & \\ [\![\langle a, t\rangle \vdash v]\!] & = & \langle a, t \cup \{[\![v]\!]\}\rangle & , v \neq [\,]. \end{array}$$

In terms of this type, a forest in which a occurs exactly once can be written in the form $\{v \dashv \langle a, t\rangle\} \uplus g$ in precisely one way.

Our goal is to define deckey$.a.k.(\{v \dashv \langle a, t\rangle\} \uplus g)$ such that the amortized costs are $O(1)$. To this end, we first note that $\langle a{-}k, t\rangle$ is a heap because $\langle a, t\rangle$ is a heap and $a{-}k \leq a$. There is, however, no guarantee that $v \dashv \langle a{-}k, t\rangle$ is a heap as well (if v is nonempty). A first attempt is therefore to remove subtree $\langle a, t\rangle$ from tree $v \dashv \langle a, t\rangle$ and to add tree $\langle a{-}k, t\rangle$, yielding forest $\{v\} \cup \{\langle a{-}k, t\rangle\} \cup g$. But the problem with this solution is, of course, that tree v is in general not binomial, as a consequence of which the relation between v's rank and v's size is destroyed.

However, as observed in the previous section, a property like $\dagger x \leq \log_\alpha \#x$, with $\alpha > 1$, is maintained when a fixed upper bound K on $|\dagger x - \dagger y|$ is used as precondition for $x \bowtie y$. By viewing a tree that has lost some subtrees as a tree of smaller rank, the choice $\{v\} \cup \{\langle a{-}k, t\rangle\} \cup g$ is not so bad provided the number of subtrees lost is not too large.

A solution is therefore to bound for each node the number of subtrees it may lose by a fixed number K, say.[1] To implement this, forests over Int $\times$ $[0..K]$ are used instead of forests over Int. Hence, each node contains in addition a value of type $[0..K]$, and trees are of the form $\langle (a, m), t\rangle$, which we abbreviate to $\langle a, m, t\rangle$. Value m will be increased each time a node loses a subtree. As a consequence, all trees in the forests in rng FH will be *pseudo-binomial*. To be able to define this class of trees explicitly, we first present an alternative characterization of binomial trees:

$$\begin{array}{ll} & \langle a, m, t\rangle \text{ is binomial} \\ \equiv & \{x : x \in t : \dagger x\} = [0..\#t) \ \wedge \ (\forall\, x : x \in t : x \text{ is binomial}). \end{array}$$

That is, the ranks of the trees in t are all different and together they exactly cover the interval $[0..\#t)$. This follows from the fact that $2^k{-}1$ can be written as a sum of powers of 2 in only one way, viz. as $1{+}2{+}\ldots{+}2^{k-1}$. As stepping-stone towards the definition of pseudo-binomial trees, we reformulate this as follows: the increasing list of the ranks of the trees in t is equal to list $[0, 1, \ldots, \#t{-}1]$.

To define the class of pseudo-binomial trees, we use $\ddagger x$ instead of $\dagger x$, which is called the *pseudo-rank of* x and is defined by $\ddagger\langle a, m, t\rangle = m{+}\#t$. The definition reads

$$\begin{array}{ll} & \langle a, m, t\rangle \text{ is pseudo-binomial} \\ \equiv & \llparenthesis x : x \in t : \ddagger x \rrparenthesis \text{ is diagonal} \ \wedge \ (\forall\, x : x \in t : x \text{ is pseudo-binomial}), \end{array}$$

where a bag B is called *diagonal* (which is short for "upper-diagonal") when the ascending list of elements of B is pointwise at least list $[0, 1, \ldots, \#B{-}1]$. The relevant properties of diagonal bags are:

(8) $\llparenthesis\ \rrparenthesis$ is diagonal

(9) B is diagonal $\wedge\ k \geq \#B\ \Rightarrow\ B \oplus \llparenthesis k \rrparenthesis$ is diagonal

(10) B is diagonal $\Rightarrow\ B \ominus \llparenthesis k \rrparenthesis$ is diagonal.

The important fact is that for any pseudo-binomial tree x,

(11) $\alpha^{\dagger x} \leq \#x$,

where constant α, $\alpha > 1$, depends on K. Assuming that $\langle a, m, t\rangle$ is pseudo-binomial, we prove this by (well-founded) induction:

$$\begin{array}{ll} & \#\langle a, m, t\rangle \\ = & \quad \{ \text{ definition of } \# \ \} \\ & 1 + (\Sigma\, x : x \in t : \#x) \\ \geq & \quad \{ \text{ ind. hyp. (11), using that } x \text{ is pseudo-binomial } \} \\ & 1 + (\Sigma\, x : x \in t : \alpha^{\dagger x}) \end{array}$$

[1] This is a generalization of Fredman and Tarjan's method [8] which corresponds to $K = 1$.

$$
\begin{array}{ll}
\geq & \{\ \ddagger x \leq K + \dagger x \text{ and } \alpha > 1\ \} \\
& 1 + (\Sigma\, x : x \in t : \alpha^{\ddagger x - K}) \\
\geq & \{\ \text{bag } \llparenthesis x : x \in t : \ddagger x \rrparenthesis \text{ is diagonal and } \alpha > 1\ \} \\
& 1 + (\Sigma\, i : 0 \leq i < \#t : \alpha^{i-K}) \\
= & \{\ \text{calculus}\ \} \\
& 1 + \frac{\alpha^{\#t} - 1}{\alpha^K(\alpha - 1)} \\
\geq & \{\ \text{take } \alpha^K(\alpha - 1) \leq 1, \text{ see below}\ \} \\
& \alpha^{\#t}.
\end{array}
$$

Taking α as large as possible, we see that we want α to be the *largest* real number satisfying

$$1 < \alpha \ \wedge\ \alpha^K(\alpha - 1) \leq 1,$$

or, equivalently, we want α to be the *unique* real number satisfying

$$1 < \alpha \ \wedge\ \alpha^{K+1} = \alpha^K + 1.$$

Since α decreases as K increases, $K = 0$ seems the best choice at this point, which corresponds to $\alpha = 2$. But, as we shall see shortly, K must be positive to ensure that the amortized costs of deckey are $O(1)$. The best value for α is therefore $\phi = (1 + \sqrt{5})/2$ for $K = 1$.

So, pseudo-binomial trees satisfy (11), which bounds the ranks of these trees. The problem is now to program the operations of FH such that the trees generated by these operations are pseudo-binomial. To achieve this for deckey, we proceed as follows. We take a definition of the form

$$\mathsf{deckey}.a.k.(\{v \dashv \langle a, m, t\rangle\} \uplus g) = cut.v \cup \{\langle a{-}k, m, t\rangle\} \cup g.$$

Since tree $v \dashv \langle a, m, t\rangle$ is pseudo-binomial, we have that $\langle a{-}k, m, t\rangle$ is pseudo-binomial as well. Tree v is, however, not necessarily pseudo-binomial. For this reason, function *cut* is introduced, which will be defined such that $cut.v$ transforms tree v into a *forest* of pseudo-binomial trees.

To see that v is not necessarily binomial, suppose $v = w \dashv \langle b, n, u\rangle$. Then $v \dashv \langle a, m, t\rangle$ and $w \dashv \langle b, n, u \uplus \{\langle a, m, t\rangle\}\rangle$ correspond to the same tree, hence both are pseudo-binomial. But $w \dashv \langle b, n, u\rangle$ is not necessarily pseudo-binomial, because $\ddagger\langle b, n, u\rangle = \ddagger\langle b, n, u \uplus \{\langle a, m, t\rangle\}\rangle - 1$ and diagonality of $B \oplus \llparenthesis k{-}1 \rrparenthesis$ does not follow from diagonality of $B \oplus \llparenthesis k \rrparenthesis$.

Notice, however, that $w \dashv \langle b, n{+}1, u\rangle$ is pseudo-binomial because $\ddagger\langle b, n{+}1, u\rangle$ equals $\ddagger\langle b, n, u \uplus \{\langle a, m, t\rangle\}\rangle$. Therefore, $cut.(w \dashv \langle b, n, u\rangle)$ is easy to define in case $n{+}1 \leq K$. To solve the remaining case $n = K$, subtree $\langle b, n, u\rangle$ is removed from $w \dashv \langle b, n, u\rangle$ and *cut* is applied recursively to w. Note that $\langle b, n, u\rangle$ is pseudo-binomial because $\langle b, n, u \uplus \{\langle a, m, t\rangle\}\rangle$ is pseudo-binomial due to property (10). The program for *cut* thus is:

$$\begin{array}{lcll} cut.[\,] & \doteq & \{\,\} & \\ cut.(w \dashv \langle b, n, u\rangle) & \doteq & \{w \dashv \langle b, n{+}1, u\rangle\} & , n < K \\ & \Box & cut.w \cup \{\langle b, 0, u\rangle\} & , n = K. \end{array}$$

To achieve $O(1)$ amortized cost, the amortized cost of the recursive alternative must be at most 0. For this purpose, it is essential that n is replaced by 0 in case $n = K$, as will be shown below. That K should be positive is clear: otherwise case $n < K$ never applies.

Remark 10.4
As explained in Chapter 6, a list v of type $\langle T\rangle_7$ represents more than a tree: it represents the root path to the last element of v. This property is exploited in the above program for $\mathsf{deckey}.a.k.(\{v \dashv \langle a, m, t\rangle\} \uplus g)$ in which v denotes the root path to $\langle a, m, t\rangle$. Recursive applications of *cut* generated during evaluation of $cut.v$ all take place along this root path. These recursive applications are called "cascading cuts" by Fredman and Tarjan in [8].
□

To complete the analysis, potential Φ is extended as follows:

$$\Phi.f = \#f + 2\Psi.f, \text{ where } \Psi.f = (\#\, a, m : (a, m) \in f : m = K),$$

i.e., Ψ counts the number of nodes that have lost the maximal number of subtrees. The role of factor 2 in the definition of Φ will be clarified below. Note that $\Phi.(f \cup g) = \Phi.f + \Phi.g$ for disjoint f and g.

We calculate $\mathcal{A}[\mathsf{deckey}]$, writing f for $\{v \dashv \langle a, m, t\rangle\} \uplus g$:

$$\begin{array}{ll} & \mathcal{A}[\mathsf{deckey}](a, k, f) \\ = & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{deckey}] \ \} \\ & \mathcal{T}[\mathsf{deckey}](a, k, f) + \Phi.(\mathsf{deckey}.a.k.f) - \Phi.f \\ = & \quad \{ \text{ definition of } \mathsf{deckey} \ \} \\ & \mathcal{T}[cut](v) + \Phi.(cut.v \cup \{\langle a{-}k, m, t\rangle\} \cup g) - \Phi.(\{v \dashv \langle a, m, t\rangle\} \uplus g) \\ = & \quad \{ \text{ property of } \Phi \ \} \\ & \mathcal{T}[cut](v) + \Phi.(cut.v) + \Phi.\{\langle a{-}k, m, t\rangle\} + \Phi.g - \Phi.\{v \dashv \langle a, m, t\rangle\} - \Phi.g \\ = & \quad \{ \text{ definition of } \Phi \text{ and } \Psi.\langle a{-}k, m, t\rangle = \Psi.\langle a, m, t\rangle \ \} \\ & \mathcal{T}[cut](v) + \Phi.(cut.v) + 1 - \Phi.\{v\} \\ = & \quad \{ \text{ definition of } \mathcal{A}[cut], \text{ see below } \} \\ & \mathcal{A}[cut](v) + 1. \end{array}$$

As amortized costs for *cut* we thus take:

$$\mathcal{A}[cut](v) = \mathcal{T}[cut](v) + \Phi.(cut.v) - \Phi.\{v\}.$$

Then $\mathcal{A}[cut]([\,]) = 0$, and $\mathcal{A}[cut](w \dashv \langle b, n, u\rangle) \leq 3$ in case $n < K$, and for case $n = K$ we derive:

$$
\begin{array}{ll}
& \mathcal{A}[cut](w \dashv \langle b,n,u\rangle) \\
= & \quad \{ \text{ definition of } \mathcal{A}[cut] \ \} \\
& \mathcal{T}[cut](w \dashv \langle b,n,u\rangle) + \Phi.(cut.(w \dashv \langle b,n,u\rangle)) - \Phi.\{w \dashv \langle b,n,u\rangle\} \\
= & \quad \{ \text{ definition of } cut \ \} \\
& 1 + \mathcal{T}[cut](w) + \Phi.(cut.w \cup \{\langle b,0,u\rangle\}) - \Phi.\{w \dashv \langle b,n,u\rangle\} \\
= & \quad \{ \text{ definition of } \Phi, \text{ using } 0 < K \text{ and } n = K \ \} \\
& 1 + \mathcal{T}[cut](w) + \Phi.(cut.w) + 1 + 0 + 2\Psi.u - \Phi.\{w\} - 2 - 2\Psi.u \\
= & \quad \{ \text{ definition of } \mathcal{A}[cut] \ \} \\
& \mathcal{A}[cut](w).
\end{array}
$$

This derivation explains why the factor 2 is needed in the definition of Φ. We conclude that $\mathcal{A}[cut](v) \leq 3$, hence that $\mathcal{A}[\mathsf{deckey}](a,k,f) \leq 4$.

To complete the definition of FH we must adapt the definition of $\bowtie$. A possible definition is

$$
\begin{array}{lcll}
\langle a,m,t\rangle \bowtie \langle b,n,u\rangle & \doteq & \langle a,m,t \cup \{\langle b,0,u\rangle\}\rangle & ,a < b \\
& [] & \langle b,n,u \cup \{\langle a,0,t\rangle\}\rangle & ,b < a,
\end{array}
$$

under precondition $\#t = \#u$. Then $x \bowtie y$ is pseudo-binomial because of property (9) and the fact that $\ddagger x \geq \dagger x$ and $\ddagger y \geq \dagger y$. Note that n and m are replaced by 0 in cases $a < b$ and $b < a$, respectively; this is done to improve the performance a little bit. (An alternative is to use $\ddagger x = \ddagger y$ as precondition for $x \bowtie y$, but then this optimization is not allowed.)

In summary, the operations of FH are

$$
\begin{array}{lcl}
\mathsf{empty} & \doteq & \{\,\} \\
\mathsf{isempty}.f & \doteq & f = \{\,\} \\
\mathsf{single}.a & \doteq & \{\langle a,0,\{\,\}\rangle\} \\
\mathsf{union}.f.g & \doteq & f \cup g \\
\mathsf{remin}.f & \doteq & (a, t \cup g) \quad |[\ \{\langle a,m,t\rangle\} \cup g = condense.f \quad ,a < \downarrow[\![g]\!]\ ]|,
\end{array}
$$

in which

$$
\begin{array}{lcll}
condense.f & = & f & ,S = \{\,\} \\
& [] & condense.(f \backslash \{x,y\} \cup \{x \bowtie y\}) \ \ |[\ (x,y) \in S\ ]| & ,S \neq \{\,\} \\
& & |[\ S = \{(x,y) \mid x,y \in f \ \wedge\ x \neq y \ \wedge\ \dagger x = \dagger y\}\ ]|,
\end{array}
$$

and

$$
\mathsf{deckey}.a.k.(\{v \dashv \langle a,m,t\rangle\} \uplus g) = cut.v \cup \{\langle a{-}k,m,t\rangle\} \cup g,
$$

where

$$
\begin{array}{lcll}
cut.[\,] & \doteq & \{\,\} & \\
cut.(w \dashv \langle b,n,u\rangle) & \doteq & \{w \dashv \langle b,n{+}1,u\rangle\} & ,n < K \\
& [] & cut.w \cup \{\langle b,0,u\rangle\} & ,n = K.
\end{array}
$$

The amortized cost of $\mathsf{remin}.f$ is bounded by $2\log_\phi \#[\![f]\!]$ if we take $K = 1$, and the amortized costs of the other operations are $O(1)$.

To sort N numbers by means of Fibonacci heaps, we see that the asymptotic bound on the number of comparisons becomes $2N\log_\phi N$, That is, at most $2.88N\log_2 N$ comparisons instead of $2N\log_2 N$ comparisons for lazy binomial queues.

10.4 Concluding remarks

We consider it a nice result that our formal description of Fibonacci heaps—which captures the essence of this data structure—fits on less than one page. The level of detail has been chosen such that this description provides a sound basis for a formal (amortized) efficiency analysis. In addition, the programs for FH can be refined further to programs operating on pointers and arrays, using the techniques described in Chapter 6. The key to this achievement is the use of type $\langle T\rangle_6$ along with its root-path view $\langle T\rangle_7$. In terms of the latter type, the programs for deckey and *cut* become remarkably concise. Moreover, the amortized costs of these programs can be determined in a calculational way.

The reason for the name "Fibonacci heaps" is that the Fibonacci sequence plays a role in the analysis by Fredman and Tarjan. In our analysis the Fibonacci sequence does not play a role anymore, but the quantity $\phi = (1+\sqrt{5})/2$, the "golden ratio", which is intimately connected with the Fibonacci sequence, still does. We remark that the number α given by

$$1 < \alpha \ \wedge \ \alpha^{K+1} = \alpha^K + 1,$$

is connected with a similar sequence H, which is defined by

$$\begin{array}{lcll} H.i & = & 1 & ,0 \le i \le K \\ H.(i{+}1) & = & H.i + H.(i{-}K) & ,i \ge K. \end{array}$$

A connection is, for instance, that for $i \ge 0$:

$$\alpha^{i-K} \le H.i \le \alpha^i.$$

To prove that each tree in a Fibonacci heap has a size at least exponential in the rank of its root, Fredman and Tarjan first prove the following lemma [8, Lemma 1]:

> Let x be any node in a Fibonacci heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the i-th child of x has a rank of at least $i{-}2$.

The formulation of this lemma as well as its proof are very operational (in the proof, they use phrases like "consider the time when y was linked to x" and "after the linking...".) We have avoided such operational reasoning altogether

by introducing the notion of pseudo-binomial trees, which precisely captures the kind of trees that arise in Fibonacci heaps. For these trees we were able to show (in one go) that their size is exponential in their rank (cf. inequality (11)).

One may wonder why we have taken the trouble to treat the general case instead of confining ourselves to case $K{=}1$—as Fredman and Tarjan did in [8]. Well, a good reason is that our analysis shows that operation *cut* can be avoided in particular applications of the graph algorithms. Take, for instance, Dijkstra's shortest path algorithm which is defined for directed graphs, and assign to K the maximal number of incoming arcs of any vertex. Then the recursive alternative of *cut* never applies, hence *deckey* can be implemented much simpler—taking $cut.v = \{v\}$ for nonempty v. As bound on the amortized number of comparisons used by $\mathsf{remin}.f$ we then get $2\log_\alpha \#[\![f]\!]$, which is reasonably small for small values of K.

Chapter 11

Path reversal, splaying, and pairing

In this chapter we analyze the amortized efficiency of three similar operations on trees. In the next section, we first investigate path reversal in detail, for which a potential function is derived in a calculational way. Since splaying and pairing are very similar to path reversal, these operations can then be analyzed along the same lines.

11.1 Path reversal

Path reversal is an operation on trees that moves a node to the root in a specific way (see Figure 11.1). It has been introduced in [32] as an alternative path compression technique, called "reversal of type zero". Such path compression techniques are used in efficient solutions to the well-known disjoint set union problem, but here we shall only be concerned with the amortized complexity of path reversal.

In the next section we first present a concise program for path reversal. Using the same cost measure as in [12], we then analyze the amortized cost of this program. We also analyze a top-down simulation of path reversal to show how this simplifies the calculations. Subsequently, the cost of a series of path reversals is determined, which is also done in [12]. Our results exactly match the results of [12], the important difference being that we *derive* the required potential function to a large extent.

11.1.1 Program

Let T be a nonempty type. Path reversal operates on trees of type $\langle T \rangle_6$, which has been introduced in Section 6.3.1 as the smallest solution of

$$X : \quad X = T \times \{X\}.$$

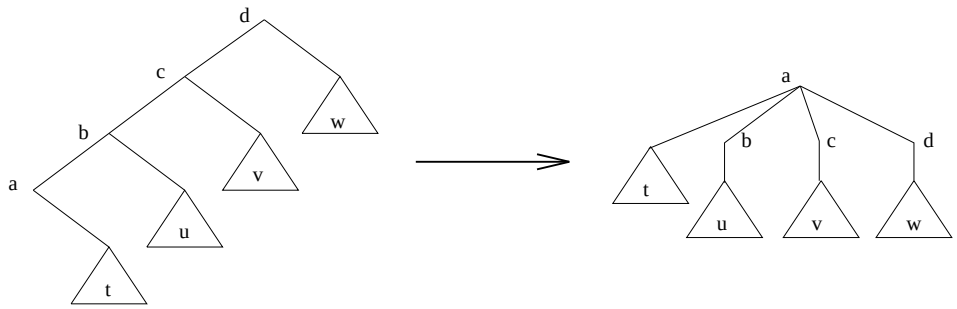

Figure 11.1: Path reversal at a (triangles denote *sets* of subtrees).

In terms of this tree type, path reversal is an operation of type $T \times \langle T \rangle_6 \curvearrowright \langle T \rangle_6$. We denote it by $\angle$, and $a \angle\, x$ is defined only if a occurs exactly once in x.

To allow for a simple description of path reversal, we use root-path view $\langle T \rangle_7$ from Section 6.3.2—which was also used in the definition of deckey in Section 10.3. In terms of this type, the program for path reversal reads

$$\begin{array}{lll} \text{(i)} & a \angle\, \langle a, t \rangle & = \langle a, t \rangle \\ \text{(ii)} & a \angle\, (y \dashv x \dashv \langle a, t \rangle) & \doteq a \angle\, (y \dashv \langle a, \{x\} \cup t \rangle), \end{array}$$

in which $t \in \{\langle T \rangle_6\}$, $x \in \langle T \rangle_6$, and $y \in \langle T \rangle_7$. The cost measure defined in this program, which we will call $\mathcal{T}$, coincides with the measure used by Ginat, Sleator, and Tarjan in [12], and therefore our results will be comparable to their results.

11.1.2 Bottom-up analysis

We analyze $(a \angle\,)$, for arbitrary a, for which the amortized cost is defined by

$$\mathcal{A}[a \angle\,](x) = \mathcal{T}[a \angle\,](x) + \Phi.(a \angle\, x) - \Phi.x.$$

Our goal is to define Φ such that

$$\text{(1)} \quad \mathcal{A}[a \angle\,](x) \leq \log_\alpha \#x,$$

and α is as large as possible—in any case larger than 1. Furthermore, we want Φ to be nonnegative and small.

The first step is to calculate a recurrence relation for $\mathcal{A}[a \angle\,]$. For case (i) we observe:

$$\begin{array}{ll} & \mathcal{A}[a \angle\,](\langle a, t \rangle) \\ = & \quad \{ \text{ definition of } \mathcal{A}[\angle\,] \ \} \\ & \mathcal{T}[a \angle\,](\langle a, t \rangle) + \Phi.(a \angle\, \langle a, t \rangle) - \Phi.\langle a, t \rangle \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } a\angle \ \} \\
& 0 + \Phi.\langle a, t\rangle - \Phi.\langle a, t\rangle \\
= & \{ \ \} \\
& 0,
\end{array}
$$

and for case (ii) we observe:

$$
\begin{array}{ll}
& \mathcal{A}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ definition of } \mathcal{A}[\angle\,] \ \} \\
& \mathcal{T}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) + \Phi.(a\angle\,(y \dashv x \dashv \langle a, t\rangle)) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ definition of } a\angle \ \} \\
& 1 + \mathcal{T}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) + \Phi.(a\angle\,(y \dashv \langle a, \{x\} \cup t\rangle)) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ definition of } \mathcal{A}[a\angle\,] \ \} \\
& \mathcal{A}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) + 1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle).
\end{array}
$$

So, the recurrence relation is:

$$
\begin{array}{lcl}
\mathcal{A}[a\angle\,](\langle a, t\rangle) & = & 0 \\
\mathcal{A}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) & = & \mathcal{A}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) \\
& & +1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle).
\end{array}
$$

The term $1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle)$ may be interpreted as the amortized cost of unfolding $a\angle\,(y \dashv x \dashv \langle a, t\rangle)$.

The obvious way to proceed is now to establish (1) by induction on $\#x$, using the above recurrence relation. Unfortunately, this is bound to fail because the sizes of the arguments of $\mathcal{A}[a\angle\,]$ are the same on both sides in case (ii). There is however an important difference between $y \dashv x \dashv \langle a, t\rangle$ and $y \dashv \langle a, \{x\} \cup t\rangle$: the subtree rooted at a increases in size (and the depth of a decreases). For this reason, the following strengthening of (1) is an appropriate induction hypothesis:

$$
(2) \quad \mathcal{A}[a\angle\,](y \dashv \langle a, t\rangle) \leq \log_\alpha \frac{\#(y \dashv \langle a, t\rangle)}{\#\langle a, t\rangle}.
$$

We apply induction on y. It is obvious that (2) holds in case (i) ($y = [\,]$), and for case (ii) we derive:

$$
\begin{array}{ll}
& \mathcal{A}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ above recurrence relation } \} \\
& \mathcal{A}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) + 1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
\leq & \{ \text{ induction hypothesis (2) } \} \\
& \log_\alpha \frac{\#(y \dashv \langle a, \{x\} \cup t\rangle)}{\#\langle a, \{x\} \cup t\rangle} + 1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
\leq & \{ \text{ see requirement on } \Phi \text{ below } \} \\
& \log_\alpha \frac{\#(y \dashv x \dashv \langle a, t\rangle)}{\#\langle a, t\rangle}.
\end{array}
$$

Hence, (2) holds provided Φ satisfies

$$1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \leq \log_\alpha \frac{\#\langle a, \{x\} \cup t\rangle}{\#\langle a, t\rangle}.$$

To remove $\log_\alpha$ from this requirement, we introduce function Ψ and define Φ of the form

$$\Phi.x = \beta \log_\alpha \Psi.x,$$

with $\beta > 0$. Constant β is introduced because there is no reason to assume that the base of the logarithm in the definition of Φ is equal to α. On account of the monotonicity of $\log_\alpha$ ($\alpha > 1$), the above requirement on Φ thus becomes

$$\alpha \left(\frac{\Psi.(y \dashv \langle a, \{x\} \cup t\rangle)}{\Psi.(y \dashv x \dashv \langle a, t\rangle)} \right)^\beta \leq \frac{\#\langle a, \{x\} \cup t\rangle}{\#\langle a, t\rangle}.$$

Defining $\sharp t = \#\langle a, t\rangle$, hence $\sharp t = 1 + (\Sigma\, x : x \in t : \#x)$, and writing x as $\langle b, u\rangle$, this may be reformulated as:

$$\alpha \left(\frac{\Psi.(y \dashv \langle a, \{\langle b, u\rangle\} \cup t\rangle)}{\Psi.(y \dashv \langle b, u \cup \{\langle a, t\rangle\}\rangle)} \right)^\beta \leq 1 + \frac{\sharp u}{\sharp t}.$$

Since y does not occur in the right-hand side, we want to eliminate y from the left-hand side as well. To cancel out y in the fraction we introduce

$$\Delta.y.x = \Psi.(y \dashv x)/\Psi.x,$$

so that

$$\frac{\Psi.(y \dashv \langle a, \{\langle b, u\rangle\} \cup t\rangle)}{\Psi.(y \dashv \langle b, u \cup \{\langle a, t\rangle\}\rangle)} = \frac{\Delta.y.\langle a, \{\langle b, u\rangle\} \cup t\rangle}{\Delta.y.\langle b, u \cup \{\langle a, t\rangle\}\rangle} \;\; \frac{\Psi.\langle a, \{\langle b, u\rangle\} \cup t\rangle}{\Psi.\langle b, u \cup \{\langle a, t\rangle\}\rangle}.$$

The important observation is now that the left factor reduces to 1, if $\Delta.y.x$ depends on y and $\#x$ only. This proviso means formally that

(3) $\quad \#x = \#x' \;\Rightarrow\; \Delta.y.x = \Delta.y.x'.$

Since Δ is defined in terms of Ψ this imposes an extra requirement on Ψ to which we return in a moment, but first we consider the right factor.

The arguments of Ψ in the numerator and denominator of the right factor have a lot structure in common, viz. the structure of t and u. To cancel out these common parts, our next step is to take a definition of Ψ of the following general form:

$$\begin{aligned} \Psi.\langle a, t\rangle &= \psi.t\ \Psi^*.t \\ \Psi^*.t &= (\Pi\, x : x \in t : \Psi.x). \end{aligned}$$

The right factor then becomes

$$\frac{\Psi.\langle a, \{\langle b,u\rangle\} \cup t\rangle}{\Psi.\langle b, u \cup \{\langle a,t\rangle\}\rangle} = \frac{\psi.(\{\langle b,u\rangle\} \cup t)}{\psi.(u \cup \{\langle a,t\rangle\})} \; \frac{\psi.u}{\Psi^*.u} \; \frac{\Psi^*.u}{\psi.t} \; \frac{\Psi^*.t}{\Psi^*.t},$$

which yields as requirement in terms of α, β, and ψ:

$$\alpha \left(\frac{\psi.(\{\langle b,u\rangle\} \cup t)}{\psi.(u \cup \{\langle a,t\rangle\})} \; \frac{\psi.u}{\psi.t} \right)^{\beta} \leq 1 + \frac{\sharp u}{\sharp t}.$$

Since the right-hand side depends on $\sharp t$ and $\sharp u$ only, we decide to let $\psi.t$ depend on $\sharp t$ in the simplest possible way:

$$\psi.t = \sharp t.$$

With this choice for ψ, (3) indeed holds, as we leave to the reader to verify. (In Section 11.4, we will show that $\psi.t$ cannot be substantially smaller than $\sharp t$.) Since $\sharp(\{\langle b,u\rangle\} \cup t) = \sharp(u \cup \{\langle a,t\rangle\})$, the above requirement reduces to

$$\alpha \left(\frac{\sharp u}{\sharp t} \right)^{\beta} \leq 1 + \frac{\sharp u}{\sharp t}.$$

As $\sharp t$ and $\sharp u$ range over all positive integers, α and β are thus required to satisfy: $\alpha > 1$, $\beta > 0$, and

$$(4) \quad (\forall\, m, n :: \alpha \left(\frac{m}{n} \right)^{\beta} \leq 1 + \frac{m}{n}).$$

To minimize the upper bound on $\mathcal{A}[a\angle\,]$, we maximize α under these constraints, and—with lower priority—we minimize β so as to keep Φ small. Assuming that $\alpha > 1$ we observe for (4):

$$\begin{array}{ll} & (\forall\, m, n :: \alpha(\frac{m}{n})^{\beta} \leq 1 + \frac{m}{n}) \\ \Rightarrow & \{ \text{ take } n = 1 \ \} \\ & (\forall\, m :: \alpha \leq \frac{m+1}{m^{\beta}}) \\ \Rightarrow & \{ \ 1 < \alpha \ \} \\ & (\forall\, m :: 1 < \frac{m+1}{m^{\beta}}) \\ \Rightarrow & \{ \text{ take } m \to \infty \ \} \\ & \beta < 1, \end{array}$$

hence β must be smaller than 1. Therefore we assume $0 < \beta < 1$, and we derive:

$$\begin{array}{ll} & (\forall\, m, n :: \alpha(\frac{m}{n})^{\beta} \leq 1 + \frac{m}{n}) \\ \equiv & \{ \ p = \frac{m}{n} \ \} \\ & (\forall\, p : p > 0 : \alpha \leq \frac{p+1}{p^{\beta}}) \\ \equiv & \{ \ \frac{x+1}{x^{\beta}} \text{ is minimal at } x = \frac{\beta}{1-\beta}, \text{ which is positive because } 0 < \beta < 1 \ \} \\ & \alpha \leq \frac{1}{\beta^{\beta}(1-\beta)^{1-\beta}}, \end{array}$$

which results in the following constraints on α and β:

$$1 < \alpha \leq \frac{1}{\beta^\beta (1-\beta)^{1-\beta}}$$
$$0 < \beta < 1.$$

To maximize α, we determine the maximum of its upper bound over all β between 0 and 1. This yields 2 as maximal value for α at $\beta = \frac{1}{2}$.

On account of (2), we thus have

$$\mathcal{A}[a\angle\,](y \dashv \langle a,t\rangle) \leq \log_2 \frac{\#(y \dashv \langle a,t\rangle)}{\#\langle a,t\rangle},$$

which implies

$$\mathcal{A}[a\angle\,](x) \leq \log_2 \#x.$$

Moreover, we have obtained as potential:

$$\Phi.\langle a,t\rangle = \tfrac{1}{2}\log_2 \#\langle a,t\rangle + (\Sigma\, x : x \in t : \Phi.x),$$

for which we have as bounds:

$$0 \leq \Phi.x \leq \tfrac{1}{2}\#x \log_2 \#x.$$

Note that $\Phi.x$ is $\Omega(\#x \log \#x)$ if each node of x has at most one subtree.

11.1.3 Top-down analysis

From the top-down analysis of bottom-up skew heaps in Section 9.4.3, we learn that the analysis of a bottom-up operation on trees may be simplified by considering a top-down "simulation". In this section we show that the calculations are simplified somewhat by analyzing the following program for path reversal:

$$\begin{array}{lllll}
\text{(i)} & a\angle\,\langle a,t\rangle & = & \langle a,t\rangle & \\
\text{(ii)} & a\angle\,\langle b,u\cup\{x\}\rangle & \doteq & \langle a,\{\langle b,u\rangle\}\cup t\rangle & |[\ \langle a,t\rangle = a\angle\, x\ ]| \quad ,a \in x.
\end{array}$$

This top-down program yields the same result as the bottom-up program, using the same number of unfoldings. The use of $\in$, however, makes this program not realistic.

To obtain a recurrence relation for $\mathcal{A}[a\angle\,]$, we take a potential of the form

$$\begin{array}{lcl}
\Phi.\langle a,t\rangle & = & \varphi.t + \Phi^*.t \\
\Phi^*.t & = & (\Sigma\, x : x \in t : \Phi.x),
\end{array}$$

and derive for case (ii):

$$
\begin{array}{ll}
& \mathcal{A}[a\angle](\langle b, u \cup \{x\}\rangle) \\
= & \quad \{ \text{ definition of } \mathcal{A}[a\angle] \ \} \\
& \mathcal{T}[a\angle](\langle b, u \cup \{x\}\rangle) + \Phi.(a\angle \langle b, u \cup \{x\}\rangle) - \Phi.\langle b, u \cup \{x\}\rangle \\
= & \quad \{ \text{ definition of } a\angle \ \} \\
& 1 + \mathcal{T}[a\angle](x) + \Phi.\langle a, \{\langle b, u\rangle\} \cup t\rangle - \Phi.\langle b, u \cup \{x\}\rangle \\
= & \quad \{ \text{ definition of } \mathcal{A}[a\angle];\ a\angle x = \langle a, t\rangle \ \} \\
& 1 + \mathcal{A}[a\angle](x) + \Phi.x - \Phi.\langle a, t\rangle + \Phi.\langle a, \{\langle b, u\rangle\} \cup t\rangle - \Phi.\langle b, u \cup \{x\}\rangle \\
= & \quad \{ \text{ definition of } \Phi \ \} \\
& 1 + \mathcal{A}[a\angle](x) + \Phi.x - \varphi.t - \Phi^*.t + \varphi.(\{\langle b, u\rangle\} \cup t) + \varphi.u + \Phi^*.u + \Phi^*.t \\
& -\varphi.(u \cup \{x\}) - \Phi^*.u - \Phi.x \\
= & \quad \{ \ \} \\
& \mathcal{A}[a\angle](x) + 1 - \varphi.t + \varphi.(\{\langle b, u\rangle\} \cup t) + \varphi.u - \varphi.(u \cup \{x\}).
\end{array}
$$

Observing that $\sharp(\{\langle b, u\rangle\} \cup t)$ is equal to $\sharp(u \cup \{x\})$ because $\#x = \#(a\angle x) = \sharp t$, the recurrence relation reduces to

$$
\begin{array}{lcl}
\mathcal{A}[a\angle](\langle a, t\rangle) & = & 0 \\
\mathcal{A}[a\angle](\langle b, u \cup \{x\}\rangle) & = & \mathcal{A}[a\angle](x) + 1 + \varphi.u - \varphi.t,
\end{array}
$$

provided $\varphi.t$ depends on $\sharp t$ only.

From this recurrence relation, it follows by induction that

$$\mathcal{A}[a\angle](x) \leq \log_\alpha \#x,$$

with $\alpha > 1$, provided φ satisfies

$$1 + \varphi.u - \varphi.t \leq \log_\alpha \frac{\#\langle b, u \cup \{x\}\rangle}{\#x}.$$

To remove $\log_\alpha$ from this requirement we define

$$\varphi.t = \beta \log_\alpha \sharp t,$$

with $\beta > 0$, so that $\varphi.t$ depends on $\sharp t$ in a simple way. This yields

$$\alpha \left(\frac{\sharp u}{\sharp t}\right)^\beta \leq \frac{\sharp u + \sharp t}{\sharp t},$$

since $\#x = \sharp t$. The analysis can now be completed as in the previous section, which leads to the conclusion that

$$\mathcal{A}[a\angle](x) \leq \log_2 \#x.$$

Remark 11.1
In the bottom-up analysis we used the following induction hypothesis:

$$\mathcal{A}[a\angle](y, \langle a, t\rangle) \leq \log_2 \frac{\#(y, \langle a, t\rangle)}{\#\langle a, t\rangle}.$$

This stronger result can also be obtained in a top-down fashion by taking it as induction hypothesis.
□

11.1.4 Series of path reversals

We want to determine the cost of a series of path reversals performed on the same tree. Therefore, we study $\mathcal{T}[\angle^*](s,x)$, with operation $\angle^*$ performing a series of reversals:

$$\begin{array}{lcl} [\,]\angle^* x & = & x \\ (s \dashv a)\angle^* x & = & s\angle^*(a\angle\, x). \end{array}$$

To amortize the cost of the applications of $\angle$, we introduce amortized costs for $\angle^*$:

$$\mathcal{A}[\angle^*](s,x) = \mathcal{T}[\angle^*](s,x) + \Phi.(s\angle^* x) - \Phi.x,$$

for which we have as recurrence relation:

$$\begin{array}{lcl} \mathcal{A}[\angle^*]([\,],x) & = & 0 \\ \mathcal{A}[\angle^*](s \dashv a,x) & = & \mathcal{A}[\angle^*](s,a\angle\, x) + \mathcal{A}[\angle\,](a,x). \end{array}$$

Since $\mathcal{A}[a\angle\,](x) \leq \log_2 \#x$, this yields by induction:

$$\mathcal{A}[\angle^*](s,x) \leq \#s \log_2 \#x,$$

from which we conclude

$$\begin{array}{ll} & \mathcal{T}[\angle^*](s,x) \\ = & \quad \{ \text{ definition of } \mathcal{A}[\angle^*] \ \} \\ & \mathcal{A}[\angle^*](s,x) + \Phi.x - \Phi.(s\angle^* x) \\ \leq & \quad \{ \text{ above bound for } \mathcal{A}[\angle^*]; \text{ bounds for } \Phi \ \} \\ & \#s \log_2 \#x + \frac{1}{2}\#x \log_2 \#x. \end{array}$$

In [12], one is interested in the average cost per reversal when the number of reversals is significantly larger than the size of the tree. For this quantity we obtain the same bound as in [12]:

$$\frac{\mathcal{T}[\angle^*](s,x)}{\#s} \leq \log_2 \#x + \frac{\#x \log_2 \#x}{2\#s}.$$

Note that the average cost of one reversal tends to the amortized cost of one reversal when the number of reversals ($= \#s$) is large. Phrased differently: the potential difference over a whole sequence of reversals is negligible when the sequence is sufficiently long.

11.2 Splaying

Splaying is the basic operation in a particular implementation of so-called *dictionaries*. A dictionary is an algebra involving subsets of an infinite, linearly ordered set (see e.g. [1, p.108]). As a simple example of a dictionary, we consider the following algebra:

$$\mathsf{D} = (\,\{\text{Int}\} \mid \{\,\}, (=\{\,\}), \in, \oplus, \ominus\,).$$

In this algebra, set Int is used as representative of an infinite, linearly ordered set. Another example of such a set is the lexicographically ordered set of identifiers of a programming language. (In that case one often speaks of "symbol tables" instead of dictionaries.)

If we replace Int in the definition of D by a finite set like $[0..N)$, the resulting algebra can be implemented very efficiently, as shown in Chapter 7. The fact that D involves sets of unbounded size forms a major obstacle that essentially excludes the use of (finite) arrays.

11.2.1 Splay trees

In [29], Sleator and Tarjan have developed an efficient implementation of dictionaries, called *splay trees*. Although the main subject here is the splaying operation, we will briefly describe splay trees in this section by presenting a refinement ST of algebra D with the following signature:

$$\mathsf{ST} = (\,BST \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{member}, \mathsf{insert}, \mathsf{delete}\,).$$

Type BST is a subset of $\langle \text{Int} \rangle$, known as the set of *binary search trees*. It is defined by

$$BST = \{x \mid x \in \langle \text{Int} \rangle \ \wedge\ \overline{x} \text{ is increasing}\},$$

where $\overline{x}$ denotes the inorder traversal of binary tree x (see Section 1.1.4). Since an increasing list does not contain duplicates, each integer can occur at most once in a binary search tree. Consequently, a subtree may be identified by the value attached to its root, a fact that will be exploited in the next section. The abstraction function is defined by $[\![x]\!] = \{a \mid a \in x\}$.

In terms of operation $\angle$ ("splay") we now present programs for the operations of ST. The type of operation $\angle$ is $(\{-\infty\} \cup \text{Int} \cup \{\infty\}) \times BST \to BST$, and it is specified by

$$a \angle \langle\,\rangle = \langle\,\rangle,$$

and, for nonempty x:

$$(5) \quad \overline{a \angle\, x} = \overline{x} \ \wedge\ (\exists\, t, b, u : a \angle\, x = \langle t, b, u \rangle : \overline{t} +\!\!+ [a] +\!\!+ \overline{u} \text{ is increasing}).$$

So, while keeping the inorder traversal intact, splaying moves a value to the root which is either equal to a or "close to a". Note that for nonempty x, $\infty \angle\, x$ results in a tree whose root contains the maximum of x, and whose right subtree is empty. This property is exploited in the program for delete:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & \langle\,\rangle \\ \mathsf{isempty}.x & \doteq & x = \langle\,\rangle \end{array}$$

$$
\begin{array}{lcll}
\mathsf{member}.a.x & \doteq & x := a \angle x & \\
 & & ;\ (\ \mathrm{return}(\mathrm{false}) & ,x = \langle\,\rangle \\
 & & \ [\!]\ \mathrm{return}(a{=}b)\ \ |[\ \langle t,b,u\rangle = x\ ]| & ,x \neq \langle\,\rangle \\
 & & \ ) & \\
\mathsf{insert}.x.a & \doteq & \langle a\rangle & ,x = \langle\,\rangle \\
 & & [\!]\ (\ \langle t,a,\langle\,\rangle,b,u\rangle & ,a < b \\
 & & \ [\!]\ \langle\langle t,b,\langle\,\rangle\rangle,a,u\rangle & ,b < a \\
 & & \ )\ \ |[\ \langle t,b,u\rangle = a \angle x\ ]| & ,x \neq \langle\,\rangle \\
\mathsf{delete}.x.a & \doteq & (\ u & ,t = \langle\,\rangle \\
 & & \ [\!]\ \langle v,b,u\rangle\ \ |[\ \langle v,b,\langle\,\rangle\rangle = \infty \angle t\ ]| & ,t \neq \langle\,\rangle \\
 & & \ )\ \ |[\ \langle t,a,u\rangle = a \angle x\ ]|. &
\end{array}
$$

The program for member is an example of a program with benevolent side-effects. For the analysis we introduce program member′ (cf. Sections 4.7 and 5.5):

$$
\begin{array}{lcll}
\mathsf{member}'.a.x & = & (\mathrm{false}, x) & ,x = \langle\,\rangle \\
 & & [\!]\ (a{=}b, \langle t,b,u\rangle)\ \ |[\ \langle t,b,u\rangle = a \angle x\ ]| & ,x \neq \langle\,\rangle.
\end{array}
$$

Remark 11.2
Starting from the definition of data refinement, the obvious implementation of $\in$ which exploits the fact that the concrete trees are binary search trees would be program f:

$$
\begin{array}{lcll}
f.a.\langle\,\rangle & = & \mathrm{false} & \\
f.a.\langle t,b,u\rangle & = & f.a.t & ,a < b \\
 & & [\!]\ \ \mathrm{true} & ,a = b \\
 & & [\!]\ \ f.a.u & ,a > b.
\end{array}
$$

But the performance of this program is poor: the evaluation of $f.a.x$ may take time proportional to $\#[\![x]\!]$ in the worst case, since trees in BST are not required to be balanced. To achieve logarithmic amortized costs for member, it is crucial that this operation transforms its tree-argument by means of operation $\angle$.
□

With member′ producing a tree, we can now associate a potential change with this "inspection" operation. The amortized costs are thus defined as follows:

$$
\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{isempty}](x) & = & \mathcal{T}[\mathsf{isempty}](x) \\
\mathcal{A}[\mathsf{member}'](a,x) & = & \mathcal{T}[\mathsf{member}'](a,x) + \Phi.(\mathsf{member}'.a.x.1) - \Phi.x \\
\mathcal{A}[\mathsf{insert}](x,a) & = & \mathcal{T}[\mathsf{insert}](x,a) + \Phi.(\mathsf{insert}.x.a) - \Phi.x \\
\mathcal{A}[\mathsf{delete}](x,a) & = & \mathcal{T}[\mathsf{delete}](x,a) + \Phi.(\mathsf{delete}.x.a) - \Phi.x,
\end{array}
$$

where $\mathcal{T}$ is an appropriate cost measure and Φ a potential. The first two operations evidently have $O(1)$ amortized costs, and splaying is implemented such that the amortized cost of each of the other operations is logarithmic.

11.2.2 Definition of splaying

In this section we introduce a simplified version of splaying which is somewhat easier to analyze. The results obtained from the analysis of this version, which are presented in the next section, are, however, sufficiently general.

We present two programs for the following restricted version of splaying, which has type $\mathrm{Int}\times BST \curvearrowright BST$, and satisfies (cf. (5)):

$$(6)\quad a \in x \;\Rightarrow\; \overline{a\angle\, x} = \overline{x} \;\wedge\; (\exists\, t, u :: a\angle\, x = \langle t, a, u\rangle).$$

So, we assume that a occurs in x, and therefore $a\angle\, x$ has root a.

Operation $(a\angle\,)$ ("splaying at a") is performed by "rotating" a towards the root, and it is the particular way these rotations are chosen that leads to an efficient implementation of splaying. Traditionally, splaying is explained by means of pictures (see Figure 11.2, transformation (ii) is called a *rotation at* a), but to allow for a systematic analysis of splaying, we introduce a special representation of binary trees so that we can describe splaying concisely and in a linear way—as opposed to a two-dimensional pictorial description. The appropriate representation to describe splaying is root-path view $\langle\mathrm{Int}\rangle_3$ from Section 6.2.3:

$$\langle\mathrm{Int}\rangle_3 = [\langle\mathrm{Int}\rangle\times\mathrm{Int} \cup \mathrm{Int}\times\langle\mathrm{Int}\rangle] \times \langle\mathrm{Int}\rangle.$$

In terms of this type, we obtain the following program for $\angle$ (cf. Figure 11.2):

$$\begin{array}{lll}
\text{(i)} & a\angle\,\langle t, a, u\rangle & = \langle t, a, u\rangle \\
\text{(ii)} & a\angle\,\langle\langle t, a, u\rangle, b, v\rangle & = \langle t, a, \langle u, b, v\rangle\rangle \\
\text{(iii)} & a\angle\,(y, \langle\langle\langle t, a, u\rangle, b, v\rangle, c, w\rangle) & = a\angle\,(y, \langle t, a, \langle u, b, \langle v, c, w\rangle\rangle\rangle) \\
\text{(iv)} & a\angle\,(y, \langle\langle v, b, \langle t, a, u\rangle\rangle, c, w\rangle) & = a\angle\,(y, \langle\langle v, b, t\rangle, a, \langle u, c, w\rangle\rangle),
\end{array}$$

where the symmetrical counterparts of (ii), (iii), and (iv), which are obtained by interchanging the role of left and right subtrees, have been omitted. (In the sequel, "symmetry" refers to this kind of symmetry.) In this program we have ignored the fact that the position of a is actually determined by starting a search at the root (as in program f in Remark 11.2); this is safe because the cost of this search is proportional to the cost of moving a to the root.

It is easily seen that this program satisfies specification (6). Note that in cases (i), (ii), and (iv) the right-hand side is fully determined by the left-hand side provided one abstains from further decomposition of the components occurring in the left-hand sides. In case (iii), however, $a\angle\,(y, \langle t, a, \langle\langle u, b, v\rangle, c, w\rangle\rangle)$ is the only remaining alternative for $a\angle\,(y, \langle t, a, \langle u, b, \langle v, c, w\rangle\rangle\rangle)$. But inspection of the resulting program learns that it is equivalent to rotating a towards the root by means of single rotations only:

$$\begin{array}{lll}
a\angle\,\langle t, a, u\rangle & = & \langle t, a, u\rangle \\
a\angle\,(y, \langle\langle t, a, u\rangle, b, v\rangle) & = & a\angle\,(y, \langle t, a, \langle u, b, v\rangle\rangle) \\
a\angle\,(y, \langle v, b, \langle u, a, t\rangle\rangle) & = & a\angle\,(y, \langle\langle v, b, u\rangle, a, t\rangle),
\end{array}$$

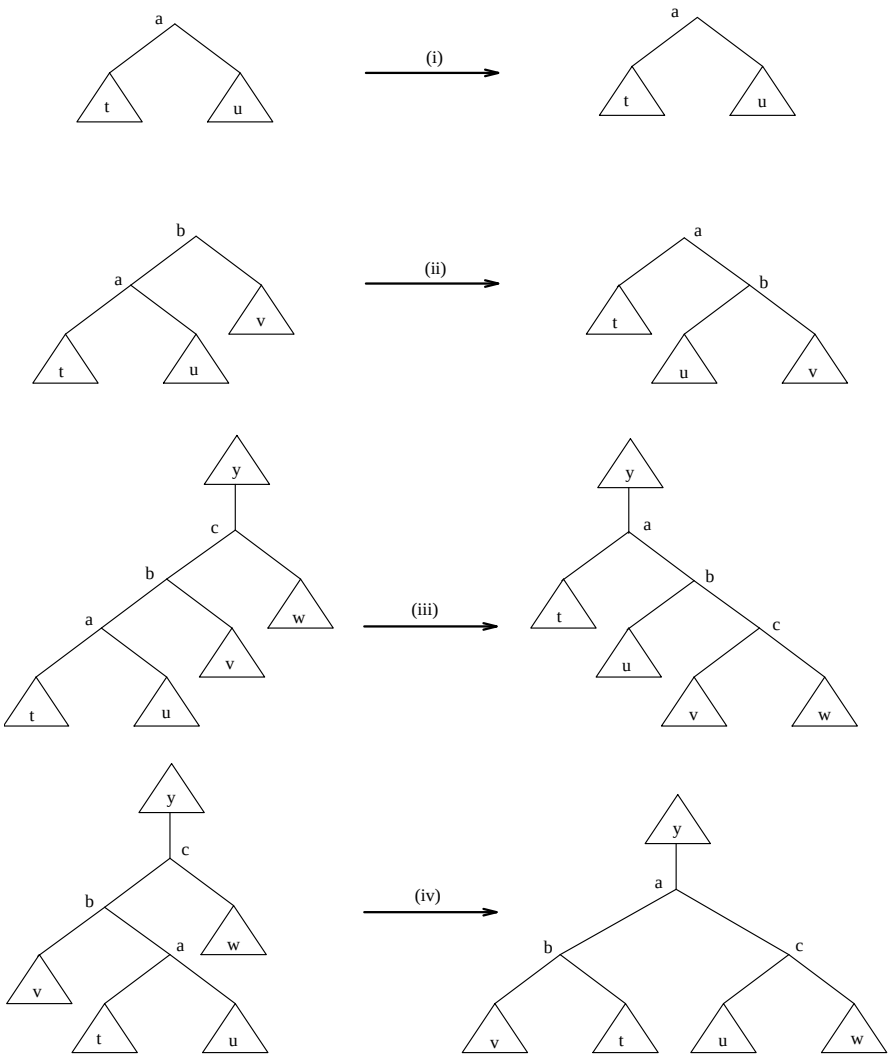

Figure 11.2: Splaying at a (cf. [29, Figure 3]).

and, consequently, the efficiency of this program is poor. (See also Remark 11.3.) The program merely distinguishes the cases "depth of a is 0" and "depth of a is greater than 0". To achieve the desired efficiency it is necessary to single out the case "depth of a is 1" and to act in the particular way described above in case "depth of a is greater than 1".

Instead of splaying in a bottom-up fashion as depicted in Figure 11.2, we may as well do it in a top-down fashion. In [29], Sleator and Tarjan present a rather complicated *iterative* method of top-down splaying. Doing this in a *recursive* way in terms of type $\langle \mathrm{Int} \rangle$, we obtain a much simpler description of top-down splaying:

$$
\begin{array}{lllll}
\text{(i)} & a\angle\,\langle t,a,u\rangle & = & \langle t,a,u\rangle & \\
\text{(ii)} & a\angle\,\langle\langle t,a,u\rangle,b,v\rangle & = & \langle t,a,\langle u,b,v\rangle\rangle & ,a{<}b \\
\text{(iii)} & a\angle\,\langle\langle x,b,v\rangle,c,w\rangle & \doteq & \langle t,a,\langle u,b,\langle v,c,w\rangle\rangle\rangle \;\; |[\; \langle t,a,u\rangle = a\angle\, x \;]| & ,a{<}b{<}c \\
\text{(iv)} & a\angle\,\langle\langle v,b,x\rangle,c,w\rangle & \doteq & \langle\langle v,b,t\rangle,a,\langle u,c,w\rangle\rangle \;\; |[\; \langle t,a,u\rangle = a\angle\, x \;]| & ,b{<}a{<}c.
\end{array}
$$

In this program the same transformations are performed as in Figure 11.2, but if the depth of a is odd, the result is a bit different. Since algebra T from Section 4.3 can be implemented in a nondestructive way, a purely-functional implementation of splay trees can be obtained from this program for $\angle$.

The cost measure defined by the dots in this program will be denoted by $\mathcal{T}$. Then $\mathcal{T}[a\angle\,](x)$ is approximately equal to the number of integer comparisons required for the evaluation of $a\angle\, x$ ($\mathcal{T}[a\angle\,](x)+2$ is a tight upper bound).

11.2.3 Analysis of top-down splaying

The analysis of top-down splaying follows the top-down analysis of path reversal very closely. To avoid too much duplication of that analysis, the derivation of the potential is therefore shortened a bit by drawing some conclusions from that analysis.

We try to derive a logarithmic bound for the amortized cost of $(a\angle\,)$, given by

$$\mathcal{A}[a\angle\,](x) = \mathcal{T}[a\angle\,](x) + \Phi.(a\angle\, x) - \Phi.x.$$

More precisely, we want to find a potential Φ such that

$$\text{(7)} \quad \mathcal{A}[a\angle\,](x) \leq \log_\alpha \sharp x,$$

with $\alpha{>}1$ and as large as possible. Recall that $\sharp x$ denotes the size of x plus one, which may defined recursively by

$$
\begin{array}{lcl}
\sharp\langle\,\rangle & = & 1 \\
\sharp\langle t,a,u\rangle & = & \sharp t + \sharp u.
\end{array}
$$

In addition, we want Φ to be nonnegative and small. Since we have omitted the symmetrical counterparts of cases (ii)–(iv) we also require Φ to be symmetric in order to cover these cases.

Setting out for an inductive argument, we first derive a recurrence relation for $\mathcal{A}[a\angle\,]$. To this end, we take Φ of the form:

$$\begin{array}{lcl}\Phi.\langle\,\rangle & = & 0\\ \Phi.\langle t,a,u\rangle & = & \Phi.t+\varphi.t.u+\Phi.u.\end{array}$$

Then we observe for case (iii):

$$\begin{array}{ll} & \mathcal{A}[a\angle\,](\langle\langle x,b,v\rangle,c,w\rangle)\\ = & \quad\{\text{ definition of }\mathcal{A}[a\angle\,]\ \}\\ & \mathcal{T}[a\angle\,](\langle\langle x,b,v\rangle,c,w\rangle)+\Phi.(a\angle\,\langle\langle x,b,v\rangle,c,w\rangle)-\Phi.\langle\langle x,b,v\rangle,c,w\rangle\\ = & \quad\{\text{ definition of }a\angle\ \}\\ & 2+\mathcal{T}[a\angle\,](x)+\Phi.\langle t,a,\langle u,b,\langle v,c,w\rangle\rangle\rangle-\Phi.\langle\langle x,b,v\rangle,c,w\rangle\\ = & \quad\{\text{ definition of }\mathcal{A}[a\angle\,];\ a\angle\,x=\langle t,a,u\rangle\ \}\\ & 2+\mathcal{A}[a\angle\,](x)+\Phi.x-\Phi.\langle t,a,u\rangle\\ & +\Phi.\langle t,a,\langle u,b,\langle v,c,w\rangle\rangle\rangle-\Phi.\langle\langle x,b,v\rangle,c,w\rangle\\ = & \quad\{\text{ definition of }\Phi\ \}\\ & 2+\mathcal{A}[a\angle\,](x)+\Phi.x-\Phi.t-\varphi.t.u-\Phi.u\\ & +\Phi.t+\varphi.t.\langle u,b,\langle v,c,w\rangle\rangle+\Phi.u+\varphi.u.\langle v,c,w\rangle+\Phi.v+\varphi.v.w+\Phi.w\\ & -\Phi.x-\varphi.x.v-\Phi.v-\varphi.\langle x,b,v\rangle.w-\Phi.w\\ = & \quad\{\ \ \}\\ & 2+\mathcal{A}[a\angle\,](x)-\varphi.t.u+\varphi.t.\langle u,b,\langle v,c,w\rangle\rangle+\varphi.u.\langle v,c,w\rangle\\ & +\varphi.v.w-\varphi.x.v-\varphi.\langle x,b,v\rangle.w.\end{array}$$

As in the analysis of path reversal, the recurrence relation can be simplified by observing that $\sharp t+\sharp\langle u,b,\langle v,c,w\rangle\rangle=\sharp\langle x,b,v\rangle+\sharp w$ because $\sharp(a\angle\,x)=\sharp x$ and $a\angle\,x=\langle t,a,u\rangle$. Hence, provided $\varphi.t.u$ depends on $\sharp t+\sharp u$ only, the amortized cost of an unfolding of case (iii) simplifies to

$$2+\varphi.u.\langle v,c,w\rangle+\varphi.v.w-\varphi.x.v-\varphi.t.u.$$

Omitting the calculations for the other cases, we thus have

$$\begin{array}{lcl}\mathcal{A}[a\angle\,](\langle t,a,u\rangle) & = & 0\\ \mathcal{A}[a\angle\,](\langle\langle t,a,u\rangle,b,v\rangle) & = & \varphi.u.v-\varphi.t.u\\ \mathcal{A}[a\angle\,](\langle\langle x,b,v\rangle,c,w\rangle) & = & \mathcal{A}[a\angle\,](x)+2+\varphi.u.\langle v,c,w\rangle+\varphi.v.w-\varphi.x.v-\varphi.t.u\\ \mathcal{A}[a\angle\,](\langle\langle v,b,x\rangle,c,w\rangle) & = & \mathcal{A}[a\angle\,](x)+2+\varphi.v.t+\varphi.u.w-\varphi.v.x-\varphi.t.u.\end{array}$$

It is obvious that (7) holds in case (i). In order that (7) follows by induction in the other cases, the following requirements must be satisfied:

$$\begin{array}{lcl}\varphi.u.v-\varphi.t.u & \le & \log_\alpha(\sharp t+\sharp u+\sharp v)\\ 2+\varphi.u.\langle v,c,w\rangle+\varphi.v.w-\varphi.x.v-\varphi.t.u & \le & \log_\alpha\frac{\sharp t+\sharp u+\sharp v+\sharp w}{\sharp t+\sharp u}\\ 2+\varphi.v.t+\varphi.u.w-\varphi.v.x-\varphi.t.u & \le & \log_\alpha\frac{\sharp t+\sharp u+\sharp v+\sharp w}{\sharp t+\sharp u}.\end{array}$$

To remove $\log_\alpha$ from these requirements, we define

$$\varphi.t.u = \beta \log_\alpha(\sharp t + \sharp u),$$

with $\beta > 0$. The requirements then reduce to

$$\begin{aligned}
\left(\frac{\sharp u+\sharp v}{\sharp t+\sharp u}\right)^\beta &\leq \sharp t + \sharp u + \sharp v \\
\alpha^2 \left(\frac{\sharp u+\sharp v+\sharp w}{\sharp t+\sharp u+\sharp v}\,\frac{\sharp v+\sharp w}{\sharp t+\sharp u}\right)^\beta &\leq 1 + \frac{\sharp v+\sharp w}{\sharp t+\sharp u} \\
\alpha^2 \left(\frac{\sharp v+\sharp t}{\sharp v+\sharp t+\sharp u}\,\frac{\sharp u+\sharp w}{\sharp t+\sharp u}\right)^\beta &\leq 1 + \frac{\sharp v+\sharp w}{\sharp t+\sharp u}.
\end{aligned}$$

As constraints on α and β we thus have: $\alpha > 1$, $\beta > 0$, and

$$(8) \quad (\forall\, k,l,m :: (\frac{l+m}{k+l})^\beta \leq k+l+m)$$

$$(9) \quad (\forall\, k,l,m,n :: \alpha^2(\frac{l+m+n}{k+l+m}\,\frac{m+n}{k+l})^\beta \leq 1 + \frac{m+n}{k+l})$$

$$(10) \quad (\forall\, k,l,m,n :: \alpha^2(\frac{k+m}{k+l+m}\,\frac{l+n}{k+l})^\beta \leq 1 + \frac{m+n}{k+l}).$$

Under these constraints, we want to maximize α and, with lower priority, we want to minimize β so as to keep Φ small.

We distinguish two cases.

Case $\beta < \frac{1}{2}$. Lemma 11.5 at the end of this section states that for $0 < \beta < \frac{1}{2}$:

$$\begin{aligned}
(8) \quad &\equiv \quad \beta \leq 1 \\
(9) \quad &\equiv \quad \alpha^2 \leq \tfrac{(1-\beta)^{1-\beta}}{\beta^\beta (1-2\beta)^{1-2\beta}} \\
(10) \quad &\equiv \quad \alpha^2 \leq 4^\beta,
\end{aligned}$$

so, taking the conjunction of the requirements on α and β, we obtain:

$$1 < \alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^\beta (1-2\beta)^{1-2\beta}} \min 4^\beta$$

$$0 < \beta < \frac{1}{2}.$$

To maximize α, we determine the maximum of its upper bound over all β satisfying $0 < \beta < \frac{1}{2}$. This yields $\sqrt[3]{4}$ as maximal value for α^2 for $\beta = \frac{1}{3}$ (see Figure 11.3).

Case $\beta \geq \frac{1}{2}$. Instantiation of (9) with "$k,l,m,n := 1,1,1,\infty$" yields that $\alpha^2 \leq \sqrt{\frac{3}{2}}$.

Since $\sqrt{\frac{3}{2}} < \sqrt[3]{4}$, we conclude from this case analysis that the optimal values for α and β are given by $\alpha = \sqrt[3]{2}$ and $\beta = \frac{1}{3}$.

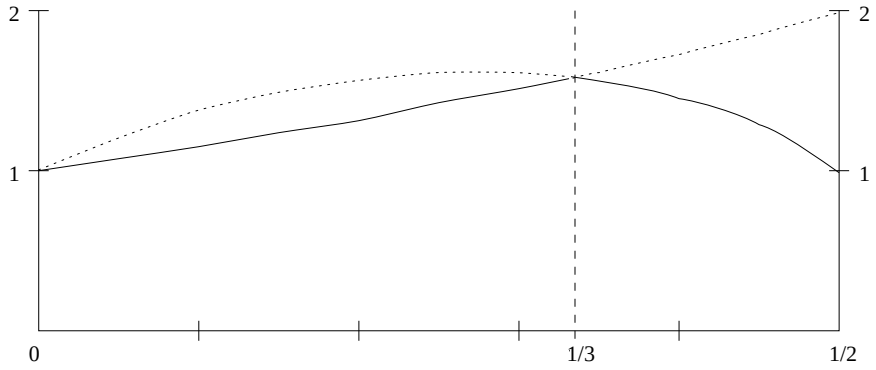

Figure 11.3: Maximum at $\beta = \frac{1}{3}$.

As result for the amortized costs of splaying we thus have (cf. (7)):

$$\mathcal{A}[a\angle\,](x) \leq 1 + 3\log_2 \sharp x.$$

Furthermore, we have derived the following potential:

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t + \log_2 \sharp\langle t,a,u\rangle + \Phi.u. \end{array}$$

Note that $\Phi.x$ is $\Omega(\sharp x \log \sharp x)$ if each node of x has at most one nonempty subtree. It would be interesting to know whether there exists a linear potential for splaying; this question will be addressed in Section 11.4.

We end this section with two remarks and the promised lemma.

Remark 11.3
The remaining alternative for (iii) mentioned in Section 11.2.2,

$$a\angle\,\langle\langle x,b,v\rangle,c,w\rangle \doteq \langle t,a,\langle\langle u,b,v\rangle,c,w\rangle\rangle \quad |[\ \langle t,a,u\rangle = a\angle\, x\ ]| \quad , a{<}b{<}c,$$

drops out because it gives rise to constraint

$$(\forall\, k,l,m,n :: \alpha^2(\frac{l+m+n}{k+l+m}\ \frac{l+m}{k+l})^\beta \leq 1 + \frac{m+n}{k+l}).$$

This implies that $\alpha \leq 1$ if we instantiate it with "$k,l,m,n := 1,\infty,1,1$", which conflicts with $\alpha > 1$.
□

Remark 11.4
The maximal value of $\frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}}$ turns out to be $\phi = \frac{\sqrt{5}+1}{2}$, the golden ratio again (for $\beta = \frac{5-\sqrt{5}}{10}$ (≈ 0.276)). So, if it were only for case (iii), constant 3 in the bound for $\mathcal{A}[a\angle\,]$ could be replaced by $2/\log_2 \phi$ (≈ 2.88).
□

Lemma 11.5
For $0 < \beta < \frac{1}{2}$,

(a) $(\forall\, k,l,m :: \left(\frac{l+m}{k+l}\right)^{\beta} \leq k+l+m) \;\equiv\; \beta \leq 1$

(b) $(\forall\, k,l,m,n :: \alpha^2 \left(\frac{l+m+n}{k+l+m}\;\frac{m+n}{k+l}\right)^{\beta} \leq 1 + \frac{m+n}{k+l}) \;\equiv\; \alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^{\beta}(1-2\beta)^{1-2\beta}}$

(c) $(\forall\, k,l,m,n :: \alpha^2 \left(\frac{k+m}{k+l+m}\;\frac{l+n}{k+l}\right)^{\beta} \leq 1 + \frac{m+n}{k+l}) \;\equiv\; \alpha^2 \leq 4^{\beta}.$

Proof The facts are proven by mutual implication.
Proof of (a), using that $\beta > 0$:

$(\forall\, k,l,m :: (\frac{l+m}{k+l})^{\beta} \leq k+l+m)$
$\Leftarrow$ { $\beta > 0$ }
$(\forall\, k,l,m :: (\frac{k+l+m}{k+l})^{\beta} \leq \frac{k+l+m}{k+l})$
$\equiv$ { }
$\beta \leq 1;$

$(\forall\, k,l,m :: (\frac{l+m}{k+l})^{\beta} \leq k+l+m)$
$\Rightarrow$ { take $k = 1$ and $l = 1$ }
$(\forall\, m :: (\frac{m+1}{2})^{\beta} \leq m+2)$
$\Rightarrow$ { take $m \to \infty$ }
$\beta \leq 1.$

Proof of (b), using that $0 < \beta < \frac{1}{2}$:

$(\forall\, k,l,m,n :: \alpha^2(\frac{l+m+n}{k+l+m}\;\frac{m+n}{k+l})^{\beta} \leq 1 + \frac{m+n}{k+l})$
$\Leftarrow$ { $\beta > 0$ }
$(\forall\, k,l,m,n :: \alpha^2(\frac{k+l+m+n}{k+l}\;\frac{m+n}{k+l})^{\beta} \leq 1 + \frac{m+n}{k+l})$
$\equiv$ { $p = \frac{m+n}{k+l}$ }
$(\forall\, p : p > 0 : \alpha^2((1+p)p)^{\beta} \leq 1+p)$
$\equiv$ { }
$(\forall\, p : p > 0 : \alpha^2 \leq \frac{(1+p)^{1-\beta}}{p^{\beta}})$
$\equiv$ { minimize $\frac{(1+x)^{1-\beta}}{x^{\beta}}$, using $0 < \beta < \frac{1}{2}$ (see below) }
$\alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^{\beta}(1-2\beta)^{1-2\beta}};$

$$
\begin{array}{ll}
 & (\forall\, k,l,m,n :: \alpha^2(\frac{l+m+n}{k+l+m}\ \frac{m+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\Rightarrow & \quad \{\ \text{take } k=m=1\ \} \\
 & (\forall\, l,n :: \alpha^2(\frac{l+n+1}{l+2}\ \frac{n+1}{l+1})^\beta \le 1+\frac{n+1}{l+1}) \\
\equiv & \quad \{\ \ \} \\
 & (\forall\, l,n :: \alpha^2 \le (1+\frac{n+1}{l+1})(\frac{l+2}{l+n+1}\ \frac{l+1}{n+1})^\beta) \\
\Rightarrow & \quad \{\ \text{take } \frac{n}{l} \to \frac{\beta}{1-2\beta} \text{ and } l \to \infty\ (\frac{\beta}{1-2\beta} > 0, \text{ since } 0<\beta<\frac{1}{2})\ \} \\
 & \alpha^2 \le (1+\frac{\beta}{1-2\beta})(\frac{1}{1+\frac{\beta}{1-2\beta}}\ \frac{1-2\beta}{\beta})^\beta \\
\equiv & \quad \{\ 1+\frac{\beta}{1-2\beta} = \frac{1-\beta}{1-2\beta}\ \} \\
 & \alpha^2 \le \frac{1-\beta}{1-2\beta}(\frac{1-2\beta}{1-\beta}\ \frac{1-2\beta}{\beta})^\beta \\
\equiv & \quad \{\ \ \} \\
 & \alpha^2 \le \frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}}.
\end{array}
$$

To complete the proof of this part, we minimize $f(x) = \frac{(1+x)^{1-\beta}}{x^\beta}$ over all $x > 0$. Then $f'(x) = 0$ is equivalent to $(1-\beta)x = (1+x)\beta$, and f turns out to be minimal at $\frac{\beta}{1-2\beta}$, which is positive because $0 < \beta < \frac{1}{2}$. So the minimum of f equals $\frac{(1+\frac{\beta}{1-2\beta})^{1-\beta}}{(\frac{\beta}{1-2\beta})^\beta}$, which in turn equals $\frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}}$, since $1+\frac{\beta}{1-2\beta} = \frac{1-\beta}{1-2\beta}$.

Proof of (c), using that $0 < \beta < \frac{1}{2}$:

$$
\begin{array}{ll}
 & (\forall\, k,l,m,n :: \alpha^2(\frac{k+m}{k+l+m}\ \frac{l+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\Leftarrow & \quad \{\ \beta > 0\ \} \\
 & (\forall\, k,l,m,n :: \alpha^2(\frac{k+m}{k+l}\ \frac{l+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\equiv & \quad \{\ p=\frac{k+m}{k+l} \text{ and } q=\frac{l+n}{k+l}\ \} \\
 & (\forall\, p,q : p>0\ \wedge\ q>0\ \wedge\ p+q \ge 1 : \alpha^2(pq)^\beta \le p+q) \\
\equiv & \quad \{\ \ \} \\
 & (\forall\, p,q : p>0\ \wedge\ q>0\ \wedge\ p+q \ge 1 : \alpha^2 \le \frac{p+q}{(pq)^\beta}) \\
\equiv & \quad \{\ \text{minimize } \frac{x+y}{(xy)^\beta}, \text{ using } 0<\beta<\frac{1}{2} \text{ (see below)}\ \} \\
 & \alpha^2 \le 4^\beta;
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall\, k,l,m,n :: \alpha^2(\frac{k+m}{k+l+m}\ \frac{l+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\Rightarrow & \quad \{\ \text{take } l=k \text{ and } m=n=1\ \} \\
 & (\forall\, k :: \alpha^2(\frac{k+1}{2k+1}\ \frac{k+1}{2k})^\beta \le 1+\frac{2}{2k}) \\
\equiv & \quad \{\ \ \} \\
 & (\forall\, k :: \alpha^2 \le (1+\frac{1}{k})(\frac{2k+1}{k+1}\ \frac{2k}{k+1})^\beta) \\
\Rightarrow & \quad \{\ \text{take } k \to \infty\ \} \\
 & \alpha^2 \le 4^\beta.
\end{array}
$$

We determine the minimum of $\frac{x+y}{(xy)^\beta}$ over all positive x and y satisfying $x+y \geq 1$. To this end, we first observe that $\frac{x+y}{(xy)^\beta}$ takes its minimal value only if $x = y$ because $(xy)^\beta$ is maximized by taking x equal to y ($\beta > 0$), and this can be achieved without changing the value of $x + y$. We thus minimize $2x^{1-2\beta}$ over all $x \geq \frac{1}{2}$. Since this function is increasing in x ($\beta < \frac{1}{2}$), it attains its minimal value of 4^β at $x = \frac{1}{2}$.
□

11.2.4 Bounds for splay trees

Cost measure $\mathcal{T}$ has been chosen such that $\mathcal{T}[a\angle](x)$ is (as good as equal to) the number of comparisons needed to locate a in x (cf. program f in Remark 11.2). This measure corresponds to Sleator and Tarjan's measure in [29]. The amortized costs satisfy

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{isempty}](x) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{member}](a,x) & \leq & 3\log_2(1{+}\#[\![x]\!]) + O(1) \text{ comparisons} \\
\mathcal{A}[\mathsf{insert}](x,a) & \leq & 4\log_2(1{+}\#[\![x]\!]) + O(1) \text{ comparisons} \\
\mathcal{A}[\mathsf{delete}](x,a) & \leq & 6\log_2(1{+}\#[\![x]\!]) + O(1) \text{ comparisons.}
\end{array}$$

For potential Φ we have as bounds

$$0 \leq \Phi.x \leq (1{+}\#[\![x]\!])\log_2(1{+}\#[\![x]\!]).$$

11.3 Pairing

In [7], Fredman, Sedgewick, Sleator, and Tarjan propose *pairing heaps* as practical alternative for Fibonacci heaps. The central operation of this data structure, called *pairing*, transforms a list of heaps into a single heap by repeatedly linking two heaps in a specific order. In this section we consider one of the numerous variants of pairing, viz. the so-called *two-pass variant*.

11.3.1 Pairing heaps

Pairing heaps can be used to implement the following algebra of priority queues, which is about the same as algebra PQ' from Chapter 10:

$$\mathsf{PQ}'' = (\,\{\mathrm{Int}\} \mid \{\,\}, (=\{\,\}), \{\cdot\}, \cup, \downarrow, \Downarrow, \mathit{deckey}\,).$$

As signature for the implementation of this algebra we use

$$\mathsf{PH} = (\,P \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{single}, \mathsf{union}, \mathsf{min}, \mathsf{delmin}, \mathsf{deckey}\,).$$

Characteristic of pairing heaps is that P is a subset of tree type

$$\langle \mathrm{Int} \rangle_9 = \{\langle\,\rangle\} \cup \langle \mathrm{Int} \rangle_4,$$

where type $\langle \mathrm{Int} \rangle_4$ has been defined in Section 6.3.1 as the smallest solution of

$$X : \quad X = \mathrm{Int} \times [X].$$

Note that for a tree $\langle a, t \rangle$ in $\langle \mathrm{Int} \rangle_9$, the trees in t are nonempty—as they are elements of $\langle \mathrm{Int} \rangle_4$. Data type P is now, informally, defined as the set of trees of type $\langle \mathrm{Int} \rangle_4$ that are free of duplicates and satisfy the heap condition.

Again, the fundamental operation on heaps is linking, which is defined on trees of type $\langle \mathrm{Int} \rangle_9$ by

$$\begin{array}{rcll} \langle\,\rangle & \bowtie & y & \doteq & y & \\ x & \bowtie & \langle\,\rangle & \doteq & x & \\ \langle a, t \rangle & \bowtie & \langle b, u \rangle & \doteq & \langle a, \langle b, u \rangle \vdash t \rangle & , a < b \\ & & & [] & \langle b, \langle a, t \rangle \vdash u \rangle & , b < a. \end{array}$$

In terms of this operator, the operations of PH are defined as follows:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & \langle\,\rangle \\ \mathsf{isempty}.x & \doteq & x = \langle\,\rangle \\ \mathsf{single}.a & \doteq & \langle a, [\,] \rangle \\ \mathsf{union}.x.y & \doteq & x \bowtie y \\ \mathsf{min}.\langle a, t \rangle & \doteq & a \\ \mathsf{delmin}.\langle a, t \rangle & \doteq & cp.t. \end{array}$$

Function cp transforms a list of heaps into a single heap by repeatedly linking two heaps. This can be done in many ways. The method analyzed in [7] is the two-pass variant: in the first pass, heaps are linked *pairwise* from left to right; in the second pass, the resulting heaps are combined from right to left. These two passes are concisely encoded in the following program for cp ("pair and combine", or "pairing" for short).

$$\begin{array}{lcl} cp.[\,] & = & \langle\,\rangle \\ cp.[x] & = & x \\ cp.(x \vdash y \vdash t) & = & (x \bowtie y) \bowtie cp.t. \end{array}$$

Note that the cost of pairing is charged to the applications of $\bowtie$.

To program deckey, we use the following root-path view of $\langle \mathrm{Int} \rangle_9$:

$$[\,[\langle \mathrm{Int} \rangle_9] \times \mathrm{Int} \times [\langle \mathrm{Int} \rangle_9]\,] \times \langle \mathrm{Int} \rangle_9,$$

which corresponds to root-path view $\langle T \rangle_8$ of tree type $\langle T \rangle_4$ from Section 6.3.2. In terms of this type, deckey is programmed as follows:

$$\mathsf{deckey}.a.k.(z, \langle a, t \rangle) \doteq z \bowtie \langle a{-}k, t \rangle,$$

where we have abbreviated $(z, \langle\,\rangle)$ to z, and $([\,], \langle a{-}k, t\rangle)$ to $\langle a{-}k, t\rangle$.

The cost measure defined by the dots in the above programs will be called $\mathcal{T}$. As for the complexity of these operations, it is still an open problem whether the amortized cost of deckey is $O(1)$. In the next section, we derive a logarithmic bound for the amortized cost of *cp*. The potential function obtained in this analysis yields a logarithmic bound for deckey.

11.3.2 Analysis of pairing

We use $\sharp t$ to denote $\#\langle a, t\rangle$, hence $\sharp t$ is positive for all lists t. The goal of the analysis of *cp* is to maximize α such that $\alpha > 1$ and

(11) $\mathcal{A}[cp](t) \leq \log_\alpha \sharp t,$

where $\mathcal{A}[cp](t)$ denotes the amortized cost of pairing, defined by

$$\mathcal{A}[cp](t) = \mathcal{T}[cp](t) + \Phi.(cp.t) - \Phi.t.$$

Note that $cp.t$ is a tree of type $\langle \mathrm{Int}\rangle_9$, whereas t is a list of type $[\langle \mathrm{Int}\rangle_9]$. Potential Φ is therefore defined both on $\langle \mathrm{Int}\rangle_9$ and $[\langle \mathrm{Int}\rangle_9]$:

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle a, t\rangle & = & \varphi.t + \Phi^*.t \\ \Phi.t & = & \varphi.t + \Phi^*.t, \ \text{ with } \ \Phi^*.t = (\Sigma\, x : x \in t : \Phi.x). \end{array}$$

Hence, $\Phi.\langle a, t\rangle = \Phi.t$. To achieve that Φ is nonnegative, we require φ to be nonnegative. In terms of the amortized costs of linking, given by

$$\mathcal{A}[\bowtie](x, y) = \mathcal{T}[\bowtie](x, y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y,$$

the following recurrence relation can now be derived from the program for *cp*:

$$\begin{array}{lcl} \mathcal{A}[cp]([\,]) & = & -\varphi.[\,] \\ \mathcal{A}[cp]([x]) & = & -\varphi.[x] \\ \mathcal{A}[cp](x \vdash y \vdash t) & = & \mathcal{A}[\bowtie](x, y) + \mathcal{A}[\bowtie](x \bowtie y, cp.t) - \varphi.(x \vdash y \vdash t) + \varphi.t \\ & & +\mathcal{A}[cp](t), \end{array}$$

using that $\Phi.(x \vdash y \vdash t) = \varphi.(x \vdash y \vdash t) + \Phi.x + \Phi.y + \Phi.t - \varphi.t$.

Since φ will be nonnegative, (11) evidently holds if $t = [\,]$ or $t = [x]$. It follows by induction in the remaining case, provided

(12) $\mathcal{A}[\bowtie](x, y) + \mathcal{A}[\bowtie](x \bowtie y, cp.t) - \varphi.(x \vdash y \vdash t) + \varphi.t \leq \log_\alpha \dfrac{\sharp(x \vdash y \vdash t)}{\sharp t}.$

To simplify this requirement on φ, we investigate $\mathcal{A}[\bowtie]$. For $x = \langle a, t\rangle$ and $y = \langle b, u\rangle$ we observe for the case $a < b$:

$$
\begin{array}{ll}
& \mathcal{A}[\bowtie](x,y) \\
= & \quad \{ \text{ definition of } \mathcal{A}[\bowtie] \ \} \\
& \mathcal{T}[\bowtie](x,y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y \\
= & \quad \{ \text{ definitions of } \bowtie \text{ and } \mathcal{T} \ \} \\
& 1 + \Phi.\langle a, y \vdash t\rangle - \Phi.\langle a, t\rangle - \Phi.y \\
= & \quad \{ \text{ definition of } \Phi \text{ and } \Phi^* \ \} \\
& 1 + \varphi.(y \vdash t) + \Phi.y + \Phi^*.t - \varphi.t - \Phi^*.t - \Phi.y \\
= & \quad \{ \ \} \\
& 1 + \varphi.(y \vdash t) - \varphi.t.
\end{array}
$$

Introducing

$$(13) \quad \delta.(y \vdash t) = \varphi.(y \vdash t) - \varphi.t,$$

we then have on account of symmetry

$$\mathcal{A}[\bowtie](x,y) = 1 + \begin{cases} \delta.(y \vdash t) & , a < b \\ \delta.(x \vdash u) & , b < a. \end{cases}$$

To avoid the case analysis in this result we let $\delta.t$ depend on $\sharp t$ only. Defining $\delta.\langle a, t\rangle = \delta.t$, we then have

$$(14) \quad \mathcal{A}[\bowtie](x,y) = 1 + \delta.(x \bowtie y),$$

since both $\sharp(y \vdash t)$ and $\sharp(x \vdash u)$ are equal to $\#(x \bowtie y)$.

In terms of δ, requirement (12) becomes:

$$(15) \quad 2 + \delta.(x \bowtie y) + \delta.((x \bowtie y) \bowtie cp.t) - \delta.(x \vdash y \vdash t) - \delta.(y \vdash t) \leq \log_\alpha \frac{\#x + \#y + \sharp t}{\sharp t}.$$

As in the previous analyses in this chapter we take

$$\delta.t = \beta \log_\alpha \sharp t,$$

and to achieve that φ is nonnegative we also take $\varphi.[\,] = 0$, which defines φ on account of (13).

Using that $\#(cp.t) = \sharp t - 1$, requirement (15) then reduces to:

$$\alpha^2 \left(\frac{\#x + \#y + \sharp t - 1}{\#x + \#y + \sharp t} \, \frac{\#x + \#y}{\#y + \sharp t} \right)^\beta \leq 1 + \frac{\#x + \#y}{\sharp t}.$$

In summary, we thus have that (11) holds and that Φ is nonnegative provided α and β satisfy: $\alpha > 1$, $\beta > 0$, and

$$(16) \quad (\forall\, k, m, n :: \alpha^2 \left(\frac{m+n+k-1}{m+n+k} \frac{m+n}{n+k} \right)^\beta \leq 1 + \frac{m+n}{k}).$$

Since $\frac{m+n+k-1}{m+n+k}$ is about 1, this set of requirements is very similar to the one in the analysis of path reversal. To maximize α, we distinguish two cases.

Case $\beta < 1$. Since constraint (16) is about equal to constraint (4) in the analysis of path reversal, we omit the calculations that lead to the following constraints on α and β:

$$1 < \alpha^2 \leq \frac{1}{\beta^\beta (1-\beta)^{1-\beta}}$$
$$0 < \beta < 1.$$

The optimal values thus are $\alpha = \sqrt{2}$ and $\beta = \frac{1}{2}$.

Case $\beta \geq 1$. Instantiating (16) with "$k, m, n := 1, \infty, 1$" yields that $\alpha^2 \leq 2$, hence $\alpha \leq \sqrt{2}$.

We conclude that the optimal values are $\alpha = \sqrt{2}$ and $\beta = \frac{1}{2}$. As result for $\mathcal{A}[cp]$ we thus have (on account of (11)):

$$\mathcal{A}[cp](t) \leq 2 \log_2 \sharp t,$$

for potential Φ defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle a, t\rangle & = & \Phi.t \\ \Phi.[\,] & = & 0 \\ \Phi.(x \vdash t) & = & \Phi.x + \log_2 \sharp(x \vdash t) + \Phi.t. \end{array}$$

11.3.3 Bounds for pairing heaps

To analyze the respective operations we recall the following result of the previous section (cf. (14)):

(17) $\Phi.(x \bowtie y) - \Phi.x - \Phi.y = \log_2 \#(x \bowtie y)$.

This yields for the first six operations:

$$\begin{array}{lcl} \mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\ \mathcal{A}[\mathsf{isempty}](x) & \text{is} & O(1) \\ \mathcal{A}[\mathsf{single}](a) & \text{is} & O(1) \\ \mathcal{A}[\mathsf{union}](x, y) & \leq & \log_2 \#[\![\mathsf{union}.x.y]\!] \text{ comparisons} \\ \mathcal{A}[\mathsf{min}](x) & \text{is} & O(1) \\ \mathcal{A}[\mathsf{delmin}](x) & \leq & 2 \log_2 \#[\![x]\!] \text{ comparisons.} \end{array}$$

These bounds imply that sorting N numbers by means of pairing heaps requires asymptotically at most $2N\log_2 N$ comparisons, instead of $2.88N\log_2 N$ for Fibonacci heaps.

To analyze deckey we must be a little careful. We use the property that

$$\Phi.(z, \langle a, t\rangle) \geq \Phi.z + \Phi.\langle a, t\rangle.$$

Then we observe, writing $x = (z, \langle a, t\rangle)$:

$$\begin{array}{ll} & \mathcal{A}[\mathsf{deckey}](a, k, x) \\ = & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{deckey}] \ \} \\ & \mathcal{T}[\mathsf{deckey}](a, k, x) + \Phi.(\mathsf{deckey}.a.k.x) - \Phi.x \\ = & \quad \{ \text{ definitions of } \mathcal{T} \text{ and } \mathsf{deckey} \ \} \\ & 1 + \Phi.(z \bowtie \langle a{-}k, t\rangle) - \Phi.(z, \langle a, t\rangle) \\ \leq & \quad \{ \text{ above property } \} \\ & 1 + \Phi.(z \bowtie \langle a{-}k, t\rangle) - \Phi.z - \Phi.\langle a, t\rangle \\ = & \quad \{ \ (17), \text{ using } \Phi.\langle a, t\rangle = \Phi.\langle a{-}k, t\rangle \ \} \\ & 1 + \log_2 \#x. \end{array}$$

Let us briefly compare pairing heaps with Fibonacci heaps. Between data type P and data type F defined in Section 10.3, we observe three major differences: (a) F consists of trees over Int $\times$ $[0..K]$ whereas trees in P are over Int, (b) trees in P can have any structure whereas trees in F are pseudo-binomial, and (c) the order of the subtrees is taken into account for trees in P, while it is irrelevant for trees in F (compare tree types $\langle T\rangle_4$ and $\langle T\rangle_6$).

Because of these differences, Fibonacci heaps are not as "self-adjusting" [7] as pairing heaps are. Combined with the fact that the implementation at pointer level is simpler and more space efficient, this makes pairing heaps interesting from a practical point of view. From a theoretical point of view, however, Fibonacci heaps are to be preferred because the amortized costs of deckey have been shown to be $O(1)$.

11.4 Why these "sum of logs" potentials?

For path reversal, splaying, and pairing, we have derived the following potential functions:

$$\Phi.\langle a, t\rangle = \tfrac{1}{2}\log_2 \#\langle a, t\rangle + (\Sigma\, x : x \in t : \Phi.x),$$

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Phi.t + \log_2 \sharp\langle t, a, u\rangle + \Phi.u, \end{array}$$

and

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle a, [\,]\rangle & = & 0 \\ \Phi.\langle a, x \vdash t\rangle & = & \Phi.x + \log_2 \#\langle a, x \vdash t\rangle + \Phi.\langle a, t\rangle. \end{array}$$

For these potential functions, the contribution of each node to the potential of the tree is proportional to the logarithm of the size of its subtrees. (In case of pairing, $\langle a, x \vdash t\rangle$ can be viewed as a node with left subtree x and right subtree $\langle a, t\rangle$.) The fact that these "sum of logarithms" potentials serve to analyze both splaying and pairing is easy to explain, since there is a clear connection between these two operations (see [7, p.121]). But the authors of [12] are at a loss to explain that such a potential can also be used to amortize the cost of path reversal, because they cannot discover a connection between splaying (and pairing) on the one hand, and path reversal on the other hand. To us, however, this fact is not a surprise anymore because we are able to analyze these three operations in the same systematic way.

In connection with the amortized complexity of deckey for pairing heaps, an important question is whether a smaller potential than "sum of logs" is possible. Since $\mathcal{A}[\mathsf{deckey}](a, k, x) \leq 1+\delta.x$, we would obtain a better bound if we were able to show that, for instance, $\delta.x = \log\log \sharp x$ is possible. Unfortunately, however, a smaller potential cannot be obtained in our analyses of path reversal, splaying, and pairing, as we will now show for path reversal.

In the bottom-up analysis of path reversal, we defined $\psi.t = \sharp t$, and this choice for ψ led to requirement (4) on α and β. To investigate whether a smaller choice for $\psi.t$ is possible, e.g. $\psi.t = \log \sharp t$, we introduce a function f on the positive integers, and take $\psi.t = \sqrt[\beta]{f.\sharp t}$. As requirement on f we then obtain formula (18) in the lemma below. In our analysis we have chosen $f.n = \sqrt{n}$ (for $\alpha = 2$ and $\beta = \frac{1}{2}$). Lemma 11.6 shows that f cannot be substantially smaller.

Lemma 11.6
Let f be a function defined on the positive integers such that $f.n \geq 1$. Suppose f satisfies

$$(18)\quad (\forall\, m, n :: \frac{f.m}{f.n} \leq \frac{m+n}{\alpha n}),$$

for some $\alpha > 1$. Then $f.n$ is $\Omega(n^\epsilon)$ and $O(n^{1-\epsilon})$, with $0 < \epsilon < \frac{1}{2}$.
Proof For a sufficiently large integer c, we instantiate (18) with $m, n := c\,n, n$ and $m, n := n, c\,n$, respectively. This yields that, for all positive n:

$$\frac{f.(cn)}{f.n} \leq \frac{c+1}{\alpha} \quad \wedge \quad \frac{f.n}{f.(cn)} \leq \frac{1+c}{\alpha c},$$

or, equivalently,

$$\frac{\alpha c}{c+1} \leq \frac{f.(cn)}{f.n} \leq \frac{c+1}{\alpha}.$$

Since $\alpha \leq 2$ (instantiate (18) with $m = n$), the lower bound does not exceed the upper bound (using $4c \leq (c{+}1)^2$). We take $c > \frac{1}{\alpha-1}$ (hence $c > 1$), and define

$$\epsilon = \log_c \tfrac{\alpha c}{c+1}.$$

Then $0 < \epsilon < \frac{1}{2}$, and for all positive n:

$$c^{\epsilon} \leq \frac{f.(cn)}{f.n} \leq c^{1-\epsilon}.$$

From these inequalities we deduce, by induction on k, that

$$c^{k\epsilon} \leq \frac{f.(c^k n)}{f.n} \leq c^{k(1-\epsilon)},$$

for all $k \geq 0$, from which we conclude that $f.n$ is $\Omega(n^{\epsilon})$ and $O(n^{1-\epsilon})$, with $0 < \epsilon < \frac{1}{2}$.
□

Along the same lines it can be shown that a potential which is substantially smaller than a "sum of logs" potential cannot be used in our analyses of splaying and pairing. For pairing the argument is as follows. The counterpart of (18), corresponding to requirement (16) on φ, is

$$(\forall\, k, m, n :: \frac{f.(m+n+k-1)}{f.(m+n+k)} \; \frac{f.(m+n)}{f.(n+k)} \leq \frac{m+n+k}{\alpha^2 k}).$$

Taking $n = 1$, this implies

$$(\forall\, k, m :: \frac{f.(m+k)}{f.(m+k+1)} \; \frac{f.(m+1)}{f.(k+1)} \leq \frac{m+k+1}{\alpha^2 k}).$$

Since $m + k \approx m + k + 1$, this formula is similar to (18); therefore, $f.n$ is again $\Omega(n^{\epsilon})$ and $O(n^{1-\epsilon})$, with $0 < \epsilon < \frac{1}{2}$.

The moral thus is that a better bound for $\mathcal{A}[\mathsf{deckey}]$ can only be obtained by using more complex potential functions—if at all possible. It is, for instance, conceivable that a potential takes the depth of the nodes into account; for example, a suitable potential for a tree x might be $\Phi.0.x$, with

$$\begin{array}{lcl} \Phi.k.\langle\,\rangle & = & 0 \\ \Phi.k.\langle t, a, u\rangle & = & \Phi.(k{+}1).t + \varphi.k.t.u + \Phi.(k{+}1).u \end{array}$$

Whether such potentials exist is an open problem.

Chapter 12

Conclusion

The original incentive that led to the research which culminated in this thesis was to increase our understanding of amortized complexity, in particular our insight in the nature of potential functions. To achieve this we have tried to derive potentials as systematically as possible for a number of data structures. The only paper we know of in which this is also an explicit goal is by Nelson [26], who derives a potential to show that snoopy caching is efficient in the amortized sense. Other work along the lines of this thesis has been recorded in [15, 18]. Furthermore, Sleator has recently shown how potentials can be computed by means of linear programming [28]. The examples in his paper, however, are restricted to operations whose amortized costs are bounded by a constant; data structures like skew heaps and splay trees are not covered.

A key factor to successful analysis of data structures is the description of the operations at the right level of detail. Our experience is that a functional programming style combined with the appropriate choice of list and tree types leads to concise programs which capture the essence of the operations. An important aspect is that these programs are recursive, which leads to recurrence relations for the actual costs as well as the amortized costs. The recurrence relations for the amortized costs are formulated in terms of a potential function, and using mathemathical induction a potential function can so be derived. Also, we have shown how the use of top-down simulations (read "algorithmic refinements") of bottom-up operations like bottom-up melding and path reversal simplifies the analysis of such operations.

In Part I we introduced the necessary tools needed to specify and (re)design several (mostly existing) data structures in Part II in an effective way. We obtained concise descriptions of these data structures and our analyses resulted in bounds that either matched or improved upon existing bounds. There are many directions in which the research reported in this thesis can be extended. Below we mention a few.

1. In Chapter 2 the notion of algebra refinement has been defined for monoalgebras only. Starting from a notion of data refinement of relations (or—perhaps better—monotonic predicate transformers), this definition may

be extended to a more general notion of algebra refinement. In this way, nondeterministic operations that, for instance, arbitrarily pick an element from a given set can be dealt with as well.

2. In Chapter 3 we have shown that any amortized analysis can be done in terms of a potential function. Some well-known data structures, such as solutions to the union-find problem (see, e.g., [32]), are only analyzed by means of the banker's method. It is interesting to look what the potential functions for these data structures look like. In particular, it is interesting to find the potential that corresponds to the banker's analysis of path reversal of Tarjan and van Leeuwen (proof of Lemma 10 in [32]) and to compare it to our analysis in Chapter 11 (where it should be noted that our analysis only gives an $O((n+m)\log n)$ bound instead on the $O(n + m \log n)$ bound in [32] for the cost of m reversals on an arbitrary initial n-node tree).

3. At the end of Chapter 10 we have seen that the fact that the number of *deckey* operations on each node is bounded by a constant K, say, gives rise to more efficient implementations. Moreover, in Section 9.4.4 (Lemma 9.12) we have shown that depending on the ratio of the number of union's and the number of delmin's different bounds for the amortized costs of these operations are obtained. The question is now how to describe additional knowledge about the use of the operations of an algebra in a formal and convenient way.

4. In Chapter 9 we conjectured that the "$\log_\phi$" bound for top-down melding is tight. To confirm this conjecture a worst-case sequence of melding operations must be constructed, which turns out to be rather difficult. A weaker—but nevertheless interesting—result would be whether the bound in Lemma 9.2 is tight for the particular potential function defined in this lemma.

5. Yet another question is under which circumstances amortized efficient data structures can be used in an efficient way in parallel programs. The problem is that a slow operation in one process may suspend several other processes due to delayed communications, which may affect the performance drastically.

Finally, to put our work in perspective, we would like to stress that we consider our quite formal approach complementary to the more intuitively based approaches of algorithm design. The latter led to new and surprising results, whereas the former can be used to polish up known results, but also to achieve new results—as we have shown in this thesis.

References

1. Aho A.V., Hopcroft J.E., Ullman J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley (1974).
2. Back R.-J.R. *Correctness preserving program refinements: proof theory and applications.* Mathematical Centre Tracts 131, Department of Computer Science, Mathematisch Centrum, Amsterdam (1980).
3. Bird R.S. *Lectures on Constructive Functional Programming.* Technical monograph PRG 69, Oxford University Computing Laboratory (1988).
4. Chen W., Udding J.-T. *Towards a calculus of data refinement.* In: J.L.A. v.d. Snepscheut (ed.), *Mathematics of Program Construction*, LNCS 375, Springer-Verlag (1989) 197–218.
5. Dijkstra E.W. *A Discipline of Programming.* Prentice-Hall (1976).
6. Dijkstra E.W., Scholten C.S. *Predicate Calculus and Program Semantics.* Springer-Verlag (1990).
7. Fredman M.L., Sedgewick R., Sleator D.D., Tarjan R.E. *The pairing heap: a new form of self-adjusting heap.* Algorithmica 1 (1986) 111–129.
8. Fredman M.L., Tarjan R.E. *Fibonacci heaps and improved network optimization algorithms.* Journal of the ACM 34 (1987) 596–615.
9. Gabow H.N., Bentley J.L., Tarjan R.E. *Scaling and related techniques for geometry problems.* Proceedings of the 16th Annual ACM Symposium on Theory of Computing (1984) 135–143.
10. Gajewska H., Tarjan R.E. *Deques with heap order.* Information Processing Letters 22 (1986) 197–200.
11. Gardiner P.H.B., Morgan C.C. *Data refinement of predicate transformers.* Theoretical Computer Science 87 (1991) 143–162.
12. Ginat D., Sleator D.D., Tarjan R.E. *A tight amortized bound for path reversal.* Information Processing Letters 31 (1989) 3–5.
13. Gries D. *The Science of Programming.* Springer-Verlag (1981).
14. Hagerup T. *On saving space in parallel computation.* Information Processing Letters 29 (1988) 327–329.
15. Heutinck R. *Priority queues.* Master's thesis (in Dutch), Department of Mathematics and Computing Science, Eindhoven University of Technology (1989).
16. Hoare C.A.R. *Proof of correctness of data representations.* Acta Informatica 1 (1972) 271–281.

17. Hoogerwoord R.R. *The design of functional programs: a calculational approach.* Ph.D. thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology (1989).
18. Hurk P.A.J.M. v.d. *Het ontwerpen van geamortiseerd efficiënte datastructuren.* Master's thesis (in Dutch), Department of Mathematics and Computing Science, Eindhoven University of Technology (1989).
19. Italiano G.F. *Amortized efficiency of a path retrieval data structure.* Theoretical Computer Science 48 (1986) 273–281.
20. Kaldewaij A. *Programming: The Derivation of Algorithms.* Prentice-Hall international series in computer science (1990).
21. Kaldewaij A., Schoenmakers B. *The derivation of a tighter bound for top-down skew heaps.* Information Processing Letters 37 (1991) 265–271.
22. McCreight E.M. *Pagination of B^*-trees with variable-length records.* Communications of the ACM 20 (1977) 670–674.
23. Mehlhorn K. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag (1984).
24. Mehlhorn K., Tsakalidis A. *Data Structures.* In: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Chapter 6, Elsevier Science Publishers B.V. (1990) 301–341.
25. Morris J. *Laws of data refinement.* Acta Informatica 26 (1989) 287–308.
26. Nelson G. *Methodical competitive snoopy-caching.* In: W.H.J. Feijen, A.J.M. v. Gasteren, D. Gries, J. Misra (eds.), *Beauty is our business: a birthday tribute to Edsger W. Dijkstra*, Chapter 38, Springer-Verlag (1990) 339–345.
27. Peyton Jones S.L. *The Implementation of Functional Programming Languages.* Prentice-Hall international series in computer science (1987).
28. Sleator D.D. *Data structures and terminating Petri nets.* In: I. Simon (ed.), *Latin'92*, LNCS 583, Springer-Verlag (1992) 488–497.
29. Sleator D.D., Tarjan R.E. *Self-adjusting binary search trees.* Journal of the ACM 32 (1985) 652–686.
30. Sleator D.D., Tarjan R.E. *Self-adjusting heaps.* SIAM Journal on Computing 15 (1986) 52–69.
31. Tarjan R.E. *Amortized computational complexity.* SIAM Journal on Algebraic and Discrete Methods 6 (1985) 306–318.
32. Tarjan R.E., Leeuwen J. v. *Worst-case analysis of set union algorithms.* Journal of the ACM 31 (1984) 245–281.
33. Turner D.A. *SASL Language Manual.* University of St. Andrews (1976).
34. Vuillemin J. *A data structure for manipulating priority queues.* Communications of the ACM 21 (1978) 309–315.
35. Vuillemin J. *A unifying look at data structures.* Communications of the ACM 23 (1980) 229–239.
36. Wadler P. *Linear types can change the world!* In: M. Broy, C.B. Jones (eds.), *Programming Concepts and Methods*, North-Holland, Amsterdam (1990) 561–581.

Glossary of notation

$\emptyset$	empty set, empty function, empty relation
$\bot$	empty tuple
$\{\bot\}$	unit type
$\mathrm{dom}\, R\,, \mathrm{rng}\, R$	domain, range of relation R
$R.x$	relation R applied to x
yRx	$(x, y) \in R$
$S \circ R$	composition of relations R and S
$R^\star$	dual of relation R
$R{\restriction}X$	restriction of relation R to X
yfx	$x \in \mathrm{dom}\, f \ \ \wedge \ \ y = f.x$ for function f
$f[x{:=}y]$	function f except that $f[x{:=}y].x = y$
$X{\rightharpoonup}Y$	partial functions from X to Y
$X{\to}Y$	(total) functions from X to Y
$?_T$	arbitrary value of type T
Bool	set of booleans, $\{\mathrm{true}, \mathrm{false}\}$
$\mathrm{Nat}, \mathrm{Int}, \mathrm{Real}$	set of natural, integer, real numbers
ϕ	golden ratio $(1 + \sqrt{5})/2$
$[a..b)$	interval $\{x \mid a \leq x < b\}$
$[a..b]$	interval $\{x \mid a \leq x \leq b\}$
$(a..b]$	interval $\{x \mid a < x \leq b\}$
$(a..b)$	interval $\{x \mid a < x < b\}$
$\{T\}, \llparenthesis T \rrparenthesis, [T], \langle T\rangle$	finite structures over T
$\#x$	size of structure x
$a \in x$	value a occurs in structure x
$\downarrow x\,, \uparrow x$	minimum, maximum of structure x
$\{T\}$	finite sets over T
$\{\,\}, \{a\}$	empty set, singleton set a
$\uplus$	disjoint set union
$S \oplus a$	$S \cup \{a\}$ for $a \notin S$
$S \ominus a$	$S \setminus \{a\}$ for $a \in S$
$\Downarrow S$	$S \setminus \{\downarrow S\}$ for nonempty set S
$\llparenthesis T \rrparenthesis$	finite bags (multisets) over T
$\llparenthesis\,\rrparenthesis, \llparenthesis a \rrparenthesis$	empty bag, singleton bag a
$\Downarrow B$	$B \ominus \llparenthesis \downarrow B \rrparenthesis$ for nonempty bag B
$\ominus\,, \oplus$	bag subtraction, bag summation

$[T]$	finite lists over T
$[\,], [a]$	empty list, singleton list a
$\vdash, hd, tl$	cons, head, tail
$\dashv, ft, lt$	snoc, front, last element
$+\!\!+$	(con)catenation
$\#s$	length of list s
$s{\uparrow}n$	prefix of length n of list s
$s{\downarrow}n$	suffix of length $\#s - n$ of list s
$s.n$	$(n{+}1)$-st element of list s
$rev.s$	reverse of list s
$\langle T \rangle$	finite binary trees over T, $(\mu X : \langle\,\rangle \in X : X \times T \times X)$
$\langle\,\rangle$	empty tree
$\langle t, a, u \rangle$	tree with left subtree t, root a, and right subtree u
$\langle a \rangle$	singleton tree a, $\langle\langle\,\rangle, a, \langle\,\rangle\rangle$
$l.t, m.t, r.t$	left subtree, root (middle), and right subtree of tree t
$\sharp t$	size of t plus one, $\sharp\langle\,\rangle = 1$ and $\sharp\langle t, a, u\rangle = \sharp t + \sharp u$
$\overline{t}$	inorder traversal of tree t
$\langle T \rangle_1$	$(\mu X :: [X \times T])$
$\langle T \rangle_2$	$(\mu X :: [X \times T \cup T \times X])$
$\langle T \rangle_3$	$[\langle T \rangle \times T \cup T \times \langle T \rangle] \times \langle T \rangle$
$\langle T \rangle_4$	$(\mu X :: T \times [X])$
$\langle T \rangle_5$	$(\mu X :: T \times (\!| X |\!))$
$\langle T \rangle_6$	$(\mu X :: T \times \{X\})$
$\langle T \rangle_7$	$[\langle T \rangle_6]$
$\langle T \rangle_8$	$[\,[\langle T \rangle_4] \times T \times [\langle T \rangle_4]\,] \times \langle T \rangle_4$
$\simeq$	coupling relation
$[\![\cdot]\!]$	abstraction function
$(\!(\cdot)\!)$	representation function
$\mathrm{rng}\,\mathsf{A}$	range of algebra A
$val(E)$	value of expression E
$\mathcal{T}, \mathcal{N}$	cost measures
$\mathcal{T}(E)$	cost of expression E
$\mathcal{T}[f]$	cost of function f
$\doteq$	unfolding counted by cost measure
$\ddot{=}$	unfolding counted twice by cost measure
$\mathcal{A}[f]$	amortized cost of function f
$\dagger x$	rank of tree x
$\ddagger x$	pseudo-rank of tree x
$\bowtie$	top-down melding/linking
$\between$	bottom-up melding
$a\angle$	path reversal/splaying at a

Index