Data Structures and Amortized Complexity in a Functional Setting

Berry Schoenmakers

Doctoral Dissertation from
Eindhoven University of Technology[1]

September 1992

[1]Current address: CWI, Kruislaan 413, 1098 SJ Amsterdam, Netherlands. E-mail: berry@cwi.nl

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 **1**

I Theoretical Aspects **7**

1 Algebras **8**

1.1 Basic data types and operations 9

1.1.1 Tuples 9

1.1.2 Relations and functions 9

1.1.3 Simple data types 10

1.1.4 Structured data types 11

1.2 Algebras 12

1.3 Monoalgebras 17

2 Algebra refinement **20**

2.1 Functional programs and specifications 20

2.2 Algorithmic refinement of functions 22

2.3 Data refinement of functions 23

2.4 Refinement of monoalgebras 26

2.5 Surjectivity and injectivity of monoalgebras 30

2.6 A closer look at surjectivity and injectivity 36

3 Amortized complexity **39**

3.1 Amortized analysis of imperative programs 40

3.2 Abstract views of amortization 44

4 Implementation aspects **49**

4.1 Functional program notation 50

4.2 Eager evaluation 51

4.3 Pointer implementation of stacks 52

4.4 Destructivity 54

4.5 Queues and concatenable deques 55

4.6 Linear usage of destructive monoalgebras 58
4.7 Benevolent side-effects 61

5 Analysis of functional programs and algebras **63**
5.1 Cost measures 63
5.2 Worst-case analysis 67
5.3 Amortized cost of functions 69
5.4 Amortized analysis 71
5.5 Amortization and linearity 74
5.6 Purely-functional deques 75
5.6.1 Specification and implementation 75
5.6.2 Some applications and their analyses 78
5.6.3 Comparison with doubly-linked list representation 82

6 More on lists and trees **84**
6.1 Lists 85
6.1.1 Stacks with deletion 86
6.1.2 Partitions 87
6.2 Binary trees 87
6.2.1 Top-down views 87
6.2.2 A skewed view 88
6.2.3 Root-path views 90
6.3 Trees and forests 92
6.3.1 Top-down views 92
6.3.2 Root-path views 93
6.4 Overview of tree types 94

II Case Studies **95**

7 Tricky representations of sets and arrays **96**
7.1 A tricky representation of sets 97
7.2 A tricky representation of arrays 99
7.3 Mehlhorn's tricky representation of sets 100
7.4 An unboundedly small potential 101
7.5 Comparisons 103

8 Maintaining the minimum of a list **104**
8.1 Stacks 104
8.2 Deques 106
8.3 Concatenable queues 106
8.4 Noninjective algebras 107

9 Skew heaps **110**
9.1 Priority queues 110
9.2 Top-down skew heaps 112
9.2.1 Introduction of $\bowtie$ 113
9.2.2 Implementation of $\bowtie$ 114
9.2.3 Analysis of $\bowtie$ 116
9.2.4 Refined analysis of $\bowtie$ 120
9.2.5 Results for priority queue operations 122
9.2.6 Comparison with Sleator and Tarjan's results 122
9.3 Intermezzo on sorting 123
9.4 Bottom-up skew heaps 126
9.4.1 Implementation of ⧖ 127
9.4.2 Bottom-up analysis of ⧖ 129
9.4.3 First top-down analysis of ⧖ 131
9.4.4 Second top-down analysis of ⧖ 139
9.4.5 Results for priority queue operations 142
9.5 Pointer implementations 142
9.6 Concluding remarks 144

10 Fibonacci heaps **146**
10.1 Lazy binomial queues 147
10.2 Intermezzo on the precondition of linking 151
10.3 Fibonacci heaps 153
10.4 Concluding remarks 158

11 Path reversal, splaying, and pairing **160**
11.1 Path reversal 160
11.1.1 Program 160
11.1.2 Bottom-up analysis 161
11.1.3 Top-down analysis 165
11.1.4 Series of path reversals 167
11.2 Splaying 167
11.2.1 Splay trees 168
11.2.2 Definition of splaying 170
11.2.3 Analysis of top-down splaying 172
11.2.4 Bounds for splay trees 178
11.3 Pairing 178
11.3.1 Pairing heaps 178
11.3.2 Analysis of pairing 180
11.3.3 Bounds for pairing heaps 182
11.4 Why these "sum of logs" potentials? 183

12 Conclusion **186**

References **188**

Glossary of notation **190**

Index **192**

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

Part II

Case Studies

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
$\perp$	empty tuple
$\{\perp\}$	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
$R{\upharpoonright}X$	restriction of relation R to X
$y\,f\,x$	$x \in \mathrm{dom}\, f \ \ \wedge \ y = f.x$ for function f
$f[x{:=}y]$	function f except that $f[x{:=}y].x = y$
$X \curvearrowright Y$	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, $\{\mathrm{true}, \mathrm{false}\}$
Nat, Int, 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 $\{x \mid a < x < b\}$
$\{T\}, \Lbag T \Rbag, [T], \langle T \rangle$	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
$\Lbag T \Rbag$	finite bags (multisets) over T
$\Lbag\, \Rbag, \Lbag a \Rbag$	empty bag, singleton bag a
$\Downarrow B$	$B \ominus \Lbag \downarrow B \Rbag$ 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
$\langle T\rangle_1$	$(\mu X :: [X \times T])$
$\langle T\rangle_2$	$(\mu X :: [X \times T \cup T \times X])$
$\langle T\rangle_3$	$[\langle T\rangle \times T \cup T \times \langle T\rangle] \times \langle T\rangle$
$\langle T\rangle_4$	$(\mu X :: T \times [X])$
$\langle T\rangle_5$	$(\mu X :: T \times \lbag X \rbag)$
$\langle T\rangle_6$	$(\mu X :: T \times \{X\})$
$\langle T\rangle_7$	$[\langle T\rangle_6]$
$\langle T\rangle_8$	$[\,[\langle T\rangle_4] \times T \times [\langle T\rangle_4]\,] \times \langle T\rangle_4$
$\simeq$	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

abstract, 20, 24
abstraction function, 29–30
algebra, 12
algebra refinement, 26
algorithmic refinement, 22
amortized analysis, 40–44, 47, 71–73
amortized cost of function, 69
arrays, 16, 96

bags, 11
banker's view, 42, 47
benevolent side-effects, 61, 74
bijective algebra, 35
binary representation, 14, 68
binary search trees, 168
binary trees, 12, 87, 94
binomial queues, 147
binomial trees, 150
booleans, 10, 13, 35

calculational style, 1
Cartesian trees, 106
circularly-linked list, 56
composition rules, 64, 69
concrete, 20, 24
conservative analysis, 47
cost measure, 64
cost of function, 64
coupling relation, 24, 26
creation operation, 17

data refinement, 24
data type, 12
deques, 57, 75
  with minimum, 106
destructivity, 16, 54
diagonal bags, 154
dictionary, 167

distribution rules, 65–67
distributive cost measure, 65
doubly-linked list, 57

eager evaluation, 51

Fibonacci heaps, 146, 157
first-order, 17–18, 49
forests, 92
fork, 58, 80, 81
function, 10
functional program, 20, 50, 53

golden ratio, 6, 144, 158, 175

higher-order, 19

infinite lists, 19
injective algebra, 33, 107
inorder traversal, 12
inspection operation, 17
intervals, 10

linear usage, 58, 62, 75
linking, 149, 151, 157, 179
lists, 11

melding
  bottom-up, 127, 129, 144
  top-down, 113, 116, 122, 123, 143
monoalgebra, 17

numerical types, 10, 13

operation, 12

pairing, 179
pairing heaps, 178, 182
parameter patterns, 50, 84, 85, 89
path reversal, 161

physicist's view, 42, 47
pointers, 16, 52–54
potential function, 42, 47, 69
  sum of logs, 144, 165, 175, 182, 183
priority queues, 16, 110, 146, 178
program, 20
program notation, 50–51, 53–54
pseudo-binomial trees, 154
pseudo-rank, 154
purely-functional, 51

queues, 15, 55
  with minimum, 106

range of algebra, 32
rank, 150
relation, 9
representation function, 29–30
root-path views, 91, 93, 153, 156, 161, 170, 179

self-adjusting, 183
sets, 11
sharing, 52, 65
signature, 27
simulation, 132, 165
singly-linked list, 53
skew heaps
  bottom-up, 127, 142
  top-down, 113–114, 122
skewed view, 88, 126
sorting
  bounds, 123, 139, 141, 151, 158, 182
  programs
    Mergesort, 60
    PQsort, 123
    *sort*1, 111
specification, 21
splay trees, 168, 178
splaying, 168, 170–172
stacks, 15, 50
  destructive, 55, 58
  purely-functional, 53
  with deletion, 86
    with minimum, 104
structures, 11
subalgebra, 13
surjective algebra, 32

transformation operation, 17
trees, 92
tuple, 9

unary representation, 13, 24, 31
unfolding, 52
unit type, 10

where-clause, 50
without loss of efficiency, 15, 76, 78
worst-case analysis, 67