Reading Guide to "Essays on Computing and Other Topics"

0 Introduction

Over the years I have written about many subjects, but I have to admit that I have published too little: as soon as I solved a problem I lost my interest and I went on. This collection is intended as a partial remedy, to make my work, at least to some extent, accessible to others.

If anything at all can be learned from this compilation about who I am, it will be that I am more an engineer, a designer, than a pure scientist. But that having been said, what also becomes clear is that my motto is: "Nothing is as Practical as a Good Theory". Sometimes, or even quite often, a designer must not be afraid to develop some appropriate theory in order to solve a practical problem. As a matter of fact, a good designer has a scientific attitude, and a good scientist also is a designer (of theories): what is the difference, actually?

* * *

I have not tried to group the essays thematically, because that might have given rise to rather arbitrary choices, if not unresolvable dilemmas. I have only separated the (few) essays written in Dutch from the ones written in English. Otherwise they are just ordered by their numbers, which more or less is chronological order. As a reading guide, Section 2 contains abstracts of all documents. The text of my Farewell Lecture is available as rh311.

In quite a few of my essays references occur to the work of Edsger W. Dijkstra's, the so-called EWDs. If so desired, these can be retrieved from the website of the University of Texas at Austin: `http://www.cs.utexas.edu/users//EWD/`.

During my career I have cooperated with, and my work has been markedly influenced by, Edsger W. Dijkstra, Wim Feijen, Netty van Gasteren, Martin Rem, Anne Kaldewaij, Jan Friso Groote, and others. It goes without saying that I am grateful for this cooperation and for their support. In particular I thank Jan Friso Groote for encouraging me to compile this selection.

1 On the notations I have used, and why

In the course of time I have adopted, and sometimes discarded, several different notations, particularly for functional-programming concepts. Here is a short summary.

1.0 Function Application

Since the appearance, in the 1970s, of the functional programming language SASL, it has become common practice in the world of functional programming to denote function application by mere juxtaposition: for example, the application of function f to argument x then is written as $f\ x$, instead of the more traditional $f(x)$. On the other hand, Dijkstra, Feijen, and van Gasteren promoted the use of an explicit infix operator for function application, for which they chose the period "."; so, they wrote $f.x$. To avoid confusion, however, with the end-of-sentence periods, and to raise the symbol to the same height as the other binary operators, I preferred to write the symbol somewhat higher, and to pronounce it "dot"; so, I write $f \cdot x$. Having an explicit symbol for such an elementary operation really helps! To quote Netty van Gasteren – unfortunately untranslatable –: "Ik kan geen onzichtbare operatoren meer zien!". Also see rh129.

1.1 List Operations: cons and snoc

At first, I adopted from SASL the symbol "$[\,]$" for the empty list and the colon "$:$" as the constructor for non-empty lists. Thus, $b:s$ is the list with b as its head element and s as its tail list.

After a while, however, I became dissatisfied with "$:$", because the symbol is *symmetric* – its own mirror image –, whereas list construction is an *asymmetric* operation: in $b:s$, for example, b is an element and s is a list, and generally $s:b$ is meaningless (unless b is a list too).

Therefore, in my PhD-thesis I adopted the semicolon to denote "cons", because it is asymmetric and yet it resembles the colon very much. So, instead of $b:s$ I now wrote $b;s$.

At a given moment I developed the need to have a symbol for the "snoc" operator, to extend a list with an element "at the other end", so to speak. In a way, this operator is the dual of "cons" and, therefore, I would like to use symbols for "cons" and "snoc" that were each other's mirror images. The (asymmetric) symbol "$;$" does allow "$;$" as its mirror image, but at that time I did not know how to encode this in LaTeX.

Therefore, from the set of standard LaTeX symbols I chose new symbols to denote "cons" and "snoc", namely "$\triangleright$" and "$\triangleleft$", respectively. These symbols somewhat resemble arrow heads, and I use them in such a way that they, when viewed as arrow heads, point towards the list arguments: $b \triangleright s$ is the list composed from element b and list s, and $s \triangleleft b$ also is composed from list s and element b. Thus, in both cases, "$\triangleright$" and "$\triangleleft$" point towards list s.

1.2 List Operations: map

For any given function f, of type $B \to C$, and for any list s with elements of type B, we can form a new list with elements of type C, obtained by applying function f to all elements of s. In many functional programming languages the (higher-order) function mapping f and s to such a result is called map; it is defined recursively by, for all f, b, and s:

$$map \cdot f \cdot [\,] = [\,] \quad \text{and:} \quad map \cdot f \cdot (b \triangleright s) = f \cdot b \triangleright map \cdot f \cdot s \quad .$$

In calculational derivations, however, long identifiers in prefix notation are awkward: binary operators written in infix notation give rise to much more concise formulae. When one considers, as I do, lists as functions, then a property of map – actually: its specification – is, for all lists s of length n:

$$map \cdot f \cdot s \cdot i = f \cdot (s \cdot i) \quad \text{, for all } i : 0 \leq i < n \quad .$$

That is, if one consider lists as (representations of) functions then map implements function composition (in this representation). Initially I used the same symbol "$\circ$" to denote both function composition and "map". Of course, this is ambiguous, because now we have: if both f and g are just functions then $f \circ g$ just denotes their composition, but if s is a list then $f \circ s$ denotes $map \cdot f \cdot s$, which is a list too. This ambiguity is harmless, provided that it is always clear what is the type of the right-hand operand of $\circ$. This gives rise to the following nice associativity property, for functions f, g and list s:

$$(f \circ g) \circ s = f \circ (g \circ s) \quad \text{, which looks much better than:}$$

$$map \cdot (f \circ g) \cdot s = map \cdot f \cdot (map \cdot g \cdot s) \quad .$$

In later writings I occasionally have used $*$ or @ to denote "map". Eventually, not being satisfied with the ambiguity of $\circ$, I adopted the bullet symbol "$\bullet$" to denote the "map" operator. On the one hand it resembles the symbol "$\cdot$" for function application, on the other hand it resembles the symbol "$\circ$" for function composition, yet it differs from both. So, nowadays I write $f \bullet s$ for $map \cdot f \cdot s$. Notice that, by means of the notational device known as *sectioning*, the partial application $map \cdot f$ in my notation becomes $(f \bullet)$. As a very simple example, the function mapping a list of integers to the list of twice the elements of the argument list can be written as: $((2*)\bullet)$. With this symbol for "map" the above associativity property looks like:

$$(f \circ g) \bullet s \;=\; f \bullet (g \bullet s) \quad .$$

1.3 List Operations: take and drop

Just as I felt the need for an infix operator for "map", I felt the need for infix operators for the "take" and "drop" functions. Initially, I used the symbols "$\uparrow$" and "$\downarrow$" to denote "take" and "drop" respectively.

After a while, however, others – myself included, occasionally – began to use these symbols as (integer) infix operators for "maximum" and "minimum". To avoid all possible confusion between these completely different uses, I then adopted the symbols "$\lceil$" and "$\lfloor$" for "take" and "drop" respectively. See "Programming by Calculation", for their definitions.

1.4 Tuples

In my PhD-thesis I adopted the convention to use lists to denote tuples as well. For example, the triple containing elements a, b, and c at that time I would write as: $[a, b, c]$. This had the additional advantage that I obtained the notation for element selection for free: in my treatment lists also are functions on the naturals, and I use this for element selection. So, we have:

$$\begin{array}{lcl} [a, b, c] \cdot 0 & = & a \\ [a, b, c] \cdot 1 & = & b \\ [a, b, c] \cdot 2 & = & c \end{array}$$

Later, however, I have adopted angular brackets $\langle \cdots \rangle$ to denote tuples; so, nowadays I write $\langle a, b, c \rangle$ to denote the very same triple. The advantage is that, syntactically, tuples are better recognizable. For element selection in tuples, I have retained function application, because it is simple and homogeneous and because, mathematically, nothing is wrong with it:

$$\begin{array}{lcl} \langle a, b, c \rangle \cdot 0 & = & a \\ \langle a, b, c \rangle \cdot 1 & = & b \\ \langle a, b, c \rangle \cdot 2 & = & c \end{array}$$

In traditional mathematical texts tuples do not seem to occur, except in the form of pairs, for which parentheses are used: then (a, b) denotes the (unordered) pair formed from a and b. Also, in functional languages like Haskell parentheses are used; in Haskell the triple containing elements a, b, and c is written as (a, b, c). Strangely enough, Haskell does not provide operators for element selection from tuples, except for pairs. In addition, in Haskell one cannot form a singleton tuple (a), because that would cause an ambiguity.

I have rejected this notation, because it is my point of view that parentheses should *only* be used to clarify the tree structure of expressions, and not in any other meaning. The ambiguity, in

Haskell, of the singleton tuple (a), is a direct consequence of this overloading the parentheses. By the way, this provides an additional reason to reject the traditional notation, $f(x)$, for function application.

2 Abstracts

RH69b An implementation of mutual inclusion

This is a relatively straightforward exercise in proving the correctness of a parallel program with semaphores. This is the first occasion where it began to dawn upon us that absence of total deadlock is a *safety property*, instead of a *liveness property*.

Published in *Information Processing Letters*, 1986.

RH71 Waarom de "energieoogstmachine" van di Wesselli niet werkt: een oefening in Separation of Concerns

In 1985 Studium Generale had an exposition of the work of a Belgian would-be artist named di Wesselli. He proposed a perpetuum mobile called the "Energieoogstmachine" – energy harvesting machine – which was based on harmonica-shaped objects. These objects had the property that, when immersed into water, their volume would increase.

Usually, the impossibility of perpetua mobiles is derived from the laws of thermodynamics but di Wesselli's concept was enterily mechanical. Therefore, I set out to pin-point the flaw in his design by purely mechanical reasoning. As was to be expected, di Wesselli did not know what he was talking about and any meaningful discussion with him turned out to be impossible: he was just a clown. But writing RH71 – in Dutch – was fun!

RH75 On degrees of productivity

Inspired by EWD792, this is my first attempt to come to grips with the notion of *productivity* of recursive definitions of infinite lists. The term "productivity" was coined by Edsger W. Dijkstra too, but in EWD749, not in EWD792.

RH79 A nice little program

Over the years, the problem how to compute $(\Sigma i : 0 \leq i < N : A^i)$ in $\mathcal{O}(\log(N))$ time, using addition and multiplication only, has become a canonical example for me. RH79 contains my first solution but, in spite of its title, this solution is not nice at all: the solution was concocted, not designed systematically. Also see RH80, however, and PbC.

RH80 A sequel to RH79

The solution presented here is one of the, by now, many ways in which this problem can be dealt with in a systematic way. The technique used here – matrix exponentiation – is attractive because it is more generally applicable. Also see PbC.

rh82a A simple theorem and it's application

In 1986, during my one of my first courses on Functional Programming, one of my brighter students invented the theorem recorded in rh82a. It is about the transformation of linearly-recursive definitions of functions on (finite) lists into tail-recursive definitions. Also see PbC.

rh89 Two exercises in functional programming

In 1986 Martin Rem was studying a class of parallel programs called "systolic arrays". For two problems which he had solved by means of such systolic arrays he asked me how I would solve these by means of functional programming. This was my answer.

rh94 On the implementation of virtual storage: a top down approach

In the context of the course on Operating Systems I was teaching, I set out to elaborate the implementation of a virtual storage system in its finest details, mainly for my own understanding and to prove to myself that this was possible by means of well-established programming techniques.

rh95a McCarthy's 91-function: a sequel to EWD845

As the title says: inspired by EWD845. By reducing the amount of case analysis, and by treating the cases separately, I constructed a proof that was less "lengthy and boring" than Dijkstra's.

rh111b A symmetric set of efficient list operations

In the common implementations of functional-programming languages, constructing a new list by adding an element in front of an existing list is an $\mathcal{O}(1)$ operation, but adding an element at the end of a list is an $\mathcal{O}(n)$ operation, where n is the length of the list. In rh111b this asymmetry is removed but the price to be paid is that one has to be content with *amortized complexity* instead of worst-case complexity.

Published in *Journal of Functional Programming*, 1992.

rh113 On the introduction of list parameters

A short technical note on the use of infinite lists to transfer information to functions in an efficient way. In a way, this somewhat resembles the good old Turing Machine.

rh128a On mathematical induction and the invariance theorem

The most common form of Mathematical Induction, with steps of the shape $P(n) \Rightarrow P(n{+}1)$, also is its weakest form. Because of this weakness I always have disliked the emphasis placed on this form, and on Structural Induction in general. Strong Mathematical Induction, with steps of the shape $(\forall i : 0{\leq}i{<}n : P(i)) \Rightarrow P(n)$, is the form to be preferred.

Dissatisfied with Netty van Gasteren's treatment – in her PhD thesis – of the Invariance Theorem, I encountered the need for a generalization: Mathematical Induction on the value of

a function. When reasoning about finite lists, as in functional programming, this boils down to Mathematical Induction on the lengths of the lists, which, indeed, is much more effective that Structural Induction alone.

Published in *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, Springer-Verlag, 1990.

rh129 Dot or juxtaposition?

Since the appearance, in the 1970s, of the functional programming language SASL, it has become common practice in the world of functional programming to denote function application by mere juxtaposition: for example, the application of function f to argument x then is written as $f\ x$.

On the other hand, Dijkstra, Feijen, and van Gasteren promoted the use of an explicit infix operator for function application, for which they chose the period ".". To avoid confusion with the end-of-sentence periods, and to raise the symbol to the same level as other binary operators, I myself have adopted the convention to write the symbol somewhat higher, and to call it "dot"; so, I write $f{\cdot}x$.

In rh129 I have compared the relative merits of the two conventions.

rh130 A sequel to rh129 and EWD1061

Here I give additional evidence for Edsger W. Dijkstra's claim that calculational reasoning with function applications may be more effective than point-free reasoning with function compositions.

rh133 A little exercise in combinator logic (mainly for the record)

A nice exercise demonstrating that from a few, seemingly innocent, axioms far-reaching conclusions can be drawn – for example: that a set is infinite –.

rh138 A surprising derivation of a postfix-code evaluator

When one considers expressions as linear, textual representations of trees, the problem of expression parsing can be viewed as the problem of reconstructing a tree from its linear, textual representation. Because linear lists and trees alike can be dealt with smoothly in functional programming, parser and compiler design become exercises in functional programming. This was my first attempt. Also see rh215 and rh224.

rh139a A calculational derivation of the CASOP algorithm

A reaction to a publication by others, because it was immediately obvious that I could improve their presentation considerably.

Published in *Information Processing Letters*, 1990.

rh140 Logisch rekenen, een voorstudie (bespot Raymond Smullyan)

In 1985 the logician Raymond Smullyan published a book, mainly for the sake of entertainment,

called "To Mock a Mockingbird", in which he presented a collection of so-called Logical Puzzles with their solutions. These solutions, however, are so annoyingly clumsy that I envisaged to write a counter text, called "To Mock a Logician", in which I would show how to solve his puzzles in an elegant and systematic way. As a first step, rh140 – in Dutch – deals with some of the Puzzles in the first chapter of Smullyan's book; it is written in a very tutorial style, because I also planned to use it for educational purposes. The main conclusion is that, with only a mild degree of abstraction, quite a few of his seemingly different puzzles are actually the same, and so are their solutions. Writing "To Mock a Logician", however, never has taken shape.

rh143 A solution to an exercise posed by S.D. Swierstra

In 1990 S.D. Swierstra – at that time a passionate proponent of *Attribute Grammars* as a programming paradigm – posed me a problem which he could only solve by means of attribute grammars. I solved his problem almost immediately by means of regular functional-programming techniques: essentially, programming by means of attribute grammars does not really differ from functional programming.

rh144 De Von Neumann machine

Written mainly as a tutorial for a course on Implementation Techniques, I have included rh144 – in Dutch – because it contains ideas that otherwise tend to be lost. In particular my observations on storage management, culminating in what I dubbed the "50%-rule", are important: any form of storage management has to strike a comprise between two (conflicting!) desires, namely maximize storage utilization and maximize processor utilization. My point is that one cannot have both at the same time; therefore, a compromise is the best achievable. Also see rh230 – in English –, where the same theme recurs in a different context.

rh146 Knaster-Tarski in disguise

Knaster-Tarski's theorem states that, in a partially ordered universe, for any monotonic function f on that universe, the equations $x : x = f \cdot x$ and $x : x \sqsubseteq f \cdot x$ have the same least solution. Definitions of recursive data types can be formulated in different ways that, eventually, happen to have much in common.

rh155 A solution to an examination exercise

Just a nice application of the technique of the *Split Binary Semaphore*, introduced by Edsger W. Dijkstra in EWD703, but attributed by him to to C.A.R. Hoare. This technique allows the systematic design of beautiful solutions for many problems, but tends to be forgotten. Also see rh214 and rh221.

rh160 Just a caveat

A one-and-a-half page warning that there is more to (functional) programming than Category Theory.

rh163 Continuations continued (for the record)

This is about the transformation of the general case of linear recursion – not assuming algebraic properties like associativity – into tail recursion. Nowadays I use a technique called *operator folding* to deal with this problem, but in rh163 I solved the problem without operator folding. Unfortunately, I had forgotten about its existence until now.

rh166 The Deutsch-Schorr-Waite graph marking algorithm (mainly for my own understanding)

As the title says, written mainly to explain this algorithm, in every detail, to myself. The best way to do such a thing is by writing about it.

rh167 e

This is about an old algorithm of Edsger W. Dijkstra's for the computation of e^x: in order to understand it, I had to derive it myself. A form of *reverse engineering* that can be quite effective.

rh168 Two problems in connection with the LRU algorithm

Two nice and not enterily trivial algorithms. Also see rh223.

rh169 A Logarithmic Implementation of Flexible Arrays

In 1983 W. Braun and M. Rem designed an efficient implementation of Dijkstra's *flexible arrays*. In rh169 I derived their solutions in a functional setting, in a calculational way.

Presented at the conference *Mathematics of Program Construction*, Oxford, 1992.

rh170a A minor variation on "On a proof of Kaplansky's Theorem"

A simple exercise in systematic proof construction, inspired by EWD1124. I had never heard of "Kaplansky's Theorem" before and its relevance is still unclear to me, but I found Dijkstra's proof not completely satisfactory, particularly its heuristics. Moreover this is about the first time I paid explicit attention to how to characterize an infinite set (something which mathematicians tend to take for granted). Also see rh133, however.

rh172 Whither Inference Rules?

In Gentzen's Natural Deduction, language, meta-language, and meta-meta-language are carefully distinguished. This is useful if one wishes to develop theories, meta-theories, meta-meta-theories, or what have you, but for all practical – construction of proofs – intents and purposes this is downright clumsy: it gives rise to an abundance of rules, to connect the several layers of abstraction and to define what are essentially the same operators in these layers; thus the essential structures are obfuscated. In rh172 I have exposed this.

rh175 Kosmisch van geen belang

A two-page essay – in Dutch – that emphasizes the cosmic futility of our existence, which, therefore, is reason enough to be humble. I wrote it after the insight that very many people seem not to be aware how huge the universe we live in really is.

Published in *GEWIS Jaarboek 1992-1993*.

rh178 The barber's paradox is no paradox

Mathematical or logical paradoxes almost always are the result of not making assumptions (sufficiently) explicit. If one does make these assumptions explicit the paradox usually disappears: the only conclusion then will be that, apparently, (the conjunction of) the assumptions are false. This one-page note settles the issue for the famous *Barber's Paradox*; there is no paradox: either such a barber simply does not exist or he is lying.

rh181 A Derivation of Huffman's Algorithm

On the systematic construction of a *Huffman encoding tree*.

Presented ad hoc at the conference *Mathematics of Program Construction*, Oxford, 1992.

rh185 Sometimes auxiliary variables are necessary

A simple example illustrating an essential observation, regarding the construction of correctness proofs in the Gries-Owicki style, for parallel programs.

rh190 On the foundations of functional programming: a programmer's point of view

Quite an elaborate study in which I set out to deal with a number of misconceptions and conventions that, at the time, seemed to be taken for granted by everbody in the functional programming community. My purpose was to show that alternatives exist and that, in particular, *Scott-Strachey Domain Theory* is not needed to understand functional programming.

Published as *Computing Science Notes 94-26*, Eindhoven University of Technology, 1994.

rh195a Avoiding real-time requirements: a case study

My position on real-time requirements is, firstly, that they had better be better avoided, and, secondly, that if that is not possible they had better be dealt with as close to the hardware as possible. In this essay, written as a tutorial for educational purposes, I demonstrate that buffering can be effectively used to weaken real-time requirements, and how to analyze such a situation.

rh196 Cantor's Diagonalisation Principle

Sometimes a mathematical or technical principle is so simple that it hardly deserves a name of its own. Yet, naming such a principle nevertheless may increase its effectiveness. Leibniz's *Rule of Equals for Equals* and the technique of *Generalization by Abstraction* are good examples of

this phenomenon.

In rh196 I have isolated the essence of what is known as *Cantor diagonalization*. Essentially, this is so simple that, indeed, it hardly deserves a name, but diagonalization constitutes a crucial step in many a mathematical proof. *Cantor's theorem* about cardinalities of power sets and the *Halting Problem* are well-known examples of this.

rh197 Concurrent processing and procedure invocations

The execution of a procedure (or: subroutine) can be viewed as a computation in isolation, independent of the computation invoking that procedure. Thus, execution of a procedure can be viewed as a restricted form of concurrent processing, in which the invoking computation is suspended during the execution of the procedure. And, indeed, both the implementation of concurrent processing and the implementation of (possibly recursive) procedures happen to have much in common.

rh199 The function "reverse"

In the 1990s Roland Backhouse and his team were playing a game they called "Constructive Type Theory". Unless I am mistaken, they seemed to believe that type information alone would be sufficient to derive (functional) programs in a calculational way. Quod non, and rh199 is one of the simplest examples demonstrating this.

rh200a Separating my concerns: a sequel to rh199

As the title says: a sequel, in which some of the more abstract patterns underlying the story in rh199 are exposed.

rh204 Yet another solution

In EWD1170 Edsger W. Dijkstra posed and solved the following problem: Does there exist an equilateral triangle whose vertices have integer (orthogonal Cartesian) coordinates? Before reading his solution I solved the problem in my own way, and my solution happened to be different from Dijkstra's.

rh205 Why $\sqrt{p}$ is irrational, for every prime p

Without exception, the usual proofs of the irrationality of $\sqrt{p}$ suffer from two drawbacks: Firstly, they are formulated as proofs by contradiction, which is quite unnecessary, and, secondly, they contain a step like: "without loss of generality we assume x and y to have no common divisors", which is equally unnecessary. Inspired by rh204. More elaborated versions of this proof appear in rh249 and rh258.

rh206 Dummy transformations and the like

Just dabbling a little with the point-free relational calculus.

rh207 Four sorting algorithms for the price of one

A unified design of four well-known sorting algorithms, by treating them as instances of one abstract algorithm: Insertion Sort, Selection Sort, Merge Sort, and Quick Sort.

rh208 Folding a binary operator: a neglected technique

In many texts on functional programming functions *foldr* and *foldl* are introduced, usually by means of some form of operational argument. In this note I introduce them by an appliciation of *Generalization by Abstraction*, which turns out to work smoothly.

rh209 How to evaluate integer expressions (part i: the simple case)

Technical notes rh209, rh210, and rh212 form a single continuing story. This story is about expression evaluation and translation, and about the representation as machine code programs for a Von Neumann type computer, in a functional setting.

rh210 The Von Neumann machine as a functional program

See rh209.

rh212 How to evaluate integer expressions (part ii: still the simple case)

See rh209.

rh213 From pointers to objects: how to avoid confusion

Many explanations of object-oriented programming confused me because, to my taste, their definitions of what constitutes an *object* are wrong. This gives rise to (for me) incomprehensible texts about objects and object references, and often it remains unclear what is what. This phenomenon is caused by the fact that in the underlying operational model essential variables are left anonymous. This essay is written mainly to analyse the issue for myself and to settle it once and for all. The conclusion is simple: the references are the objects, not the records, holding the object's attributes, referred to by these references. As a result, there can be no such thing as a "static object", just like there is no such thing as a "static integer".

rh214 Eventvariables: an implementation with semaphores

Yet another nice application of the technique of the Split Binary Semaphore, written just for the fun of it. Also see rh155 and rh221.

rh215d Functional-Program Inversion, with an application to Parser Construction

If one considers expressions as linear, textual representations of trees, then the parsing problem – that is: reconstructing a tree from its linear representation – can be treated as an exercise in program inversion: a parser then is the inverse of a tree traversal. Simple as the technique itself

may seem, the mathematical treatment in detail turned out to be quite tedious. The applications are quite nice, though. For an explanation of the notations used: see PbC.

rh218 Two Exercises with Real-Time Programs

Having moved to Dieter Hammer's group, in which real-time programming was studied, this is about my first attempt to come to grips with the problem of specifying and designing programs with real-time requirements.

rh221 Two Applications of the Split Binary Semaphore

Written mainly for educational purposes, but it cannot be denied: I like this technique, for its elegance and effectiveness. Also see rh155 and rh214.

rh223 A truly efficient implementation of LRU stacks

Again, this note was written because I was utterly disappointed by a publication by others. Unfortunately, I forgot to submit it for publication. Also see rh168.

rh224 Expression Evaluation Revisited

As a contribution to the liber amicorum for Frans Kruseman Aretz I decided to brush up an existing story that I thought would match his interests. Also see rh138, rh209, and PbC.

Published in *Simplex Sigillum Veri*, Technische Universiteit Eindhoven, 1995.

rh227 De Suiker en de Weegschaal

In Dutch: A surprising – at least to me – solution to a problem in physics from the 1995 "Nationale Wetenschapsquiz".

rh228 Causal Delivery with Vector Clocks

In a sequential process the *events* – state changes – are totally ordered. In a distributed system with asynchronous message transmission as the only means of interaction between processes, the sending of a message always preceeds the reception of that message. This is called *causality*, and together with the sequential ordering of events this induces a partial ordering relation of the events in the system that is called *causal precedence*. The problem of *causal delivery* is how to guarantee that messages arriving at a single process are delivered to that process in an order that respects causal precedence. This can be considered as a generalization of First-In-First-Out delivery. This essay is my attempt to come to grips with this problem, again, out of dissatisfaction with a publication by others. By the way, the word "clock" is misleading: the subject is the relative ordering of events, in which time plays no role whatsoever. Also see rh235.

rh230 An Introduction to Garbage Collection

A tutorial on Garbage Collection, mainly written for my students, and for myself. In contrast

to other texts on the subject, I have paid attention to performance considerations, by analysing the relation between processor utilization and storage utilization. Also see rh144.

rh234 Implementeren is Programmeren

This essay – in Dutch – has been written for educational purposes. It contains some of the material from rh144. The latter is much more elaborate, but in rh234 I have formulated explicit proof rules for "jump"-instructions, which makes it easier to establish the correctness of machine-code programs formally. Also the transition from (binary) machine code to assembly language has been elaborated, and a section on the importance of allocation independence has been added.

rh235c Leslie Lamport's Logical Clocks: a tutorial

As the title says, this report is mainly written for tutorial reasons. In addition to a discussion of the logical clocks, it also contains a presentation of a distributed algorithm for Mutual Exclusion, which, to my own surprise, can be fully understood without knowledge of or reference to the logical clocks. Also see rh228.

Published as *Computing Science Reports 02-01*, Eindhoven University of Technology, 2002.

rh244 Let's not make things worse

Here I criticize the prevailing folklore in the functional programming community that it would not be necessary to formulate *specifications* of functional programs. In this folklore one would just write down a trivially correct, and possibly inefficient, functional program for the problem at hand, which is then considered an executable specification. In rh244 I explain why this is not a good idea, and I illustrate that by means of an example.

rh246 Distributed Summation

The problem of *Distributed Summation* I consider prototypical for truly distributed computations. This is my first, very informal, presentation of the algorithm for distributed summation. From later experience, also with teaching, I have learned that more formal presentations of this algorithm actually are simpler and more elegant than this informal rendering. Also see rh254.

rh249 Laten we het niet erger maken

This essay – in Dutch – has been written on the occasion of Lex Bijlsma's leaving our university. It contains a selection of techniques and observations.

rh253a Mathematical Induction, Well-Foundedness, and the Axiom of Choice

There exist three different characterizations that, at first sight, do not seem to be related but that actually are equivalent. The proof of one of these equivalences requires an appeal to the *Axiom of Choice*, which seems to be unavoidable. I recall having seen other "proofs" of the same equivalences, but in such an informal way that the appeal to the Axiom of Choice is swept under the rug.

rh254 A Formal Development of Distributed Summation

This is a completely formal design of the algorithm for *Distributed Summation*, in the Owicki-Gries style for proving properties of parallel program. Even the progress properties could be proven easily in a formal way. This report demonstrates convincingly that designing distributed algorithms does not really differ from designing parallel algorithms in general. Also see rh246.

Published as *Computing Science Reports 00-09*, Eindhoven University of Technology, 2000.

rh258 Formality Works

On the occasion of Edsger W. Dijkstra's retirement (and his 70th birthday) I was invited to submit a contribution to a special issue of Information Processing Letters. In it I have presented several examples of formal, calculational reasoning.

Published in *Information Processing Letters*, 2001.

rh274 The Longest Upsequence: a derivation

The problem of the *The Longest Upsequence* is a relatively old programming problem. Triggered by a question by Wim Fijen, I constructed a completely calculational derivation of a functional program for this problem.

rh275 Range/Term Trading: a generalization

In formulae with universal or existential quantification, the range of the formula and its term both are predicates. Therefore, it is possible to formulate algebraic rules for moving parts of these predicates from the range to the term and back. By introducing an additional operator these manipulative possibilities can be extended to quantified formulae with other quantors. Actually, I have felt the need for this additional operator at earlier occasions, but it was the writing of rh274 that inspired me write rh275 as well.

rh277 Time to wake up

A nicely calculational solution to a problem about students falling asleep during a lecture.

rh278 The Thue-Morse sequence: a nice exercise

Somewhere I encountered a mention of the, so-called, *Thue-Morse sequence*, and the problem how to derive a functional program to generate it as an *infinite list* – also called *stream* –. This is a nice exercise, but also it is nothing more than that: just an exercise, and as such I have posed it often to my students (who had no difficulties with it whatsoever). Also see PbC, for the theory of infinite lists and productivity.

rh280b A Formal Derivation of a Sliding Window Protocol

Again – see, for instance, rh228, rh235, and rh254 – this is a demonstration that non-trivial

distributed algorithms can be designed systematically and with such a degree of formality that, on the one hand, correctness of the design is established and, on the other hand, clarity of its presentation is retained. The Sliding Window Protocol definitely is a non-trivial algorithm, because of the many different concerns that play a role in its design.

Published as *Computing Science Reports 06-31*, Eindhoven University of Technology, 2006.

rh281b Computing celebrities

The problem of the *Celebrity Clique* is a nice generalization of the well-known celebrity problem. I wrote my solution out of dissatisfaction with the incredibly clumsy presentation, of what seems – the paper does not even contain a summary of the final algorithm! – to be the same solution, by Richard Bird and Sharon Curtis in *Journal of Functional Programming*. It illustrates the saying: *If your only tool is a hammer every problem looks like a nail.*

I have sent rh281 to Richard Bird, who also is an editor of this journal. He turned out to be a bad loser: he refused to publish it!

rh284 A Unique Representation of Sets

By the end of 2007 Jan Friso Groote posed me the problem how to represent (finite) sets of integers, and how to implement elementary operations on such sets, under the additional requirement that every set is represented *uniquely*. I did so by means of standard functional programming techniques. After having chosen the representation – some form of balanced binary trees – the implementation of the operations turned out to be quite straightforward. The price for uniqueness is high, though: the operations to insert or delete an element turn out to have linear time complexity – instead of logarithmic, as is the case with AVL-trees –, and we have reasons to believe that this is the best attainable.

rh294 ∃*-elimination revisited

In the chapters on reasoning in a Natural Deduction style, the textbook used in our introductory course on logic contains a rule called *∃*-elimination*. In this rule a bound variable is used, the scope of which is not clearly delineated. This is my attempt to repair this. The author of the textbook admitted that my solution is technically better and that, actually, it is well-known and that he has used it in his teaching for many years. His experience, however, was that students found it difficult to grasp and, as a result, made many errors with it. My experience with teaching the current version is that students still make many errors with it. So, although students also make many errors with bound variables, the ill-defined scope in the current version of the rule does not seem to be the source of the problems with ∃*-elimination. Writing rh294 was a nice exercise, though, because it also made explicit that ∃*-elimination and ∀-introduction are faces of the same coin: they are each others duals.

rh295 A Useful Classification

This note embroiders on a theme already present in Netty van Gasteren's PhD-thesis (1988). Although, to my taste, this helps to understand and remember the properties of and differences between surjective, injective, and bijective functions, students still tend to find this very difficult.

That is to say, during examinations many of them have shown not to master this subject.

rh309 Representation Conversion Revisited

In Chapter 8 of PbC I have derived solutions for the problem how to convert binary representations of (natural) numbers into equivalent ternary representations. The full elaboration of these solutions into sequential programs, however, has never been completed. This note fills that gap.

rh310 Generalization by Abstraction, once more

Until recently, an efficient solution, in functional programming, for the well-known problem of the *Maximal Segment Sum* could only be obtained by means of the technique of tupling. Generalization by Abstraction, however, also does the job, and even better. See PbC for a more general discussion of this technique, and for the problem of the Maximal Segment Sum.

rh311 29 Years of (Functional) Programming: what have we learned?

The text of my farewell lecture, to be delivered on 29 August 2013. It provides an overview of the more important lessons we have learned, with an illustration of a few programming techniques. The most important of these is the principle of *Generalization by Abstraction*. In addition some misconceptions encountered over the years will be exposed and refuted.

PhD The design of functional programs: a calculational approach

My PhD-thesis (1989), my first comprehensive treatment of the subject. Many themes from it recur in later essays, but not all of them.

PbC Programming by Calculation

My great *Unvollendete*. Already right after having obtained my doctor's degree, by the end of 1989, the idea rose that it would be worthwhile to write a text book on calculational functional programming and its applications. It never was completed. The current text I have used for many years during a master's course with the very same title. It is not bad, but it is unbalanced, it contains way too few exercises, and more recent insights have not been incorporated. It is included here as is.

Eindhoven, 26 August 2013

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven