Making Formality Work For Us

Roland Backhouse
Department of Computing Science,
Rijksuniversiteit Groningen,
Postbox 800,
9700 AV Groningen,
The Netherlands

May 12, 1989

Abstract

Formal reasoning is notoriously long and arduous; in order to use it to reason effectively in the construction of programs it is, therefore, paramount that we design our notations to be both clear and economical. Taking examples from AI, from imperative programming, from the use of the Bird-Meertens formalism and from category theory we demonstrate how the right choice of what to denote and how it is denoted can make significant improvements to formal calculations. Brief mention is also made of the connection between economical notation and properties of type.

Foreword

Earlier this year I was an invited speaker at the 5th British Computer Society Theoretical Computer Science Colloquium held at Royal Holloway and Bedford New College, London. Before you is the text of my lecture, almost but not quite as given at the conference. (Perhaps the best way to describe the present paper is as the lecture that I *should* have given.)

The publication of the text of the lecture will, I hope, be of scientific value to members of the audience but that is not my main reason for wanting to see it published in its present form. Rather, it is that I wanted to provide an example of the organisation of a lecture that younger researchers might criticise and, possibly, learn from.

New lecturers have a right to seek advice from their supervisors about lecture presentation but I suspect that that advice is rarely offered. Understandably so because, as so aptly stated by Donald A. Bligh [1], "anyone who has the temerity to offer it lays himself open to be shot at whenever he goes into the lecture room, ... since it is often assumed that he will be the embodiment of perfection". I certainly do not regard my own lecturing technique as "the embodiment of perfection" and, indeed, am just as sensitive to criticism as any other. Nevertheless, I feel very strongly about the importance of improving the standard of communication — whether it be oral or written — within the computing science community, so much so that I am willing to expose myself to being "shot at" — so long as the net effect is a positive one.

The written word is a different medium than the overhead projector and so it is not possible for this text to reproduce faithfully the presentation of the lecture. Roughly, however, each figure here corresponds to a slide in the lecture. In addition some (groups of) equations that appear in-line in the text occupied a slide. (Note that, apart from the sequence of titles in the opening paragraph, none of the textual material was reproduced on a slide; I have a strong abhorrence of lecturers who read aloud from their slides — after all, the audience can read text on a slide much faster than it can be read out, and, in any case, such a practice is a sure sign of an ailing memory.)

The rule that each figure formed a slide is not valid in several cases where, within the text, I make a comparison between two calculations (for example, figs. 8 and 9). During the lecture I was able to point to the most significant parts of

[1] *What's the use of lectures? 3rd edition,* Penguin Books, Harmondsworth, England (1972).

a calculation as well as occasionally using overlays for greater emphasis. I also decided at the last minute to curtail some of the discussion by omitting one or two slides. Nevertheless, I had a somewhat large number of slides for a one hour lecture and had to deliver it at a brisk pace.

Several colleagues have made comments on the text which I have gratefully incorporated, either within the text itself or as footnotes. In addition, during the course of the lecture J.R. Hindley made a very valuable comment on my discussion of Warshall's algorithm that I have made use of. My thanks to all of these people.

This year — in fact this very conference — has seen the first issue of a new journal — Formal Aspects of Computing — of which I am proud to be one of its editors. It is therefore a fitting occasion, first, for me to congratulate those who have brought the journal into being — in particular D. John Cooke — and, second, to examine the implications and importance of formal reasoning for computing science.

As is the tradition for invited lecturers I have pondered long and deeply over the title of my talk (perhaps more so than over its content?). There are many permutations of the specific words that I have chosen that, some would argue, offer a more defensible topic of discussion. For instance I could have entitled my talk

Formality Makes Work For Us

or

Making Formality Make Work For Us.

I could have entitled it

Making Formality Work For The Computer

Formality Makes Work For The Computer

Making Formality Make Work For The Computer

or, indeed,

Making Formality Make Work For Us And The Computer

Each of these titles would, I am sure, find more sympathy among many computing scientists and, if followed, have more prospect of success than the one that I have chosen. But no! What I wish to discuss is how we can make formal reasoning *help* (not hinder) *us* (not the computer) in the construction of high-quality computer programs.

So that there is no danger of misunderstanding let me say what I mean by "formal reasoning". I mean, simply, uninterpreted calculation, the sort of mechanical manipulation of symbols that we undertake when, say, performing long division. (See figure 1.) In the process of doing such a calculation we do not construct mental pictures of which the operations are representations (in the example given we do not picture the rearrangement of 316 789 apples

$$
\begin{array}{rcllllll}
 & & & 1 & 3 & 7 & 7 & 3 \\
\cline{2-8}
23 &) & 3 & 1 & 6 & 7 & 8 & 9 \\
 & & 2 & 3 & & & & \\
\cline{3-4}
 & & & 8 & 6 & & & \\
 & & & 6 & 9 & & & \\
\cline{4-5}
 & & & 1 & 7 & 7 & & \\
 & & & 1 & 6 & 1 & & \\
\cline{4-6}
 & & & & 1 & 6 & 8 & \\
 & & & & 1 & 6 & 1 & \\
\cline{5-8}
 & & & & & & 7 & 9 \\
 & & & & & & 6 & 9 \\
\cline{7-8}
 & & & & & & 1 & 0
\end{array}
$$

Figure 1: Long division.

into blocks of 23). Our calculations are completely *syntactic* and divorced from *semantics.*[2]

The complex nature of the programming task is such that any programming calculus will be much more like the integral calculus than the differential calculus. At the outset of a calculation it will be unclear what rules or what properties we will need to invoke. But it should always be clear what the goal of a calculation is . Here is an example to illustrate the point. (See figure 2.)

Given a property, F, of predicates the *circumscription* of F, denoted here by $F\odot$, is also a property of predicates, introduced by McCarthy [14] in his work on logic databases.

The circumscription of F involves a second-order quantification, i.e. a quantification over predicates. The question posed by Lifschitz [12] was,

[2]N.G. de Bruijn makes the following illuminating remarks [6] "Here 'formal' means: without 'understanding' the 'meaning', and therefore it has to be possible to instruct computers how to check the correctness. Indeed, the fact that we do have computers will be one of the reasons why our generation has better chances than those who tried to have similar claims in the past, like Leibniz, Peano and Hilbert. Even if we do not actually *use* computers, they are there to set the standard of what is 'formal' verification."

For function F mapping predicates to *Bool* the *circumscription* of F, denoted by $F\odot$, is the function defined as follows:

$$\begin{array}{rrcl} & F \odot p & \equiv & F.p \wedge \forall(q : F.q : \neg(q < p)) \\ \text{where} & q \leq p & \equiv & \forall(x : q.x : p.x) \\ \text{and} & q < p & \equiv & (q \leq p) \wedge \neg(q \equiv p) \end{array}$$

Theorem (Lifschitz [12])
Let $F.p$ be $U \leq p$. Then

$$F \odot p \equiv (U \equiv p)$$

Figure 2: Circumscription.

"when can such a quantification be reduced to a first-order formula?" Part of Lifschitz's analysis consisted of considering properties of the form $\lambda([p]U \leq p)$, for some predicate U, as shown in the figure.

Now, I know of only two rules that remove a quantification — the one-point rule (called "singleton range" in [2])

$$\oplus(i :\ i = e :\ f.i) = f.e$$

and the empty range rule

$$\oplus(i :\ \textbf{false} :\ f.i) = 1_{\oplus}$$

(where $1_{\oplus}$ is the identity of $\oplus$). An investigation of when $F \odot p$ can be reduced to a first-order formula thus amounts to an investigation of when the latter two rules are applicable. Indeed, for the particular case that F is $\lambda([p]U \leq p)$, a straightforward calculation shown in fig. 3 suffices to reduce $F \odot p$ to such a form.

The point to note about this calculation is that it begins with a clear goal: given a formula involving a universal quantification reduce it to an equivalent formula not involving any quantification. We cannot know in advance the final answer (that, after all, is the purpose of the calculation); nor indeed are we guaranteed success. Nevertheless, having begun our calculation we proceed by pure syntactic considerations — in this case seeking to invoke the one-point rule or the empty range rule, or both — until the goal is reached or we decide that the goal must be modified or postponed.

$$
\begin{array}{ll}
 & F \odot p \\
= & \qquad \{\text{defn.}\} \\
 & (U \leq p) \wedge \forall(q : U \leq q : \neg(q < p)) \\
= & \qquad \{\text{range splitting}\} \\
 & (U \leq p) \wedge \forall(q : U = q : \neg(q < p)) \wedge \forall(q : U < q : \neg(q < p)) \\
= & \qquad \{\text{one pt. rule}\} \\
 & (U \leq p) \wedge \neg(U < p) \wedge \forall(q : U < q : \neg(q < p)) \\
= & \qquad \{\text{prop. calc.}\} \\
 & (U \equiv p) \wedge \forall(q : U < q : \neg(q < p)) \\
= & \qquad \{\text{context rule}\} \\
 & (U \equiv p) \wedge \forall(q : U < q : \neg(q < U)) \\
= & \qquad \{\text{trading}\} \\
 & (U \equiv p) \wedge \forall(q : (U < q) \wedge (q < U) : \ \mathbf{false}) \\
= & \qquad \{\text{empty range}\} \\
 & (U \equiv p) \wedge \ \mathbf{true} \\
= & \qquad \{\text{prop. calc.}\} \\
 & U \equiv p
\end{array}
$$

Figure 3: Proof of Lifschitz's theorem.

So much for the nature of formal reasoning. Before discussing how we might improve our formal-reasoning skills I want to say a few words about the challenge with which we are faced. One of the earliest exercises I know of in purely formal reasoning was Jutting's encoding of Landau's "Grundlagen der Mathematik" in the language of Automath [11]. Essentially, Jutting used a computer program to verify the entire contents of Landau's book. According to de Bruijn [6] the process of going from a mathematical text to a formal input incurred at least a *ten-fold* increase in the size of the text.[3] For those

[3] "Jutting (1977) gives some details about experiences with the checking of a relatively large text (viz. the translation of Landau's "Grundlagen", Landau (1930)). The coded version (Jutting (1976)) consists of about 5.10^7 bits. This may seem very large (may be 10 to 50 times as large as a direct encoding of the words and symbols Landau wrote himself), but it is still of the order of what a single cassette tape can contain".

Note that de Bruijn's remarks quoted here and elsewhere refer to "verifying" or "check-

$$
\begin{array}{ll}
|[& A, B, C \in \text{ Type} \\
; & \oplus \in A \times A \to A \qquad \{\oplus \text{ is associative}\} \\
; & \otimes \in A \times B \to A \\
 & \qquad \{\forall(x, y, z :: (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z))\} \\
; & e \in A \\
\triangleright & \\
 & \oplus/ \circ (\![e, \otimes]\!) \not\to\!\!+ \circ \mathit{segs} = \mathit{fst} \circ (\![< e, e >, \circledast]\!) \not\to \\
]| &
\end{array}
$$

$$
\begin{array}{lccc}
\text{where} & \circledast & = & ... \\
 & \mathit{segs} & = & ... \\
 & \mathit{tails} & = & ... \\
 & \mathit{inits} & = & ... \\
 & \eta & = & ... \\
 & \theta & = & ... \\
 & +\!\!+ & = & ... \\
 & + & = & ... \\
 & * & = & ... \\
 & \not\to & = & ...
\end{array}
$$

Figure 4: Aggregated Segment Sum Theorem.

of you who have never seen Landau's text I should also emphasise that, by the standards of mathematicians, it is already very formal. Jutting did not choose any old mathematics textbook as the basis for his work; he consciously chose one where the step to the language of Automath was minimal.

I could provide several other examples where the difference between informal and formal involves a scaling factor of around ten or more. Recently, I myself completed a formal proof about a certain form of computation over list structures. (See figures 4 and 5.) The contents of the theorem I proved are not relevant here but some statistics about its proof are. Essentially, the theorem states that, under certain conditions, two functions defined over lists are extensionally equal; the first entails an $O(n^3)$ computation, the second an

ing" mathematics; our goal is somewhat more ambitious (and more difficult) viz. to use formal methods to *construct* computer programs.

$$\underbrace{\underbrace{\oplus/ \circ (\![e,\otimes]\!)\!\not\to\!\!+ \circ \mathit{segs}}_{\substack{O(n^3)\text{ executable specification}\\ 2\text{ type constructors}\\ 9\text{ auxiliary functions}}} \quad = \quad \underbrace{\mathit{fst} \circ (\![< e, e >, *]\!)\!\not\to}_{\substack{O(n)\\ 1\text{ type constructor}\\ 3\text{ auxiliary functions}}}}_{\substack{32\text{ lemmas}\\ 147\text{ proof steps}}}$$

Figure 5: Statistics on a formal construction.

$O(n)$ computation. In order to program the first function (the less efficient one) it is necessary to define two type constructors and 9 auxiliary functions. In order to program the second function (the more efficient one) one type definition is needed together with the definition of three auxiliary functions. The texts of both programs, written in SML, are included in the appendix. (It is worth noting in passing that the longer program is less efficient and corresponds much more closely to the original problem specification. Executable specifications may be substantially longer than efficient implementations!) In my proof of their equivalence, which was completely formal although done by hand, I counted 32 lemmas and 147 proof steps. All this to construct a program occupying less than one page of code!

Such statistics, in the wrong hands, could be used to spread alarm and discontent among the formal methods community. On first encounter they seem to provide ample justification for the argument that formality simply makes extra work for us. But I do not regard the figures as daunting: rather, they reflect the challenge that we face, one which, as we all know, is no newcomer to computing science.

The way to improve our calculational method — the only way — is to improve on the notations that we use. Having attempted some years ago, however, to discuss choice of notation I know full well that it is a subject fraught by inculcated prejudice. For many, “familiarity” (and familiarity alone) is the only ground for choosing a notation. I know also that formal calculation is not a spectator sport: it is very boring to watch someone else perform a series of syntactic manipulations. Thus, in two ways my task is difficult: the calculations I am about to show you are just that — formal calculations — and they use notations that will be unfamiliar to all but a small minority of you.

Several of my examples are taken from the calculus developed by Richard Bird and Lambert Meertens, and sometimes called the "Bird-Meertens formalism" and sometimes called "Squiggol" [5, 15]. One way to summarise their work is that it is a calculus of the reduce operator (denoted in the sequel by $/$) and the map operator (denoted in the sequel by $*$). Up till now they have tended to remain within the realm of finite lists, but it is becoming increasingly obvious how to extend their work to an arbitrary type constructor and its corresponding reduce operator (see e.g. Malcolm [13]).

From a calculational perspective their work represents for me the most major advance that I have encountered in my career as a computing scientist. From the traditional, foundational perspective, however, their work offers no surprises; the basic theorems and methods that they use are well-known and have been so, in most cases, for several decades. The advance that they have made is to show how concise notation designed around fundamental algebraic properties can substantially increase the effectiveness of the calculational method.

The two properties that are the mainstay of their approach are, first, that the type $A \times B \to C$ is isomorphic to $A \to (B \to C)$ and, second, that function composition is associative. The first of these permits one to "curry" a binary operator — the well-known process that defines, for example, $(2+)$ to be the function that given x computes $2 + x$.

These two properties are used silently in their calculations emphasising a principle that I shall reiterate several times, namely: one should always endeavour to *design one's notation so that the use of the most important properties becomes (almost) invisible.* In particular, they use the infix operators "/", standing for reduce, and "$*$", standing for map, primarily in their curried forms. Thus, for join lists reduce is a polymorphic function of type

$$[\![(\alpha \times \alpha \to \alpha) \times \alpha* \to \alpha]\!]$$

and if $\oplus \in A \times A \to A$ then

$$\oplus/ \in A* \to A$$

$$
\begin{array}{ll}
 & \oplus/ \circ f* \circ +\!\!+/ \\
= & \qquad \{* \text{ promotion}\} \\
 & \oplus/ \circ +\!\!+/ \circ f** \\
= & \qquad \{/ \text{ promotion}\} \\
 & \oplus/ \circ \oplus/* \circ f**
\end{array}
$$

Hence,

$$
\begin{array}{ll}
 & \oplus/ \circ \oplus/* \circ +\!\!+/ \circ +\!\!+/* \\
= & \qquad \{\text{above}, f := \oplus/\} \\
 & \oplus/ \circ \oplus/* \circ \oplus/** \circ +\!\!+/* \\
= & \qquad \{* \text{ distributivity}\} \\
 & \oplus/ \circ (\oplus/ \circ \oplus/* \circ +\!\!+/)* \\
= & \qquad \{\text{above}, g := \oplus/\} \\
 & \oplus/ \circ (\oplus/ \circ \oplus/* \circ \oplus/**)* \\
= & \qquad \{* \text{ distributivity}\} \\
 & \oplus/ \circ \oplus/* \circ \oplus/** \circ \oplus/***
\end{array}
$$

Figure 6: Squiggoling.

Similarly, again for join lists, map is a polymorphic function of type

$$[\![(\alpha \to \beta) \times \alpha* \to \beta*]\!]$$

and if $f \in A \to B$ then

$$f* \in A* \to B*$$

(Note also the use of the symbol "$*$" both as a type constructor and as an operator on functions. This, too, is an important economy of notation introduced by category theory indicating that "$*$" is a functor.)

Figure 6 is a somewhat concocted example of a calculation in Squiggol. The objective of the calculation is to reduce the expression

$$\oplus/ \circ \oplus/* \circ +\!\!+/ \circ +\!\!+/*$$

to an equal expression not involving "$+\!\!+$". (Here "$+\!\!+$" denotes the join constructor and "$\oplus$" denotes an arbitrary binary operator.)

The details of the calculation are not important and, anyway, it makes use of properties that I have no time to explain. What is important is just how often associativity of function composition is used. The first three lines establish a minor lemma that is used in the subsequent main calculation. Now, in the first step of the main calculation the grouping of the functions that is used is

$$(\oplus/ \circ \oplus/* \circ +\!\!\!+/) \circ +\!\!\!+/*$$

In the second step it is

$$\oplus/ \circ (\oplus/* \circ \oplus/** \circ +\!\!\!+/*)$$

In the preliminary calculation the grouping of the functions is, firstly,

$$\oplus/ \circ (f* \circ +\!\!\!+/)$$

and, in the subsequent step, it is

$$(\oplus/ \circ \oplus/*) \circ f**$$

Nowhere is it explained that associativity of function composition is being applied — but nor is such an explanation necessary. The property is so fundamental that it is rightly used without mention.

(The above proof could be repeated without making use of function composition, replacing it by function application. But then (a) extensionality must be used requiring explicit introduction of variables and (b) the statement of the $*$ and $/$ promotion properties becomes longer and hence their applicability is less easy to identify.)

Am I saying the obvious?! Something that an audience such as this needs no telling?! Perhaps. But my next example is one where the opportunity to introduce concise notation has been overlooked and is taken from that very school that has always advocated higher-level notations. I am referring to the presentation of a unification algorithm by David Rydeheard and Rod Burstall in [18] and [17].

In their paper [18] Rydeheard and Burstall defined the composition of a parallel pair of morphisms, F, with a morphism, f, which they denoted by $F \circ f$. They also used the symbol "$\cdot$" for composition of morphisms. (See figure 7.) Now there is a very important property that is invisible in diagrammatic form, namely:

$$(F \circ f) \circ g = F \circ (f \cdot g)$$

A capital letter denotes a parallel pair of morphisms.
A small letter denotes a morphism.

$$\bullet \overset{F}{\rightrightarrows} \bullet \xrightarrow{f} \bullet \qquad\qquad F \circ f$$

$$\bullet \xrightarrow{f} \bullet \xrightarrow{g} \bullet \qquad\qquad f \cdot g$$

$$\bullet \overset{F}{\rightrightarrows} \bullet \xrightarrow{f} \bullet \xrightarrow{g} \bullet$$

$$(F \circ f) \circ g = F \circ (f \cdot g)$$

Use ";" for both " $\circ$ " and "$\cdot$"

$$(F; f); g = F; (f; g)$$

Thus write $\qquad F; f; g$

Figure 7: Composition of morphisms.

It can be made invisible in the linear notation if, instead of using different symbols "$\circ$" and "$\cdot$" for the two types of composition, we use just one — say ";". For then the law becomes the statement that ";" is associative:

$$(F; f); g = F; (f; g)$$

and we are free to omit the parentheses.

Rydeheard and Burstall state but do not prove two theorems which form the basis of their presentation of the unification algorithm. I did go through that exercise — formally — in order to determine what sort of improvement this overloading of notation would produce.

The proof of their theorem 1 is very short. Essentially it consists of a

$F \oplus G$	denotes the coalesced sum of F and G
$F*$	denotes the coequaliser of F
$=/F$	$F = (f, f)$ for some f, i.e. F is a parallel pair of *equal* arrows

$$
\begin{array}{ll}
(P1) & =/((F \oplus G); h) \;\equiv\; =/(F; h) \;\wedge\; =/(G; h) \\
(P2) & =/(F; F*) \\
(P3) & =/(F; F*; h)
\end{array}
$$

$$
\begin{array}{ll}
 & =/((F \oplus G); F*; (G; F*)*) \\
\equiv & \quad \{(P1)\} \\
 & =/(F; F*; (G; F*)*) \;\wedge\; =/(G; F*; (G; F*)*) \\
\equiv & \quad \{(P2) F := G; F*\} \\
 & =/(F; F*; (G; F*)*) \\
\equiv & \quad \{(P3) F, h := F, (G; F*)*\} \\
 & \textbf{true}
\end{array}
$$

Figure 8: Lemma in the construction of $(F \oplus G)*$.

proof of property P1 shown in figure 8 together with the three step proof that follows. Now examine the latter proof and you will see that the associativity of ";" is used three times. This is made explicit in figure 9 where the Rydeheard and Burstall notation has been reintroduced. Note that (P0) has been used twice in step 2, once for each argument of the conjunction.

Three steps saved in a proof of length six does not amount to much but bear in mind that a factor of two saving on something that may be "10 to 50" times the length of a mathematics textbook does. More importantly, the choice of notation can greatly influence the recognition or non-recognition of basic proof steps.

Rydeheard and Burstall's presentation of unification brings me neatly to a topic that is dear to my heart since it is the topic with which I began my research career. In the notation that I have introduced, their theorem 1 reads

$$(F \oplus G)* = F*; (G; F*)*$$

where "$\oplus$" constructs the coalesced sum of two pairs of parallel morphisms,

$$
\begin{array}{ll}
(P0) & (F \circ f) \circ g = F \circ (f \cdot g) \\
(P1) & =/((F \oplus G) \circ h) \;\equiv\; =/(F \circ h) \wedge =/(G \circ h) \\
(P2) & =/(F \circ F*) \\
(P3) & =/((F \circ F*) \circ h)
\end{array}
$$

$$
\begin{array}{ll}
 & =/((F \oplus G) \circ (F* \cdot (G \circ F*)*)) \\
\equiv & \qquad \{(P1)\} \\
 & =/(F \circ (F* \cdot (G \circ F*)*)) \;\wedge\; =/(G \circ (F* \cdot (G \circ F*)*)) \\
\equiv & \qquad \{(P0)\} \\
 & =/(F \circ (F* \cdot (G \circ F*)*)) \;\wedge\; =/((G \circ F*) \circ (G \circ F*)*) \\
\equiv & \qquad \{(P2)\} \\
 & =/(F \circ (F* \cdot (G \circ F*)*)) \\
\equiv & \qquad \{(P0)\} \\
 & =/((F \circ F*) \circ (G \circ F*)*) \\
\equiv & \qquad \{(P2)\} \\
 & \mathbf{true}
\end{array}
$$

Figure 9: Making $P0$ explicit.

"$H*$" is the coequaliser of H and ";" is composition. This expression is immediately recognisable as a fundamental property of a regular algebra, indeed the property basic to several closure algorithms such as Warshall's algorithm for constructing the transitive closure of a boolean matrix, or Gaussian elimination for computing the inverse of a real matrix [3].

My first attempt to discuss the design of notation was at a workshop in Sweden four years ago where I presented a formal derivation of Warshall's algorithm. My presentation was greeted by consternation mainly because I had dared to diverge from conventional, familiar notation but also because I claimed to have a formal derivation that used fewer steps than the total number of stated axioms. I must admit to having been over-zealous in my statistics — a "step" often involved a combination of more than one property. Nevertheless, it is possible that a derivation can involve fewer steps than there are axioms: if the notation is so chosen that the most frequently used axioms become "invisible".

I shall try once again to present one of the more interesting parts of that

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdot & \cdot & \cdot \\ a_{10} & a_{11} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & & \\ \cdot & \cdot & & \cdot & \\ \cdot & \cdot & & & \cdot \end{bmatrix}$$

$$i \cdot \mathbf{A} = \begin{bmatrix} & & & & \\ & & \mathbf{0} & & \\ a_{i0} & a_{i1} & \cdot & \cdot & \cdot \\ & & \mathbf{0} & & \\ & & & & \end{bmatrix}$$

$$\mathbf{A} \cdot j = \begin{bmatrix} & a_{0j} & \\ & a_{1j} & \\ \mathbf{0} & \cdot & \mathbf{0} \\ & \cdot & \\ & \cdot & \end{bmatrix}$$

Figure 10: Extraction of rows and columns.

---

derivation. A warning beforehand! Four years ago I didn't dare to diverge too far from standard notation; now I have more courage.

The "standard notation" that I am referring to is matrix notation. Given a square matrix $\mathbf{A}$ and an index i, I shall also use i to denote a matrix of the same size as $\mathbf{A}$ consisting of all 0's excepting for the (i,i)th element which is defined to be 1. The symbol "$\cdot$" is used to denote matrix multiplication.

With this notation, a matrix consisting of all 0's excepting for the ith row of $\mathbf{A}$ is denoted $i \cdot \mathbf{A}$.(See figure 11.) Similarly, a matrix of the same size as $\mathbf{A}$ consisting of all 0's excepting for the jth column of $\mathbf{A}$ is denoted by $\mathbf{A} \cdot j$.

The importance of this notation is that (as is well known) matrix multiplication is associative. Thus, for example, we can immediately observe that the extraction of the (i,j)th element of a matrix can be performed by first extracting the jth column followed by the ith row or, vice versa, by first extracting the ith row followed by the jth column. For our purposes the

$$
\begin{array}{llcll}
(P0) & i \cdot (\mathbf{A} \cdot j) & = & (i \cdot \mathbf{A}) \cdot j & \\
\text{Write} & & i \cdot \mathbf{A} \cdot j & & \\
 & & & & \\
(P1) & i \cdot (\mathbf{A} \cdot \mathbf{B}) & = & (i \cdot \mathbf{A}) \cdot \mathbf{B} & \\
\text{Write} & & i \cdot \mathbf{A} \cdot \mathbf{B} & & \\
 & & & & \\
(P2) & \mathbf{A} \cdot (i \cdot \mathbf{B}) & = & (\mathbf{A} \cdot i) \cdot \mathbf{B} & \\
\text{Write} & & \mathbf{A} \cdot i \cdot \mathbf{B} & & \\
 & & & & \\
(P3) & i \cdot j & = & \mathbf{0} & \text{if } i \neq j \\
 & & = & i & \text{if } i = j \\
 & & & & \\
(P4) & i \cdot \mathbf{0} & = & \mathbf{0} & \\
(P5) & \mathbf{0} \cdot i & = & \mathbf{0} &
\end{array}
$$

Figure 11: Matrix properties.

order is irrelevant and so we choose to write (see figure 11)

$$i \cdot \mathbf{A} \cdot j$$

Similarly, extracting the ith row of $\mathbf{A}$ multiplied by $\mathbf{B}$ is the same as multiplying the matrix formed from the ith row of $\mathbf{A}$ by the matrix $\mathbf{B}$. Indeed, any sequence of matrices and indices can be interpreted in a variety of ways as a succession of operations involving matrix multiplication and the extraction of rows or columns, each interpretation corresponding to a particular ordering of the "$\cdot$" operations. But, it is distinctly preferable *not* to fix on any ordering and to adopt a notation that makes no such commitment.

The four step proof in fig. 12 is one that I thought worthy of only one step in the paper referred to above. This time I have chosen to spell it out some more. Taken together, (P3), (P4) and (P5) (see fig. 11) imply that when two unequal indices are adjacent in an expression the whole expression may be reduced to the null matrix. Formally, the details are more long-winded. In the first step we parenthesise the expression in a way that anticipates the use of property (P3). Subsequently the properties of $\mathbf{0}$ are used to reduce the quantified expression to the null matrix and then properties of summation yield the final expression. In reality, therefore, a complete formal proof of

$$
\begin{array}{ll}
 & \sum(i:\ i \neq k:\ k \cdot \mathbf{A} \cdot k \cdot i \cdot \mathbf{A} \cdot k) \\
= & \qquad \{\text{associativity}\} \\
 & \sum(i:\ i \neq k:\ k \cdot \mathbf{A} \cdot (k \cdot i) \cdot \mathbf{A} \cdot k) \\
= & \qquad \{P3, \text{ context}\} \\
 & \sum(i:\ i \neq k:\ k \cdot \mathbf{A} \cdot \mathbf{0} \cdot \mathbf{A} \cdot k) \\
= & \qquad \{P4, P5\} \\
 & \sum(i:\ i \neq k:\ \mathbf{0}) \\
= & \qquad \{\text{quantifiers}\} \\
 & \mathbf{0}
\end{array}
$$

Figure 12: The notation in use.

this little lemma is about 10 steps.

Two points about this example are worth stressing. First, considerable advantage has been gained by choosing to name matrices *of the same size* rather than row or column *vectors*. The importance of what and what not to name is discussed in some detail by van Gasteren [9]. Second, the chosen notation enables the eye to group together indices much more so than the conventional notation

$$\mathbf{A}_{kk} \cdot \mathbf{A}_{ik}$$

As you will have noticed by now, the rôle of type information is crucial in determining the boundaries of these notational games. A simple example will suffice to demonstrate that it is not always possible to overload one symbol with the meaning of two different operators. The equivalence ($\equiv$) and inequivalence ($\not\equiv$) operators are both associative and symmetric and together enjoy the property

$$a \equiv (b \not\equiv c) = (a \equiv b) \not\equiv c$$

But it would be extremely foolhardy to denote them both by the same symbol!

The reliance on type information has been taken one step too far, in my view, by Meertens [15] in an otherwise outstanding paper. He observes that function application (here denoted by "·") and function composition (here denoted by "$\circ$") together enjoy the property that

$$(f \circ g).x = f.(g.x)$$

He therefore chooses to denote both by juxtaposition (and dubs it "apposition"), thus:

$$fgx$$

This not uncommon notational convention is bedevilled by the fact that "fgx" can be interpreted in six semantically different ways not all of which can be distinguished by type considerations. The use of "x" misleadingly suggests that it always denotes a value and not a function, but in the latter case errors can occur. Consider, for example,

$$\mathit{double}\,(+1)(+2)$$

where $\mathit{double}.f = f \circ f$
Then

$$(\mathit{double}.(+1)) \circ (+2) = +4$$

but

$$\mathit{double}.((+1) \circ (+2)) = +6$$

I.e. "$\mathit{double}\,(+1)(+2)$" has two unequal interpretations. Meertens was well aware of the problems and explicitly refers to the fact that his use of the notation is limited to situations where the type of functions involved is not so complex as that of *double*, (typically in proofs by induction of the properties of list-processing functions) and, consequently, where the danger of error is absent. Figure 13 illustrates a case where possible misinterpretations (for instance, interpreting the second line as

$$+\!\!\!+/\tau*(s +\!\!\!+ \tau*t)$$

) are prohibited by the supplied type information. In such cases the convention can be beneficial. However, as our ability to reason about type structures increases our use of higher-order functions is becoming commonplace. For this reason, in my own work [1] I have rejected Meertens' convention.

I would like to return, at this point, to the aggregated segment sum theorem mentioned at the beginning of this lecture for the proof of the theorem has considerably increased my understanding of the relevance of categorical

$(P1) \quad +\!\!+/x +\!\!+ y = +\!\!+/x +\!\!+ +\!\!+/y$
$(P2) \quad f*x +\!\!+ y = f*x +\!\!+ f*y$

$|[\quad s,t \in A*$
$; \quad +\!\!+/\tau*s = s$
$; \quad +\!\!+/\tau*t = t$
$\triangleright$

$$
\begin{array}{ll}
 & +\!\!+/\tau*s +\!\!+ t \\
= & \quad \{(P2)\} \\
 & +\!\!+/\tau*s +\!\!+ \tau*t \\
= & \quad \{(P1)\} \\
 & +\!\!+/\tau*s +\!\!+ +\!\!+/\tau*t \\
= & \quad \{\text{induction hypo.}\} \\
 & s +\!\!+ t
\end{array}
$$

$]|$

Figure 13: Exploiting type information to the extreme.

concepts to program construction. I should warn you, however, that my understanding and knowledge of category theory remains very weak; what understanding I have is largely due to Hoare's [10] refreshing approach to the theory.

Some very brief background information will help to set the picture for what I have to say. In Eindhoven many "segment problems" have been conceived as illustrations of their program design method. Some are included in the book by Dijkstra and Feijen [8]; some are included in Martin Rem's column in the Science of Computer Programming. In the course of his Marktoberdorf lectures [5] Richard Bird stated a theorem applicable to some such problems. His theorem, however, contains a large number of unnecessary assumptions; my own theorem does not and is consequently much more widely applicable.

In order to exclude all unnecessary assumptions I had to pay very great

Functors

non-empty lists	type constructor	$A+$
	map operator	$f+$
possibly-empty lists	type constructor	$A*$
	map operator	$f*$

(Polymorphic) natural transformations

$$\begin{array}{lcl} \theta & : & + \stackrel{\cdot}{\longrightarrow} * \\ \psi & : & ** \stackrel{\cdot}{\longrightarrow} * \\ \eta & : & +* \stackrel{\cdot}{\longrightarrow} *+ \end{array}$$

Figure 14: Types and natural transformations used in aggregated segment sum theorem.

---

attention to whether a list could be empty or not. I did so by making a clear *type* distinction between *non-empty* lists and *possibly-empty* lists. Thus, I defined two functors $+$ and $*$. For type A, $A+$ is the type of non-empty lists with elements taken from A and for function $f \in A \to B$, $f+$ is the function of type $A+ \to B+$ that is executed by "mapping" f to each element of a given list. $A*$ and $f*$ are defined in the same way but for possibly-empty lists. (See figure 14.)

A consequence of the decision to separate the two types was that I needed to introduce several (polymorphic) natural transformations.[4] The basic transformations I used are listed in figure 14. Note that juxtaposition is used to denote the (horizontal) composition of two functors. Juxtaposition will also be used to denote application of a functor to a natural transformation.

During the course of my calculations I had reason to consider an expression somewhat like

$$\eta \circ \psi \circ \theta* \circ f{+}{+}* \qquad (1)$$

[4]It is known that, in the category of sets, all polymorphic functions from functor F to functor G are natural transformations ([19, 7]). However not all natural transformations are polymorphic.

1. $\theta \circ u{+} = u{*} \circ \theta$, for all u
2. $\psi \circ v{*}{*} = v{*} \circ \psi$, for all v
3. $\eta \circ w{+}{*} = w{*}{+} \circ \eta$, for all w

$$
\begin{array}{ll}
 & \eta \circ \psi \circ \theta{*} \circ f{+}{+}{*} \\
= & \qquad \{ * \text{ distributivity}\} \\
 & \eta \circ \psi \circ (\theta \circ f{+}{+}){*} \\
= & \qquad \{(1),\ u := f{+}\} \\
 & \eta \circ \psi \circ (f{+}{*} \circ \theta){*} \\
= & \qquad \{ * \text{ distributivity}\} \\
 & \eta \circ \psi \circ f{+}{*}{*} \circ \theta{*} \\
= & \qquad \{(2), v := f{+}\} \\
 & \eta \circ f{+}{*} \circ \psi \circ \theta{*} \\
= & \qquad \{(1),\ w := f\} \\
 & f{*}{+} \circ \eta \circ \psi \circ \theta{*}
\end{array}
$$

Figure 15: Another natural transformation.

---

and to transform it into an expression of the form

$$f\dagger \circ \xi$$

where ξ and $\dagger$ were at that time unknown to me. In other words my objective was to commute f from one side of the expression to the other. (In actual fact I have grossly simplified the problem to one where the pattern in the calculation could be recognised during a lecture such as this.)

The ardent squiggolist will have no hesitation in grasping (1) and applying his beloved style of equational reasoning. After a few minutes he will announce that the problem is trivial and display a calculation somewhat like the one in figure 15.

The problem is trivial; nevertheless tremendous benefit is accrued by regarding it somewhat critically. Look, for instance, at how often symbols are carried down from one line to the next; the “action” takes place within a localised region at each step but it is surrounded by other symbols that only obscure the calculation.

$$\varepsilon : \dagger \stackrel{.}{\longrightarrow} \ddagger \equiv \forall(f :: \varepsilon \circ f\dagger = f\ddagger \circ \varepsilon)$$

$$\frac{\varepsilon : \dagger \stackrel{.}{\longrightarrow} \ddagger}{\varepsilon : \wr\dagger \stackrel{.}{\longrightarrow} \wr\ddagger} \qquad \text{poly}$$

$$\frac{\varepsilon : \dagger \stackrel{.}{\longrightarrow} \ddagger}{\varepsilon\wr : \dagger\wr \stackrel{.}{\longrightarrow} \ddagger\wr} \qquad \text{distributivity}$$

$$\frac{\begin{array}{l}\varepsilon_1 : \dagger \stackrel{.}{\longrightarrow} \ddagger \\ \varepsilon_2 : \ddagger \stackrel{.}{\longrightarrow} \wr\end{array}}{\varepsilon_2 \circ \varepsilon_1 : \dagger \stackrel{.}{\longrightarrow} \wr} \qquad \text{composition}$$

$$\begin{array}{rcl} \theta & : & + \stackrel{.}{\longrightarrow} * \\ \psi & : & ** \stackrel{.}{\longrightarrow} * \\ \eta & : & +* \stackrel{.}{\longrightarrow} *+ \end{array}$$

$$\begin{array}{rcll} \theta* & : & +* \stackrel{.}{\longrightarrow} ** & \text{(distributivity)} \\ \psi \circ \theta* & : & +* \stackrel{.}{\longrightarrow} * & \text{(composition)} \\ \psi \circ \theta* & : & ++* \stackrel{.}{\longrightarrow} +* & \text{(poly)} \\ \eta \circ \psi \circ \theta* & : & ++* \stackrel{.}{\longrightarrow} *+ & \text{(composition)} \end{array}$$

Hence, by definition,

$$\eta \circ \psi \circ \theta* \circ f{+}{+}{*} = f{*}{+} \circ \eta \circ \psi \circ \theta*$$

Figure 16: Removing the crud.

The form of the three premises strongly suggest a notational improvement. Note that they all have the same form: the first and last symbol is repeated in each case, and each expresses an equality between a pair of function compositions. Above all, notice that the dummies u, v and w play no rôle except as placeholders. We are led, therefore, to introduce a special notation for this type of property. Specifically (see figure 16) we write

$$\varepsilon : \dagger \stackrel{\cdot}{\longrightarrow} \ddagger$$

in place of

$$\forall (f :: \varepsilon \circ f\dagger = f\ddagger \circ \varepsilon)$$

(Lest there be some misunderstanding let me say at the outset that the notation is not my invention; it is standard to many category theory texts although sometimes the dot above the arrow is omitted.)

Having introduced a new notation it behoves us to examine its properties. Three of these are shown in figure 16 in the form of inference rules. The first I have called *poly* because it depends on the natural transformation ε being polymorphic; the second I have called *distributivity* because it is a consequence of the distributivity requirement for $\wr$ to be a functor; the third, *composition*, is a consequence of the associativity of function composition.

With these rules a derivation of the required property can be written down that contains no unnecessary clutter. It is shown in figure 16. Note that it has precisely the same length as the previous proof, but it involves considerably less writing. Indeed, the calculation has been so simplified that it is hardly necessary to write down the four initial steps.

It is now time to shift the direction of this lecture — for a short while — towards type theory. Elsewhere, I and my colleagues [4] have argued that the prevalent notion that

$$\text{type} \equiv \text{statically checkable}$$

is a major obstacle to further progress. How much so became abundantly clear to me when John Hughes of Glasgow University visited us and talked about polymorphic functions of type

$$\alpha\dagger \rightarrow \alpha\ddagger$$

for all α, for some functors $\dagger$ and $\ddagger$. His observation — that such functions are necessarily natural transformations — made me aware how ignorant I

and	**T**	**F**
T	**T**	**F**
F	**F**	**F**

$$\mathbf{and} \in Bool \times Bool \longrightarrow Bool$$
$$\wedge \in (\alpha \to Bool) \times (\alpha \to Bool) \longrightarrow (\alpha \to Bool)$$
$$(p \wedge q).x = (p.x) \ \mathbf{and} \ (q.x)$$

Figure 17: Lifting **and**.

was of the properties of polymorphism. (I know why I am ignorant; it is because discussion of the subject has been dominated for many years by its use in automatic type inference.) Following on from Hughes' talk Peter de Bruin [5] recently discovered a property of polymorphism that may prove to be very far-reaching. He observed that, given the introduction rules for a type constructor, $*$ say, it is possible to systematically construct an operation on binary relations, which he also denotes by $*$, such that if $f \in \alpha*$ for all α (i.e. f is polymorphic in α) then

$$< f, f > \ \in R*$$

for all relations R (of appropriate type). By choosing suitable relations de Bruin was not only able to verify Hughes' observation but also to verify the promotion theorems put forward by Malcolm [13]. In other words, many of the properties on which our calculations rely are consequences of the fact that the functions we use are very often *polymorphic*.

The discovery that polymorphism is yet richer than I had ever imagined led me to speculate on a notational convention that is ubiquitous but that

[5]It has been pointed out to me that "de Bruin's" theorem was in fact discovered by Reynolds [16] and called the "abstraction theorem". Philip Wadler of Glasgow University has recognised its relevance and used it to prove a wide variety of properties of familiar operators like reduce, map and filter [19].

$$
\begin{array}{lcl}
\bullet & \in & \omega \to (\alpha \to \omega) \\
\circ & \in & (\beta \to \omega) \to (\alpha \to \beta) \to (\alpha \to \omega) \\
\odot & \in & (\beta \times \gamma \to \omega) \to ((\alpha \to \beta) \times (\alpha \to \gamma)) \to (\alpha \to \omega) \\
\circledcirc & \in & (\beta \times \gamma \times \delta \to \omega) \to ((\alpha \to \beta) \times (\alpha \to \gamma) \times (\alpha \to \delta)) \to (\alpha \to \omega)
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{true} & = & \mathbf{T}\bullet \\
\neg & = & \mathbf{not} \circ \\
\wedge & = & \mathbf{and} \odot \\
\mathbf{if} & = & \mathbf{IF} \circledcirc
\end{array}
$$

$\mathbf{not} \in \mathit{Bool} \to \mathit{Bool}$	$\neg \in (\alpha \to \mathit{Bool}) \to (\alpha \to \mathit{Bool})$
$b \ \mathbf{and} \ c = c \ \mathbf{and} \ b$	$p \wedge q = q \wedge p$
$\mathbf{IF\ T\ THEN}\ x\ \mathbf{ELSE}\ y = x$	$\mathbf{if\ true\ then}\ f\ \mathbf{else}\ g = f$

Figure 18: Lifting combinators.

I have never seen *formally* justified. The convention is indeed so ubiquitous that one may be forgiven for not being aware of its existence. I shall refer to it by the name "lifting". It is the convention whereby the same symbol is used to denote an operator of type $A \times B \times ... \times Y \to Z$ and an operator of type $(\alpha \to A) \times (\alpha \to B) \times ... \times (\alpha \to Y) \to (\alpha \to Z)$. Consider, for example, the Boolean **and** operator defined by the truth table shown in figure 17. This operator can easily be "lifted" into an operator on predicates, denoted here by "$\wedge$", as defined in figure 17. Note that "**and**" and "$\wedge$" have quite different types. However, common practice is to denote them both by the same symbol — "$\wedge$".

The same holds for many operators, for example "$\neg$" (logical negation) and conditionals. Indeed, there is a sequence of polymorphic combinators; the ith element in the sequence lifts a function of i arguments into a function of $i+1$ arguments of which the first i are higher-order (see figure 18). The 0th element of this sequence is the well-known K combinator that constructs a constant function from a given argument. The second is the curried form of the function composition operator. Note that all are polymorphic.

A large number of examples — I have yet to see any proof — suggests that these combinators enjoy a remarkable property that fully justifies their never being denoted, namely that, given any valid judgement (an assertion about the type of an object, or an assertion about the equality of two objects), any lifted form of the judgement that is "type correct" is also valid. For example, we have the judgement

$$\mathbf{not} \in \mathit{Bool} \to \mathit{Bool}$$

Lifting it we obtain

$$\mathbf{not}\circ \in (\alpha \to \mathit{Bool}) \to (\alpha \to \mathit{Bool})$$

which is clearly valid. Inspection of the truth table for conjunction reveals the well-known law

$$a \text{ } \mathbf{and} \text{ } b =_{\mathit{Bool}} b \text{ } \mathbf{and} \text{ } a$$

The lifted form is also not surprising:

$$p \wedge q =_{A \to \mathit{Bool}} q \wedge p$$

Similarly, assuming that $\mathbf{IF} \in \mathit{Bool} \to \alpha \to \alpha \to \alpha$ is defined by

$$\mathbf{IF}\ \mathbf{T}\ \mathbf{THEN}\ x\ \mathbf{ELSE}\ y = x$$

$$\mathbf{IF}\ \mathbf{F}\ \mathbf{THEN}\ x\ \mathbf{ELSE}\ y = y$$

it is easily verified that

$$(\mathbf{IF}\circledcirc)\ \mathbf{T}\bullet\ \mathbf{THEN}\ f\ \mathbf{ELSE}\ g =_{A \to B} f$$

(where f, $g \in A \to B$).

(Some people are never convinced by simple examples. A more complicated example is

$$f \ \circ\ (\oplus/) = (\oplus/)\ \circ\ f* \ \Rightarrow\ (f\circ)\ \circ\ ((\oplus\odot)/) = ((\oplus\odot)/)\ \circ\ ((f\circ)*).)$$

The lifting combinators offer the most appropriate example with which I should conclude this lecture because they illustrate very effectively how notation can be designed so as to make important properties invisible, and thus their use automatic. That the reasons for choosing not to denote them are, as far as I am aware, nowhere discussed also emphasises just how blind we can be in our acceptance of the conventions introduced by our predecessors. I can only hope that this talk has opened your eyes to some of the possibilities.

References

[1] R.C. Backhouse. *An Exploration of the Bird-Meertens Formalism.* Technical Report CS8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[2] R.C. Backhouse. *Program Construction and Verification.* Prentice-Hall International, London, 1986.

[3] R.C. Backhouse and B.A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and its Applications*, 15:161–186, 1975.

[4] R.C. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[5] R.S. Bird. Constructive functional programming. International Summer School on Constructive Methods in Computing Science, Marktoberdorf, 24th July to 5th August 1988.

[6] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, 1980.

[7] Peter J. de Bruin. Naturalness of polymorphism. 1989. Department of Mathematics and Computing Science, University of Groningen.

[8] E.W. Dijkstra and W.H.J. Feijen. *Een Methode van Programmeren.* Academic Service, Den Haag, 1984. Now available as *A Method of Programming*, Addison-Wesley, Reading, Mass., 1988.

[9] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1988.

[10] C.A.R. Hoare. Notes on an approach to category theory for computer scientists. International Summer School on Constructive Methods in Computing Science, Marktoberdorf, 24th July to 5th August 1988.

[11] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* PhD thesis, Eindhoven University of Technology, 1977. Mathematical Centre Tracts nr. 83, Math. Centre Amsterdam 1979.

[12] Vladimir Lifschitz. Computing circumscription. In *Proc. 9th International Conference on AI*, pages 121–127, 1985.

[13] G. Malcolm. *Homomorphisms and Promotability.* Computing Science Notes CS 8815, Department of Mathematics and Computing Science, University of Groningen, 1988.

[14] John McCarthy. Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[15] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, North-Holland, 1986.

[16] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E. Mason, editor, *IFIP '83*, pages 513–523, Elsevier Science Publishers, 1983.

[17] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* Prentice-Hall International, 1988.

[18] D.E. Rydeheard and R.M. Burstall. *The Unification of Terms: a Category Theoretic Account.* Technical Report UMCS–85–8–1, Department Computer Science, University of Manchester, 1985.

[19] Philip Wadler. Theorems for free! March 1989. Draft report, Dept. Comp. Science, University of Glasgow.

Appendix. SML Implementation of the segment sum functions

```
fun fst (x, _)  =  x;

(*  data type definitions
    >*  snoc lists
    >+  non-empty snoc lists
*)
datatype 'a**  =  !!        |  >* of 'a** * 'a
     and 'a++  =  T of 'a   |  >+ of 'a++ * 'a
infix >* >+;

(*  curried versions of the constructors
*)
fun >>* l a  =  l >* a
and >>+ l a  =  l >+ a;

(*  reduce functions
*)
fun */ (c, _) !!         =  c
  | */ (c, i) (l >* a)  =  i ( */ (c, i) l) a
and +/ (f, _) (T a)     =  f a
  | +/ (f, i) (l >+ a)  =  i ( +/ (f, i) l) a
fun // f = +/ (fn x=>x, f);

(*  compose constructors with auxiliary function
*)
fun >*~ f l a  =  l >* f a
and >+~ f l a  =  l >+ f a;

(*  map functions
*)
fun ** f  =  */ (!!, >*~ f)
and ++ f  =  +/ (T o f, >+~ f);

\newpage
```

```
(*  join function on >*
*)
fun +-+ l  =  */ (l, >>* );

(*  natural transformations
*)
val theta  =  +/ ( >>* !! , >>* );
val eta  =  let fun add x = */ (T x, >>+)
            in  add !!  o  ** theta
            end;

(*  list processing functions
*)
val tails  =  let fun f x a = (++ (fn y => y >+ a) x) >+ (T a)
              in  +/ (T o T, f)
              end
and inits  =  let fun f (x, g) a = (x >* g a, >>+ (g a))
              in  fst  o  */ ((!!, T) ,f)
              end
val segs   =  eta  o  */ (!!, +-+)  o  ** (theta o tails)  o  inits;

(*  aggregated segment sum theorem
*)
fun lhs p c e  =  // p  o  ++ ( */ (e, c))  o  segs
and rhs p c e  =  let fun f (x,y) a = (p x (c y a), p (c y a) e)
                  in  fst  o  */ ((e,e), f)
                  end;
```