Reprinted from PROGRAM SPECIFICATION AND TRANSFORMATION, L.G.L.T. Meertens, ed.

Two Exercises Found in a Book on Algorithmics

Richard S. Bird
Programming Research Group
University of Oxford, UK

Lambert Meertens
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

1. INTRODUCTION.

A major test of a good notation is how suggestive it is of new relationships between the objects described, and how susceptible it is to the manipulations required to establish such relationships formally. Indeed, if the associated calculus is sufficiently attractive to use, new relationships can come to light simply by the process of manipulating formulae.

The term 'algorithmics' was coined in [Geurts & Meertens 2]: 'Suppose a textbook has to be written for an advanced course in algorithmics. Which vehicle should be chosen to express the algorithms? Clearly, one has the freedom to construct a new language, not only without the restraint of efficiency considerations, but without any considerations of implementability whatsoever.' It was elaborated upon in [Meertens 3], and stands now for both a notation and a calculus for manipulating algorithmic expressions, designed with the aim of meeting the criteria enunciated above. Algorithmics corresponds, broadly speaking, to what is currently known as Transformational Programming, but the level of abstraction is arguably higher than one would normally encounter, and a wide range of specific notation is emphasised. The subject is still in its infancy and it is not the purpose of the present paper to give a comprehensive account. Instead we want to present, in as simple and direct a fashion as possible, two exercises in manipulation as they might appear in some future book on Algorithmics. If, in studying these problems and their solutions, the reader is by turns puzzled, intrigued and finally enlightened as to the real possibilities for a useful calculus of algorithm derivation, then we shall have achieved what we set out to do.

In order to describe the two problems without further preamble, it is necessary to refer to certain concepts without giving them a formal definition; consequently the statement of exactly what is provided and what is required will not be very precise. The first part of the paper is devoted to developing enough of the calculus of algorithmics to remedy this deficiency. We shall then be in a position in the last two sections both to state the two problems precisely and solve them simply by a process of formula manipulation.

Problem 1. The reduction operator '$/$' of APL takes some binary operator $\oplus$ on its left and a vector x of values on its right. The meaning of $\oplus/x$ for $x = [a, b, \ldots, z]$ is the value $a \oplus b \oplus \cdots \oplus z$. For this to be well-defined in the absence of brackets, the operation $\oplus$ has to be associative. Now there is another operator '$\backslash$' of APL called 'scan'. Its effect is closely related to reduction in that we have

$$\oplus\backslash x = [a, a \oplus b, a \oplus b \oplus c, \ldots, a \oplus b \oplus \cdots \oplus z].$$

The problem is to find some definition of scan as a reduction. In other words, we have to find some function f and an operator $\otimes$ so that

$$\oplus\backslash x = (fa) \otimes (fb) \otimes \cdots \otimes (fz).$$

Problem 2. This problem was suggested to us by Phil Wadler. Define a line to be a sequence of characters not containing the newline character NL. It is easy to define a function *Unlines* that converts a non-empty sequence t of lines into a sequence of characters by inserting newline characters

between every two lines of t. Indeed, *Unlines* can be written as a simple reduction as described in Problem 1. Since *Unlines* is injective, the function *Lines*, which converts a sequence of characters into a sequence of lines by splitting on newline characters, can be specified as the inverse of *Unlines*. The problem, just as in Problem 1, is to find a definition by reduction of the function *Lines*.

It is worth remarking that neither problem, both of which are fairly simple to solve, is just an academic exercise of no practical relevance; both illustrate quite serious and important concerns in computation. The former seeks to replace a quadratic time algorithm with a linear one, while the latter is an instance of the problem of finding a computationally effective definition of some operation, given only an effective definition for its inverse. This problem arose in interactive text-formatting.

2. Notation

Our formulae will be (equations between) expressions. The class of allowable expressions (actually, a simplified version) is described below. Certain equations are taken as definitions of the operator or function appearing on the left hand side; these equations may define the function recursively. As simple examples we have

$$fact\ n = \times/[1..n]$$

and

$$fib\ 0 = 0$$

$$fib\ 1 = 1$$

$$fib\ (n+2) = fib\ (n+1) + fib\ n\,.$$

The class of expressions is given by the following BNF syntax, in which the meta-brackets '{' and '}' signify zero or more occurrences of the enclosed part, and signs in double quotes are literals.

expression ::= term {op term} | term op | op term | op

term .:= {primary} primary

primary ::= constant | identifier | "(" expression ")" | "[" expression-list "]"

expression-list ::= | expression {"," expression}

Here, 'op' is short for 'operator'. We shall use symbols such as $\oplus$, $\otimes$, $/$ and $\cdot$ to denote operators. The precedence rule between operators is that all have the same precedence and all are right associative. (So $a \oplus b \otimes c$ means $a \oplus (b \otimes c)$.) This is the convention adopted in [Bird 1]. Function application is denoted by a space; this operator is of higher precedence than others and is left associative. (So $f\ a\ b \otimes c$ means $((f\ a)\ b) \otimes c$.) Again this is the convention adopted in [1] and is one that is familiar to many functional programmers, being copied from Turner's KRC [4].

The forms (term op), (op term) and (op) are known as sections. If they stand alone, no brackets are needed. A presection, of the form $(x \otimes)$, is a prefix function with the property

$$(x \otimes)\, y = x \otimes y\,.$$

A postsection, of the form $(\otimes x)$, is a prefix function with the property

$$(\otimes x)\, y = y \otimes x\,.$$

A full section, of the form $(\otimes)$, is a prefix function with the property

$$(\otimes)\, x\, y = x \otimes y\,.$$

Certain operators, for example the reduction operator '/' of Problem 1, expect operators as their left argument. However, productions such as $\otimes/$ are not allowed by the foregoing grammar, and one has to write $(\otimes)/$. It is a harmless abuse of notation to permit the brackets to be dropped in such a situation, and we shall henceforth do so.

This is all there is to say about syntax. The equal precedence and right-associative rule for all operators except application takes a little getting used to, but turns out to be very convenient, at

least for formulae not involving common arithmetic operations, where other precedence rules are deeply ingrained.

3. STRUCTURES AND HOMOMORPHISMS.

As was stated before, $\oplus/[a, b, \ldots, z] = a \oplus b \oplus \cdots \oplus z$, in which no brackets are needed if $\oplus$ is associative. Henceforth we shall require $\oplus$ to be associative if used in reductions. This means that it is unnecessary to specify the order in which the reduction is performed: left to right, right to left and recursive computation by splitting in two halves all yield the same result. Such 'underspecifications' are generally helpful in algorithmic developments, since they allow one to explore various strategies. In fact, it may be argued that the imposition of an order if it is irrelevant is an *over*specification, an undue commitment that may stand in the way of a useful transformation.

Since the computation order for reduction by an associative operator $\oplus$ is immaterial we can give a symmetric recursive characterisation of $\oplus/$, which can be taken as its formal definition. Let $+\!\!+$ stand for sequence concatenation, so $[a, b] +\!\!+ [c, d, e] = [a, b, c, d, e]$. Also let the identifiers x and y stand for arbitrary sequences. Then

$$\oplus/[a] = a;$$

$$\oplus/(x +\!\!+ y) = (\oplus/x) \oplus (\oplus/y).$$

If the operation $\oplus$ has a unit, then reduction of an empty sequence, $\oplus/[\,]$, stands for that unit. Otherwise, such a reduction is undefined and x and y must not be empty.

The sequence concatenation operator $+\!\!+$ is associative. Associativity is also the requirement on $\oplus$ for $\oplus/$ to be meaningful. This is, of course, not a coincidence. We may, likewise, define reduction over a set, so that $\oplus/\{a, b, \ldots, z\} = a \oplus b \oplus \cdots \oplus z$. The formal definition is similar to the one for sequences, with $\{\cdot\}$ replacing $[\cdot]$ and $\cup$ replacing $+\!\!+$. Not only is set union associative, but also commutative and idempotent. These are precisely the requirements that $\oplus$ has to meet in order to make the given definition of reduction over sets unambiguous. In general, one may consider *structures* that are built by taking singletons from some domain and by applying a binary construction operation to previously erected structures. In the absence of specific properties for the construction operator, we obtain the set of binary trees whose leaves are labelled with atoms. As we have seen, familiar algebraic properties give other familiar data structures: sequences and sets. The algebraic properties of associativity and commutativity together yield yet another familiar data structure: bags or multisets. This means we can give a generic definition of reduction, and also obtain generic algorithmic laws and developments that have, as yet, no commitment to a choice of specific data structure. As the examples in the present paper are only concerned with sequences, this point will not be elaborated upon here.

The identity law (which is an algebraic property that the construction operation may, or may not, have) corresponds to the empty structure (tree, sequence, bag or set, as the case may be). It may happen, and indeed it often does, that an operation $\oplus$ has no unit, but that an algorithmic development naturally leads to forms $\oplus/x$ in which x may be empty. This is a common nuisance that would require special measures to cater for 'exceptional' cases, causing complication of the algorithmic specifications under consideration, which has to be dragged along in the development. Fortunately, in many cases it is possible to employ an expedient stratagem: extend the domain of $\oplus$ with a 'fictitious value', an adjoined element, that assumes the role of the missing unit. For example, the binary operation of taking the minimum value of two operands has no unit in the domain of real numbers. By adding a fictitious value, which we might call ∞, we can assign a meaning to the minimum reduction of an empty structure. This can help to simplify algorithmic developments, and sometimes very much so. It is not uncommon that such fictitious values only have an ephemeral role in a derivation. This is similar to the mathematical 'trick' of solving problems concerning real numbers through a temporary excursion into the complex domain.

Another high-level operation is the map operator '$\cdot$', which takes a function on its left and a sequence, or in general a structure, on its right and replaces each element by its image under the function. For example, we have $f \cdot [a, b, \ldots, z] = [fa, fb, \ldots, fz]$. As for reduction, we can give a recursive characterisation:

$$f \cdot [a] = [fa];$$

$$f\bullet(x +\!\!+ y) \;=\; (f\bullet x) +\!\!+ (f\bullet y).$$

For an empty sequence, we must have $f\bullet[\,] \;=\; [\,]$.

DEFINITION. A function h defined on sequences over a given domain is a 'homomorphism' if there exists an operation $\oplus$ such that, for all sequences x and y over that domain:

$$h\,(x +\!\!+ y) \;=\; h\,x \oplus h\,y.$$

If h is defined on empty sequences, then, moreover, $h\,[\,]$ must be the (possibly fictitious) unit of $\oplus$. The generalisation to other structures is obvious. Both reductions and maps are examples of homomorphisms. This is immediate from their recursive definitions. For a given homomorphism h, the operation $\oplus$ is uniquely determined (on the range of h) by h, and we shall refer to it as 'the' operation of h. The operation of a reduction $\oplus/$ is, of course, $\oplus$. That of a map $f\bullet$ is $+\!\!+$. Not all functions on structures are homomorphisms. A counterexample is the 'number of elements' function $\#$ on sets. On sequences and bags, however, $\#$ is a homomorphism. Moreover, all injective functions are homomorphisms. The importance of homomorphisms is essentially the same as mentioned before for reductions: they allow a variety of computational strategies, among which such important paradigms as iteration (left-to-right construction) and divide-and-rule. The assumption is that $\oplus$ (and h on singleton structures) are relatively cheap to compute. The formulation of a function as a homomorphism shows then also how to develop an incremental approach, as in formal-difference methods.

Although there are other homomorphisms than reductions and maps, these can be viewed as the stuff homomorphisms are made of:

HOMOMORPHISM LEMMA. *A function h is a homomorphism if and only if there exist an operation $\oplus$ and a function f such that $h \;=\; (\oplus/)\circ(f\bullet)$.*

PROOF. The 'if' part follows straightforwardly from the definitions of reduction, map and homomorphism. For the 'only if' part, use induction on the size of the argument sequence, taking for $\oplus$ the operation required by the definition of homomorphism and putting $f \;=\; h\circ[\cdot]$, in which the function $[\cdot]$ turns an element into a singleton sequence.

For short, we say then that h is the homomorphism $(\oplus, f)$. The reduction $\oplus/$ is the homomorphism $(\oplus, \mathrm{id})$, in which 'id' stands for the identity function, and the map $f\bullet$ can be written as $(+\!\!+, [\cdot]\circ f)$. Although certainly not all of algorithmics can be reduced or mapped to the construction of homomorphisms, this is a major constructive paradigm.

4. LAWS

We give, without proof, some simple laws about homomorphisms.

LAW 1. $(f\circ g)\bullet \;=\; (f\bullet)\circ(g\bullet)$.

LAW 2. *Let f, $\oplus$ and $\oplus'$ satisfy $f(x\oplus y) \;=\; (f\,x)\oplus'(f\,y)$ and $f(\oplus/[\,]) \;=\; \oplus'/[\,]$.*
Then $f\circ(\oplus/) \;=\; (\oplus'/)\circ(f\bullet)$.

LAW 3. *Let h be a homomorphism with operation $\oplus$.*
Then $h\circ(+\!\!+/) \;=\; (\oplus/)\circ(h\bullet)$.

Laws 2 and 3 are applications of the homomorphism lemma. For the proofs we refer to [3]. Law 3 can also be derived as a special case of law 2. The second condition of law 2 may be left out if the functions in its conclusion are not required to work on empty sequences. From law 3 we also derive

COROLLARY. (a) $(\oplus/)\circ(+\!\!+/) \;=\; (\oplus/)\circ((\oplus/)\bullet)$.

(b) $(f\bullet)\circ(+\!\!+/) \;=\; (+\!\!+/)\circ((f\bullet)\bullet)$.

Conversely, law 3 follows from the successive application of (b) and (a) of the corollary (using the homomorphism lemma), followed by an application of law 1. From these laws and the corollary one can derive many standard program transformations. For example, some forms of loop fusion can be viewed as an application of law 1, and, as we shall see, filter promotion can also be derived from these laws. The importance of the corollary is that in contrast to laws 2 and 3 it needs no applicability condition.

As example, we give a simple application of law 2. Define the function *last* on non-empty sequences by $last\,(x +\!\!+ [a]) = a$. If we define the operator $\gg$ by $a \gg b = b$, the function *last* can be expressed as a reduction: $last = \gg/$. Let f be a strict function, that is, f 'undefined' = 'undefined'. (Note that $\gg/[\,]$ is undefined.) Since $f(a \gg b) = (fa) \gg (fb)$, law 2 gives us now:

$$f \circ last = f \circ (\gg/) = (\gg/) \circ (f\cdot) = last \circ (f\cdot).$$

The following plays no role in the exercises to follow.
Let $P \triangleleft x$ be the notation for filtering the sequence x with the predicate P. For example, if *even* is the predicate testing for the property of being even, then $even \triangleleft [1, 2, 3, 5, 8] = [2, 8]$. It is easy to see that a filter is a homomorphism, with operation $+\!\!+$. So we have, by law 3,

$$(P\triangleleft) \circ (+\!\!+/) = (+\!\!+/) \circ ((P\triangleleft)\cdot).$$

This corresponds to the filter-promotion or generate-and-test paradigm: rather than filtering one huge structure, we can divide it into smaller structures, filter each of these, and collect the outcomes.

5. THE SCAN–REDUCE PROBLEM.
First of all, we must give a precise definition of the scan operator '$\backslash$'. This is done with the help of a function α that takes a non-empty sequence x and returns the sequence of non-empty initial subsequences of x, arranged in order of increasing length. We have

$$\alpha[a] = [[a]] \tag{1}$$

$$\alpha(x +\!\!+ y) = (\alpha x) +\!\!+ (x +\!\!+) \cdot \alpha y \tag{2}$$

and now we can define

$$\oplus\backslash x = (\oplus/) \cdot \alpha x. \tag{3}$$

The task before us is to find a homomorphism $(\otimes, f)$ so that

$$\oplus\backslash x = \otimes/f\cdot x. \tag{4}$$

First we determine f:

$$\begin{aligned}
fa &= \otimes/[fa] && \text{(definition of } / \text{ on singletons)}\\
&= \otimes/f\cdot[a] && \text{(definition of } \cdot)\\
&= \oplus\backslash[a] && \text{(by (4))}\\
&= (\oplus/)\cdot\alpha[a] && \text{(by (3))}\\
&= (\oplus/)\cdot[[a]] && \text{(by (1))}\\
&= [\oplus/[a]] && \text{(definition of } \cdot)\\
&= [a] && \text{(definition of } / \text{ on singletons).}
\end{aligned}$$

Next we determine $\otimes$ by calculating $x \otimes y$. Suppose $x = \otimes/f\cdot x'$ and $y = \otimes/f\cdot y'$: equivalently, we have $x = \oplus\backslash x'$ and $y = \oplus\backslash y'$. We may assume that x' and y' are non-empty sequences. Then

$$\begin{aligned}
x \otimes y &= (\otimes/f\cdot x') \otimes (\otimes/f\cdot y') && \text{(definition of } x \text{ and } y)\\
&= \otimes/(f\cdot x') +\!\!+ (f\cdot y') && \text{(definition of } /)\\
&= \otimes/f\cdot(x' +\!\!+ y') && \text{(definition of } \cdot)\\
&= \oplus\backslash x' +\!\!+ y' && \text{(by (4))}
\end{aligned}$$

$$
\begin{aligned}
&= (\oplus/) \cdot \alpha (x' +\!\!+ y') && \text{(by (3))}\\
&= (\oplus/) \cdot (\alpha x') +\!\!+ (x' +\!\!+) \cdot \alpha y' && \text{(by (2))}\\
&= ((\oplus/) \cdot \alpha x') +\!\!+ (\oplus/) \cdot (x' +\!\!+) \cdot \alpha y' && \text{(definition of } \cdot \text{).}
\end{aligned}
$$

In the last line, $(\oplus/) \cdot \alpha x' = \oplus\backslash x' = x$, and $(\oplus/) \circ (x' +\!\!+) = ((\oplus/x') \oplus) \circ (\oplus/)$, so that

$$
\begin{aligned}
x \otimes y &= x +\!\!+ ((\oplus/x') \oplus) \cdot (\oplus/) \cdot \alpha y'\\
&= x +\!\!+ ((\oplus/x') \oplus) \cdot y,
\end{aligned}
$$

using definition (3) again. The last expression still contains a reference to x', which remains to be eliminated. We note that x' is the last element of the sequence $\alpha x'$. So, using the rule found for *last* in Section 4 from law 2, $\oplus/x' = \oplus/last\,(\alpha x') = last\,((\oplus/) \cdot \alpha x')$. By definition (3), the argument of *last* equals $\oplus\backslash x' = x$, and so $\oplus/x' = last\ x$. Hence

$$x \otimes y = x +\!\!+ ((last\ x) \oplus) \cdot y,$$

and we are done.

6. The Lines–Unlines problem.

Suppose CH is some set of characters, including the newline character NL. Let $CH' = CH - \{\mathsf{NL}\}$. The function *Unlines* has type $\mathrm{Seq}_+\,(\mathrm{Seq}\ CH') \to \mathrm{Seq}\ CH$, and is defined by

$$Unlines = \oplus/ \tag{1}$$

$$x \oplus y = x +\!\!+ [\mathsf{NL}] +\!\!+ y. \tag{2}$$

The reason we insist *Unlines* takes a non-empty sequence as argument is that the operator $\oplus$ does not have a unit, i.e., $\oplus/[\,]$ is not well-defined. It is easy to show that *Unlines* is injective, that is, $Unlines\ xs = Unlines\ ys$ implies $xs = ys$, and so we can specify the function *Lines* to be

$$Lines = Unlines^{-1}. \tag{3}$$

The task before us is to find a homomorphism $(\otimes, f)$ so that $Lines\ x = \otimes/f \cdot x$. We shall discover the definitions of $\otimes$ and f simply by making use of the identity $Lines(Unlines\ xs) = xs$, or, in other words,

$$\otimes/f \cdot \oplus/xs = xs. \tag{4}$$

First we determine f for arguments $a \neq \mathsf{NL}$:

$$
\begin{aligned}
f\,a &= \otimes/[f\,a] && \text{(definition of } / \text{ on singletons)}\\
&= \otimes/f \cdot [a] && \text{(definition of } \cdot \text{)}\\
&= \otimes/f \cdot \oplus/[[a]] && \text{(definition of } / \text{ on singletons)}\\
&= [[a]] && \text{(by (4)).}
\end{aligned}
$$

Also

$$
\begin{aligned}
f\,\mathsf{NL} &= \otimes/[f\,\mathsf{NL}] && \text{(definition of } / \text{ on singletons)}\\
&= \otimes/f \cdot [\mathsf{NL}] && \text{(definition of } \cdot \text{)}\\
&= \otimes/f \cdot [\,] +\!\!+ [\mathsf{NL}] +\!\!+ [\,] && \text{(as } [\,] \text{ is unit of } +\!\!+ \text{)}\\
&= \otimes/f \cdot \oplus/[[\,], [\,]] && \text{(definition of } \oplus \text{ and } / \text{)}\\
&= [[\,], [\,]] && \text{(by (4)).}
\end{aligned}
$$

Next we must determine $\otimes$. Since each argument of $\otimes$ is a non-empty sequence we need only consider the definition of $(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys)$. We have

$$
\begin{aligned}
&(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys)\\
&= (\otimes/f \cdot \oplus/xs +\!\!+ [x]) \otimes (\otimes/f \cdot \oplus/[y] +\!\!+ ys) && \text{(by (4))}\\
&= \otimes/(f \cdot \oplus/xs +\!\!+ [x]) +\!\!+ (f \cdot \oplus/[y] +\!\!+ ys) && \text{(definition of } / \text{)}
\end{aligned}
$$

$$
\begin{aligned}
&= \otimes/f\cdot(\oplus/xs +\!\!+ [x]) +\!\!+ (\oplus/[y] +\!\!+ ys) && \text{(definition of } \cdot\text{)}\\
&= \otimes/f\cdot((\oplus/xs)\oplus(\oplus/[x])) +\!\!+ ((\oplus/[y])\oplus(\oplus/ys)) && \text{(definition of } /\text{)}\\
&= \otimes/f\cdot(\oplus/xs) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/[x]) +\!\!+ (\oplus/[y]) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/ys) && \text{(definition of } \oplus\text{)}\\
&= \otimes/f\cdot(\oplus/xs) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/[x +\!\!+ y]) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/ys) && \text{(definition of } /\text{)}\\
&= \otimes/f\cdot(\oplus/xs)\oplus(\oplus/[x +\!\!+ y])\oplus(\oplus/ys) && \text{(definition of } \oplus\text{)}\\
&= \otimes/f\cdot\oplus/xs +\!\!+ [x +\!\!+ y] +\!\!+ ys && \text{(definition of } /\text{)}\\
&= xs +\!\!+ [x +\!\!+ y] +\!\!+ ys && \text{(by (4)),}
\end{aligned}
$$

and we are done. Note that the above derivation actually juggles with some potentially fictitious values. We have assigned no meaning to $\oplus/[\,]$, and yet the term $\oplus/xs$ appears above in a context where it is not required that xs be non-empty. No confusion can arise because $\oplus$ does not have a unit in Seq CH, so $\oplus/[\,]$ is adequately defined by the properties it must have. It is consistent with these properties to add the laws

$$(\oplus/[\,]) +\!\!+ [\mathsf{NL}] \;=\; [\mathsf{NL}] +\!\!+ (\oplus/[\,]) \;=\; [\,],$$

which shows how to do actual computations in the extended domain (Seq CH) $\cup\ \{\oplus/[\,]\}$. Note also from the definition of $\otimes$ that its unit must be $[[\,]]$; for example, we can calculate

$$
\begin{aligned}
(xs +\!\!+ [x])\otimes[[\,]] &= (xs +\!\!+ [x])\otimes([[\,]] +\!\!+ [\,])\\
&= xs +\!\!+ [x +\!\!+ [\,]] +\!\!+ [\,]\\
&= xs +\!\!+ [x]\,.
\end{aligned}
$$

This follows also from *Lines* $[\,] = \otimes/[\,]$ and *Unlines* $[[\,]] = [\,]$, since we find $\otimes/[\,] =$ *Lines* (*Unlines* $[[\,]]) = [[\,]]$.

References

1. R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming 6* (1986) 159–189.
2. L. Geurts & L. Meertens. Remarks on Abstracto. *ALGOL Bull. 42* (1978), 56–63.
3. L. Meertens. Algorithmics—Towards programming as a mathematical activity. *Proc. CWI Symp. on Mathematics and Computer Science*, CWI Monographs Vol. 1 (J.W. de Bakker, M. Hazewinkel and J. K. Lenstra, eds.) 289–334. North-Holland, 1986.
4. D. Turner. Recursion equations as a programming language. *Functional Programming and its Applications*, Cambridge University Press, Cambridge, UK, 1982.