SPECIFICATION OF THE FAST FOURIER TRANSFORM ALGORITHM AS A TERM REWRITING SYSTEM

P.H. Rodenburg

Department of Philosophy
University of Utrecht
University of Groningen

D.J. Hoekzema

Departments of Physics and Philosophy
University of Utrecht

Logic Group
Preprint Series
No. 27

Department of Philosophy

University of Utrecht

SPECIFICATION OF THE FAST FOURIER TRANSFORM ALGORITHM AS A TERM REWRITING SYSTEM

P.H. Rodenburg
D.J. Hoekzema

December 1987

Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

SPECIFICATION OF THE FAST FOURIER TRANSFORM ALGORITHM AS A TERM REWRITING SYSTEM

P.H. Rodenburg
Department of Philosophy, Rijksuniversiteit Utrecht
Department of Philosophy, Rijksuniversiteit Groningen

D.J. Hoekzema

Departments of Physics and Philosophy, Rijksuniversiteit Utrecht

We specify an algorithm for multiplying polynomials with complex coefficients incorporating the Fast Fourier Transform algorithm of Cooley and Tukey [CT]. The specification formalism we use is a variant of the formalism ASF described in [BHK]. The difference with ASF is essentially a matter of semantics: we only use pure equations, and this, with some extra restrictions, allows us to consider the specification as a term rewriting system, and take for its meaning the canonical algebra of normal forms.

Key Words & Phrases: Algebraic Specification, Term Rewriting System, Completeness, Semicompleteness, Fast Fourier Transform, Rational Numbers, Polynomial Multiplication.

1986 CR Categories: D.2.1 [**software engineering**]: Requirements/specifications; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - *Algebraic Approaches to Semantics.*

1980 Mathematics Subject Classification: 68B10 [**Software**] Analysis of programs - *Semantics.*

Note: Partial support received from the Dutch government through SPIN under the PRISMA project (Parallel Inference and Storage Machine).

1. Introduction

There is a sense in which many algebraic specifications[1] are more than they seem. Suppose we specify the lower semilattice **2** by the following equations E:

$$\begin{aligned} \mathtt{T} \wedge \mathtt{T} &= \mathtt{T} \\ \mathtt{F} \wedge \mathtt{p} &= \mathtt{F} \\ \mathtt{p} \wedge \mathtt{F} &= \mathtt{F} \end{aligned}$$

Under the initial algebra semantics, E describes **2** as the initial object in the variety determined by E. In general, this object is given as the quotient $C/\sim$ of a suitable set C of closed terms over an equivalence $\sim$ induced by E. Thus, **2** is presented as a pair of classes, one consisting of the terms $\mathtt{T}$, $\mathtt{T} \wedge \mathtt{T}$, $(\mathtt{T} \wedge \mathtt{T}) \wedge \mathtt{T}$ etc., the other of $\mathtt{F}$, $\mathtt{F} \wedge \mathtt{F}$, $\mathtt{T} \wedge (\mathtt{F} \wedge \mathtt{T})$, ... In this way, an interesting operational aspect of E is lost: when we write the specification, we think of it as explaining the operation 'greatest lower bound' on an ordered pair of things, one named $\mathtt{F}$, the other $\mathtt{T}$; but in the end, $\mathtt{F}$ is explained as

[1]For algebraic specification theory, we refer to [BT].

$\mathtt{T} \wedge (\mathtt{F} \wedge \mathtt{T})$ just as much as the other way round. In other words, E was intended as a system of *oriented* equations, but the orientation has no place in equational specification; and the semantics that goes with it calls on rather sizeable constructs to explain simple things.

The intended directionality can be taken serious by reading E as a set of rewrite rules[2]. E then says: in an element of C,

you may replace a subterm $\mathtt{T} \wedge \mathtt{T}$ by $\mathtt{T}$,

and subterms of the form $\mathtt{F} \wedge t$ or $t \wedge \mathtt{F}$ by $\mathtt{F}$.

In this particular case — and, as we will show, in many other cases — we get an elegant semantics in the bargain. We can replace the equivalence classes by the normal forms of the rewriting system (these happen to be $\mathtt{T}$ and $\mathtt{F}$) and define the operation $\wedge$ by: $x \wedge y$, for x, y in $\{\mathtt{T}, \mathtt{F}\}$, is the normal form of the term $x \wedge y$.

Another reason for assigning direction to equations has to do with *testing* specifications. As with many other large constructions, it is hard to see by mere looking that a large algebraic specification is what one intended it to be. As a last resort, one might want to just do a number of deductions, and see if they lead to unexpected results. But a deduction is simply a sequence of rewritings, in which the axioms may be used both from left to right and from right to left. The rewriting system represented by this procedure will of course be rather complex; so here too, the question whether one really wants an equation to be used in two directions is worth considering.

The greater part of this paper is taken up by a large specification. Its purpose is, first of all, to add to the body of experience in specifying in the formalism ASF of Bergstra, Heering and Klint (introduced in [BHK]). In the second place, it is an experiment to see to what extent algorithms of practical interest can be specified as reasonable term rewriting systems — where reasonability should at least imply confluency and weak normalization. Thirdly, we intend to use it as a test object for mechanical rewriting. This intention has influenced our choice of the algorithm to be specified: we chose polynomial multiplication and the Fast Fourier Transform because it is well known that these lend themselves to parallelization (see [V]). It seems to be generally thought that term rewriting (like the execution of functional programs, cf. [V]) should be done in parallel to be feasible; it then is of some interest to study how the parallelism inherent in an algorithm can be passed on to a parallel implementation of a term rewriting system. At the other extreme, as an algorithm with little inherent parallelism, we have specified the Gauß elimination algorithm, in [HR]; this combination should help to obtain information about the potential for parallelization of term rewriting *by itself*. Actually, even the present specification contains a part with little inherent parallelism, to wit, the specifications of numbers; it would be quite a success if these could be implemented at all. Such an implementation is not really necessary: the imports in the later modules can be replaced by references to an off-the-peg number system.

2. The semantics

A signature consists of a set of sort symbols and a set of function symbols of fixed types, as usual; terms are defined as usual. We assume in our discussions that there are closed (i.e. variable-free) terms of every sort. We write $C(\Sigma)$ for the set of closed terms over signature Σ. Rewrite rules must make sense as equations; moreover, variables occurring in the right hand side of a rewrite rule must also occur in the left hand side. A side effect of this restriction is that closed terms never rewrite to terms that contain variables.

The cumbersome expression 'term rewriting system' will be abbreviated to TRS. Likewise, 'strongly normalizing' becomes SN, 'weakly normalizing' WN.

We call a TRS *complete* if it is confluent (Church-Rosser) and SN; *semicomplete* if it is confluent and WN. In semicomplete systems, every term has a unique normal form. Thus it is for semicomplete systems that the suggestion of the first section works.

Definition. Let Σ be a signature. A *structure of type* Σ is a mapping $\mathbf{A}$ with domain Σ such that for each $S \in \Sigma$, $\mathbf{A}(S)$ is a nonempty set, and if $G \in \Sigma$ is of type $S_1 \times \ldots \times S_n \to S_0$, $\mathbf{A}(G)$ is a mapping from $\mathbf{A}(S_1) \times \ldots \times \mathbf{A}(S_n)$ into $\mathbf{A}(S_0)$.

There are other ways of presenting structures. One that we shall use, for finite $\Sigma = \{S_1, \ldots, S_n, G_1, \ldots, G_k\}$, is: as tuples $\langle \mathbf{A}(S_1), \ldots, \mathbf{A}(S_n), \mathbf{A}(G_1), \ldots, \mathbf{A}(G_k) \rangle$. When in the sequel there is little need for differentiating between structures for the same signature, we shall write $\mathbf{A}(A)$, for $A \in \Sigma$, as A.

Let $\mathbf{A}$ be a structure of type Σ. We extend $\mathbf{A}$ over $C(\Sigma)$ by

$$\mathbf{A}(G(t_1, \ldots, t_n)) = \mathbf{A}(G)(\mathbf{A}(t_1), \ldots, \mathbf{A}(t_n)).$$

If $\mathbf{A}$ is a structure of type Σ, and Σ' is any signature, then the restriction $\mathbf{A}|\Sigma'$ is simply the restriction of the mapping $\mathbf{A}$ to $\Sigma' \cap \Sigma$. (There are no real difficulties with empty signatures.)

Definition. Let $\mathcal{R} = (\Sigma, R)$ be a semicomplete TRS, with rewrite rules R, over a signature Σ. Let C° be the set of closed normal forms of $\mathcal{R}$; and for each sort $S \in \Sigma$, $C^\circ_S \subseteq C^\circ$ the set of closed normal forms of sort S. The canonical model of $\mathcal{R}$ is the Σ-structure $\mathbf{C}$ $(=\mathbf{C}_{\mathcal{R}})$ such that

for each $S \in \Sigma$, $\mathbf{C}(S) = C^\circ_S$;
if $G: S_1 \times \ldots \times S_n \to S_0$, and $t_1 \in \mathbf{C}(S_1)$, ..., $t_n \in \mathbf{C}(S_n)$, then $\mathbf{C}(G)(t_1, \ldots, t_n)$ is the normal form of $G(t_1, \ldots, t_n)$.

The canonical model of $\mathcal{R}$ is (indeed) a model of R — in particular, it is isomorphic to the quotient $C/\sim$ of the set of all terms over the congruence induced by R.

3.The role of hiding.

To specify a data structure algebraically, one often uses (and sometimes *must* use, cf. [BT]) hidden operations. The need of hiddens persists when one focuses on canonical models as the meanings of specifications. Indeed, the operational aspect of rewrite rules, and the importance of concepts such as completeness, that concern the operation of these rules, leads to a natural extension of the significance of hiding. Suppose we want to specify the structure $(\mathbb{N}, 0, S, \boldsymbol{P})$ of the natural numbers with zero and successor, and an enumeration $\boldsymbol{P}$ of the prime numbers: $\boldsymbol{P}(0) = 2$, $\boldsymbol{P}(1) = 3$, etc.. To do this, we need some operations that are fairly natural (addition, multiplication, division); but also some complicated operations which represent the process of calculating the next prime (see the module PRIM below). In sensible calculations, only some special combinations of intermediate values occur. Now one might insist that any term involving hidden operations be confluent and normalizing; but this would often require careful stipulations too arbitrary to be of any interest. Instead, the hiddens will be left out of the canonical models altogether.

A *TRS with hiddens* is a triple $\mathcal{R} = (\Sigma, \Sigma^*, R)$ such that (Σ^*, R) is a TRS, and Σ is a subsignature of Σ^*. In the sequel we suppress the adjunct "with hiddens". We call Σ the *visible* signature of $\mathcal{R}$, Σ^* the *full* signature. $\mathcal{R}$ is *complete* if every closed term over Σ is confluent and strongly normalizes to a closed term over Σ; *semicomplete* if every such term is confluent and weakly normalizes to a closed term over Σ. A term over Σ^* is *attainable* if it can be obtained by rewriting a closed Σ-term. The definition of (semi-)completeness may then be restated as: $\mathcal{R}$ is (semi-)complete if every attainable term is confluent and strongly (weakly) normalizes to a term over Σ. If we wish to emphasize that a TRS has no hiddens, we may call it *flat*. The operation of *flattening* turns a system $\mathcal{R} = (\Sigma, \Sigma^*, R)$ into the associated flat system $\mathrm{Flat}(\mathcal{R}) = (\Sigma^*, R)$.

For a set E of equations (or rewrite rules interpreted as equations) over a signature Σ^*, and a subsignature $\Sigma \subseteq \Sigma^*$, we denote by $\mathrm{Eq}_\Sigma(E)$ the set of all equations over Σ that follow from E. A *model* for a TRS $\langle \Sigma, \Sigma^*, R \rangle$ is a Σ-structure that is a model of $\mathrm{Eq}_\Sigma(R)$. Equivalently, a model of $\mathcal{R}$ is any $\Sigma(\mathcal{R})$-structure that has an extension that can be expanded to a model of $\mathrm{Flat}(\mathcal{R})$. We call such a model *separating* if its values on distinct closed normal forms over Σ are distinct. If a weakly normalizing TRS has a separating model, it is confluent.

This hiding mechanism is more versatile than is necessary for the actual specification below. In every case, $\mathcal{R}$ will be such that its models can be immediately expanded to models of $\mathrm{Flat}(\mathcal{R})$. We stick to the generalization because it seems natural.

Definition. Let $\mathcal{R} = \langle \Sigma, \Sigma^*, R \rangle$ be a semicomplete TRS. Let C° be the set of closed normal forms of $\mathcal{R}$ over Σ; and for each sort $S \in \Sigma$, $C_S^\circ \subseteq C^\circ$ the set of closed normal forms over Σ of sort S. The *canonical model* of $\mathcal{R}$ is the Σ-structure $\mathbf{C}$ ($=\mathbf{C}_{\mathcal{R}}$) such that

for each $S \in \Sigma$, $\mathbf{C}(S) = C_S^\circ$;
if $G: S_1 \times ... \times S_n \to S_0$ is a function symbol of Σ, and $t_1 \in \mathbf{C}(S_1)$, ..., $t_n \in \mathbf{C}(S_n)$, then
$\mathbf{C}(G)(t_1, ..., t_n)$ is the normal form of $G(t_1, ..., t_n)$.

This definition involves a liberalization of the ordinary initial algebra semantics, as announced above. In general, the canonical model of a TRS $\langle \Sigma, \Sigma^*, R \rangle$ with hiddens is a subalgebra of a restriction of the initial model of $\langle \Sigma^*, R \rangle$ — neglecting isomorphism. The canonical model of a semicomplete TRS is separating. Observe that all separating minimal models of a given weakly normalizing TRS are isomorphic.

4. Normal extensions.

Specifications tend to be large; so one tries to develop them step by step, as a series of modules imported one in the other. It is highly desirable that we are able to prove important properties of specifications in the same stepwise fashion. Since for us confluency and normalization are paramount, we must have a method for deducing these properties from properties of component modules.

Let $\mathcal{R}_0 = \langle \Sigma_0, \Sigma_0^*, R_0 \rangle$ and $\mathcal{R}_1 = \langle \Sigma_1, \Sigma_1^*, R_1 \rangle$ be TRSes. We say $\mathcal{R}_0$ is a subsystem of $\mathcal{R}_1$ (notation: $\mathcal{R}_0 \subseteq \mathcal{R}_1$) if $\Sigma_0^* \subseteq \Sigma_1^*$ and $R_0 \subseteq R_1$.

Definition. Let $\mathcal{R}_0$ and $\mathcal{R}_1$ be TRSes (as above), with $\mathcal{R}_0 \subseteq \mathcal{R}_1$. We call $\mathcal{R}_1$ a *normal extension* of $\mathcal{R}_0$ if

(i) any term formed by applying a function symbol in Σ_1 to closed $\mathcal{R}_0$-normal forms over $\Sigma_0 \cap \Sigma_1$ (in accordance with its type) weakly normalizes (in $\mathcal{R}_1$) to a closed term over $\Sigma_0 \cap \Sigma_1$;

(ii)if $\mathbf{A} \models \mathcal{R}_0$, then $\mathbf{A}|\Sigma_1$ has a subalgebra that can be expanded to a model of $\mathcal{R}_1$.

$\mathcal{R}_1$ is a *strongly normal extension* of $\mathcal{R}_0$ if (i) also holds with "weakly" replaced by "strongly".

Condition (i) implies in particular that constants of Σ_1 reduce to $\mathcal{R}_0$-normal forms over $\Sigma_0 \cap \Sigma_1$. In general, it may be said to require that $\mathcal{R}_1$ is about the same objects as $\mathcal{R}_0$, and by rewriting in $\mathcal{R}_1$ we find the irreducible names these objects had in $\mathcal{R}_0$. Condition (ii) requires that if $\mathcal{R}_1$ proves $t_1 = t_2$, with t_1 and t_2 closed terms over $\Sigma_0 \cap \Sigma_1$, $t_1 = t_2$ was provable in $\mathcal{R}_0$ already — or in logical terms, that $\mathcal{R}_1$ is conservative over $\mathcal{R}_0$.

In every concrete case below, we shall have $\Sigma_0 \subseteq \Sigma_1$. The definition above accommodates *hiding on import*, i.e. hiding part of the visible (export) signature of an imported module. We have been able to avoid this, but the option may seem desirable.

We call a TRS *regular* if it is both left-linear and nonambiguous.

Theorem. If $\mathcal{R}_0$ and $\mathcal{R}_1$ are TRSes such that $\mathcal{R}_0 \subseteq \mathcal{R}_1$, then

(i) (Bergstra & Klop [BK]) if $\mathcal{R}_0$ is semicomplete, and $\mathcal{R}_1$ is a normal extension of $\mathcal{R}_0$, then $\mathcal{R}_1$ is semicomplete;

(ii) if $\mathcal{R}_0$ is complete, $\mathcal{R}_1$ is regular, and $\mathcal{R}_1$ is a strongly normal extension of $\mathcal{R}_0$, then $\mathcal{R}_1$ is complete.

Proof. Suppose $\mathcal{R}_i = \langle \Sigma_i, \Sigma^*_i, R_i \rangle$, $i \in \{0, 1\}$.

(i). By a simple induction on closed terms, $\mathcal{R}_1$ is WN. Since $\mathcal{R}_0$ is semicomplete, it has a separating model $\mathbf{A}$. Let $\mathbf{B}$ be a model of $\mathcal{R}_1$, constructed by expanding some subalgebra of $\mathbf{A}|\Sigma_1$. If s and t are closed $\mathcal{R}_1$-normal forms over Σ_1, then by (i) of the above definition s and t are also closed $\mathcal{R}_0$-normal forms over Σ_0. Then $\mathbf{B} \models s=t$ implies $\mathbf{A} \models s=t$; hence $s \equiv t$ since $\mathbf{A}$ is separating. It follows that $\mathbf{B}$ is separating as well, so $\mathcal{R}_1$ must be confluent.

(ii). We use induction on terms to show that every closed Σ_1-term strongly normalizes to a closed term over $\Sigma_0 \cap \Sigma_1$.

Suppose $t \equiv \mathrm{F}(t_1,...,t_n)$, and $t_1,...,t_n$ strongly normalize to closed terms over $\Sigma_0 \cap \Sigma_1$. t need not be a redex, but it can become one by contractions in $t_1,...,t_n$. The reverse cannot happen: if $\mathrm{F}(s_1,...,s_n)$ is an instance of some rewrite rule [r], and $s_i \twoheadrightarrow r_i$, then $\mathrm{F}(r_1,...,r_n)$ is an instance of the same rewrite rule by regularity. If t has an infinite reduction sequence

$$t \equiv t^0 \to t^1 \to t^2 \to ...,$$

this cannot be the fault of $t_1,...,t_n$: at the least, these must have been multiplied ad infinitum. So if $\overset{\cdot}{t}_1,...,\overset{\cdot}{t}_n$ are the normal forms of $t_1,...,t_n$, $t' \equiv \mathrm{F}(\overset{\cdot}{t}_1,...,\overset{\cdot}{t}_n)$ should also have an infinite reduction sequence. But by (i) of the definition of strongly normal expansion, this is impossible.¶

5.Polynomial multiplication and the Fast Fourier Transform

We shall explain modules as we go along; but we make an exception to this rule for the final modules. These concern multiplying polynomials, and the discrete Fourier transform.

Let n be a natural number, $N = 2^n$, and a a polynomial of degree at most $N-1$ with complex coefficients: say

$$a = a_0 + a_1X + ... + a_{N-1}X^{N-1},$$

with $a_0, a_1, ..., a_{N-1} \in \mathbb{C}$. For $j \in \mathbb{Z}$, we set $W_n(j) = e^{\pi ij2^{1-n}}$ (we shall often drop the subscript n). Then

(1) $W_n(0) = 1$.

By the *Fourier transform* $F_n(a)$ of a let us understand the polynomial

$$\sum_{k=0}^{N-1} X^k \cdot \sum_{l=0}^{N-1} W_n(kl)a_l \quad ;$$

the *inverse Fourier transform* (up to a factor $1/N$) $G_n(a)$ is the polynomial

$$\sum_{k=0}^{N-1} X^k \cdot \sum_{l=0}^{N-1} W_n(-kl)a_l \quad .$$

If $b = b_0 + b_1X + \ldots + b_{N-1}X^{N-1}$ and $c = c_0 + c_1X + \ldots + c_{N-1}X^{N-1}$ are polynomials, then their diagonal product is

$$a \wedge b := \sum_{k=0}^{N-1} b_k c_k X^k \quad .$$

Now let a and b be polynomials; we assume that the degree of their product $a \cdot b$ is less than N. Since $e^{(\pi+x)i} = -e^{xi}$,

$$(2)\ W_n(\frac{N}{2}+j) = -W_n(j);$$

whence

$$(3)\ \sum_{k=0}^{N-1} W_n(kp) = 0$$

for every integer $p \not\equiv 0 \pmod{N}$. Also note that

$$(4)\ W(m) \cdot W(n) = W(m+n).$$

The following calculation proves an equality reminiscent of the convolution theorem for ordinary Fourier transforms:

$$\frac{1}{N} G_n(F_n(a) \wedge F_n(b)) = \frac{1}{N} G_n\left(\sum_{k=0}^{N-1} X^k \cdot \sum_{l=0}^{N-1} W(kl)a_l \cdot \sum_{l'=0}^{N-1} W(kl')b_{l'} \right)$$

$$= \frac{1}{N} G_n \left(\sum_{k=0}^{N-1} X^k \cdot \sum_{l=0}^{N-1} \sum_{l'=0}^{N-1} W(k(l+l'))a_l \cdot b_{l'} \right) \qquad \text{(distribute)}$$

$$= \frac{1}{N} \sum_{m=0}^{N-1} X^m \cdot \sum_{k} W(-km) \cdot \sum_{l;l'} W(k(l+l'))a_l \cdot b_{l'}$$

$$= \frac{1}{N} \sum_{m=0}^{N-1} X^m \cdot \sum_{l;l'} a_l \cdot b_{l'} \cdot \sum_{k} W(k(l+l'-m)) \qquad \text{(distribute, rearrange, and use (4))}$$

$$= \frac{1}{N} \sum_{m=0}^{N-1} X^m \cdot \sum_{l+l'=m} N a_l \cdot b_{l'} \qquad \text{(by (1) and (3))}$$

$$= a \cdot b.$$

If, for the moment, we do not count the Fourier transformations, the above calculation suggests a cheap way of multiplying polynomials. For suppose $a \cdot b$ has degree $< N$: the number of complex multiplications and additions needed to normalize $a \cdot b$ by straightforwardly applying the distributive law is of the order of N^2 (notation: $O(N^2)$); whereas for $F_n(a) \wedge F_n(b)$, N multiplications suffice.

It just remains to be seen that Fourier transformations are cheap. The cost obviously depends on the method; we choose the wellknown algorithm of Cooley and Tukey [CT]. Let a be as above, with degree less than $N := 2^n$, and $n > 0$. We can express $F_n(a)$ in $F_{n-1}(\_)$ as follows. Let $Even(a)$ be the polynomial of degree $< 2^{n-1}$ with coefficients $a_0, a_2, \ldots, a_{N-2}$; and $Odd(a)$ the polynomial with coefficients $a_1, a_3, \ldots, a_{N-1}$. Let $M = \frac{N}{2}$. We have

(5) $W_n(2k) = W_{n-1}(k)$ and $W_n(2k+1) = W_n(1) \cdot W_{n-1}(k)$

from (4). Let w_n be the polynomial

$$\sum_{k=0}^{M-1} X^k W_n(k) .$$

Now we calculate

$$F_n(a) = \sum_{k=0}^{N-1} X^k \cdot \sum_{l=0}^{N-1} W_n(kl)a_l$$

$$= \sum_{k=0}^{N-1} X^k \cdot \left(\sum_{l=0}^{M-1} W_{n-1}(kl)a_{2l} + W_n(k) \cdot \sum_{l=0}^{M-1} W_{n-1}(kl)a_{2l+1} \right) \text{ by (5),}$$

$$= \sum_{k=0}^{M-1} X^k \sum_{l=0}^{M-1} W_{n-1}(kl)a_{2l} + \sum_{k=M}^{N-1} X^k \sum_{l=0}^{M-1} W_{n-1}(kl)a_{2l}$$

$$+ \sum_{k=0}^{M-1} W_n(k) \cdot X^k \sum_{l=0}^{M-1} W_{n-1}(kl)a_{2l+1} + \sum_{k=M}^{N-1} W_n(k) \cdot X^k \sum_{l=0}^{M-1} W_{n-1}(kl)a_{2l+1}$$

$$= (1 + X^M)F_{n-1}(Even(a)) + (1 - X^M)(w_n \wedge F_{n-1}(Odd(a))) \quad \text{(the minus sign comes from (2)).}$$

In words: the sequence of coefficients in $F_n(a)$ may be calculated cheaply (in O(N) multiplications and additions) from those in $F_{n-1}(Even(a))$ and $F_{n-1}(Odd(a))$, and a sequence w_n of complex roots of unity. Similarly, $G_n(a)$ is obtained from $G_{n-1}(Even(a)),G_{n-1}(Odd(a))$, and roots of unity $W_n(-k)$. Since this process must be repeated n times, the number of operations needed to obtain the Fourier transform is of the order of $N\log_2(N)$.

We shall specify this multiplication algorithm from scratch, first working through a series of number systems.

6.The specification formalism

We have taken some slight notational liberties with regard to ASF; e.g. in the module BIN (near the end) we introduce postfix operation symbols. Another feature we have assumed, as it becomes desirable when one does not use conditional equations, is projections for tuple types. It so happens that it is not much trouble to specify them. Here is a list of our deviations from ASF:

— We use non-ASCII characters, such as arrows, and subscripts and bold print.

— We have operator names that are letters; the colon is used as an operator name.

— We use square brackets in identifiers.

— -- sometimes does *not* signify the beginning of a comment; namely, when a human reader would not expect it to.

— Binding is not mentioned when it is to objects with the same name. For instance, if parametrized module MOD_1 has a sort X, in parameter M, and MOD_2 has a sort of the same name in parameter M', and M is bound to M', then X is tacitly bound to X.
— Priority is not always indicated with brackets; thus, we write `x + y·z` instead of `x + (y·z)`.
— We have postfix operators.
— Functions may have tuples as arguments. So given `F: X → Y × Z, G: Y × Z → U`, we write `G(F(x))`.

In all, the differences are slight, and they accommodate the human reader. Corrected versions have been typechecked by F. Wiedijk and P. Hendriks.

Given modules MOD_0 and MOD_1, we denote by $MOD_0 + MOD_1$ the following module:

```
module MOD0 + MOD1
begin
  imports MOD0, MOD1
end MOD0 + MOD1
```

7.Normalization

The meaning of a TRS, we may say, is its canonical model. A suitable specification consisting of just one module, without imports, may be regarded as a TRS in a straightforward way. Thus to attach meaning to modularized specifications, it will suffice if we can normalize them to single modules, eliminating the `imports` clauses. This is not entirely unproblematic. Since we find the normalization algorithm of [BHK] hard to understand, we shall give an informal explanation of our own.

Basically, a normalization step that eliminates a clause

```
imports    MOD1
```

from a module MOD_0 consists of adding the items under the various headings of MOD_1 to the lists under the same headings in MOD_0. This goes wrong in two respects:

1° name clashes may result, notably of variables;

2° if the `imports` clause involves renamings, these must be carried out some way — particularly if MOD_1 was parametrized.

Our cavalier attitude towards these difficulties is as follows. First, one should be careful to avoid undesired name clashes. In practice this means that merging is to be preceded by some preventive disambiguation; this can be effected in various ways. In the case of variables it suffices to affix sort tags. Second, renamings *change* a module before it is fitted into a larger module. If a sort is renamed, it is changed everywhere in the module expression; in particular, sort tags of variables and types of functions may change. After these changes, one simply unites as indicated above. —This description leaves much to be desired, but it should do for normalization from the inside out, that is, for the elimination of normalized imports.

8.The specification

The following listing of modules with their imports may help the reader to trace the origin of imported functions. Parametrized modules may be imported repeatedly, the parameters being bound differently each time.

BOOL
NAT
COMP imports BOOL and NAT
MON imports NAT
DIV imports COMP and MON
PRIM imports DIV
SIM
PINT imports NAT
DIRE imports COMP and MON
FAC imports PRIM, SIM, PINT and DIRE
REV imports SIM
EXP imports NAT
MUL imports PRIM, REV and EXP
INT imports PINT and MON
LADD imports SIM
REX imports INT, REV and LADD
RAT imports REX
SING imports INT
PROJ
RED imports REX, SING, SIM and PROJ
LITR imports SIM twice
RADD imports RAT, RED, LITR (twice), MUL, FAC and PROJ
CORT imports RADD
POW imports CORT
ARCTAN imports RADD
PI imports ARCTAN
RUN imports PI and POW
SCAM imports CORT and SIM
MERGE imports SIM
LRUN imports RUN, SCAM, LITR and MERGE
SPLIT imports SIM
DIM imports SIM
DFT imports LRUN, SPLIT, LADD, DIM, LITR and REV

BIN
POL imports CORT and SIM
DEG imports POL and BIN
UL imports NAT and BIN
MULP imports DFT, POL, DEG and UL

Now it is time for the actual work. As a TRS, the specification below is complete. The practical way to prove this is module by module. Indeed, every module will be complete in the sense of §3; and the completeness will either be easy to prove by standard methods, or it will follow from the completeness of some simple combination of earlier modules by the normal extension theorem (§4). Proofs will at best be sketched; but some of the earlier proof sketches are rather laborious. To attempt formal proofs here would be mistaken. They would still rely heavily on the structure of the data at hand, which never is such that more formalism would help to understand it. Some of the earlier applications of the normal extension theorem (notably with PRIM and FAC) also happen to be among the most difficult. In any case, the reader who wishes to economize on mental effort may be content with convincing himself of semicompleteness.

```
module BOOL                                    --booleans
begin
  exports
    begin
      sorts    B
      functions    T: → B
                   F: → B
                   _∧_: B × B → B
    end
  equations
  [1] T ∧ T = T
  [2] T ∧ F = F
  [3] F ∧ T = F
  [4] F ∧ F = F
end BOOL
```

This module is SN; the lower semilattice **2** is a separating model. Hence as remarked in §3, BOOL is confluent, hence complete. Moreover, **2** is minimal, which, again by §3, shows it is the structure specified. (Another way of seeing BOOL is confluent is by noting it is regular.)

```
module NAT                                --natural numbers
begin
  exports
    begin
      sorts N
      functions  0: → N
                 S_: N → N
                 _+_: N × N → N
                 _·_: N × N → N
    end
  variables
    m, n: → N
  equations
  [1] m + 0 = m
  [2] m + Sn = S(m+n)
  [3] m · 0 = 0
  [4] m · Sn = m·n + m
end NAT
```

This module is complete; it describes the standard model $\mathbf{N} = \langle \mathbb{N}, +, \cdot, S, 0 \rangle$ of arithmetic. To see this, first note that NAT is SN: it is easy to weight the operation symbols in such a way that right hand sides always weigh less than left hand sides; then SN follows by recursive path ordering. NAT is confluent since it is regular. Alternatively, one may check that **N** is a separating model of NAT; and since it is minimal, it is the canonical model.

```
module COMP              --comparison of natural numbers
begin
  exports
    begin
      functions    _≤_: N × N → B
                   _<_: N × N → B
    end
  imports BOOL, NAT
  variables
    m, n: → N
  equations
  [1] 0 ≤ m = T
  [2] Sm ≤ 0 = F
  [3] Sm ≤ Sn = m ≤ n
  [4] m < n = Sm ≤ n
end COMP
```

COMP is clearly regular; and if s and t are normal forms of NAT, then $s \leq t$ and $s < t$ strongly normalize to T or F. It is easy to expand the disjoint union of **N** and **2** with suitable predicates (≈ functions to truth values) ≤ and <. This proves that COMP is a strongly normal extension of BOOL + NAT; we get its canonical model in the bargain.

```
module  MON                --cut-off subtraction (monus)
begin
  exports
    begin
      functions     _÷_: N × N → N
    end
  imports NAT
  variables
    m, n: → N
  equations
  [1] 0 ÷ m = 0
  [2] m ÷ 0 = m
  [3] Sm ÷ Sn = m ÷ n
end MON


module DIV                                  --divisibility
begin
  exports
    begin
      functions     _|_: N × N → B
    end
  imports COMP, MON
  functions    D: B × N × N → B
  variables
    m, n: → N
  equations
  [1] m | 0 = T
  [2] m | Sn = m ≤ Sn ∧ m|(Sn÷m)
end DIV
```

DIV is complete, being regular and a strongly normal extension of COMP + MON, which in its turn is a strongly normal extension of COMP. It so happens that *every* closed term is SN.

```
module PRIM                                          --primes
begin
  exports
    begin
      functions  P_: N → N
    end
  imports DIV
  functions S0: B × N × N → N
            S1: B × N × N → N
  variables
    m, n: → N
  equations
  [1] P0 = SS0
  [2] PSm = S0(0 < m, SPm,0)
  [3] S0(F, m, n) = m
  [4] S0(T, m, 0) = S0(PS0·PS0 ≤ Sm, Sm, S0)
  [5] S0(T, m, Sn) = S1(PSn | m, m, Sn)
  [6] S1(T, m, n) = S1(P0 | Sm, Sm, 0)
  [7] S1(F, m, n) = S0(PSn·PSn ≤ m, m, Sn)
end PRIM
```

PRIM is supposed to give an enumeration of the prime numbers $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, etc. The local functions S_0 and S_1 will turn out to implement the sieve of Eratosthenes. We shall use the normal extension theorem to show that the module works, and that it is complete. Note that PRIM is regular. The normal forms of DIV that are terms of PRIM are precisely the booleans T and F, and the numerals S^n0 (for $n \in \mathbb{N}$, setting $S^00 \equiv 0$ and $S^{n+1}0 \equiv S(S^n0)$).

We show first that $P(S^n0)$ strongly normalizes to a numeral, for each $n \in \mathbb{N}$. We use induction. For P0 there is only one reduction, yielding SS0. Now assume that $P(S^k0)$ strongly normalizes to a numeral, for each $k \leq n$; consider $P(S^{n+1}0)$. This reduces only by [2], to

$$S_0(0 < S^n0,\ SP(S^n0),\ 0).$$

This contains redexes $0 < S^n0$ and $SP(S^n0)$; the first we know to be SN, and the second is SN by assumption. The entire term becomes a redex when $0<S^n0$ is normalized. Suppose $SP(S^n0)$ has been reduced to some term t — the form of t does not matter; we know t strongly normalizes to the numeral for successor of the n-th prime. If $0 < S^n0 \twoheadrightarrow F$ (i.e. if $n = 0$), we must apply [3], and get t, and eventually S^30. (Observe that 3 is the only prime that immediately succeeds another prime.) If $0 < S^n0 \twoheadrightarrow T$ (i.e. if $n > 0$), we must apply [4]. The result is

$$S_0(PS0 \cdot PS0 \leq St,\ St,\ S0).$$

Since $n \geq 1$, the induction hypothesis guarantees that the proper subterms strongly normalize. The entire term becomes a redex only when $\mathtt{P}\mathtt{S}0 \cdot \mathtt{P}\mathtt{S}0 \leq \mathtt{S}t$ is normalized. Take any t' that $\mathtt{S}t$ reduces to: we know that t' strongly normalizes to the numeral for p_n+2. If $\mathtt{P}\mathtt{S}0 \cdot \mathtt{P}\mathtt{S}0 \leq \mathtt{S}t \twoheadrightarrow \mathtt{F}$, i.e. if p_n+2 < 9, p_n+2 is prime. Then indeed the rule to be applied is [3], which produces t'. If $\mathtt{P}\mathtt{S}0 \cdot \mathtt{P}\mathtt{S}0 \leq \mathtt{S}t \twoheadrightarrow \mathtt{T}$, we apply [5] and get

$$\mathtt{S}_1(\mathtt{P}\mathtt{S}0 \mid t', t', \mathtt{S}0).$$

Again, the proper subterms strongly normalize. We may disregard reductions of t'. The entire term becomes a redex once $\mathtt{P}\mathtt{S}0 \mid t'$ is normalized. We distinguish two cases.

I. $\mathtt{P}\mathtt{S}0 \mid t' \twoheadrightarrow \mathtt{T}$. Then p_n+2 is not prime. To continue, we must use [6], and get

$$\mathtt{S}_1(\mathtt{P}0 \mid \mathtt{S}t', \mathtt{S}t', 0).$$

Since p_n+3 is even, we have to repeat [6], and try, in vain, whether $2 \mid p_n$+4. This brings us to the other case.

II. $\mathtt{P}\mathtt{S}0 \mid t' \twoheadrightarrow \mathtt{F}$. We must apply [7]. The reduct is

$$\mathtt{S}_0(\mathtt{P}\mathtt{S}\mathtt{S}0 \cdot \mathtt{P}\mathtt{S}\mathtt{S}0 \leq t', t', \mathtt{S}\mathtt{S}0).$$

If $\mathtt{P}\mathtt{S}\mathtt{S}0 \cdot \mathtt{P}\mathtt{S}\mathtt{S}0 \leq t' \twoheadrightarrow \mathtt{F}$, we next apply [3] and end up with the normal form of t'. And indeed, we knew that p_n+2 is odd, and we have checked that 3 does not divide it, so that it must be prime if it is less than 25. If $\mathtt{P}\mathtt{S}\mathtt{S}0 \cdot \mathtt{P}\mathtt{S}\mathtt{S}0 \leq t' \twoheadrightarrow \mathtt{T}$, we apply [5]. This comes down to trying if 5 divides p_n+2. If it does, we go on with p_n+3 (and case I.); if it does not, we try if 7 is too large, and if it is not, if it divides p_n+2; and so on.

Thus we cycle through [5], [6] and [7], increasing m and n in turn, until we hit a prime m. This m will be represented by a term of DIV, which strongly normalizes.

Suppose $\mathbf{A}$ is a separating model of DIV; let Σ be the export signature of PRIM. Let $\mathbf{B} \subseteq \mathbf{A}|\Sigma$ be the minimal subalgebra. Expand $\mathbf{B}$ with an operation P defined by $P(S^n0) = S^{p_n}0$; the resulting algebra we call $\mathbf{C}$. To show that $\mathbf{C}$ is a model of PRIM, it will suffice to define an expansion of $\mathbf{C}$ that is a model of Flat(PRIM). The following operations S_0 and S_1 will do the job:

$S_0(F, m, n) = m$;

$S_0(T, m, 0) = Sm$ if $m < 8$, or Sm is a power of 2, or $Sm < p^2$ for the least prime $p > 2$ that divides Sm;

$= \mu k > Sm$ (k is prime) (i.e. the least prime greater than Sm) otherwise;

$S_0(T, 0, Sn) = 1$;

$S_0(T, Sm, Sn) = Sm$ if Sm has no prime factors greater than p_n, or $p_{Sn} \nmid Sm$ and $Sm < p^2$ for the least prime $p > p_{Sn}$ that divides Sm;

$= \mu k > Sm$ (k is prime) otherwise;

$S_1(T, 0, n) = 1$;

$S_1(T, 1, n) = 3$;

$S_1(T, m, n) = \mu k > m$ (k is prime) if $m > 1$;

$S_1(F, m, n) = m$ if m has no prime factors greater than p_n, or $m < p^2$ for the least prime $p > p_n$ that divides m;

$= \mu k \geq m$ (k is prime) otherwise.

```
module SIM                              --simple lists
begin
  parameters  M
    begin
      sorts   X
    end M
  exports
    begin
      sorts   LX
      functions     _;_: X × LX → LX
                    [X]: → LX
    end
end SIM


module PINT                             --positive integers
begin
  exports
    begin
      sorts   P
      functions     1: → P
                    S_: P → P
                    N_: P → N
                    P_: N → P
                    _≤_: P × N → B
    end
  imports COMP
  variables
    p: → P
    n: → N
  equations
  [1] N1 = S0
  [2] NSp = SNp
  [3] P0 = 1        --arbitrary.
  [4] PS0 = 1
  [5] PSSn = SPSn
  [6] p ≤ n = Np ≤ n
end PINT
```

```
module DIRE                              --division with remainder
begin
  exports
    begin
      functions    _:_: N × N → N
                   If: B × N × N → N
    end
  imports COMP, MON
  variables
    m, n, k: → N
  equations
  [1] If(T, m, n) = m
  [2] If(F, m, n) = n
  [3] m : 0 = 0                    --arbitrary
  [4] m : Sn = If(m≤n, 0, S((m÷Sn):Sn))
end DIRE
```

m:*n* is the integer part of the quotient of *m* over *n*. Rule [3] fixes a result for division by zero. In a way, it would be more satisfactory to have

```
[*]   m : n = If(m<n, 0, S((m÷n):n))
```

instead of [3] and [4], but our theory does not deal with partial algebras. A similar comment could be made about rule [3] in PINT.

```
module FAC                                        --factorization
begin
  exports
    begin
      functions F_: P → LN
                S_: LN → LN
    end
  imports PRIM, SIM {M bound by [X → N] to PRIM,
                          renamed by [LX → LN], [[X] → [N]]},
          PINT, DIRE
  functions F1: B × N × N × LN → LN
            F2: B × N × N × LN → LN
  variables
    m, n: →N
    p: →P
    a: →LN
  equations
  [1] S[N] = [N]
  [2] S(m;a) = Sm ; a
  [3] Fp = F1(p ≤ S0, Np, 0, 0;[N])
  [4] F1(T, m, n, a) = a
  [5] F1(F, m, n, a) = F2(Pn|m, m, n, a)
  [6] F2(F, m, n, a) = F2(PSn|m, m, Sn, 0;a)
  [7] F2(T, m, n, a) = F1(m:Pn ≤ S0, m:Pn, n, Sa)
end FAC
```

FAC is complete; we shall check this by the same method as we used with PRIM. First, we introduce an auxiliary module $\mathcal{M}$, which is the result of instantiating SIM with PRIM, renaming X to N and LX to LN, and adding PINT and DIRE. Completeness of $\mathcal{M}$ follows from that of PRIM, PINT and DIRE, by simple applications of the normal extension theorem. We want to show that FAC is a strongly normal extension of $\mathcal{M}$.

The first condition to be satisfied boils down to: any term F(S^{m}1) strongly normalizes to a list of numerals S^{n}0. Note that FAC is regular.

Evidently, F1 normalizes to 0;[N]. If S^{m}1 is any other numeral, m+1 has a unique representation

$$m+1 = p_k^{n_k} \cdot \ldots \cdot p_0^{n_0}$$

with $n_k > 0$.

Now F(S^{m}1) is a redex for [3], and [3] only. The reduct is

$$F_1(S^m1 \leq S0,\ NS^m1,\ 0,\ 0;[N]).$$

The only redexes are $\mathtt{S}^m\mathtt{1} \le \mathtt{S0}$ and $\mathtt{NS}^m\mathtt{1}$. Since $\mathcal{M}$ is complete, we know these subterms strongly normalize; the one to F (since $m > 0$), the other to $\mathtt{S}^{m+1}\mathtt{0}$. The entire term becomes a redex, for [5], when $\mathtt{S}^m\mathtt{1} \le \mathtt{S0}$ has been normalized. Say we have reduced $\mathtt{NS}^m\mathtt{1}$ to s. [5] produces

$$(*)\quad \mathtt{F}_2(\mathtt{P0} \mid s,\ s,\ \mathtt{0},\ \mathtt{0};\mathtt{[N]}).$$

Let us disregard possible reductions in s. Then (*) can only be attacked by normalizing $\mathtt{P0} \mid s$. There are two cases.

I. $\mathtt{P0} \mid s \twoheadrightarrow \mathtt{T}$. This means that $n_0 > 0$. We must continue with [7], to

$$\mathtt{F}_1(s{:}\mathtt{P0} \le \mathtt{S0},\ s{:}\mathtt{P0},\ \mathtt{0},\ \mathtt{S}(\mathtt{0};\mathtt{[N]})).$$

To continue with the entire term, we must normalize the first argument. The other arguments are SN: $s{:}\mathtt{P0}$ goes to the numeral for $m' := p_k^{n_k} \cdot \ldots \cdot p_0^{n_0 - 1}$; $\mathtt{S}(\mathtt{0};\mathtt{[N]})$ normalizes to $\mathtt{S0};\mathtt{[N]}$ by [2]. If $s{:}\mathtt{P0} \le \mathtt{S0}$ normalizes to T, implying $k = 0$ and $n_0 = 1$, we are done with factorizing, and we reach $\mathtt{S0};\mathtt{[N]}$ through [4] and (possibly) [2].

Now suppose $s{:}\mathtt{P0} \le \mathtt{S0} \twoheadrightarrow \mathtt{F}$. Say we have done $s{:}\mathtt{P0} \twoheadrightarrow t_1$, $\mathtt{S}(\mathtt{0};\mathtt{[N]}) \twoheadrightarrow t_2$. Then

$$\mathtt{F}_1(\mathtt{F},\ t_1,\ \mathtt{0},\ t_2) \to \mathtt{F}_2(\mathtt{P0} \mid t_1,\ t_1,\ \mathtt{0},\ t_2)$$

by [5], and we are back in the position of (*) with, essentially, m replaced by m'; that is, this time we try if $2 \mid m'$.

II. If $\mathtt{P0} \mid s \twoheadrightarrow \mathtt{F}$, then $n_0 = 0$. By [6] we reach

$$\mathtt{F}_2(\mathtt{PS0} \mid s,\ s,\ \mathtt{S0},\ \mathtt{0};(\mathtt{0};\mathtt{[N]})),$$

and we are back in position (*), with $\mathtt{[N]}$ replaced by $\mathtt{0};\mathtt{[N]}$, the third argument by S0, and P0 by PS0. In effect, we now try if $3 \mid m$.

Continuing in this way, we see that $\mathtt{F}(\mathtt{S}^m\mathtt{1})$ strongly normalizes to

$$\mathtt{S}^{n_k}\mathtt{0};\ldots;\mathtt{S}^{n_0}\mathtt{0};\mathtt{[N]}.$$

(We have omitted brackets; we establish the convention that ; associates to the right, which in fact is the only way the above term makes sense.)

For the second condition, it will suffice to show that the canonical model M of $\mathcal{M}$ can be expanded to a model of Flat(FAC). M has four sorts:

N, the natural numbers;

P, the positive integers, which we may take to be the natural subset of N;

B, the booleans; and

LN, the finite sequences $\langle n_0,\ldots,n_{k-1}\rangle$ ($k \in \mathbb{N}$) of natural numbers.

We use ε for the empty sequence, and bold letters $\boldsymbol{n}$ to vary over sequences $\langle n_0,\ldots,n_{k-1}\rangle$.The following expansion by operations S, F, F_1 and F_2 will do the job (note that the argument of F may be assumed > 0). Set

$S(\varepsilon) = \varepsilon$;

$S(n_0, n_1, \ldots,n_k) = \langle n_0+1, n_1, \ldots,n_k\rangle$;

$F(1) = \langle 0\rangle$;

$F(p_0^{n_0}\cdot\ldots\cdot p_k^{n_k}) = \langle n_k, \ldots,n_0\rangle$, if $n_k > 0$;

$F_1(T, m, n, \boldsymbol{l}) = \boldsymbol{l}$;

$F_1(F, m, n, \boldsymbol{l}) = \varepsilon$ if $m \leq 1$ or $p_j \mid m$ for some $j < n$;

$F_1(F, p_i^{n_0}\cdot\ldots\cdot p_{k+i}^{n_k}, i, \langle l_0,\ldots,l_{j-1}\rangle) = \langle n_k,\ldots,n_1, n_0+l_0, l_1,\ldots,l_{j-1}\rangle$ (= $\langle n_k,\ldots,n_1\rangle$ if $j = 0$),

with $n_k > 0$;

$F_2(F, m, n, \boldsymbol{l}) = \varepsilon$ if $m \leq 1$ or $p_j \mid m$ for some $j \leq n$;

$F_2(F, p_{i+1}^{n_1}\cdot\ldots\cdot p_{k+i+1}^{n_{k+1}}, i, \langle l_0,\ldots,l_{j-1}\rangle) = \langle n_k,\ldots,n_1, l_0, \ldots,l_{j-1}\rangle$, with $n_{k+1} > 0$;

$F_2(T, m, n, \boldsymbol{l}) = S(\boldsymbol{l})$ if $m{:}p_n \leq 1$,

$= \varepsilon$ if $m{:}p_n > 1$ and $p_j \mid m$ for some $j < n$;

$F_2(T, p_i^{n_0}\cdot\ldots\cdot p_{k+i}^{n_k}, i, \langle l_0,\ldots,l_{j-1}\rangle) = \langle n_k,\ldots,n_1, n_0+l_0, l_1,\ldots,l_{i-1}\rangle$ (= $\langle n_k,\ldots,n_1\rangle$ if $j = 0$),

with $n_k > 0$.

Of the following modules, REV is easily seen to be a strongly normal extension of SIM, and EXP of NAT. Hence they are complete.

```
module    REV                                          --reversion
begin
  exports
    begin
      functions    _;_: LX × LX → LX
                   R_: LX → LX
    end
  imports SIM
  variables
    a,b: → LX
    x: → X
  equations
  [1] [X];a = a
  [2] (x;a) ; b = x; (a;b)
  [3] R[X] = [X]
  [4] R(x;a) = Ra;(x;[X])
end REV


module EXP                                        --exponentiation
begin
  exports
    begin
      functions    _↑_: N × N → N
    end
  imports NAT
  variables
    m, n: → N
  equations
  [1] m ↑ 0 = S0
  [2] m ↑ Sn = (m↑n) · m
end EXP
```

```
module MUL                          --multiplying out
begin
  exports
    begin
      functions   N_: LN → N
    end
  imports PRIM, REV {M bound by [X → N] to PRIM, renamed by [LX →
                        LN], [[X] → [N]]},
          EXP
  functions M: LN × N × N → N
  variables
    l, m, n: → N
    a: → LN
  equations
  [1] Na = M(Ra, 0, S0)
  [2] M(l;a, m, n) = M(a, Sm, n·(Pm↑l))
  [3] M([N], m, n) = n
end MUL
```

The purpose of this module is to regain natural numbers from lists of exponents of their prime factors. One may check that it is regular. Let $\mathcal{M}$ be the module that results from eliminating the functions N and M. $\mathcal{M}$ is easily seen to be complete.

Let $t \equiv S^{n_{k-1}}0;\ldots;S^{n_0}0;[N]$ be a list of numerals. It is straightforward to check that Nt strongly normalizes to the numeral for $\prod_{j<k} p_j^{n_j}$. Next, let $\mathbf{M}$ be the canonical model of $\mathcal{M}$. We expand $\mathbf{M}$ to a model of Flat(MUL) by operations N and M defined by:

$$N(n_{k-1}, \ldots, n_0) = \prod_{j<k} p_j^{n_j};$$

$$M(\langle n_{k-1}, \ldots, n_0\rangle, m, n) = n \cdot \prod_{j<k} p_{j+m}^{n_j}.$$

Thus MUL is a strongly normal extension of $\mathcal{M}$; hence MUL is complete, by the normal extension theorem.

```
module     INT                                  --integers
begin
  exports
    begin
      sorts   Z
      functions 0: → Z
                +_: P → Z
                -_: P → Z
                -_: Z → Z
                _+_: Z × Z → Z
                _-_: Z × Z → Z
                I_: N → Z
                J_: N → Z
                A_: Z → N
    end
  imports PINT, MON
  variables
    p, q: → P
    y, z: → Z
    n: → N
  equations
  [1] -0 = 0
  [2] --p = +p
  [3] -+p = -p
  [4] I0 = 0
  [5] ISn = +PSn
  [6] J0 = 0
  [7] JSn = -PSn
  [8] y + 0 = y
  [9] 0 + y = y
  [10] +p + +q = I(Np + Nq)
  [11] -p + -q = J(Np + Nq)
  [12] +p + -q = I(Np÷Nq) + J(Nq÷Np)
  [13] -p + +q = J(Np÷Nq) + I(Nq÷Np)
  [14] y - z = y + -z
  [15] A0 = 0
  [16] A+p = Np
  [17] A-p = Np
end INT
```

To see that INT works, note that for $m, n \in \mathbb{N}$, at least one of $m \div n$ and $n \div m$ is zero. For the rest, the normal forms are conspicuous and completeness is straightforward.

Another presentation of the integers, which in some ways is rather natural, constructs them from zero by a successor function and a predecessor function. In our case this would lead to complications: something like *Pred*(*Suc*(*Pred*(*z*))) would probably be an ambiguous redex.

```
module LADD                                    --list addition
begin
  parameters  M'
    begin
      sorts X
      functions    _+_: X × X → X
    end M'
  exports
    begin
      functions    _+_: LX × LX → LX
    end
  imports SIM {M bound to M'}
  variables
    x, y: → X
    a, b: → LX
  equations
  [1] a + [X] = a
  [2] [X] + a = a
  [3] x;a + y;b = x+y ; a+b
end LADD
```

```
module     REX   --positive rationals as lists of exponents
begin
  exports
    begin
      functions _·_: LZ × LZ → LZ
                1:_: LZ → LZ
                _:_: LZ × LZ → LZ
    end
  imports INT, REV {M bound by [X → Z] to INT, renamed by [LX →
                        LZ], [[X] → [Z]]},
          LADD {M' bound by [X → Z] to INT, renamed by [LX → LZ],
                        [X] → [Z]}
  variables
    y, z: → Z
    p, q: → LZ
  equations
  [1] p · q = R(Rp + Rq)
  [2] 1:[Z] = [Z]
  [3] 1:(y;p) = -y;(1:p)
  [4] p : q = p·(1:q)
end REX
```

In the light of the normal extension theorem, it is easy to see that REX is complete. Its significance is as follows. Positive rational numbers have unique representations $p_{k-1}^{n_{k-1}} \cdot \ldots \cdot p_0^{n_0}$, with $p_0, p_1, \ldots$ the succession of primes, $n_0, \ldots, n_{k-1} \in \mathbb{Z}$, and $n_{k-1} > 0$ (setting the empty product at 1). Thus there is an injection

$$p_{k-1}^{n_{k-1}} \cdot \ldots \cdot p_0^{n_0} \mapsto n_{k-1}; \ldots; n_0; [Z]$$

(with n_i the unique normal form of INT that corresponds with n_i) from $\mathbb{Q}^+$ to the set of normal forms of sort LZ. Now if p corresponds to p, q to q, the normal form of p · q stands for $p \cdot q$; that of 1:p for $1/p$, and that of p:q for p/q.

We have not quite got the rationals yet, since $0; n_{k-1}; \ldots; n_0; [Z]$ is normal, whereas $p_k^0 \cdot p_{k-1}^{n_{k-1}} \cdot \ldots \cdot p_0^{n_0} = p_{k-1}^{n_{k-1}} \cdot \ldots \cdot p_0^{n_0}$. This matter will be dealt with by the next module, which also introduces positive, negative and zero.

```
module RAT                                          --rationals
begin
  exports
    begin
      sorts Q
      functions 𝟘: → Q
                +_: LZ → Q
                -_: LZ → Q
                -_: Q → Q
                1:_: Q → Q
                _·_: Q × Q → Q
                _:_: Q × Q → Q
    end
  imports REX
  variables
    a, b: → LZ
    p, q: → Q
  equations
  [1] +a·+b = +(a·b)
  [2] +a·-b = -(a·b)
  [3] -a·+b = -(a·b)
  [4] -a·-b = +(a·b)
  [5] 1:+b = +(1:b)
  [6] 1:-b = -(1:b)
  [7] 1:𝟘 = 𝟘                          --arbitrary
  [8] p:q = p·(1:q)
  [9] -𝟘 = 𝟘
  [10] -+a = -a
  [11] --a = +a
  [12] q·𝟘 = 𝟘
  [13] 𝟘·q = 𝟘
  [14] +(0;a) = +a
  [15] -(0;a) = -a
end RAT
```

Next we describe the reduction of simple fractions p^{y}_{i} , p^{z}_{i} to a common denominator. Our procedure starts with two integers, to be thought of as exponents of some prime number, and results in a triple ‹numerator of first argument, numerator of second argument, common denominator› of natural numbers. In symbols: if $y \oplus z = \langle l, m, n \rangle$, then $x^{y}+x^{z} = \frac{x^{l}+x^{m}}{x^{n}}$. Completeness is straightforward. .

```
module    SING                    --single factor reduction
begin
  exports
    begin
      functions _⊕_: Z × Z → N × N × N
    end
  imports INT
  variables
    p, q: → P
  equations
  [1] 0 ⊕ 0 = ‹0, 0, 0›
  [2] 0 ⊕ +p = ‹0, Np, 0›
  [3] 0 ⊕ -p = ‹0, 0, Np›
  [4] -p ⊕ 0 = ‹0, 0, Np›
  [5] +p ⊕ 0 = ‹Np, 0, 0›
  [6] +p ⊕ +q = ‹Np, Nq, 0›
  [7] +p ⊕ -q = ‹Np+Nq, 0, Nq›
  [8] -p ⊕ +q = ‹0, Np+Nq, Np›
  [9] -p ⊕ -q = ‹Nq÷Np, Np÷Nq, Np+ (Nq÷Np)›
end SING
```

Note that $m + (n \div m) = \max(m, n)$.

```
module PROJ                                    --projection
begin
  parameters  M
    begin
      sorts X
    end M
  exports
    begin
      functions @0: X × X × X → X
                @1: X × X × X → X
                @2: X × X × X → X
    end
  variables
    x, y, z: →X
  equations
  [1] @0(x, y, z) = x
  [2] @1(x, y, z) = y
  [3] @2(x, y, z) = z
end PROJ
```

```
module RED                                                --reduction
begin
  exports
    begin
      functions _⊕_: LZ × LZ → LN × LN × LN
    end
  imports REX, SING, SIM {M bound by [X → N] to NAT, renamed by
                                [LX → LN], [[X] → [N]]},
          PROJ {M bound by [X → N] to NAT}
  functions R1: LZ × LZ × LN × LN × LN→ LN × LN × LN
  variables
    p, q: → LZ
    a, b, c: → LN
    y, z: → Z
  equations
  [1] p ⊕ q = R1(Rp, Rq, [N], [N], [N])
  [2] R1(y;p, z;q, a, b, c) =
                  R1(p, q, @0(y⊕z);a, @1(y⊕z);b, @2(y⊕z);c)
  [3] R1(y;p, [Z], a, b, c) =
                  R1(p, [Z], @0(y⊕0);a, b, @2(y⊕0);c)
  [4] R1([Z], z;q, a, b, c) =
                  R1([Z], q, a, @1(0⊕z);b, @2(0⊕z);c)
  [R5] R1([Z], [Z], a, b, c) = ‹a, b, c›
end RED
```

It is routine to check that RED is complete. Its action is as follows. Given two positive rationals p and q, represented in the manner discussed under REX, first the representations are reversed, so that one does not have to count before one can say which prime the first integer in the list is supposed to be the exponent of. The pair of reversed rationals is then transformed to a triple $\langle \boldsymbol{l}, \boldsymbol{m}, \boldsymbol{n} \rangle$ of lists of natural numbers, with the following property: if N is the function from sequences of natural numbers to $\mathbb{Z}^+$ specified in MUL above, then

$$p + q = \frac{N\boldsymbol{l} + N\boldsymbol{m}}{N\boldsymbol{n}} \ .$$

To help us complete the specification of the rationals, we introduce a parametrized module, to be applied to translate lists of natural numbers to lists of integers; then, finally, we define addition.

```
module LITR                                    --list translation
begin
  parameters  M'
    begin
      sorts X, Y
      functions  I_: X → Y
    end M'
  exports
    begin
      functions  I_:LX → LY
    end
  imports SIM {M bound to M'}, SIM {M bound by [X → Y] to M',
            renamed by  [LX → LY], [[X] → [Y]]}
  variables
    x: →X
    a: →LX
  equations
  [1] I[X] = [Y]
  [2] I(x;a) = Ix; Ia
end LITR
```

```
module RADD                                    --rational addition
begin
  exports
    begin
      functions  _+_: Q × Q → Q
                 _-_: Q × Q → Q
    end
  imports RAT, RED, LITR {M' bound by  [X → N], [Y → Z] to INT,
            renamed by [LX → LN], [[X] → [N]], [LY → LZ],
            [[Y] → [Z]]},
          LITR {M' bound by [X → N], [Y → Z], [I → J] to INT,
            renamed by [LX → LN], [[X] → [N]], [LY → LZ],
            [[Y] → [Z]], [I → J]}  --the binding concerns I: X→Y,
              the renaming I: LX→LY
          MUL, FAC, PROJ {M bound by [X → LN] to MUL}
  functions A: Z × LZ → Q
  variables
    p: →P
    q: →Q
    a, b: →LZ
    n: →N
  equations
  [1] q + 𝟘 = q
  [2] 𝟘 + q = q
  [3] +a + +b = +(IFP(N@0(a⊕b) + N@1(a⊕b)) · J@2(a⊕b))
  [4] -a + -b = -(IFP(N@0(a⊕b) + N@1(a⊕b)) · J@2(a⊕b))
  [5] +a + -b = A(IN@0(a⊕b) + JN@1(a⊕b), J@2(a⊕b))
  [6] -a + +b = A(IN@1(a⊕b) + JN@0(a⊕b), J@2(a⊕b))
  [7] A(0, a) = 𝟘
  [8] A(+p, a) = +(IFp · a)
  [9] A(-p, a) = -(IFp · a)
end RADD
```

Recall that J takes a natural number $n \in N$ to its inverse $-n \in Z$; N takes the product of a sequence of natural numbers; P turns natural numbers into positive integers, F factorizes these, and I embeds the natural numbers into the integers. Note that $N(l) > 0$, for any sequence l of natural numbers. Hence applying P is safe in the sense that it is not important that we remember what we said $P(0)$ was.

The patient reader will find on inspection that this module is regular. The sum $\mathcal{M}$ of the imports is easily shown to be complete, by repeated applications of the normal expansion theorem. Given that $\mathcal{M}$ is complete, completeness of RADD is another straightforward application of the normal extension theorem.

Real numbers may be thought of as converging sequences $r_0, r_1,\ldots,r_n,\ldots$ of rational numbers. In a concrete case, and for an irrational number x, such a sequence will represent a process of calculation producing ever closer approximations of x. One uses these rational approximations in numerical operations with x. Now it may be supposed that the closer the approximation, the more laborious is the calculation that leads to it. So one tries to strike a balance between feasibility and precision. In order to estimate the latter, one needs a modulus m associated with x and the process $(r_n)_{n\in\mathbb{N}}$, which gives for every $n\in\mathbb{N}$ a number $m(n)$ such that for all $k > n$: $|r_n - r_k| \leq m(n)$; and one must know how errors propagate under the various operations performed on the approximating rational. Of course, complex numbers are approximated in analogous fashion by complex rationals.

Accordingly, we shall specify irrational numbers as functions F from N to Q; and in each case we shall declare an unspecified constant in N, and stipulate that $F(N)$ is the approximation to be used. The important matter of errors and their propagation we leave aside.

Thus, real numbers are actually a particular way of operating with rationals. Similarly we approach the complex numbers through the complex rationals.

```
module    CORT                          --complex rationals
begin
  exports
    begin
      sorts   C
      functions    (_,_): Q × Q → C
                   _+_: C × C → C
                   _·_: C × C → C
                   -_: C → C
                   _*: C → C
                   C_: Q → C
                   i: → C
                   0: → C
                   1: → C
    end
  imports RADD
  variables
    x, y, u, v: → Q
  equations
  [1] (x,y) + (u,v) = (x+u, y+v)
  [2] (x,y) · (u,v) = ( x·u - y·v, x·v + y·u )
  [3] -(x, y) = (-x, -y)
  [4] (x, y)* = (x, -y)       --conjugate
  [5] i = (0,+[Z])
  [6] 0 = (0,0)
  [7] 1 = (+[Z],0)
  [8] Cx = (x,0)
end CORT
```

```
module POW                              --the exponential function
begin
  parameters  M
    begin
      functions  N: → N
    end M
  exports
    begin
      functions  E_: C → C
    end
  imports CORT
  functions    E: C × N × N → C
  variables
    x: →C
    m, n: →N
  equations
  [1] Ex = E(x, N, N)
  [2] E(x, m, 0) = 1 + x·C+(Jm;[Z])
  [3] E(x, m, Sn) = E(x, m, n) · E(x, m, n)
end POW
```

POW specifies e^x, for $x \in \mathbb{C}$, to be the sequence $\left(1 + \frac{x}{2^n}\right)^{2^n}$. The actual element of this sequence that will be used is determined by the parameter N. $E(x, m, n)$ is $\left(1 + \frac{x}{2^m}\right)^{2^n}$ (recall that J: $n \mapsto -n$ embeds the natural numbers into the integers). Indeed, [2] says, in ordinary notation, that $E(x, m, 0) = 1 + \frac{x}{2^m}$, and [3] doubles the exponent for the next value of n.

```
module ARCTAN                                    --arcus tangens
begin
  parameters  M'
    begin
      functions    M: → N
    end M'
  exports
    begin
      functions    A_: Q → Q
    end
  imports RADD
  functions    T0: Q × N → Q
               T1: Q × N → Q
               A: Q × N → Q
  variables
    x: →Q
    n: →N
  equations
  [1] T0(x, 0) = x
  [2] T0(x, Sn) = -x·x · T0(x, n)
  [3] T1(x, n) = T0(x, n) · +JFPS(n + n)
  [4] A(x, 0) = x
  [5] A(x, Sn) = A(x, n) + T1(x, Sn)
  [6] Ax = A(x, M)
end ARCTAN
```

ARCTAN specifies arctan x as the series $\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{2n+1}$. Recall that P converts natural numbers into positive integers, which are factorized by F, the result of which is a list of natural numbers; and J turns a list $\langle n_0,\ldots,n_{k-1}\rangle$ of natural numbers into the list $\langle -n_0,\ldots,-n_{k-1}\rangle$ of integers. $T_0(x, n) = (-1)^n x^{2n+1}$, $T_1(x, n)$ is the nth term of the series, and $A(x, n)$ is the nth partial sum. M is the natural number parameter which determines how far the series is to be developed.

```
module PI                                   --calculation of π
begin
  exports
    begin
      functions    π: → Q
    end
  imports ARCTAN
  equations
  [π] π = +(+S1;[Z]) · (+(+S1;[Z])·A(+(-1;0;0;[Z])) -
                  A(+(-1;0;0;0;0;0;0;0;0;0;0;0;0;
                       0;0;0;0;0;0;0;0;0;0;0;0;0;
                       0;0;0;0;0;0;0;0;0;0;0;0;0;
                       0;0;0;0;0;0;0;0;0;0;0;0;0;[Z])))
end PI
```

The sequence of prime numbers begins with

2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239.

So 239 is the fifty-second prime; and PI specifies π as $4(4\arctan\frac{1}{5} - \arctan\frac{1}{239})$.

```
module RUN                                  --complex roots of unity
begin
  exports
    begin
      functions    R_: N → C
    end
  imports PI, POW
  variables
    n: →N
  equations
  [R] Rn = E(𝕆, π·+((+1-In);[Z]))
end
```

In the notation of §5, $Rn = W_n(1) = e^{i\pi 2^{1-n}}$. The function *I*, introduced in INT, embeds the natural numbers into the integers.

```
module SCAM                                  --scalar multiplication
begin
  exports
    begin
      functions _·_: C × LC → LC
    end
  imports CORT, SIM {M bound by [X → C] to CORT, renamed by
                        [LX → LC], [[X] → [C]]}
  variables
    x, y: →C
    a: →LC
  equations
  [1] x·[C] = [C]
  [2] x·(y;a) = (x·y);(x·a)
end SCAM


module MERGE                                 --merging lists
begin
  exports
    begin
      functions    _‖_: LX × LX → LX
    end
  imports SIM
  variables
    x, y: →X
    a, b: →LX
  equations
  [1] a ‖ [X] = [X]
  [2] [X] ‖ a = [X]
  [3] (x;a) ‖ (y;b) = x;(y;(a‖b))
end MERGE
```

```
module LRUN                          --lists of roots of unity
begin
  exports
    begin
      functions    W_: N → LC
                   V_: N → LC
    end
  imports RUN, SCAM, LITR {M' bound by [X → C], [Y → C], [I → _*]
                              to RUN, renamed by [LX → LC],
                              [[X] → [C]], [LY → LC], [[Y] → [C]],
                              [I → _*]},
          MERGE {M bound by [X → C] to RUN, renamed by [LX → LC],
                  [[X] → [C]]}
  variables
    n: →N
  equations
  [1] W0 = 1;[C]
  [2] WSn = Wn ‖ (RSn·Wn)
  [3] Vn = (Wn)*
end LRUN
```

$W(n)$ is the list of coefficients of the polynomial w_n in §5. $V(n)$ consists of the conjugates of the elements of$W(n)$; it is used analogously, to calculate G_n. $W(n+1)$ is obtained from $W(n)$ by

$$W(n+1) = \langle W_n(0),\ W_{n+1}(1)\cdot W_n(0),\ W_n(1), W_{n+1}(1)\cdot W_n(1),\ \ldots\rangle.$$

```
module    SPLIT          --splitting lists into odd and even parts
begin
  exports
    begin
      functions  E_: LX → LX
                 O_: LX → LX
    end
  imports SIM
  variables
    x: →X
    a: →LX
  equations
  [1] E[X] = [X]
  [2] O[X] = [X]
  [3] E(x;a) = x;Oa
  [4] O(x;a) = Ea
end SPLIT
```

```
module DIM                                --diagonal multiplication
begin
  parameters  M'
    begin
      sorts X
      functions  _·_: X × X → X
    end M'
  exports
    begin
      functions  _∧_: LX × LX → LX
    end
  imports SIM {M bound to M'}
  variables
    x, y: →X
    a, b: →LX
  equations
  [1] a ∧ [X] = [X]
  [2] [X] ∧ a = [X]
  [3] (x;a) ∧ (y;b) = (x·y);(a∧b)
end DIM
```

```
module DFT                              --discrete Fourier transform
begin
  exports
    begin
      functions  F: N × LC → LC
                 G: N × LC → LC
    end
  imports LRUN, SPLIT {M bound by [X → C] to LRUN, renamed by
                          [LX → LC], [[X] → [C]]},
          LADD {M' bound by [X → C] to LRUN, renamed by
                 [LX → LC], [[X] → [C]]},
          DIM {M' bound by [X → C]to LRUN, renamed by [LX → LC],
                 [[X] → [C]]},
          LITR {M' bound by [X → C], [Y → C], [I → -_] to RUN,
                 renamed by [LX → LC], [[X] → [C]], [LY → LC],
                 [[Y] → [C]], [I → -_]},
          REV {M bound by [X → C] to RUN, renamed by [LX → LC],
                 [[X] → [C]]}
  variables
    a: →LC
    n: →N
  equations
  [1] F(0, a) = a
  [2] F(Sn, a) = (F(n, Ea);F(n, Ea)) +
                              ((WSn ∧ F(n, Oa));(-WSn ∧ F(n, Oa)))
  [3] G(0, a) = a
  [4] G(Sn, a) = (G(n, Ea);G(n, Ea)) +
                              ((VSn ∧ G(n, Oa));(-VSn ∧ G(n, Oa)))
end DFT
```

```
module    BIN            --natural numbers in binary notation
begin
  exports
    begin
      sorts    ℕ
      functions  #: → ℕ
                 _0: ℕ → ℕ
                 _1: ℕ → ℕ
                 _+_: ℕ × ℕ → ℕ
    end
  variables
    m, n: →ℕ
  equations
  [1] #0 = #
  [2] # + n = n
  [3] n + # = n
  [4] m0 + n0 = (m+n)0
  [5] m0 + n1 = (m+n)1
  [6] m1 + n0 = (m+n)1
  [7] m1 + n1 = ((m+n)+#1)0
end BIN
```

The representation of polynomials as lists of coefficients is not quite correct, since distinct lists of complex numbers may correspond with the same polynomial. For example, ‹0, 0› ≠ ‹0›, whereas $0 + 0 \cdot X = 0$. We now present the true polynomials.

```
module    POL                                          --polynomials
begin
  exports
    begin
      sorts Pol
      functions  P_: LC → Pol
                 E: → Pol
                 _;_: C × Pol → Pol
                 L_: Pol → LC
    end
  imports CORT, SIM {M bound by [X → C] to CORT, renamed by
                        [LX → LC], [[X] → [C]]}
  functions K: LC → LC
  variables
    x, y: →C
    a: →LC
    p: →LZ
    q: →Q
    f: →Pol
  equations
  [1] P[C] = E
  [2] P(x;a) = x ; Pa
  [3] (𝟘,𝟘) ; E = E
  [4] LE = [C]
  [5] L(x;f) = K(x;Lf)
  [6] K((𝟘,𝟘);[C]) = [C]
  [7] K((+p,q);[C]) = (+p,q);[C]
  [8] K((-p,q);[C]) = (-p,q);[C]
  [9] K((𝟘,+p);[C]) = (𝟘,+p);[C]
  [10] K((𝟘,-p);[C]) = (𝟘,-p);[C]
  [11] K(x;(y;a)) = (x;(y;a))
end POL
```

The auxiliary function *K*, and the case distinctions [6]-[11] are in order to ensure regularity.

```
module    DEG                    --degree in binary notation
begin
  exports
    begin
      functions  D_: Pol → ℕ
    end
  imports POL, BIN
  functions   D0: LC → ℕ
  variables
    x, y: →C
    f: →Pol
    a: →LC
  equations
  [1] Df = D0(Lf)
  [2] D0([C]) = #
  [3] D0(x;[C]) = #
  [4] D0(x;(y;a)) = D0(y;a) + #1
end DEG
```

If $f = x_0 \cdot X^0 + \ldots + x_{n-1} \cdot X^{n-1}$, then $D(f)$ is the least number $i \geq 0$ such that $x_i \neq 0$.

```
module         UL              --unary length of a binary numeral
begin
  exports
    begin
      functions  L_: ℕ → N
    end
  imports NAT, BIN
  functions   K_: N → N
  variables
    m: →ℕ
    n: →N
  equations
  [1] L(#) = 0
  [2] L(m0) = KLm
  [3] L(m1) = SLm
  [4] K0 = 0
  [5] KSn = SSn
end UL
```

```
module     MULP                    --multiplication for polynomials
begin
  exports
    begin
      functions  _·_: Pol × Pol → Pol
    end
  imports DFT, POL, DEG, UL
  variables
    f, g: →Pol
  equations
  [M] f·g   = P(C+(JL(Df+Dg);[Z]) ·
                G(L(Df+Dg), F(L(Df+Dg), Lf) ∧ F(L(Df+Dg), Lg)))
end MULP
```

If f and g are polynomials, neither identically 0, then $D(f) + D(g) = D(f{\cdot}g)$. We ignore the shortcut that is possible when $f{\cdot}g = 0$. If n represents the natural number n in BIN, and $k = \mu m.2^m > n$, then `L(`n`)` normalizes to $\mathtt{S}^k\mathtt{0}$.

9.Parallelism

The fundamental reason why TRSes invite parallel implementation is that long terms may contain many redexes, which can be treated in any order — or all at once. Some qualifications may be necessary, but these will be rather natural (e.g. one might want to require confluency); and with complete TRSes such as the above, there is not much that can go wrong.

However, employing a large number of calculators does not guarantee a quick job. The effort will need to be organized, and the resulting bureaucracy may become unwieldy. One of the things that will help to prevent this, is: to make reasonably sure that the jobs being distributed are not small, and can be done fairly independently. There are two levels in the implementation of a TRS on which one can try to distinguish such jobs. The lower level is that of the implementation of term rewriting, and concerns the recognition of redexes, singling out redexes to be rewritten, and the actual reduction process. We shall not deal with it here. On a higher level, there may be possibilities for such "coarse-grained" parallelism in the algorithms being specified, which carry over to the specification. We list a few for the specification above.

(DFT) The calculation of $F_{n+1}(a)$ may be split into calculations of $F_n(Odd\,(a))$ and $F_n(Even\,(a))$ (repeatedly); the same holds for G_{n+1}.

(DIM) Diagonal multiplications can be split into separate multiplications of elements of the lists, in particular in MULP and DFT.

(SCAM) A multiplication $x{\cdot}y$ of a scalar with a list may be split into multiplications of the scalar with elements of the list; in particular in LRUN and MULP.

(ARCTAN) Terms of a series may be calculated in parallel (this is relevant to PI).

(CORT) A complex addition or multiplication produces two rational additions, which may be worth executing in parallel (two is not much, of course).

(RED) `R(p,q)` produces a list of terms $@_i(y \oplus z)$, which may sometimes be worth handling in parallel.

(LADD) List additions reduce to additions of elements: in REX and DFT (in the latter the jobs will more likely be worth distributing).

(MUL) Exponentiating may be done in parallel.

10. Completeness and semicompleteness

We have not particularly strived for completeness; rather we tried to get a TRS that worked, and found that we might as well have a complete one. We *were* careful to avoid overlapping redex patterns; but then, owing to the size of the system, the ambiguities that we tried led to perplexity and loss of confluency.

At first sight, the necessity of proving (semi-)completeness may seem a burden. As may be seen above, however, the proofs mainly consist in checking that one has specified what one set out to specify — which is not generally a matter of course for any formal specification method.

ACKNOWLEDGEMENTS

Canonical models as semantics for equational specifications were proposed in [BK]. In writing the specification above we profited by [V]. We thank F.-J. de Vries and F. Wiedijk for their comments.

REFERENCES

[BHK] J.A. BERGSTRA, J. HEERING & P. KLINT, ASF-An Algebraic Specification formalism. Report CS-R8705, Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[BK] J.A. BERGSTRA & J.W. KLOP, De Semicomplete TRS. Kluwer, 1987.

[BT] J.A. BERGSTRA & J.V. TUCKER, Algebraic Specifications of Computable and Semicomputable Datatypes. Report CS-R8619, Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[CT] J.W. COOLEY & J.W. TUKEY, An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation XIX (1965), 297-301.

[HR] D.J. HOEKZEMA & P.H. RODENBURG, Gaußian Elimination as a Term Rewriting System. To appear in the Logic Group Preprint Series, Department of Philosophy, University of Utrecht.

[K] J.W. KLOP, Term Rewriting Systems: a Tutorial. Bulletin of the European Association for Theoretical Computer Science, June 1987.

[V] F.-J. DE VRIES, A Functional Program for the Fast Fourier Transform. Logic Group Preprint Series No. 19, Department of Philosophy, University of Utrecht, March 1987.

Logic Group Preprint Series
Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

nr. 1 C.P.J. Koymans, J.L.M. Vrancken, *Extending Process Algebra with the empty process*, September 1985.
nr. 2 J.A. Bergstra, *A process creation mechanism in Process Algebra*, September 1985.
nr. 3 J.A. Bergstra, *Put and get, primitives for synchronous unreliable message passing*, October 1985.
nr. 4 A.Visser, *Evaluation, provably deductive equivalence in Heyting's arithmetic of substitution instances of propositional formulas*, November 1985.
nr. 5 G.R. Renardel de Lavalette, *Interpolation in a fragment of intuitionistic propositional logic*, January 1986.
nr. 6 C.P.J. Koymans, J.C. Mulder, *A modular approach to protocol verification using Process Algebra* , April 1986.
nr. 7 D. van Dalen, F.J. de Vries, *Intuitionistic free abelian groups*, April 1986.
nr.8 F. Voorbraak, *A simplification of the completeness proofs for Guaspari and Solovay's R*, May 1986.
nr.9 H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, *A semantic framework for the COLD-family of languages*, May 1986.
nr.10 G.R. Renardel de Lavalette, *Strictheidsanalyse*, mei 1986.
nr.11 A. Visser, *Kunnen wij elke machine verslaan? Beschouwingen rondom Lucas' argument*, Juli 1986.
nr.12 E.C.W. Krabbe, *Naess's dichotomy of tenability and relevance*, June 1986.
nr.13 Hans van Ditmarsch, *Abstractie in wiskunde, expertsystemen en argumentatie*, Augustus 1986
nr.14 A.Visser, *Peano's Smart Children, a provability logical study of systems with built-in consistency* , October 1986.
nr.15 G.R.Renardel de Lavalette, *Interpolation in natural fragments of intuitionistic propositional logic*, October 1986.
nr.16 J.A. Bergstra, *Module Algebra for relational specifications*, November 1986.
nr.17 F.P.J.M. Voorbraak, *Tensed Intuitionistic Logic*, January 1987.
nr.18 J.A. Bergstra, J. Tiuryn, *Process Algebra semantics for queues*, January 1987.
nr.19 F.J. de Vries, *A functional program for the fast Fourier transform*, March 1987.
nr.20 A. Visser, *A course in bimodal provability logic*, May 1987.
nr.21 F.P.J.M. Voorbraak, *The logic of actual obligation, an alternative approach to deontic logic*, May 1987.
nr.22 E.C.W. Krabbe, *Creative reasoning in formal discussion*, June 1987.
nr.23 F.J. de Vries, *A functional program for Gaussian elimination*, September 1987.
nr.24 G.R. Renardel de Lavalette, *Interpolation in fragments of intuitionistic propositional logic*, October 1987.(revised version of no. 15)
nr.25 F.J. de Vries, *Applications of constructive logic to sheaf constructions in toposes*, October 1987.
nr.26 F.P.J.M. Voorbraak, *Redeneren met onzekerheid in expertsystemen*, November, 1987.
nr.27 P.H. Rodenburg, D.J. Hoekzema, *Specification of the fast Fourier transform algorithm as a term rewriting system*, December, 1987.