A PSF library of data types

S. Mauw* J. C. Mulder*

*Dept. of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email: {sjouke,hansm}@win.tue.nl

Abstract. We present a library of basic data types for the process specification formalism PSF. The specification is written in the algebraic specification formalism ASF, which is a sublanguage of PSF.

1 Introduction

PSF (Procees Specification Formalism, [MV93]) is a language for the formal specification of distributed systems. The dynamic part is based on the Algebra of Communicating Processes (ACP [BW90, BK84]) and the static part on the Algebraic Specification Formalism (ASF [BHK89]). In this paper we will only discuss the static part.

The elegance of an algebraic specifiation formalism such as ASF partly comes from the fact that, in contrast to modern programming languages, it has no built in data types. This leads to a semantics that is easy to understand, but also to a quite cumbersome specification style. Users have to give their own definitions of, say, Booleans and Natural numbers, which may distract the attention from the actual target of specification. Since the ASF language allows for reuse of specifications by means of modularisation and parameterisation constructs, it is possible to collect frequently used primitive data types into a library.

The library specification in a language such as ASF is at the same level as the target specification. This has an advantage over the case where a library is defined on a lower level, which is the case for, e.g. COBOL, namely, the semantics of the library are defined by the semantics of the specification language. Furthermore the existing tools don't need to be extended since the library is defined within the specification language. A drawback is that the implementation of the library data types by means of the existing tools for term rewriting may not be as efficient as a tailored implementation.

There exist several libraries for algebraic specification languages. We mention the LOTOS library [ISO89], the SDL library [IT88] and the library from [Mos92]. The first two libraries can be considered subsets of our library (neglecting the language specific data types in the SDL library). The library from [Mos92] is about as extensive as ours, but has a rather different goal. The language used there has a rich syntax and semantics allowing for short and expressive specifications. By contrast, PSF is rather spartan. Under the restriction that our specification must be algebraically correct, we tried to make a specification as efficient as possible. This is not the target of [Mos92], in which readability was the primary goal.

Experience has shown that it is difficult to write a library which satisfies the needs of all users. An example of a particularly hard problem is to provide a sufficiently flexible error handling mechanism. Even if one comes close to a complete library, the so-called *Not Invented Here* syndrome causes people to write their own basic data type specifications anyway.

For PSF two attempts for a general library of basic data types are in [MV93] and [vW93]. The first only contains the modules that are shared by the specifications in [MV93]. The second is a more serious attempt to make a general library. Both are currently being distributed with the PSF-Toolkit [Vel93]. The library from [vW93]

Proceedings of the ASF+SDF95 workshop on generating tools from algebraic specifications, May 11 & 12, 1995, CWI, Amsterdam

has several shortcomings. First of all, due to its size, compilation takes quite some time. Furthermore its performance is ratheru slow since the algorithms used were not optimal with respect to term rewriting. Next, the interfaces of the data types are not completely uniform and some useful data types were missing. Finally it contains some errors.

The library presented in this paper is adapted from the library in [vW93] where we resolved the above mentioned shortcomings. We don't think this is the final PSF library. Practical experience has to point out new shortcomings.

The library consists of the following data types: Booleans, Sequences, Tables, Naturals (decimal, binary and unary), Integers (decimal), Floats (decimal), Characters (ASCII), Strings and Finite sorts with equality.

We considered including Cartesian product types (also known as records). The language would be marginally easier to use if this construct were built in. As it is, we can simulate products using parameterized modules, but binding such a module is almost as much work as creating a product type from scratch. Consequently, we decided to leave them out of the library.

Our aim was to write a library within the existing PSF language, restricted by the possibilities of the current term rewriting tool. Thus, we do not add special features to the language, although it would certainly be more efficient to solve some of the problems by changing the language or the tools. Therefore, we consider so-called hybrid implementations as cheating. Furthermore, we adhere to the initial algebra semantics, so we do not have special features, such as partial functions, negative conditions and ordered equations, which would have resulted in more efficient and concise specifications. The consequence of not having partial functions is that we have to make all functions total by choosing default outcomes. For example, we chose to specify $x/0 = 0$.

Furthermore, we have the rigid syntax of ASF, which is less flexible than the formalism ASF+SDF allows for. The natural number 42 is, e.g. represented by the expression `nat(^4^2)`. Since one of our objectives was to make an executable specification, we did not include abstract data types such as sets. Nevertheless, we do provide a module called Quasi-Sets, which defines set operations on sets implemented as lists.

The specification can be interpreted as a term rewriting system (TRS) by ordering the equations from left to right. The TRS obtained in this way is complete. We will not give a proof of this property.

The complete PSF-Toolkit, including the new library can be obtained by anonymous ftp from ftp://ftp.win.tue.nl/pub/psf/psf.0.9.5.tar.Z.

2 Overview of the library

In order to avoid summing up lengthy specifications, we will not present the complete library here. A typical user of the library will only need the visible signatures of the relevant data types plus an intuition of the meaning. This is the interface between his specification and the library specification.

For each data type from the library we will informally explain its meaning, give the exported signature, describe the normal forms and give some examples. Notice that for reasons of conciseness, we do not adhere strictly to the PSF syntax. For instance, we will contract the declarations of two functions if they have the same type.

2.1 Booleans

This module defines the standard Boolean values and functions.

module: `Booleans`

sort: `BOOLEAN`

functions:

```
        true, false : -> BOOLEAN
        not : BOOLEAN -> BOOLEAN
        eq, ne, and, or, xor : BOOLEAN # BOOLEAN -> BOOLEAN
```

typical normal form: `true`

example: `or(true, false) = true`

2.2 Sequences

This first two modules below define two flavours of sequences, with or without the empty sequence. Both modules are parameterized with the type of the elements of the sequence. The disadvantage of having an empty sequence is that you need an error element in that type. Sequences are built from the constructors `empty-sequence` and infix ^; non-empty sequences from prefix ^ and infix ^.

Quasi-sets are actually sequences on which we defined some typical set operators.

module: `Sequences`

parameter: `Elements`
sort: `ITEM`
functions:

```
        eq : ITEM # ITEM -> BOOLEAN
        error-element : -> ITEM
```

sort: `SEQ`

functions:

```
        empty-sequence : -> SEQ
        _^_ : SEQ # ITEM -> SEQ
        ^_ : ITEM -> SEQ
        first, last : SEQ -> ITEM
        tail, reverse : SEQ -> SEQ
        concat : SEQ # SEQ -> SEQ
        eq, ne : SEQ # SEQ -> BOOLEAN
        elem : ITEM # SEQ -> BOOLEAN
```

with `Elements` bound to `Booleans`:
typical normal form: `(empty-sequence ^ true) ^ false`

example: `elem(false, ^ true ^ false) = true` module: `Non-Empty-Sequences`

parameter: `Elements`
sort: `ITEM`
function:
```
        eq : ITEM # ITEM -> BOOLEAN
```

sort: `SEQ`

functions:
```
        ^_ : ITEM -> SEQ
        _^_ : SEQ # ITEM -> SEQ
        first, last : SEQ -> ITEM
        tail, reverse : SEQ -> SEQ
        concat : SEQ # SEQ -> SEQ
        eq, ne : SEQ # SEQ -> BOOLEAN
        elem : ITEM # SEQ -> BOOLEAN
```

with `Elements` bound to `Booleans`:
typical normal form: `^(true) ^ false`

example: `elem(false, ^ true ^ false) = true`

module: `Quasi-sets`

like `Sequences`, except:
functions:
```
        insert, delete : ITEM # SEQ -> SEQ
        is-member : ITEM # SEQ -> BOOLEAN
        is-subset, equal, not-equal : SEQ # SEQ -> BOOLEAN
        union, intersection, difference : SEQ # SEQ -> SEQ
```

with `Elements` bound to `Booleans`:
typical normal form: `^(true) ^ false`

example: `intersection(^ true, ^false) = empty-sequence`

2.3 Tables

Lookup tables are defined in the next two modules. The parameters `Keys` and `Items` are separate, so we can bind them to two different modules. The module `Tables` uses a linear list of key-value pairs, while the module `Binary-Trees` uses a more efficient structure.

module: `Tables`

parameter: `Keys`

sort: `KEY`
function:
```
        eq : KEY # KEY -> BOOLEAN
```
parameter: `Items`
sort: `ITEM`

sorts: `TABLE, ENTRY, RESULT-OF-LOOKUP`

functions:
```
        empty-table : -> TABLE
        _~_ : ENTRY # TABLE -> TABLE
        entry : KEY # ITEM -> ENTRY
        insert : TABLE # KEY # ITEM -> TABLE
        delete : TABLE # KEY -> TABLE
        lookup : TABLE # KEY -> RESULT-OF-LOOKUP
        found : ITEM -> RESULT-OF-LOOKUP
        not-found : -> RESULT-OF-LOOKUP
```

with `Keys and Items` bound to `Booleans`:
typical normal form: `entry(false, true) ~ empty-table`

examples:
```
        insert(empty-table, true, false)
            = entry(false, true) ~ empty-table
        lookup(entry(false, true) ~ empty-table, false)
            = found(true)
```

module: `Binary-Trees`

parameter: `Keys`
sort: `KEY`
functions:
```
        eq, lt : KEY # KEY -> BOOLEAN
```
parameter: `Items`
sort: `ITEM`

sorts: `TABLE, ENTRY, RESULT-OF-LOOKUP`

functions:
```
        empty-table : -> TABLE
        node : ENTRY # TABLE # TABLE -> TABLE
        entry : KEY # ITEM -> ENTRY
        insert : TABLE # KEY # ITEM -> TABLE
        delete : TABLE # KEY -> TABLE
        lookup : TABLE # KEY -> RESULT-OF-LOOKUP
        found : ITEM -> RESULT-OF-LOOKUP
        not-found : -> RESULT-OF-LOOKUP
```

with `Keys and Items` bound to `Booleans`:
typical normal form:
```
        node(entry(false, true), empty-table, empty-table)
```

example:

```
        insert(empty-table, false, true)
            = node(entry(false, true),
                 empty-table, empty-table)
        lookup(node(entry(false, true),
                 empty-table, empty-table), false)
            = found(true)
```

2.4 Arithmetic

2.4.1 Decimal naturals

The natural numbers are implemented as sequences of decimal digits. We have to introduce the digits in a separate module so that we can bind the parameter of the non-empty sequences to it.

The module `Naturals` has a lot of sub-modules which we do not describe here. For convenience we provide abbreviations for the first 101 numbers.

module: `Digits`

sort: `DIGIT`

functions:

```
        0, 1, ..., 9 : -> DIGIT
```

typical normal form: `7`

module: `Naturals`

sort: `NATURAL`

functions:

```
        nat : DIGIT-SEQUENCE -> NATURAL
        nat0, nat1, nat2, ...., nat100 : -> NATURAL
        succ, pred : NATURAL -> NATURAL
        _+_, _-_, _*_, _/_, mod, _**_ : NATURAL # NATURAL -> NATURAL
        eq, ne, gt, ge, lt, le : NATURAL # NATURAL -> BOOLEAN
        if : BOOLEAN # NATURAL # NATURAL -> NATURAL
        min, max : NATURAL # NATURAL -> NATURAL
```

typical normal form: `nat(^(4) ^ 2)`

example: `nat3 + nat4 = nat(^(7))`

2.4.2 Binary naturals

The binary naturals are similar to the decimal naturals.

module: `Bits`

sort: `BIT`

functions:
```
        0b, 1b : -> BIT
```

typical normal form: `1b`

module: `Binary-Naturals`

sort: `BINARY-NATURAL`

functions:
```
        bin : BIT-SEQUENCE -> BINARY-NATURAL
        bin0, bin1, bin10, ...., bin1000000 : -> BINARY-NATURAL
        succ, pred : BINARY-NATURAL -> BINARY-NATURAL
        _+_, _-_, _*_, _/_, mod, _**_ :
                    BINARY-NATURAL # BINARY-NATURAL -> BINARY-NATURAL
        eq, ne, gt, ge, lt, le :
                    BINARY-NATURAL # BINARY-NATURAL -> BOOLEAN
        if : BOOLEAN # BINARY-NATURAL # BINARY-NATURAL
                    -> BINARY-NATURAL
        min, max : BINARY-NATURAL # BINARY-NATURAL -> BINARY-NATURAL
```

typical normal form: `bin(((((^(1b) ^ 0b) ^ 1b) ^ 0b) ^ 1b) ^ 0b)`

example: `bin11 + bin100 = bin((^(1b) ^ 1b) ^ 1b)`

2.4.3 Unary naturals

For completeness we also provide natural numbers in the usual `succ(zero)` notation, which is admittedly not very efficient.

module: `Unary-Naturals`

sort: `UNARY-NATURAL`

functions:
```
        zero : -> UNARY-NATURAL
        unary0, unary1, ..., unary10 : -> UNARY-NATURAL
        succ, pred : UNARY-NATURAL -> UNARY-NATURAL
        _+_, _-_, _*_, _/_, mod, _**_ :
                    UNARY-NATURAL # UNARY-NATURAL -> UNARY-NATURAL
        eq, ne, gt, ge, lt, le :
                    UNARY-NATURAL # UNARY-NATURAL -> BOOLEAN
        if : BOOLEAN # UNARY-NATURAL # UNARY-NATURAL -> UNARY-NATURAL
        min, max : UNARY-NATURAL # UNARY-NATURAL -> UNARY-NATURAL
```

typical normal form: `succ(succ(succ(succ(succ(succ(succ(zero)))))))`

example:
```
        unary3 + unary4
        = succ(succ(succ(succ(succ(succ(succ(zero)))))))
```

2.5 Integers

An integer is a pair of a sign and a natural number. For convenience we provide a series of mixed mode arithmatic operators. The constructor for integers is the function `int : SIGN # DIGIT-SEQUENCE -> INTEGER`, however this function is hidden for reasons of efficiency. The user may call the function `make-int` which rewrites to a normalized integer.

module: `Signs`

sort: `SIGN`

functions:

```
        pos, neg : -> SIGN
```

typical normal form: `pos`

module: `Integers`

sort: `INTEGER`

functions:

```
        make-int: SIGN # DIGIT-SEQ -> INTEGER
        nat-to-int: NATURAL -> INTEGER
        int-to-nat: INTEGER -> NATURAL
        sign : INTEGER -> SIGN

        _-, abs, succ, pred : INTEGER -> INTEGER
        _+_, _-_, _*_, _/_, mod : INTEGER # INTEGER -> INTEGER
        eq, ne, gt, ge, lt, le : INTEGER # INTEGER -> BOOLEAN
        if : BOOLEAN # INTEGER # INTEGER -> INTEGER
        min, max : INTEGER # INTEGER -> INTEGER

        _- : NATURAL -> INTEGER
        _+_, _-_, _*_, _/_, mod : NATURAL # INTEGER -> INTEGER
        eq, ne, gt, ge, lt, le : NATURAL # INTEGER -> BOOLEAN
        if : BOOLEAN # NATURAL # INTEGER -> INTEGER
        min, max : NATURAL # INTEGER -> INTEGER

           _+_, _-_, _*_, _/_, mod : INTEGER # NATURAL -> INTEGER
        eq, ne, gt, ge, lt, le : INTEGER # NATURAL -> BOOLEAN
        if : BOOLEAN # INTEGER # NATURAL -> INTEGER
        min, max : INTEGER # NATURAL -> INTEGER
```

typical normal form: `int(pos, nat(^(4) ^ 2))`

example: `-nat6 * -nat7 = int(pos, nat(^(4) ^ 2))`

2.6 Floating point arithmetic

The module `Floating-Points-M-P` is parameterized with two natural numbers. The first number (`M`) is the precision used for normal forms. The second number (`P`) denotes the precision used while computing. Shown here is the binding `[M->3, P->4]`.

The normal forms of the sort `FP` are constructed with the hidden function `fp : INTEGER # INTEGER -> FP`. The user has access to a wide range of `make-fp` functions which accept various combinations of argument types. To analyze a floating point number one may use the two projection functions `mantissa` and `exponent`.

module: `Floating-Points-3-4`

sort: `FP`

functions:

```
        abs : FP -> FP
        sign : FP -> SIGN
        mantissa : FP -> INTEGER
        exponent : FP -> INTEGER

        _+_, _-_, _*_, _/_ : FP # FP -> FP
        eq, ne, gt, ge, lt, le : FP # FP -> BOOLEAN
        if : BOOLEAN # FP # FP -> FP
        min, max : FP # FP -> FP

        _+_, _-_, _*_, _/_ : INTEGER # FP -> FP
        eq, ne, gt, ge, lt, le : INTEGER # FP -> BOOLEAN
        if : BOOLEAN # INTEGER # FP -> FP
        min, max : INTEGER # FP -> FP

        _+_, _-_, _*_, _/_ : FP # INTEGER -> FP
        eq, ne, gt, ge, lt, le : FP # INTEGER -> BOOLEAN
        if : BOOLEAN # FP # INTEGER -> FP
        min, max : FP # INTEGER -> FP

        _+_, _-_, _*_, _/_ : NATURAL # FP -> FP
        eq, ne, gt, ge, lt, le : NATURAL # FP -> BOOLEAN
        if : BOOLEAN # NATURAL # FP -> FP
        min, max : NATURAL # FP -> FP

        _+_, _-_, _*_, _/_ : FP # NATURAL -> FP
        eq, ne, gt, ge, lt, le : FP # NATURAL -> BOOLEAN
        if : BOOLEAN # FP # NATURAL -> FP
        min, max : FP # NATURAL -> FP

        make-fp        : SIGN # DIGIT-SEQ # DIGIT-SEQ # INTEGER -> FP
        make-fp        : SIGN # DIGIT-SEQ # DIGIT-SEQ           -> FP
        make-fp        : SIGN # DIGIT-SEQ             # INTEGER -> FP
        make-fp        : SIGN # DIGIT-SEQ                       -> FP
        make-fp        :        DIGIT-SEQ # DIGIT-SEQ # INTEGER -> FP
        make-fp        :        DIGIT-SEQ # DIGIT-SEQ           -> FP
        make-fp        :        DIGIT-SEQ             # INTEGER -> FP
        make-fp        :        DIGIT-SEQ                      -> FP
        make-fp        : SIGN # NATURAL              # INTEGER -> FP
        make-fp        : SIGN # NATURAL                        -> FP
        make-fp        :        NATURAL              # INTEGER -> FP
        make-fp        :        NATURAL                        -> FP
        make-fp        :        INTEGER              # INTEGER -> FP
        make-fp        :        INTEGER                        -> FP
```

typical normal form: `fp(int(pos, nat(^(4) ^ 2)), int(pos, nat(^(0))))`

examples:

```
        make-fp(^3) + make-fp(^4)
            = fp(int(pos, nat(^(7))), int(pos, nat(^(0))))

        make-fp(nat2) / make-fp(nat3)
            = fp(int(pos, nat((^(6) ^ 6) ^ 7)), int(neg, nat(^(3))))
```

2.7 Characters and strings

The ASCII characters and non-empty sequences of the same are described in two rather simple modules.

module: `Characters`

sort: `CHARACTER`

functions:

```
        'nul', 'soh', 'stx', 'etx', 'eot', 'enq', 'ack',
        'bel', 'bs',  'ht',  'nl',  'vt',  'ff',  'cr',
        'so',  'si',  'dle', 'dc1', 'dc2', 'dc3', 'dc4',
        'nak', 'syn', 'etb', 'can', 'em',  'sub', 'esc',
        'fs',  'gs',  'rs',  'us',  'spa', 'exc', 'dqu',
        'hsh', 'dlr', 'prc', 'amp', 'squ', 'lpa', 'rpa',
        'ast', 'pls', 'cma', '-',   'dot', 'fsl', '0',
        '1',   '2',   '3',   '4',   '5',   '6',   '7',
        '8',   '9',   'col', 'sco', 'lth', 'eql', 'grt',
        'que', 'ats', 'A',   'B',   'C',   'D',   'E',
        'F',   'G',   'H',   'I',   'J',   'K',   'L',
        'M',   'N',   'O',   'P',   'Q',   'R',   'S',
        'T',   'U',   'V',   'W',   'X',   'Y',   'Z',
        'lbr', 'bsl', 'rbr', 'hat', 'und', 'bqu', 'a',
        'b',   'c',   'd',   'e',   'f',   'g',   'h',
        'i',   'j',   'k',   'l',   'm',   'n',   'o',
        'p',   'q',   'r',   's',   't',   'u',   'v',
        'w',   'x',   'y',   'z',   'lac', 'bar', 'rac',
        'tld', 'del' : -> CHARACTER
        first-char, last-char : -> CHARACTER
        succ, pred : CHARACTER -> CHARACTER
        eq, ne, gt, ge, lt, le : CHARACTER # CHARACTER -> BOOLEAN
        ord : CHARACTER -> NATURAL
        char : NATURAL -> CHARACTER
```

typical normal form: `'x'`

example: `ord('x') = nat((^(1) ^ 2) ^ 0)`

module: `Strings`

sort: `CHAR-SEQ`

This is just `Non-Empty-Sequences` bound to `Characters`.
typical normal form: `(((^('h') ^ 'e') ^ 'l') ^ 'l') ^ 'o'`

example: `elem('l', ^'h'^'e'^'l'^'l'^'o') = true`

2.8 Finite sorts with equality

In practice one often needs a finite sort with an explicit equality function. As shown in [BMW91] it is not easy to do this in a manner which is both correct and efficient. Therefore we included specifications of finite sorts up to 25 elements and a perl script to generate larger sorts. It would be nice to do this in a parameterized fashion, but in [BMW91] it was shown that this is impossible. Consequently, we provide them as separate modules. One can use renaming to create a sort with more meaningful names. A drawback is that it is not easy to create two different types with the same number of elements in this way. By using simple renamings, the origin rule [BHK89] would make them equal.

module: `f5`
sort: `f5`

functions:
```
        0f5, 1f5, 2f5, 3f5, 4f5 : -> f5
        succ, pred : f5 -> f5
        eq, ne, lt, le, gt, ge : f5 # f5 -> BOOLEAN
        ord : f5 -> NATURAL
        f5 : NATURAL -> f5
```

typical normal form: `2f5`

example: `succ(2f5) = 3f5`

3 Conclusions

Experiments showed that the new library compiles faster and is considerably more efficient. Faster compilation is due to the reduced length gained by combining several lengthy tables. The increase in efficiency comes from the fact that we used more efficient algorithms. It was often possible to reduce, say, quadratic algorithms to linear ones.

Since all numerical data types define the same functions, we think that our specification has a more consistent interface than [vW93].

We encountered several shortcomings in the ASF specification formalism which influenced the efficiency. The ability to use so-called *default conditions* in many cases would reduce the duplication in the reduction of conditions. Further, the possibility to only specify total functions not only leads to some extra equations, but in some cases it also influences the efficiency because extra conditions are needed.

We think, however, that the simplicity of the initial algebra approach outweighs the benefits gained from these nice but semantically complex features.

Furthermore, we encountered the problem that no distinction can be made between the representation used for internal calculations (i.e. the normal forms of the specification) and the representation of the terms showed to the user of the system. It is in general not the case that the representation suited for calculations is the same as the representation preferred by the user. Explicit input and output conversions would improve the readability.

The possibility offered by ASF of writing parameterized specifications showed very useful for the specification of several generic data types.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification.* ACM Press frontier series. ACM Press, 1989.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60:109–137, 1984.

[BMW91] J.A. Bergstra, S. Mauw, and F. Wiedijk. Uniform algebraic specifications of finite sets with equality. *Int. J. of Foundations of Computer Science*, 1(2):43–65, 1991.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[ISO89] ISO - International Organization for Standardization. *Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour, IS 8807.* 1989.

[IT88] ITU-TS. *ITU-TS Recommendation Z.100: Specification and Description Language (SDL).* ITU-TS, Geneva, 1988.

[Mos92] P. D. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic specification of communication protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[Vel93] G.J. Veltink. The PSF Toolkit. *Computer Networks and ISDN Systems*, 25(7):875–898, 1993.

[vW93] J.J. van Wamel. A library for PSF. Report P9301, Programming Research Group, University of Amsterdam, 1993.