Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.R.H. Hendriks

ASF system user's guide

Computer Science/Department of Software Technology Report CS-R8823 May

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

69D21, 69D41, 69D43, 69D44, 69F31, 69F32

ASF System User's Guide

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

A simple environment is described for compiling and testing ASF (Algebraic Specification Formalism) specifications. It consists of the following components: a parser and typechecker, a normalizer which removes the modular structure from the specification, a code generator which translates the specification to Prolog, a reduction machine which reduces input terms for a given specification, and simple tracing facilities for the reduction machine. An overview is given of the architecture of the ASF system and of the implementation techniques applied in it. Limitations and planned extensions of the current implementation are discussed.

Key Words & Phrases: Software Engineering, Algebraic Specifications, Specification Languages, Executable Specifications, Prolog.

1980 Mathematics Subject Classification (1985): 68Nxx **[Software]:** 68N20 Compilers and generators; 68Qxx **[Theory of computing]:** 68Q50 Grammars, rewriting systems; 68Q65 Abstract data types.

1987 CR Categories: D.2.1 **[Software Engineering]:** Requirements/Specifications; D.3.1 **[Programming Languages]:** Formal Definitions and Theory; D.3.3 **[Programming Languages]:** Language Constructs - Abstract data types; D.3.4 **[Programming Languages]:** Processors; F.3.1 **[Logics and Meanings of Programs]:** Specifying and Verifying and Reasoning about Programs - *Specification techniques*; F.3.2 **[Logics and Meanings of Programs]:** Semantics of Programming Languages - *Algebraic approaches to semantics.*

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

Note: This paper will be submitted for publication elsewhere.

1. Introduction

The goal of ESPRIT-project 348 (GIPE - Generation of Interactive Programming Environments) is to make a system which can generate an interactive programming environment for a programming language from the specification of that language. In this context the Algebraic Specification Formalism ASF [BHK87, BHK85] has been developed as an intermediate step in the development of a formalism to specify languages. However, one cannot only specify languages in ASF, but also general abstract data types.

One can derive executable prototypes from a given algebraic specification in several ways. In most approaches, the original specification is translated into a - preferably equivalent - form that can be executed efficiently. We have, for instance, experimented with translations to input for the Equation Interpreter [ODo85] and C-Prolog [PWBBP85]. These experiments are described in [Hen87, HK86].

In this paper, we describe a system for compilation and execution of ASF specifications. The equations in the specification are interpreted as rewrite rules from left to right.

A short introduction to ASF is given in section 2 and the system itself is described in section 3. Section 4 deals with soundness and completeness of the implementation and section 5 discusses some possible

Report CS-R8823
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

improvements of the current implementation. Finally, section 6 contains some conclusions related to the current implementation of the system.

2. Introduction to ASF

ASF [BHK87, BHK85] is an algebraic specification formalism similar to OBJ2 [FGJM85], ACT-ONE [EM85], and RAP [Hus86]. The general idea is to give a signature and a set of (conditional) equations over that signature. The signature consists of a set of sorts and a set of functions over these sorts.

The meaning of an ASF-specification is its initial algebra [MG85]. An example of the initial algebra is obtained by first taking the free term algebra over the signature of the specification. Next, a congruence relation is defined on this algebra by defining two closed terms to be equal if and only if their equality can be deduced from the set of (conditional) equations using many-sorted conditional equational logic [GM82]. Finally, closed terms that are in the same congruence class are identified.

ASF allows for modular division of the specification. It has several features to support this:

- Exports:
  Each module may have an `exports` section consisting of a (possibly incomplete) signature. These sorts and functions are visible outside the module.
- Hidden sorts and functions:
  Sorts and functions that are local to a module are declared in the `sorts` and `functions` sections.
- Imports:
  The `imports` section contains the names of modules that have to be incorporated in a module. While importing a module it is possible to bind its parameters, to rename its signature (see below) or to perform a combination of these.
- Parameters:
  Parameters are declarations of (possibly incomplete) signatures which are formal parameters of the module. They are declared in the `parameters` section. They can be bound to actual sorts and functions of a module when the parameterized module is imported.
- Renamings:
  Upon import of a module we can rename parts of the signature of the module if changes in names of sorts or functions are desirable to avoid, for instance, name clashes.

Modularization has been studied separately in [BHK86], which gives an algebraic specification of the main concepts of modularization, except for parameterization. In [BHK87] an extensive description of ASF is given and it also describes the normalization strategy, which defines how compound modules have to be evaluated in the context of the total specification to which they belong. The result of normalization is a module without imports, but some unbound parameters may remain.

3. The ASF system

3.1. User-level architecture

At the user level, the ASF system consists of three commands:

- `asfcheck`:
  Performs syntax checking and typechecking of an ASF specification.
- `asf`:
  Performs syntax checking, typechecking, normalization and generation of a prototype.
- `asfex`:
  Reduces a set of input terms using a previously generated prototype. The steps in these reductions may be displayed by setting the trace option of `asfex`.

Strictly speaking, the `asfcheck` command is redundant. It only exists for reasons of efficiency. The user-level architecture of the ASF system is shown in figure 1.

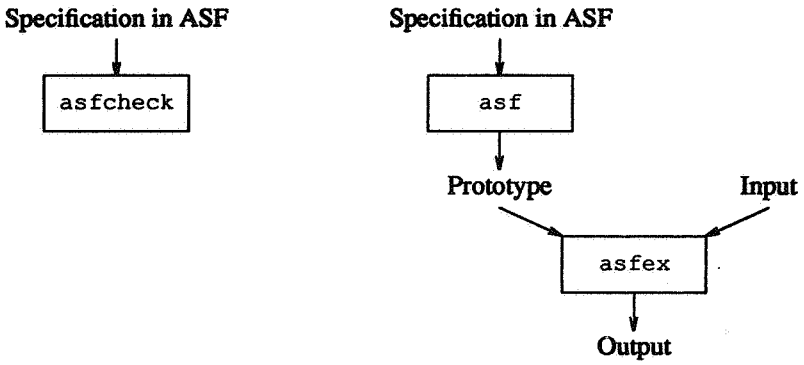

Fig. 1: User-level architecture of ASF system

3.2. An example

To illustrate the system we will use the following specification of natural numbers as an example:

```
module Natural-Numbers
begin

   exports
   begin
      sorts NAT
      functions
         zero :            -> NAT
         succ : NAT        -> NAT
         plus : NAT # NAT -> NAT
   end

   variables
      x, y : -> NAT

   equations

      [1]     plus(x, zero)     = x
      [2]     plus(x, succ(y)) = succ(plus(x, y))

end Natural-Numbers
```

Assume that this specification resides in a file named `example.asf`. First, this specification is checked and compiled into a prototype by the command:

```
asf example.asf
```

Now assume that the following input module resides on file `example.inp`:

```
module Natural-Numbers
begin

   variables
      x, y : -> NAT

   terms

      [1]     plus(zero, succ(zero))
      [2]     plus(succ(x), succ(y))

end Natural-Numbers
```

The command

```
asfex example.inp
```

will check the input module and produce the following:

```
module Natural-Numbers
begin

   [1]   plus(zero, succ(zero))
         = succ(zero)

   [2]   plus(succ(x), succ(y))
         = succ(plus(succ(x), y))

end Natural-Numbers
```

When the trace option of `asfex` is set, this output will also show the intermediate steps used in the reductions.

3.3. Internal structure

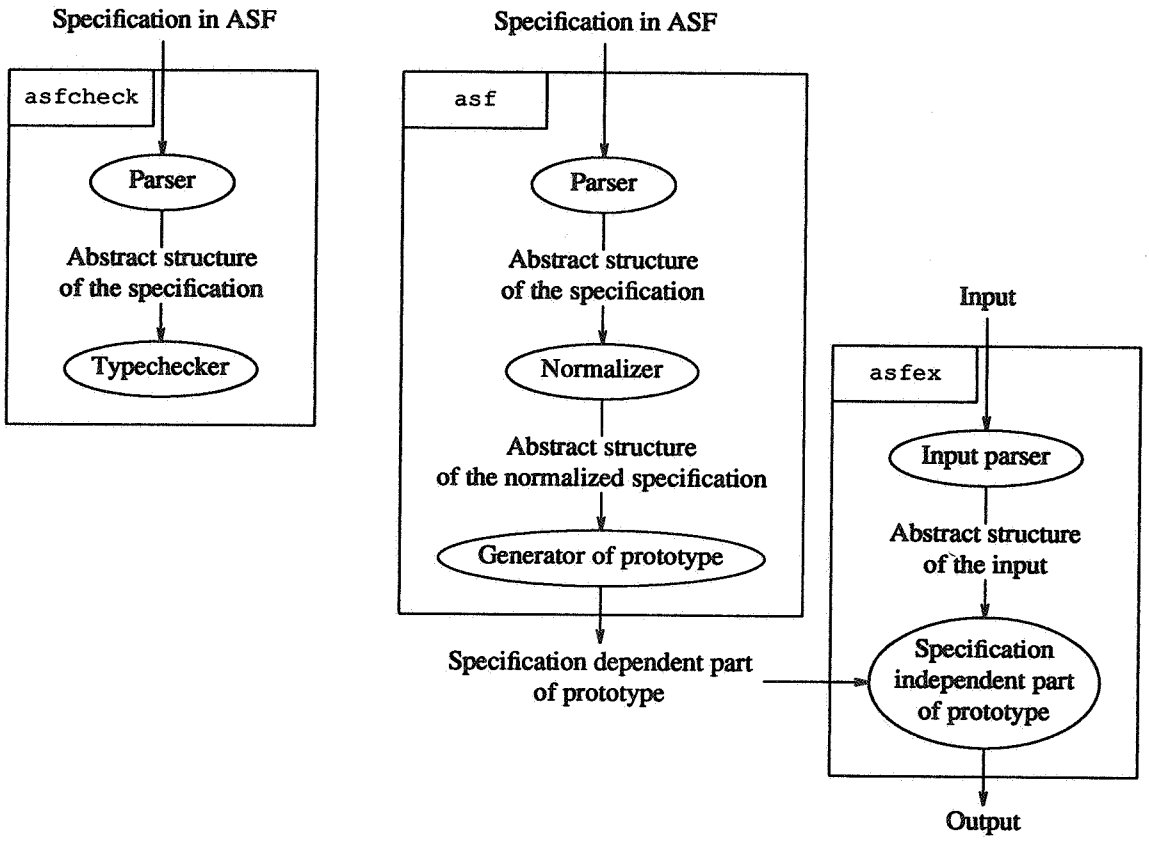

Fig. 2: Internal structure of ASF system

The *internal* structure of the ASF system is shown in figure 2. It consists of the following major steps:

- `asfcheck`:

  The typechecker of ASF acts on the abstract structure of a specification which is generated by a parser. This parser of ASF specifications is available as `asfparse`.

- `asf`:
  The same abstract structure is used in `asf` as input for the normalizer and the generator of the prototype.
  - `asfnorm`:
    The normalizer removes the modular structure from all modules in the specification. Thus, in the output of the normalizer all imports have been removed and all renamings and parameter bindings have been carried out.
  - `asfimpl`:
    The generator of the prototype adds type information to the equations and creates the specification dependent part of the prototype.
- `asfex`:
  The generated code plus the code for a "reduction machine" (the specification independent part) constitute the prototype. The input for the prototype is first transformed to abstract structures using the input parser, which is available as `inpparse`.

3.3.1. Typechecking and normalization

Although typechecking and normalization may be considered to be different at the user level, they have a lot in common in their implementation. The typechecker has to do a great deal of normalization, because it must at least construct the visible signature (the combination of the export signature and the signatures of the parameters) of each module. Therefore, the typechecker and the normalizer have been implemented by one combined program.

The normalizer generates the abstract structure of the normalized specification. This abstract structure is analogous to the one generated by the parser. This is done to allow addition of an unparser in a future version of the system.

The normalizer renames hidden sorts and functions when the module in which they are declared is imported in another module. The user of the system will only observe these renamings in traces of terms in which imported hidden functions occur. Function symbols and operators which consist of an identifier surrounded by dots are renamed by postfixing the identifier with a hyphen and a natural number. Operators which consist of sequences of one or more operator symbols are postfixed with one or more `*`'s. This assures that the new function symbols and operators are legal ASF function symbols and operators. These automatic renamings are such that name clashes with other functions and operators are avoided.

3.3.2. Generation of the prototype

A generated prototype consists of two parts:

- The specification dependent part:
  This part consists mainly of the Prolog code generated from the equations and the signatures of each (normalized) module. The signatures are needed to typecheck input modules. The equations are disambiguated to prevent incorrect use of equations due to overloading of function symbols.
- The specification independent part:
  This part of the prototype consists of C-Prolog clauses for typechecking input modules, for reducing terms to normal form, for printing terms, and for creating traces of reductions.

The generator of the prototype only creates the specification dependent part. Upon execution this code is added to the specification independent part.

Code is generated using a modified version of the compilational approach of van Emden and Yukawa [EY87, HK88]. It implements a leftmost innermost reduction strategy. For each n-ary function in the specification a $(n+1)$-ary predicate and a n-ary function are created. The predicate represents the graph of the function: its first argument is the result of the application of the function to the rest of the arguments. The function represents the case of normal forms (i.e. irreducible terms). A catch-all rule is used to build

the term if no equation is applicable. The following illustrates the generated code for the previously given example of natural numbers:

```
/* >>> Equations                                                    <<< */

/* [1] */
plus(X, X, zero).

/* [2] */
plus(Res, X, succ(Y))
   :- plus(I1, X, Y),
      succ(Res, I1).

/* >>> Catch-all                                                   <<< */

zero(zero).
succ(succ(I1), I1).
plus(plus(I1, I2), I1, I2).
```

(In section 5.2 an improved version of this code will be presented.)

Before generating code for an equation the use of variables is checked. First, let *V* be the set of variables used in the left-hand side of the conclusion of the equation. Next, the conditions (if present) of the equation are checked in the order in which they are specified. There are two possibilities for each condition:

1. The condition is correct if all variables of both sides are elements of *V*.
2. The condition is also correct if all variables of only one side of the condition occur in *V*. The extra variables in the other side are added to *V*.

Finally, it is checked that all variables in the right-hand side of the conclusion of the equation are members of the resulting set *V*. Hence, an error-message is given if in both sides of a condition or in the right-hand side of an equation variables are used which have not been introduced before.

The code generated for a condition is influenced by the above mentioned use of variables in it. In case 1 the generated code will reduce both sides of the condition and the condition succeeds if the results are equal. In the case 2 the side of the condition in which no new variables are introduced is reduced and the result has to be unified with the other side to fulfil the condition.

3.3.3. Parsing of specification and input modules

The ASF system contains two parsers: one for ASF specifications and one for input modules. Both parsers are written in LEX, YACC and C. They both generate abstract structures of their input represented in C-Prolog clauses.

The input of the prototype is a number of input-modules, each consisting of a set of terms labeled with the name of a module. These module names state which equations may be used while reducing a term to normal form. This is necessary because in this version of the ASF system the generated prototype is an implementation for all modules of the specification. The input should have the following concrete syntax:

```
<input>          ::= <input-module>+ .
<input-module>   ::= "module" <module-ident>
                     "begin"
                           [ <variables> ]
                           [ <terms> ]
                     "end" <module-ident> .
<terms>          ::= "terms" <tagged-term>+ .
<tagged-term>    ::= <tag> <term> .
```

Where <module-ident>, <variables>, <tag> and <term> are defined as in [BHK87]. Each input-module has two optional sections:

- `variables` section:

  The variables used in the `terms` section of the module should be declared with their sort in this section. These sorts should of course be defined in the signature of the corresponding module in

the specification after normalization.

- `terms` section:
  This section contains the terms to be reduced to normal form. Terms are always typechecked before they are reduced.

3.3.4. Reduction of terms to normal form

Terms in an input-module are translated in a similar manner as terms in specification modules. The term `plus(zero, succ(zero))` will, for instance, be translated to the Prolog goals:

```
?- zero(R1),
   zero(R2),
   succ(R3, R2),
   plus(Res, R1, R3).
```

In this way the arguments of a term are first reduced to normal form in left to right order. The standard Prolog interpretation ensures the reduction machine to search for the first equation (in the normalized specification) whose left-hand side matches. If the equation at hand is a conditional one, first all the conditions must be satisfied. The generated code is such that all subterms of the right-hand side of the equation are reduced to normal form before returning the result of evaluation.

If during the reduction of a term a conditional equation is encountered, its conditions are evaluated in the order in which they are specified. As mentioned before, the use of variables in a condition determines how it is evaluated.

Evaluation of open terms (terms in which variables occur) is supported by the system. Note, however, that variables are treated as constants in the reduction of an open term. This assures the prototype to return normal forms of open terms which are equationaly provable from the equations given in the specification. In particular no induction on the structure of terms is used to deduce equality of terms.

When the trace option is set, each instance of an equation used in the reduction of the terms in the input is printed. This is achieved by Prolog clauses which simulate an Prolog interpreter of the generated code which has the side effect of printing the trace information. Doing this the generated code of the prototype does not have to be changed to provide the trace-option. This avoids the need to regenerate the prototype when the trace option is activated or deactivated.

4. Soundness and completeness

The generated prototype interprets the equations of the specification as rewrite rules for a (conditional) term rewriting system.

The generated interpretation is sound, i.e. for all (eventually open) terms t_1 and t_2 the following holds: if the implementation I returns t_2 as the result of evaluation of t_1, then the equality of t_1 and t_2 can be proved using the equations E of the specification. In short notation:

$$\mathrm{I} \models t_1 \longrightarrow\!\!\!\!\!\rightarrow t_2 \;\Rightarrow\; \mathrm{E} \vdash t_1 = t_2$$

The proof of this is similar to the proof of the correctness of the compilational approach in [EY87].

More interesting is the question whether the converse holds. Or, more precisely, if two terms t_1 and t_2 are given such that they can be proved equal using E, can we use the implementation to show them to be equal:

$$\mathrm{E} \vdash t_1 = t_2 \;\Rightarrow\; \exists t\ \mathrm{I} \models t_1 \longrightarrow\!\!\!\!\!\rightarrow t \wedge \mathrm{I} \models t_2 \longrightarrow\!\!\!\!\!\rightarrow t$$

In general, this is too much to hope for, because it is undecidable whether an equation is derivable from a given set of equations. Incompleteness might be caused by non-termination of the implementation, non-confluence, and the inability to solve conditions. We will treat these aspects in short and for an extensive treatment we refer to [Kap87]. In general, it is undecidable whether any of these three aspects holds for a set of equations. However, syntactic criteria exits which assure the term rewriting system corresponding to a set of (conditional) equations to be complete. Such criteria have not been implemented in the current version of the system.

4.1. Termination

It is very easy to write a set of equations which, when interpreted as rewrite rules, will not terminate. Some of the easiest examples are:

```
[1]     a = a
```

and commutative laws like:

```
[1]     x + y = y + x
```

Several articles [Kap87, JW87] investigate the use of simplification orderings to prove termination of term rewriting systems. A simplification ordering [Der85, Rus85] is a wellfounded ordering $>$ on open terms such that:

- each term t is less than a term in which it occurs:

$$f(..., t, ...) > t$$

- and the ordering preserves contexts:

$$t_1 > t_2 \Rightarrow f(..., t_1, ...) > f(..., t_2, ...)$$

In [DF85] a description is given of an algorithm that will construct a simplification ordering that proves termination of a term rewriting system or terminates in failure. It tries to construct a simplification ordering from a given set of equations by assuming that all terms in the conditions and the right-hand side of the conclusion of an equation have to be smaller than the left-hand side of the conclusion.

Unfortunately, it is sometimes necessary to change the intended meaning of a specification in order to assure termination of the corresponding term rewriting system. It is, for example, impossible to give a specification of the datatype sets such that the corresponding term rewriting system terminates without imposing an ordering on the elements in the sets.

4.2. Confluence

A term rewriting system is confluent if for all terms t, t_1 and t_2 such that t reduces to t_1 as well as t_2, there exists a term v such that both t_1 and t_2 reduce to v. In short:

$$t \longrightarrow\!\!\!\!\!\rightarrow t_1 \wedge t \longrightarrow\!\!\!\!\!\rightarrow t_2 \Rightarrow \exists v \; t_1 \longrightarrow\!\!\!\!\!\rightarrow v \wedge t_2 \longrightarrow\!\!\!\!\!\rightarrow v$$

An example of a specification of which the corresponding term rewriting system is not confluent is the following:

```
[1]     a = b
[2]     a = c
```

In such a specification the implementation is incapable of proving the equality of, for example, `b` and `c`.

It is possible to transform a given set of (conditional) equations into a confluent (and terminating) term rewriting system if a simplification ordering on terms is given. In [Kap87] a sketch of such a completion procedure is given which is again improved in [JW87].

There exist syntactic criteria, like regularity, which assure confluence. A term rewriting system is regular if it is left-linear (no variable occurs more than once in the left-hand side of an equation) and non ambiguous (no two rules exist with overlapping left-hand sides). Such criteria are easy to check but they have not been implemented because they are useless if one wants to reduce open terms.

4.3. Conditions

To implement conditions correctly it is neccesary to be able to solve equations over a given set of equa-

tions. Again, confluence and termination are needed to find a solution of an equation. In

```
[1]     a = b
[2]     a = c
[3]     a = c ===> d = e
```

the generated code cannot reduce `d` to `e` because it cannot deduce the equality of `a` and `c`. The following example shows a set of equations for which the generated code will not terminate if `a` or `c` are to be reduced.

```
[1]     a = b ===> c = d
[2]     c = d ===> a = b
```

If in a condition of an equation variables occur which do not occur in the left-hand side of the conclusion, the implementation has to find terms which solve the condition. In Kaplan's article [Kap87] such conditions are forbidden. As mentioned before, our system is more liberal in allowing conditions in which variables may be introduced (see section 3.3.2).

A warning is given if variables are not used in the above mentioned way, but this does not guarantee completeness of the implementation as becomes clear in the following example:

```
module Natural-Numbers
begin

   exports
   begin
      sorts BOOL, NAT
      functions
         true  :            -> BOOL
         false :            -> BOOL
         0     :            -> NAT
         s     : NAT        -> NAT
         _+_   : NAT # NAT  -> NAT
         lt    : NAT # NAT  -> BOOL
   end

   variables
      x, y, z : -> NAT

   equations

      [1]    x + 0    = x
      [2]    x + s(y) = s(x + y)

      [3]    lt(x, x) = false
      [4]    x + s(y) = z ===> lt(x, z) = true
      [5]    x + s(y) = z ===> lt(z, x) = false

end Natural-Numbers
```

The generated code of this specification, for example, will show the term `lt(0, s(0))` to be irreducable. The term matches with the left-hand side of the conclusion of `[4]`, whereafter the term substituted for `z` (in this case: `s(0)`) is reduced and unified with `0 + s(y)`. This unification fails. Next, equation `[5]` is examined, but in this case the terms `0` and `s(0) + s(y)` have to be unifiable. No further equation is applicable and hence the term is irreducable.

5. Possible improvements

The ASF system described here is completely operational and can be used to compile specifications of reasonable size (e.g. 50 pages). We see, however, many potential improvements to the system. These will now be discussed briefly.

5.1. Normalization versus modular compilation

In the current system the modular structure of the specification is not reflected in the generated code.

The user is often interested in just one module of his specification, but in the current system code is generated for all modules. Each module is normalized and code is generated for it independently of the code generated for the other ones. This will not only increase the compilation time of specifications, but also the size and execution time of generated code.

While the user will often only change a few modules, the current ASF system will completely typecheck, normalize and generate code for all modules in the specification including the ones that have not been changed. A modular implementation would only process the changed modules. Currently, we are investigating modular implementation techniques.

5.2. Optimization of generated code

In the example of natural numbers (see section 3.2) the generated code could be simplified by observing that the functions `zero` and `succ` are never used as head symbol in the left-hand side of an equation. Hence, there are no clauses for the predicates of these functions except for the catch-all rules. As a consequence, the generated code could be simplified to:

```
/* >>> Equations                                                    <<< */

/* [1] */
plus(X, X, zero).

/* [2] */
plus(succ(I1), X, succ(Y))
   :- plus(I1, X, Y).

/* >>> Catch-all                                                   <<< */

plus(plus(I1, I2), I1, I2).
```

We could also do without the catch-all rule for `plus` if only closed expressions were to be reduced. This same information could also be used in the decomposition of an input term into Prolog goals. The goal

```
?- plus(Res, zero, succ(zero)).
```

would now be the result of decomposition of the term `plus(zero, succ(zero))`.

This simplification is not yet implemented, because it requires global information of the specification. As in the example of natural numbers and natural numbers modulo 2 in appendix I it can be useful to specify a function without any equations in one module and import this module in another one in which equations are added for the function. Using such global information is difficult to reconcile with our desire to achieve a modular implementation of ASF.

5.3. Optimization of the reduction machine

It frequently occurs that the same (sub)term is reduced more than once during the reduction of a given input term. Such repeated reductions can be avoided by storing terms and their computed normal form in a database. Before reducing a term, the reduction machine can consult the database to see whether or not it has been reduced previously and the stored normal form can be used. Of course, some equilibrium will have to be found between storing all intermediate results and recomputing them. Initial experiments show that this techniques might lead to substantial savings in execution time.

6. Concluding remarks

The current ASF implementation has been in use over the past year for the development of many small and several large specifications. It has been ported to various machines (Vax, Sun, and Gould) and institutes. The implementation consists of 2600 lines of Prolog, and 1600 lines of other supporting programs (e.g., C, LEX, and YACC). This shows that a simple, experimental, system for the processing of algebraic specifications can be build with a modest effort. However, still much research is needed to improve

compilation techniques and to improve the performance of the generated prototypes.

Acknowledgements

Niek van Diepen, Paul Klint and Emma van der Meulen commented on earlier versions of this paper. Discussions with Roland Bol, Niek van Diepen, Jan Heering, Paul Klint, Monique Logger, Jan Rekers, Pum Walters and Freek Wiedijk helped to build the system.

References

[BHK85] J.A. Bergstra, J. Heering, and P. Klint, "Algebraic definition of a simple programming language," Report CS-R8504, Centre for Mathematics and Computer Science, Amsterdam (1985).

[BHK86] J.A. Bergstra, J. Heering, and P. Klint, "Module algebra," Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam (1986).

[BHK87] J.A. Bergstra, J. Heering, and P. Klint, "ASF - an algebraic specification formalism," Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam (1987).

[Der85] N. Dershowitz, "Termination," pp. 180-224 in *Proceedings of the First International Conference on Rewriting Techniques and Applications*, ed. J.-P. Jouannaud, Lecture Notes in Computer Science **202**, Springer-Verlag, Dijon, France (1985).

[DF85] D. Detlefs and R. Forgaard, "A procedure for automatically proving the termination of a set of rewrite rules," pp. 255-270 in *Proceedings of the First International Conference on Rewriting Techniques and Applications*, ed. J.-P. Jouannaud, Lecture Notes in Computer Science **202**, Springer-Verlag, Dijon, France (1985).

[EM85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications*, vol. I, *Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science **6**, Springer-Verlag (1985).

[EY87] M.H. van Emden and K. Yukawa, "Logic programming with equations," *Journal of Logic Programming* **4**, pp. 265-288 (1987).

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," pp. 52-66 in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, New Orleans (1985).

[GM82] J.A. Goguen and J. Meseguer, "Universal realization, persistent interconnection and implementation of abstract modules," pp. 265-281 in *Proceedings 9th International Conference on Automata, Languages and Programming*, Lecture Notes in Computer Science **140**, Springer-Verlag (1982).

[HK86] J. Heering and P. Klint, "The efficiency of the Equation Interpreter compared with the UNH Prolog interpreter (extended abstract)," *SIGPLAN Notices* **21**(2), pp. 18-21 (1986).

[HK88] J. Heering and P. Klint, "Towards shorter algebraic specifications: a simple language definition and its compilation to Prolog," Report CS-R8814, Centre for Mathematics and Computer Science, Amsterdam (1988).

[Hen87] P.R.H. Hendriks, "Type-checking mini-ML: an algebraic specification with user-defined syntax," Report CS-R8737, Centre for Mathematics and Computer Science, Amsterdam (1987), Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands*, pp. 21-38, SION, Amsterdam (1987).

[Hus86] H. Hussmann, *Rapid prototyping for algebraic specifications - RAP system user's manual, Version 2.0, Draft*, Universität Passau (1986).

[JW87] J.-P. Jouannaud and B. Waldmann, "Reductive conditional term rewriting systems," pp. 223-244 in *Formal Description of Programming Concepts - III*, ed. M. Wirsing, Elsevier (1987).

[Kap87] S. Kaplan, "Simplifying conditional term rewriting systems: unification, termination and confluence," *Journal of Symbolic Computation* **4**, pp. 285-334, Academic Press Limited (1987).

[MG85] J. Meseguer and J.A. Goguen, "Initiality, induction, and computability," pp. 459-541 in *Algebraic Methods in Semantics*, ed. M. Nivat, and J.C. Reynolds, Cambridge University

Press (1985).

[ODo85] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press (1985).

[PWBBP85] F. Pereira, D. Warren, D. Bowen, L. Byrd, and L. Pereira, *C-Prolog User's Manual, Version 1.5*, SRI International, Menlo Park, California (1985).

[Rus85] M. Rusinowitch, "Path of subterms ordering and recursive decomposition ordering revisited," pp. 225-240 in *Proceedings of the First International Conference on Rewriting Techniques and Applications*, ed. J.-P. Jouannaud, Lecture Notes in Computer Science **202**, Springer-Verlag, Dijon, France (1985).

Appendix I. An example

I.1. Example

```
module Natural-Numbers
begin

   exports
   begin
      sorts NAT
      functions
         0    :              -> NAT
         succ : NAT          -> NAT
         plus : NAT # NAT -> NAT
         mult : NAT # NAT -> NAT
   end

   variables
      n, n1, n2 : -> NAT

   equations

      [n1]    plus(n, 0)          = n
      [n2]    plus(n1, succ(n2)) = succ(plus(n1, n2))

      [n3]    mult(n, 0)          = 0
      [n4]    mult(n1, succ(n2)) = plus(mult(n1, n2), n1)
      [n5]    mult(0, n)          = 0
      [n6]    mult(succ(n1), n2) = plus(mult(n1, n2), n2)

end Natural-Numbers

module Natural-Numbers-mod-2
begin

   imports
      Natural-Numbers
         {  renamed by
               [  NAT  -> NAT-2,
                  succ -> s,
                  plus -> _+_,
                  mult -> _*_ ] }

   equations

      [m1]    s(s(0)) = 0

end Natural-Numbers-mod-2
```

I.2. Example in abstract form (constructed by `asfparse`)

```
/*****************************************************************************/
/*                                                                           */
/*     Natural-Numbers                                                       */
/*                                                                           */
/*****************************************************************************/

sorts('Natural-Numbers','**exp','NAT').
function('Natural-Numbers','**exp',fnc('0',[],'NAT')).
function('Natural-Numbers','**exp',fnc(succ,['NAT'],'NAT')).
function('Natural-Numbers','**exp',fnc(plus,['NAT','NAT'],'NAT')).
function('Natural-Numbers','**exp',fnc(mult,['NAT','NAT'],'NAT')).
variable('Natural-Numbers',n2,'NAT').
variable('Natural-Numbers',n1,'NAT').
variable('Natural-Numbers',n,'NAT').
equation('Natural-Numbers','[n1]',
   eq(plus(n,'0'),n),
   []).

equation('Natural-Numbers','[n2]',
   eq(plus(n1,succ(n2)),succ(plus(n1,n2))),
   []).

equation('Natural-Numbers','[n3]',
   eq(mult(n,'0'),'0'),
   []).

equation('Natural-Numbers','[n4]',
   eq(mult(n1,succ(n2)),plus(mult(n1,n2),n1)),
   []).

equation('Natural-Numbers','[n5]',
   eq(mult('0',n),'0'),
   []).

equation('Natural-Numbers','[n6]',
   eq(mult(succ(n1),n2),plus(mult(n1,n2),n2)),
   []).

module_names('Natural-Numbers','Natural-Numbers').


/*****************************************************************************/
/*                                                                           */
/*     Natural-Numbers-mod-2                                                 */
/*                                                                           */
/*****************************************************************************/

binop('_+_','+').
binop('_*_','*').
imports('Natural-Numbers-mod-2','Natural-Numbers',
   ren([rename('NAT','NAT-2'),rename(succ,s),rename(plus,'_+_'),rename(mult,'_*_')]),
   '').
equation('Natural-Numbers-mod-2','[m1]',
   eq(s(s('0')),'0'),
   []).

module_names('Natural-Numbers-mod-2','Natural-Numbers-mod-2').
```

I.3. Normalized version of Example (abstract form)

```
/*****************************************************************************/
/*                                                                           */
/*     Natural-Numbers                                                       */
/*                                                                           */
/*****************************************************************************/

module_names('Natural-Numbers','Natural-Numbers').
sorts('Natural-Numbers','**exp','NAT').
function('Natural-Numbers','**exp',fnc('0','[]','NAT')).
function('Natural-Numbers','**exp',fnc(succ,['NAT'],'NAT')).
function('Natural-Numbers','**exp',fnc(plus,['NAT','NAT'],'NAT')).
function('Natural-Numbers','**exp',fnc(mult,['NAT','NAT'],'NAT')).
variable('Natural-Numbers',n2,'NAT').
variable('Natural-Numbers',n1,'NAT').
variable('Natural-Numbers',n,'NAT').
equation('Natural-Numbers','[n1]',eq(plus(n,'0'),n),'[]').
equation('Natural-Numbers','[n2]',eq(plus(n1,succ(n2)),succ(plus(n1,n2))),'[]').
equation('Natural-Numbers','[n3]',eq(mult(n,'0'),'0'),'[]').
equation('Natural-Numbers','[n4]',eq(mult(n1,succ(n2)),plus(mult(n1,n2),n1)),'[]').
equation('Natural-Numbers','[n5]',eq(mult('0',n),'0'),'[]').
equation('Natural-Numbers','[n6]',eq(mult(succ(n1),n2),plus(mult(n1,n2),n2)),'[]').


/*****************************************************************************/
/*                                                                           */
/*     Natural-Numbers-mod-2                                                 */
/*                                                                           */
/*****************************************************************************/

module_names('Natural-Numbers-mod-2','Natural-Numbers-mod-2').
sorts('Natural-Numbers-mod-2','**exp','NAT-2').
function('Natural-Numbers-mod-2','**exp',fnc('0','[]','NAT-2')).
function('Natural-Numbers-mod-2','**exp',fnc(s,['NAT-2'],'NAT-2')).
function('Natural-Numbers-mod-2','**exp',fnc('_+_',['NAT-2','NAT-2'],'NAT-2')).
function('Natural-Numbers-mod-2','**exp',fnc('_*_',['NAT-2','NAT-2'],'NAT-2')).
variable('Natural-Numbers-mod-2',n2,'NAT-2').
variable('Natural-Numbers-mod-2',n1,'NAT-2').
variable('Natural-Numbers-mod-2',n,'NAT-2').
equation('Natural-Numbers-mod-2','[n1]',eq('_+_'(n,'0'),n),'[]').
equation('Natural-Numbers-mod-2','[n2]',eq('_+_'(n1,s(n2)),s('_+_'(n1,n2))),'[]').
equation('Natural-Numbers-mod-2','[n3]',eq('_*_'(n,'0'),'0'),'[]').
equation('Natural-Numbers-mod-2','[n4]',eq('_*_'(n1,s(n2)),'_+_'('_*_'(n1,n2),n1)),'[]').
equation('Natural-Numbers-mod-2','[n5]',eq('_*_'('0',n),'0'),'[]').
equation('Natural-Numbers-mod-2','[n6]',eq('_*_'(s(n1),n2),'_+_'('_*_'(n1,n2),n2)),'[]').
equation('Natural-Numbers-mod-2','[m1]',eq(s(s('0')),'0'),'[]').


/*****************************************************************************/
/*                                                                           */
/*     Binary Operators                                                      */
/*                                                                           */
/*****************************************************************************/

binop('_*_',*).
binop('_+_',+).
```

I.4. Normalized version of Example (unparsed)

```
module Natural-Numbers
begin

   exports
   begin
      sorts NAT
      functions
         0    :            -> NAT
         succ : NAT        -> NAT
         plus : NAT # NAT -> NAT
         mult : NAT # NAT -> NAT
   end

   variables
      n, n1, n2 : -> NAT

   equations

      [n1]    plus(n, 0)          = n
      [n2]    plus(n1, succ(n2)) = succ(plus(n1, n2))

      [n3]    mult(n, 0)          = 0
      [n4]    mult(n1, succ(n2)) = plus(mult(n1, n2), n1)
      [n5]    mult(0, n)          = 0
      [n6]    mult(succ(n1), n2) = plus(mult(n1, n2), n2)

end Natural-Numbers

module Natural-Numbers-mod-2
begin

   exports
   begin
      sorts NAT-2
      functions
         0   :                -> NAT-2
         s   : NAT-2          -> NAT-2
         _+_ : NAT-2 # NAT-2 -> NAT-2
         _*_ : NAT-2 # NAT-2 -> NAT-2
   end

   variables
      n, n1, n2 : -> NAT-2

   equations

      [n1]    n + 0       = n
      [n2]    n1 + s(n2) = s(n1 + n2)

      [n3]    n * 0       = 0
      [n4]    n1 * s(n2) = n1 * n2 + n1
      [n5]    0 * n       = 0
      [n6]    s(n1) * n2 = n1 * n2 + n2

      [m1]    s(s(0)) = 0

end Natural-Numbers-mod-2
```

I.5. Specification dependent part of the prototype generated for Example

```
/*****************************************************************************/
/*                                                                           */
/*     Natural-Numbers                                                       */
/*                                                                           */
/*****************************************************************************/

module('Natural-Numbers').
sorts('Natural-Numbers','NAT').
function('0~1','Natural-Numbers','0','[]','NAT').
function('succ~1','Natural-Numbers',succ,['NAT'],'NAT').
function('plus~1','Natural-Numbers',plus,['NAT','NAT'],'NAT').
function('mult~1','Natural-Numbers',mult,['NAT','NAT'],'NAT').

/* >>> Equations                                                    <<< */

/* [n1] */
'plus~1'(_168,_168,'0~1')
   :- !.

/* [n2] */
'plus~1'(_341,_189,'succ~1'(_226))
   :- 'plus~1'(_372,_189,_226),
      'succ~1'(_341,_372),
      !.

/* [n3] */
'mult~1'(_237,_185,'0~1')
   :- '0~1'(_237),
      !.

/* [n4] */
'mult~1'(_360,_189,'succ~1'(_226))
   :- 'mult~1'(_392,_189,_226),
      'plus~1'(_360,_392,_189),
      !.

/* [n5] */
'mult~1'(_237,'0~1',_207)
   :- '0~1'(_237),
      !.

/* [n6] */
'mult~1'(_366,'succ~1'(_212),_231)
   :- 'mult~1'(_398,_212,_231),
      'plus~1'(_366,_398,_231),
      !.

/* >>> Catch-all                                                    <<< */

'0~1'('0~1').
'succ~1'('succ~1'(_173),_173).
'plus~1'('plus~1'(_180,_181),_180,_181).
'mult~1'('mult~1'(_180,_181),_180,_181).
```

```
/*****************************************************************************/
/*                                                                           */
/*     Natural-Numbers-mod-2                                                 */
/*                                                                           */
/*****************************************************************************/

module('Natural-Numbers-mod-2').
sorts('Natural-Numbers-mod-2','NAT-2').
function('0~2','Natural-Numbers-mod-2','0',[],'NAT-2').
function('s~1','Natural-Numbers-mod-2',s,['NAT-2'],'NAT-2').
function('_+_~1','Natural-Numbers-mod-2','_+_',['NAT-2','NAT-2'],'NAT-2').
binop('_+_~1',+).
function('_*_~1','Natural-Numbers-mod-2','_*_',['NAT-2','NAT-2'],'NAT-2').
binop('_*_~1',*).

/* >>> Equations                                                    <<< */

/* [n1] */
'_+_~1'(_347,_347,'0~2')
   :- !.

/* [n2] */
'_+_~1'(_520,_368,'s~1'(_405))
   :- '_+_~1'(_551,_368,_405),
      's~1'(_520,_551),
      !.

/* [n3] */
'_*_~1'(_416,_364,'0~2')
   :- '0~2'(_416),
      !.

/* [n4] */
'_*_~1'(_539,_368,'s~1'(_405))
   :- '_*_~1'(_571,_368,_405),
      '_+_~1'(_539,_571,_368),
      !.

/* [n5] */
'_*_~1'(_416,'0~2',_386)
   :- '0~2'(_416),
      !.

/* [n6] */
'_*_~1'(_545,'s~1'(_391),_410)
   :- '_*_~1'(_577,_391,_410),
      '_+_~1'(_545,_577,_410),
      !.

/* [m1] */
's~1'(_426,'s~1'('0~2'))
   :- '0~2'(_426),
      !.

/* >>> Catch-all                                                    <<< */

'0~2'('0~2').
's~1'('s~1'(_352),_352).
'_+_~1'('_+_~1'(_359,_360),_359,_360).
'_*_~1'('_*_~1'(_359,_360),_359,_360).
```

Appendix II. Messages of the system

This appendix contains a list of all messages the ASF system can generate. The parts of the messages printed in italic are placeholders for pieces of text which will be filled in with appropriate information. Most messages are prefixed with an indication of the program from which they originate. These prefixes are:

`ASF Parser:`	Parser of input
`ASF Check:`	Typechecker
`ASF Norm:`	Normalizer
`ASF Impl:`	Generator of prototype
`ASF Input Parser:`	Parser of input
`ASF Ex:`	The generated prototype

There are two types of messages which are not prefixed by one of the above mentioned labels. All messages of Prolog start with an exclamation mark `!`. And, finally, the user is warned if commands are used incorrectly:

- `Usage: asf [-o outputfile] [files]`
  In case the command: asf is used with the -o option and without further names of files.
- `Usage: asfex [-t] [-i inputfile] [files]`
  The -i option is given without file names.

The system not only generates error messages but it also notifies the user after succesful completion of major operations on the input. These messages are:

- `ASF Parser: File has been parsed`
- `ASF Check: Typechecking is finished`
- `ASF Norm: Normalization is finished`
- `ASF Impl: Compilation is finished`
- `ASF Input Parser: File has been parsed`
- `ASF Ex: File has been executed`

II.1. Error messages of the parsers

Both parsers in the system can give error messages. In case of the parser for ASF specifications the messages are prepended with `ASF Parser:`. The messages of the parser for input terms are preceded by `ASF Input Parser:`. These messages are:

- `Can't open file "File" for input`
- `Usage: asfparse [-o outputfile] [files]`
- `Can't open file "File" for output`
- `"File" line Nr: "Symbol" unexpected; "Symbol1" or "Symbol2" expected`
- `"File" syntax error in state Nr`

II.2. Error messages of typechecker and normalizer

Each of the messages in the following list is preceded by either `ASF Check:` or `ASF Norm:` depending on the program from which the message originates.

- `Can't open file "File" for output`
- `No abstract syntax loaded`
- `Can't open file "File" for input`
- `module "Mod" defined more than once`
- `module "Mod1" begin ... end "Mod2": "Mod1" not equal to "Mod2 "`
- `module "Mod": "Parm1" begin ... end "Parm2": "Parm1" not equal to "Parm2"`
- `module "Mod2": imported module "Mod1" not yet defined`
- `module "Mod": [ ... "Name" -> "Name1" ... "Name" -> "Name2" ... ]: multiple renaming of "Name"`
- `module "Mod2": [ ... "Name1" -> "Name2" ... ]: no visible name "Name1" in module "Mod1"`

- module "*Mod2*": [... "*Name1*" -> "*Name2*" ...]: no formal name "*Name1*" in parameter "*Parm*" of module "*Mod1*"
- module "*Mod2*": module "*Mod1*" does not have parameter "*Parm*"
- module "*Mod*": hidden sort "*Sort*" in declaration of visible function "*Fnc*: -> *Sort*"
- module "*Mod*": undeclared sort "*Sort*" in function declaration "*Fnc*: -> *Sort*"
- module "*Mod*": undeclared sort "*Sort*" in variable declaration "*Var*: -> *Sort*"
- module "*Mod*": equation [*Tag*]: left-hand side has type "*Sort1*" and right-hand side type "*Sort2*"
- module "*Mod*": equation [*Tag*]: term "*Term*" cannot be typed
- module "*Mod*": illegal renaming of sort "*Sort*" to unary operator "*Un-Op_*"
- module "*Mod*": illegal renaming of sort "*Sort*" to binary operator "*_Bin-Op_*"
- module "*Mod*": illegal renaming of function "*Fnc*: *Sort* -> *Sort*" to binary operator "*_Bin-Op_*"
- module "*Mod*": illegal renaming of function "*Fnc*: *Sort* # *Sort* -> *Sort*" to unary operator "*Un-Op_*"
- module "*Mod*": illegal renaming of function "*Fnc*: *Sort* # *Sort* # *Sort* -> *Sort*" to unary operator "*Un-Op_*"
- module "*Mod*": illegal renaming of function "*Fnc*: *Sort* # *Sort* # *Sort* -> *Sort*" to binary operator "*_Bin-Op_*"
- module "*Mod2*": [... "*Sort*" -> "*Sort1*" ...]: binding to hidden sort "*Sort1*" of actual module "*Mod1*"
- module "*Mod2*": [... "*Fnc*" -> "*Fnc1*" ...]: binding to hidden function "*Fnc1*: -> *Sort*" of actual module "*Mod1*"
- module "*Mod2*": [... "*Sort*" -> "*Sort1*" ...]: no sort "*Sort1*" in actual module "*Mod1*"
- module "*Mod2*": [... "*Fnc*" -> "*Fnc1*" ...]: no function "*Fnc1*: -> *Sort*" in actual module "*Mod1*"
- module "*Mod*": parameter "*Parm*" defined more than once
- module "*Mod*": sort "*Sort*" has different origins: modules "*Mod1*" and "*Mod2*"
- module "*Mod*": function "*Fnc*: -> *Sort*" has different origins: modules "*Mod1*" and "*Mod2*"
- module "*Mod*": sort "*Sort*" has different origins: exports section and hidden section
- module "*Mod*": function "*Fnc*: -> *Sort*" has different origins: exports section and hidden section
- module "*Mod*": sort "*Sort*" has different origins: parameter "*Parm*" and exports section
- module "*Mod*": function "*Fnc*: -> *Sort*" has different origins: parameter "*Parm*" and exports section
- module "*Mod*": sort "*Sort*" has different origins: parameter "*Parm*" and hidden section
- module "*Mod*": function "*Fnc*: -> *Sort*" has different origins: parameter "*Parm*" and hidden section
- module "*Mod*": sort "*Sort*" has different origins: parameter "*Parm1*" and parameter "*Parm2*"
- module "*Mod*": function "*Fnc*: -> *Sort*" has different origins: parameter "*Parm1*" and parameter "*Parm2*"
- module "*Mod*": sort "*Sort*" has different origins: sorts "*Sort1*" and "*Sort2*"
- module "*Mod*": function "*Fnc*: -> *Sort*" has different origins: functions "*Fnc1*: -> *Sort1*" and "*Fnc2*: -> *Sort2*"
- module "*Mod*": overloaded functions "*Fnc*: -> *Sort1*" and "*Fnc*: -> *Sort2*"

- module "*Mod*": overloaded function "*if: BOOL # Sort1 # Sort1 -> Sort2*" with predefined if-function
- module "*Mod*": overloading of variable "*Name: -> Sort*" with constant "*Name: -> Sort*"
- module "*Mod*": overloaded variables "*Var: -> Sort1*" and "*Var: -> Sort2*"

II.3. Error messages of the generator of prototypes

Each of the following error messages of the generator of prototypes is preceded by `ASF Impl:`.

- Can't open file "*File*" for output
- No abstract syntax loaded
- Can't open file "*File*" for input
- module "*Mod2*" imports "*Mod1*"
- module "*Mod*" left-hand side of equation [*Tag*] is a variable
- module "*Mod*" left-hand side of equation [*Tag*] is a tuple
- module "*Mod*" wrong use of "*Var1, Var2*" in condition of equation [*Tag*]
- module "*Mod*" wrong use of "*Var1, Var2*" in right-hand side of equation [*Tag*]

NAME

asfcheck – parser and typechecker for the algebraic specification formalism ASF

SYNOPSIS

asfcheck [*files*]

DESCRIPTION

Asfcheck parses and typechecks a specification written in the algebraic specification formalism *ASF*.

It reads from standard input if no *files* are specified, otherwise the contents of *files* are concatenated and will be treated as one algebraic specification. Searching for files proceeds in two stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.asf* and a second attempt is made. The specification is parsed using *asfparse*, and if parsing is successful it is checked.

An interpreter for a correct specification can be generated using *asf*. It is recommended to use *asfcheck* to parse and typecheck a specification before using *asf*, because normalization is a time-consuming process and it is useless if the algebraic specification is type-incorrect.

FILES

Current Directory

out*.as	abstract syntax of specification

Bin-Directory for ASF

asfcheck	shell-script for this command
asfparse	parser
exec.norm	code for the typechecker

SEE ALSO

- asf(7), asfparse(7), asfnorm(7), asfimpl(7), asfex(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

ASF Parser: *text*
Messages from the parser, see asfparse(7).

ASF Check: Typechecking is finished
ASF Check: No abstract syntax loaded
ASF Check: module "*name*" *text*
Error messages of the typechecker.

! *text*
Messages from Prolog

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

NAME

asf – code generator for the algebraic specification formalism ASF

SYNOPSIS

asf [**–o** *outputfile*] [*files*]

DESCRIPTION

Asf generates an interpreter for a specification written in the algebraic specification formalism *ASF*.

It reads from standard input if no *files* are specified, otherwise the contents of *files* are concatenated and will be treated as one algebraic specification. Searching for files proceeds in two stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.asf* and a second attempt is made. The specification is parsed using *asfparse*, and if parsing is successful it is normalized with *asfnorm*. The specification dependant part of the interpreter is generated with *asfimpl* in the standard file *out.impl* in the current directory if no **–o** option is given, otherwise it is written in *outputfile*.

The generated interpreter can be used by *asfex* to reduce terms to their normal form.

Normalization is a time-consuming process and it is useless if the algebraic specification is type-incorrect. It is recommended to use *asfcheck* to parse and typecheck a specification before using *asf*.

FILES

Current Directory

out.impl	standard output (without **–o** option)
out*.as	abstract syntax of specification
out*.norm	abstract syntax of normalized specification

Bin-Directory for ASF

asf	shell-script for this command
asfparse	parser
asfnorm	normalizer
exec.norm	code for the normalizer
asfimpl	code generator
exec.impl	code for the code generator

SEE ALSO

- asfparse(7), asfcheck(7), asfnorm(7), asfimpl(7), asfex(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

Usage: asf [-o outputfile] [files]
: If the **–o** option is given without outputfile.

ASF Parser: *text*
: Messages from the parser, see asfparse(7).

ASF Norm: *text*
: Messages from the normalizer, see asfnorm(7).

ASF Impl: *text*
: Messages from the code generator, see asfimpl(7).

! *text*
: Messages from Prolog

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

NAME

asfex – reduction machine for the algebraic specification formalism ASF

SYNOPSIS

asfex [–i *inputfile*] [–t] [*files*]
asfex [–t] [–i *inputfile*] [*files*]

DESCRIPTION

Asfex reduces terms to their normal form. It needs the code (generated by *asf*) from a specification written in the algebraic specification formalism *ASF*.

Asfex looks for the generated code from the specification in the *inputfile* if the –i option is given, otherwise it looks for the standard file *out.impl* in the current directory. *Asfex* reads from standard input if no *files* are specified and in that case the output of the reduction will be written in *out.ex*. The *files* are parsed using *inpparse*, and if parsing is successful the reduction machine will reduce each term to its normal form. Searching for files proceeds in two stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.inp* and a second attempt is made. The output of the reduction is written in the corresponding file with the suffix *.ex*.

The reduction machine uses leftmost innermost reduction to reduce terms to their normal form and a trace of the reduction is given when the –t option is specified.

SYNTAX OF INPUT

The input should have the following syntax:

```
<input>          ::= <input-module>+ .
<input-module>   ::= "module" <module-ident>
                     "begin"
                         [ <variables> ]
                         [ <terms> ]
                     "end" <module-ident> .
<terms>          ::= "terms" <tagged-term>+ .
<tagged-term>    ::= <tag> <term> .
```

Where <module-ident>, <variables>, <tag> and <term> are defined as in *ASF*.

FILES

Current Directory

out.impl	standard input (without –i option)
out.ias	abstract syntax of input (without *files* specified)
out.ex	standard output (without *files* specified)
file.ias	abstract syntax if *file* is one of the *files* specified
file.ex	outputfile if *file* is one of the *files* specified

Bin-Directory for ASF

asfex	shell-script for this command
inpparse	parser for input
exec.red	code for the reduction machine

SEE ALSO

- asf(7), asfparse(7), asfcheck(7), asfnorm(7), asfimpl(7), inpparse(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

Usage: asfex [-t] [-i inputfile] [files]
If the –i option is given without *inputfile* and without *files*.

ASF Ex: *inputfile* does not exist

ASF Input Parser: *text*
Messages from the parser, see inpparse(7).

ASF Ex: *file* has been executed
ASF Ex: module "*name*" *text*
Error messages of the interpreter.

! *text*
Messages from Prolog

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

NAME

asfparse – parser for the algebraic specification formalism ASF

SYNOPSIS

asfparse [**–o** *outputfile*] [*files*]

DESCRIPTION

Asfparse parses a specification written in the algebraic specification formalism *ASF*.

It reads from standard input if no *files* are specified, otherwise the contents of *files* are concatenated and will be treated as one algebraic specification. Searching for files proceeds in two stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.asf* and a second attempt is made. The abstract syntax of the algebraic specification is written in the standard file *out.as* in the current directory if no **–o** option is given, otherwise the abstract syntax is written in *outputfile*.

The generated abstract syntax can be used by *asfnorm* to normalize the specification and by *asfimpl* to generate an interpreter for all modules without **imports** in the specification.

Asfparse is used in *asf* and *asfcheck*.

FILES

Current Directory
out.as standard output (without **–o** option)

Bin-Directory for ASF
asfparse this command

SEE ALSO

- asf(7), asfcheck(7), asfnorm(7), asfimpl(7), asfex(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

ASF Parser: *file* has been parsed
Message after parsing of *file*.
ASF Parser: Usage: asfparse [-o outputfile] [files]
If the **–o** option is given without outputfile.
ASF Parser: Can't open file "*file*" for input
ASF Parser: Can't open file "*file*" for output
ASF Parser: "*file*" line *linenumber*: "*token*" unexpected; "*tokens*" expected
ASF Parser: "*file*" syntax error in state *number*

BUGS

The parser will not recover from all errors: It skips until the next newline if an error occurs in a function- or variable declaration.

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

NAME

asfnorm – normalizer for the algebraic specification formalism ASF

SYNOPSIS

asfnorm [–o *outputfile*] [*files*]

DESCRIPTION

Asfnorm normalizes a specification written in the algebraic specification formalism *ASF*. Its input is the abstract syntax of the specification as generated by the parser for ASF using *asfparse*. The output is also in the form of the abstract syntax of a specification. This generated abstract syntax can be used by *asfimpl* to generate an interpreter.

Asfnorm reads from standard input if no *files* are specified, otherwise the contents of *files* are concatenated and will be treated as one algebraic specification. Searching for files proceeds in two stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.as* and a second attempt is made.

Asfnorm is used in *asf* to generate an interpreter for an algebraic specification. It is recommended to use *asfcheck* to parse and typecheck a specification before using *asf*, because normalization is a time-consuming process and it is useless if the algebraic specification is type-incorrect.

FILES

Current Directory

out.norm	standard output (without –o option)

Bin-Directory for ASF

asfnorm	shell-script for this command
exec.norm	code for the normalizer

SEE ALSO

- asf(7), asfparse(7), asfcheck(7), asfimpl(7), asfex(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

ASF Norm: Usage: asfnorm [-o outputfile] [files]
: If the –o option is given without outputfile.

ASF Norm: Normalization is finished
: Message after successful normalization.

ASF Norm: No abstract syntax loaded

ASF Norm: Can't open file "*file*" for input

ASF Norm: Can't open file "*file*" for output

ASF Norm: module "*name*" *text*
: Error messages of the normalizer.

! *text*
: Messages from Prolog

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

NAME

asfimpl – code generator for the algebraic specification formalism ASF

SYNOPSIS

asfimpl [**–o** *outputfile*] [*files*]

DESCRIPTION

Asfimpl generates an interpreter for a specification written in the algebraic specification formalism *ASF*. Its input is the abstract syntax of the specification as generated by the parser for ASF using *asfparse* or the normalizer *asfnorm*.

Asfimpl reads from standard input if no *files* are specified, otherwise the contents of *files* are concatenated and will be treated as one algebraic specification. Searching for files proceeds in three stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.norm* and a second attempt is made. Finally, an attempt is made using the suffix *.as*.

Asfimpl is used in *asf* to generate an interpreter for an algebraic specification written in ASF.

FILES

Current Directory

out.impl	standard output (without –o option)

Bin-Directory for ASF

asfimpl	shell-script for this command
exec.impl	code for the code generator

SEE ALSO

- asf(7), asfparse(7), asfcheck(7), asfnorm(7), asfex(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

ASF Impl: Usage: asfimpl [-o outputfile] [files]
: If the –o option is given without outputfile.

ASF Impl: Compilation is finished
: Message after successful code generation.

ASF Impl: No abstract syntax loaded

ASF Impl: Can't open file "*file*" for input

ASF Impl: Can't open file "*file*" for output

ASF Impl: module "*name*" *text*
: Error messages of the code generator.

! *text*
: Messages from Prolog

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

NAME

inpparse – parser for input-modules

SYNOPSIS

inpparse [–o *outputfile*] [*files*]

DESCRIPTION

Inpparse parses input for the interpreter generated from a specification written in the algebraic specification formalism *ASF*.

It reads from standard input if no *files* are specified, otherwise the contents of *files* are concatenated and will be treated as one algebraic specification. Searching for files proceeds in two stages. First, each filename, as given, is opened. Only if this fails, the name is extended with the default suffix *.inp* and a second attempt is made. The abstract syntax of the algebraic specification is written in the standard file *out.ias* in the current directory if no –o option is given, otherwise the abstract syntax is written in *outputfile*.

Inpparse is used in *asfex* to reduce terms to their normal form.

SYNTAX OF INPUT

The input should have the following syntax:

```
<input>          ::= <input-module>+ .
<input-module>   ::= "module" <module-ident>
                     "begin"
                         [ <variables> ]
                         [ <terms> ]
                     "end" <module-ident> .
<terms>          ::= "terms" <tagged-term>+ .
<tagged-term>    ::= <tag> <term> .
```

Where <module-ident>, <variables>, <tag> and <term> are defined as in *ASF*.

FILES

Current Directory

out.ias	standard output (without –o option)

Bin-Directory for ASF

inpparse	this command

SEE ALSO

- asfex(7).
- J.A. Bergstra, J. Heering, and P.Klint,
  ASF — An algebraic specification formalism,
  Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

DIAGNOSTICS

ASF Input Parser: *file* has been parsed
Message after parsing of *file*.

ASF Input Parser: Usage: inpparse [-o outputfile] [files]
If the –o option is given without outputfile.

ASF Input Parser: Can't open file "*file*" for input

ASF Input Parser: Can't open file "*file*" for output

ASF Input Parser: "*file*" line *linenumber*: "*token*" unexpected; "*tokens*" expected

ASF Input Parser: "*file*" syntax error in state *number*

BUGS

The parser will not recover from all errors: It skips until the next newline if an error occurs in a variable declaration.

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

EXTRACTING ASF FROM TAPE

After extracting the *Asf*-Implementation from tape using:

tar xv

a directory named asfimpl is created in the current directory. It contains a Makefile, this text (README) and the input for this text in manual format (Readme.text). Furthermore it contains five subdirectories:

bin	Bin-directory, which is empty.
eqs	Sources to handle asf specifications.
input	Sources to handle the input.
man	Manual pages.
test	Test input.

REQUIREMENTS

In order to be able to install and use the *Asf*-Implementation it is necessary to have an implementation of *C-Prolog*. The default name for *C-Prolog* which the system uses, is *prolog*. This default value can be changed by editing the variable *PROLOG* in the header of the Makefile in the asfimpl-directory. Upon installation of the system it is possible to modify the default stacksizes of *C-Prolog* used throughout the system by editing the variable *STACK* in the same file.

INSTALLATION

To install the *Asf*-Implementation go to the directory asfimpl and use

make

to create several commands in the bin-directory.

After this, make these commands available by creating links to the files:

asf, *asfcheck*, *asfparse*, *asfnorm*, *asfimpl*, *asfex* and *inpparse*

in the bin-directory or by adding the pathname of the bin-directory to your searchpath.

TESTING

The *Asf*-Implementation can be tested by going to the test-directory: asfimpl/test. It contains an asf-specification of the booleans (in bool.asf) and natural numbers (in nat.asf).

To check the specifications use:

asfcheck bool nat

The given specification is correct and hence the following messages should be generated:

ASF Parser: bool has been parsed
ASF Parser: nat has been parsed
ASF Check: Typechecking is finished

It should be possible to generate an interpreter for this specification using:

asf bool nat

The following messages should be generated:

ASF Parser: bool has been parsed
ASF Parser: nat has been parsed
ASF Norm: Normalization is finished
ASF Impl: Compilation is finished

and a file out.impl will be created.

The testinput for this specification is in the file term.inp. To reduce the terms in this file to normal form use the command

asfex term or
asfex -t term

It generates the following messages:

ASF Input Parser: term has been parsed
ASF Ex: term has been executed

and a file term.ex in which the output of the reduction (without or with trace information) can be found.

MANUAL PAGES

Manual pages for the distinct commands can be found in the directory: asfimpl/man. These can be read using the manual package of ditroff.

SEE ALSO

- J.A. Bergstra, J. Heering, and P.Klint,
ASF — An algebraic specification formalism,
Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

AUTHOR

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.