Syntax and Semantics of a High-Level Intermediate Representation for ASF+SDF

J.A. Bergstra, M.G.J. van den Brand

Software Engineering (SEN)

SEN-R0030 November 30, 2000

Report SEN-R0030
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Syntax and Semantics of a High-Level Intermediate Representation for ASF+SDF

J.A. Bergstra
email: janb@wins.uva.nl

Utrecht University, Department of Philosophy
P.O. Box 80126, NL-3508 TC
Utrecht, The Netherlands
University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ
Amsterdam, The Netherlands

M.G.J. van den Brand
email: Mark.van.den.Brand@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
Developing a compiler for ASF+SDF has been a challenging task. The compilation of ASF+SDF is performed using an intermediate language μASF, an abstract syntax representation of ASF+SDF. Although ASF+SDF is quite simple to use, it provides a number of features which have an unclear semantics. By means of a number of examples in μASF and a more mathematical notation a number of these semantic issues are clarified. These examples are helpful for both the compiler designer and the specification writer.

1998 ACM Computing Classification System: D.3.1, D.3.2, F.4.2
Keywords and Phrases: rewriting system compiler, language definition, semantics definition language
Note: Work carried out under project SEN1.4, ASF+SDF

1. Introduction

Since its definition in [BHK89] developing a compiler for ASF and ASF+SDF [DHK96, HHKR89] has been a challenging task. Several methods to obtain fast interpreters and/or compilers have been investigated. We mention [Dik89, Hen91, Wal91, KW93, Moo94, Kam96, FKW98]. Writing a compiler for ASF+SDF is not a simple task for several reasons. One of them, addressed in this article, is that ASF+SDF provides various tricky features, among others:

- default rules,
- negative conditions,
- overlapping left-hand sides of rewrite rules, and
- list matching in combination with conditions.

The semantics of these features is not crystal clear. However, to obtain a correct compiler for ASF+SDF these features should be clarified. One option is to use an intermediate format to which ASF+SDF can be translated and for which a compiler can be written more easily. μASF, a subset of ASF, is such an intermediate format. It removes the SDF part and all modularisation. μASF allows the denotation of single-sorted, flat rewrite systems with positive and negative conditions, and default rules. μASF is an abstract syntax representation of ASF+SDF. In fact, by describing the semantics of μASF we are also clarifying the semantics of ASF+SDF itself. Reasoning about the semantics in terms of an abstract version of ASF+SDF is much simpler because of the lack of syntactic sugar.

In order to describe the dynamic semantics of μASF the notions of stable and liberal rewriting will be introduced. Stable rewriting means that at no point there is a choice between two (or more) applicable rules of the same priority level which lead to different terms. Liberal rewriting allows the application of a rule if its left-hand side matches and all its conditions are satisfied. Given the notions of stable and liberal rewriting the stability conditions are defined in order to ensure that all liberal normalisations are in fact stable.

Given a μASF specification the translation to C [KR78] is achievable. Furthermore, μASF ensures that the translation is kept simple and maintainable.

This paper defines the static semantics of μASF. Furthermore it illustrates the dynamic semantics of μASF by means of many small examples. The context-free syntax definition of μASF can be found in Appendix 1. In Section 4 the notions of stable and liberal rewriting are introducted. In Section 5 these notions are extended for list matching. Section 6 presents the combination of outermost and innermost evaluation. It concludes with the implementation criteria for an ASF+SDF compiler and guidelines for a specification writer. This paper is not an ASF+SDF design rationale. It describes the status quo of ASF+SDF largely without further motivation. See, for instance, [DHK96] for the use of ASF+SDF and more sophisticated examples.

2. μASF Static Semantics

μASF is an abstract syntax representation of ASF+SDF. It is an algebraic specification formalism which only allows prefix notation. It is single-sorted and allows the use of conditions (positive and negative) and the use of default equations. The default mechanism is used to prioritize rewrite rules. There is no import mechanism in μASF but it is possible to split a specification into several μASF modules.

A statically correct μASF specification has to satisfy a number of criteria. These criteria are formulated informally, but it is possible to implement them in a typechecker for μASF. The specification formalism is single-sorted, thus only the arity of functions has to be checked. In fact each μASF module is self-contained. This means that all information concerning the functions is available in the module. Variables are not explicitly declared. There are no restrictions concerning left-linearity. Each function in the signature which is not attributed “external” is implicitly declared as “local”. The rules to be checked are as follows:

1. No overloading of function names is allowed.
2. If a function occurs as an outermost function symbol in the left-hand side of a (conditional) equation it must be declared as “local” in the signature.
3. Every function used in the equations must be declared. Such a declaration is implicitly local unless the “external” attribute is added.
4. Functions declared with the attribute “constructor” may not occur as outermost function symbol in the left-hand side of an equation.
5. The arity of an occurrence of a function in an equation must be equal to the arity in its declaration in the signature.
6. For the use of variables in a (conditional) equation the following rules hold:
   (a) All variables in the right-hand side of a (conditional) equation should occur in the left-hand side or in one of the positive conditions.

```
imports Layout
exports
  sorts  BOOL
  context-free syntax
    true                  → BOOL  {constructor}
    false                 → BOOL  {constructor}
    BOOL "|" BOOL         → BOOL  {left}
    BOOL "&" BOOL         → BOOL  {left}
    BOOL "xor" BOOL       → BOOL  {left}
    not BOOL              → BOOL
    "(" BOOL ")"          → BOOL  {bracket}
  variables
    Bool [0-9']* → BOOL
  priorities
    BOOL "|"BOOL → BOOL < BOOL "xor"BOOL → BOOL < BOOL "&"BOOL → BOOL
    < notBOOL → BOOL
equations


[B1]  true | Bool     = true
[B2]  false | Bool    = Bool
[B3]  true & Bool     = Bool
[B4]  false & Bool    = false
[B5]  not false       = true
[B6]  not true        = false
[B7]  true xor Bool   = not Bool
[B8]  false xor Bool  = Bool
```

Figure 1: ASF+SDF specification of the Booleans.

(b) All variables occuring in the left-hand side and right-hand side of a negative condition should occur in the left-hand side of the equation or in one of the preceeding positive conditions.

(c) It is not allowed to have new variables, i.e., variables not occurring in the left-hand side of the equation or in the preceeding conditions, on *both* sides of a positive condition.

3. Examples

A good way to get some feeling for μASF is to show some ASF+SDF specifications and their "derived" μASF equivalent.

3.1 Booleans

The first example for which we want to show how the translation from ASF+SDF to μASF can be performed is based on the ASF+SDF specfication of the Booleans.
The ASF+SDF specification of Figure 1 can be translated into the following μASF code. The names, such as `true`, `or`, etc., used in this specification are abbreviations of the names which will be generated if this translation is performed automatically.

```
module Booleans
signature
```

```
  false {constructor};
  true {constructor};
  and(_,_);
  or(_,_);
  xor(_,_);
  not(_)
rules
  [B1] or(true,Bool)   = true;
  [B2] or(false,Bool)  = Bool;

  [B3] and(true,Bool)  = Bool;
  [B4] and(false,Bool) = false;

  [B5] not(false)      = true;
  [B6] not(true)       = false;

  [B7] xor(true,Bool)   = not(Bool);
  [B8] xor(false,Bool)  = Bool
```

Note that syntactic issues like associativity and priorities of context-free syntax rules disappear because they play no role in the semantics of ASF+SDF.

3.2 List Manipulation

The second example is a bit more complicated. It is the ASF+SDF specification of a simplified version of a type environment used when specifying type checkers for programming languages. A number of illustrative examples of type checkers can be found in [DHK96]. The specification in Figure 2 shows the use of list matching, default equations, and positive and negative conditions.

For the ASF+SDF specification of Figure 2 the following μASF code is derived.

```
module Type-environment
signature
  "{list}"(_) {constructor,external};
  conc(_,_) {constructor,external};
  null {constructor,external};
  nil-type {constructor};
  pair(_,_) {constructor};
  type-env(_) {constructor};
  lookup(_,_);
  add-to(_,_,_)
rules
  [l-1] lookup(Id,type-env("{list}"(conc(*Pair1,
                                    conc(pair(Id,Type),
                                         *Pair2))))) = Type;
  [l-2] default: lookup(Id,Tenv) = nil-type;

  [at-1] add-to(Id,Type1,
                type-env("{list}"(conc(pair(Id,Type2),*Pair2)))) =
           type-env("{list}"(conc(pair(Id,Type1),*Pair2)));
  [at-2] Id1 != Id2 &
         add-to(Id1,Type1,"{list}"(*Pair1))) == type-env("{list}"(*Pair2))
         ==>
         add-to(Id1,Type1,
                type-env("{list}"(conc(pair(Id2,Type2),*Pair1)))) =
           type-env("{list}"(conc(pair(Id2,Type2),*Pair2)));
  [at-3] add-to(Id,Type,type-env("{list}"(null))) =
imports Layout
exports
  sorts ID
  lexical syntax
    [a-z][a-z0-9]* → ID
  sorts TYPE
  context-free syntax
    natural  → TYPE  {constructor}
    string   → TYPE  {constructor}
    nil-type → TYPE  {constructor}
  sorts TENV PAIR
  context-free syntax
    "(" ID ":" TYPE ")"        → PAIR   {constructor}
    "[" PAIR* "]"              → TENV   {constructor}

    lookup(ID, TENV)           → TYPE
    add-to(ID, TYPE, TENV)     → TENV
hiddens
  variables
    Id [0-9]*        → ID
    Type [0-9]*      → TYPE
    Pair [0-9]*      → PAIR
    Pair "*"[0-9]*   → PAIR*
    Tenv [0-9]*      → TENV
equations
```

[l-1] $\text{lookup}(\mathit{Id},\ [\mathit{Pair}_1^*\ (\mathit{Id} : \mathit{Type})\ \mathit{Pair}_2^*]) = \mathit{Type}$

[l-2] $\text{lookup}(\mathit{Id},\ \mathit{Tenv}) = \text{nil-type}$ **otherwise**

[at-1] $\text{add-to}(\mathit{Id},\ \mathit{Type}_1,\ [(\mathit{Id} : \mathit{Type}_2)\ \mathit{Pair}_2^*]) = [(\mathit{Id} : \mathit{Type}_1)\ \mathit{Pair}_2^*]$

[at-2] $$\frac{\begin{array}{rcl} \mathit{Id}_1 & \neq & \mathit{Id}_2, \\ \text{add-to}(\mathit{Id}_1,\ \mathit{Type}_1,\ [\mathit{Pair}_1^*]) & = & [\mathit{Pair}_2^*] \end{array}}{\text{add-to}(\mathit{Id}_1,\ \mathit{Type}_1,\ [(\mathit{Id}_2 : \mathit{Type}_2)\ \mathit{Pair}_1^*]) \ = \ [(\mathit{Id}_2 : \mathit{Type}_2)\ \mathit{Pair}_2^*]}$$

[at-3] $\text{add-to}(\mathit{Id},\ \mathit{Type},\ []) = [(\mathit{Id} : \mathit{Type})]$

Figure 2: ASF+SDF specification of a type environment.

```
          type-env("{list}"(pair(Id,Type)))
```

The function `null` represents the empty list and the function `conc` is used as an (associative) concatenation operator on lists. The variables prefixed by a star represent list variables. The function `{list}` is necessary to make a distinction between a list with only one element and an ordinary term, e.g., `{list}(a)` vs. `a` or `{list}(V)` vs. `V`.

4. Semantic Considerations

We will provide only an informal description of the (dynamic) semantics of μASF by means of examples. Clearly a lot can be said about the rewriting theory of μASF but for the purpose of writing a compiler this theory is less essential. We attempt to provide the basic intuitions only in order to give the compiler designer some anchors when developing transformations and a code generator. Furthermore, these examples can be used as test cases to check whether the generated code behaves as expected.

We restrict ourselves to the rewriting of closed terms. Rewriting of open terms in combination with default rules and negative conditions is problematic. The concept of list matching will not be addressed in this section, but it will be discussed in Section 5.

First of all, it must be stated that μASF will be based mainly on innermost rewriting (call-by-value). Innermost rewriting is a good choice for several reasons:

- Most users are familiar with call-by-value from C and other imperative languages.
- It facilitates compilation to and interfacing with C and other imperative languages.
- It simplifies the understanding of default rules (compare, e.g., [BBKW89]).
- Its behavior is more predictable than that of other strategies, an important consideration when rewrite systems become large.

Secondly, we wanted the notion of a normal form to be semi-decidable (which is already substantially more complicated than in the case of unconditional rewriting without priorities). So, for t to be in normal form we require a definitive failure of each legitimate attempt to rewrite t. Furthermore, we want to avoid semantic paradoxes. Consider, e.g., the following μASF specification:

```
module muS0
signature
  a
rules
  [1] a == a ==> a = a
```

If a is in normal form then `[1]` applies so a is not in normal form; if a is not in normal form then clearly `[1]` can be applied so $a == a$ has to succeed and hence a must be in normal form (assuming that the test $a == a$ is based on normalisation). This is a paradox unless we allow that a may not be in normal form and cannot be reduced at the same time.

We will escape from these paradoxes by imposing strict conditions on the cases where we will require a compiled specification to show a specified behaviour. In case of `muS0` an implementation must diverge on input a, in other words the evaluation of a will not terminate.

4.1 Classification of the Example Specifications

Quite a number of small examples will be presented in this paper to illustrate the various semantic issues of μASF and indirectly of ASF+SDF. In order to improve the accessibility of these examples we present a classification of the most important features to be discussed and the corresponding examples.

Simple equations S_1

(Simple) conditional evaluation S_2, S_4, S_6, S_{12},

Divergence due to recursion S_3, S_5, S_7, S_{10}

Non-proper choice S_8, S_{13}

Use of defaults S_9, S_{10}, S_{11}

Use of variables and defaults S_{15}, S_{16}, S_{17}

Use of assignment conditions S_{19}, S_{20}, S_{21}

Liberal rewriting and choice S_{22}, S_{23}, S_{24}, S_{26}, S_{27}

Liberal rewriting and normal forms S_{28}, S_{29}

Weak vs. strong normalisation S_{30}, S_{31}

List matching S_{32}, S_{33}

4.2 Informal Mathematical Notation

Specifications in μASF use a familiar catalogue of symbols but in some cases with a less familiar meaning. As an example consider the μASF specification

```
module muS1
signature
  a; b; c; f(_)
rules
  [1] a = b
  [2] f(a) = c
```

At first sight one might think that when reducing the term `f(a)` the rewrite `f(a)` $\Rightarrow$ `c` is used and as a consequence the identity `f(a) = c` follows from `muS1`. However, as stated above the intended meaning of μASF rests on innermost reduction. From that it follows that `f(b)` is the (unique) normal form of `f(a)`.

In order to avoid confusion we will write $t \rightrightarrows t'$ if t can be rewritten into t' in one or more steps, corresponding to the intended meaning of μASF. So, in our mathematical notation we will write the specification S_1 as

$$\begin{array}{lll} S_1: & \texttt{[1]} & a \rightrightarrows b \\ & \texttt{[2]} & f(a) \rightrightarrows c \end{array}$$

The equality sign also occurs in the conditions, it checks the syntactic equality of the normal forms obtained by reducing the terms in the left-hand side and the right-hand side. We will give three examples of its use.

First of all, consider:

$$\begin{array}{lll} S_2: & \texttt{[1]} & a \rightrightarrows b \\ & \texttt{[2]} & c \rightrightarrows b \\ & \texttt{[3]} & a = c \Rightarrow f(b) \rightrightarrows a \end{array}$$

Here we find that a and c have a common reduct b and so $f(b) \rightrightarrows a$ (and $a \rightrightarrows b$, so $f(b) \rightrightarrows b$).

Secondly, consider:

$$\begin{array}{lll} S_3: & \texttt{[1]} & a \rightrightarrows a \\ & \texttt{[2]} & a = b \Rightarrow c \rightrightarrows d \\ & \texttt{[3]} & e \rightrightarrows h \\ & \texttt{[4]} & e = f \Rightarrow g \rightrightarrows h \end{array}$$

In S$_3$ c has no normal form. The condition $a = b$ is evaluated by normalizing both a and b and then checking for syntactical identity of the normal forms. Applying rule `[2]` to c leads to a cycle (divergence). So rule `[2]` cannot be applied, but we do not say that c is a normal form. For that the rule must explicitly fail. This happens, for instance, with term g. Rule `[4]` fails because the condition can not be satisfied and hence g is in normal form. We see that in conditions an equality can either fail or succeed (or else diverge). Both for success and failure of a condition it is required that both its left- and right-hand side normalize.

Finally, consider:

$$\begin{array}{lll} \mathrm{S}_4: & \texttt{[1]} & a \rightharpoondown b \\ & \texttt{[2]} & c \rightharpoondown d \\ & \texttt{[3]} & e \rightharpoondown f \\ & \texttt{[4]} & f \rightharpoondown e \\ & \texttt{[5]} & a = c \wedge e = f \Rightarrow h \rightharpoondown d \end{array}$$

We are interested in h. It is in normal form when rule `[5]` fails. Now $a = c$ fails because $a \rightharpoondown b$ and $c \rightharpoondown d$ where b and d are different normal forms (no matching left-hand side). The connective $\wedge$ has to be evaluated from left to right. It is one of the McCarthy connectives. If evaluation of its first argument fails, its second argument is not evaluated. Following the notation of [BBR95] we write ${}_{\circ}\!\wedge$ for these occurrences of the conjunction. Now `[5]` fails in spite of the fact that e and f do not even have a normal form. So, S$_4$ will be rewritten with rule `[5]`:

$$\texttt{[5]} \quad a = c \;{}_{\circ}\!\wedge\; e = f \Rightarrow h \rightharpoondown d$$

4.3 Some Examples of Reductions and Normal Forms

Given the mathematical notation presented in Section 4.2 various semantic issues of μASF will be discussed. If necessary new notation will be introduced to deal with specific features.

Consider the specification S$_5$:

$$\mathrm{S}_5: \quad \texttt{[1]} \quad a = b \Rightarrow a \rightharpoondown b$$

In S$_5$ a has no normal form. Indeed normalization of a diverges because it contains itself as a proper subtask. A term is in normal form if each of its proper subterms is in normal form and all rules fail when for each rule an attempt is made to apply it.

$$\mathrm{S}_6: \quad \texttt{[1]} \quad e = f \Rightarrow b \rightharpoondown c$$

Here b is in normal form because $e = f$ evaluates to false.

Consider

$$\begin{array}{lll} \mathrm{S}_7: & \texttt{[1]} & a \rightharpoondown b \\ & \texttt{[2]} & c \rightharpoondown b \\ & \texttt{[2]} & a = c \Rightarrow b \rightharpoondown d \end{array}$$

The only normal form of this system is d. Normalisation of a and c leads to b, which requires the normalisation of a and c, hence it loops.

4.4 Normal Forms and Stable Reductions

In this section we will define a very restricted form of innermost rewriting based on stable reductions. A reduction is stable if at each step in the reduction process exactly one rule is applicable.

Regarding normal forms we use the following notations, which will be explained in more detail below: let $S = (\Sigma, E)$ be a specification, t a closed Σ-term. We say that:

$$t \in \mathrm{NF}_{\mathrm{stable}}(S)$$

(read "t is in stable normal form") if all rules of E explicitly fail on t (using stable reductions). Furthermore, we say that:

$$t \in \mathrm{HNF}_{\mathrm{stable}}(S)$$

where HNF stands for "has normal form" and stable refers to the requirement that t has a so-called stable reduction path to $t' \in \mathrm{NF}_{\mathrm{stable}}(S)$. A reduction path is stable if at no stage it has a proper choice between two (or more) applicable rules. The choice is called proper if:

- either *all* rules are default or *all* rules are non-default, thus the rules must have the same priority.
- application leads to different terms.

We write $S : t \Rightarrow_{\mathrm{stable}} t'$ if t reduces in a stable and innermost way to t'. When checking that conditions fail in order to prove $t \in \mathrm{NF}_{\mathrm{stable}}(S)$, the conditions must allow normalisation in a stable fashion.

We will illustrate these notions by some further examples:

$$\begin{array}{lll} \mathrm{S}_8 : & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & a \Rrightarrow c \end{array}$$

then $b, c \in \mathrm{NF}(\mathrm{S}_8)$, so $b, c \in \mathrm{HNF}_{\mathrm{stable}}(\mathrm{S}_8)$. But $a \notin \mathrm{HNF}_{\mathrm{stable}}(\mathrm{S}_8)$. Indeed right from the start there are two options to reduce a, leading to different results. So $\mathrm{S}_8 : a \Rightarrow_{\mathrm{stable}} b$ is not true.

$$\begin{array}{lll} \mathrm{S}_9 : & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & \texttt{default} : a \Rrightarrow c \end{array}$$

In this case the default rule has lower priority, so $\mathrm{S}_9 : a \Rightarrow_{\mathrm{stable}} b$ is true.

$$\begin{array}{lll} \mathrm{S}_{10} : & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & c \neq c \Rightarrow a \Rrightarrow a \\ & \texttt{[3]} & \texttt{default} : c \Rrightarrow c \\ & \texttt{[4]} & \texttt{default} : a \Rrightarrow c \end{array}$$

In S_{10} the only stable reduction will be $a \Rightarrow b$ by means of rule [1] if the condition of rule [2] fails (rule [2] can be applied, because its left-hand side matches, and it has the same priority). But checking the condition leads to a cycle in the evaluation of c. A consequence of the fact that the evaluation of rule [2] diverges is that a has no stable reduct and indeed no stable normal form at all.

$$\begin{array}{lll} \mathrm{S}_{11} : & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & \texttt{default} : c = c \Rightarrow a \Rrightarrow a \\ & \texttt{[3]} & c \Rrightarrow c \\ & \texttt{[4]} & \texttt{default} : a \Rrightarrow c \end{array}$$

Now $\mathrm{S}_{11} : a \Rightarrow_{\mathrm{stable}} b$ and $b \in \mathrm{NF}_{\mathrm{stable}}(\mathrm{S}_{11})$. The non-default rules are examined first. Only if each of those fails default rules need to be applied. So, the implicit divergence of rule [2] is now avoided.

$$\begin{array}{lll} \mathrm{S}_{12} : & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & e = f \Rightarrow a \Rrightarrow a \\ & \texttt{[3]} & e \Rrightarrow g \\ & \texttt{[4]} & f \Rrightarrow h \end{array}$$

Now $S_{12} : a \Rightarrow_{\text{stable}} b$ and $b \in NF_{\text{stable}}(S_{12})$. This caused by the fact that the evaluation of rule [2] fails because $S_{12} : e \Rightarrow_{\text{stable}} g \in NF_{\text{stable}}(S_{12})$ and $S_{12} : f \Rightarrow_{\text{stable}} h \in NF_{\text{stable}}(S_{12})$ but $g \neq h$.

The following example demonstrates a non-stable reduction because during normalization of the condition a non proper choice between rule [3] and [4] occurs.

S_{13} : [1] $a \mathrel{=\!\!\rightharpoondown} b$
[2] $e = f \Rightarrow a \mathrel{=\!\!\rightharpoondown} c$
[3] $e \mathrel{=\!\!\rightharpoondown} g1$
[4] $e \mathrel{=\!\!\rightharpoondown} g2$
[5] $g1 \mathrel{=\!\!\rightharpoondown} g$
[6] $g2 \mathrel{=\!\!\rightharpoondown} g$
[7] $f \mathrel{=\!\!\rightharpoondown} h$

We see that refuting [2] for rewriting a (in order to allow application of [1]) invokes the normalization of e. But e has no stable normal form, both [3] and [4] can be applied. So, e cannot be normalized in a stable manner. For the purpose of stable normalization refuting [2] fails and application of [1] is not allowed. So the requirement of stable normalization includes that all normalizations performed to either accept or reject conditions of rules must be stably normalizing as well.

If we mark, for instance, [3] as default thus obtaining S_{14} the situation changes and $a \in HNF_{\text{stable}}(S_{14})$.

Variables Sofar we have restricted ourselves to rewrite rules which only contains constants. The equations in μASF specifications contain normally variables as well. The rest of this section is devoted to the definition of stable reduction in combination with variables, conditions, and defaults. We will introduce the notion of assignment conditions which is a condition in which on one side a new variable is introduced.

First, we consider the effect of variables.

S_{15} : [1] $a \mathrel{=\!\!\rightharpoondown} c$
[2] `default` : $f(X) \mathrel{=\!\!\rightharpoondown} b$
[3] $f(a) \mathrel{=\!\!\rightharpoondown} d$

The first example of the use of variables is rather straightforward. We observe $S_{15} : f(a) \Rightarrow_{\text{stable}} f(c) \Rightarrow_{\text{stable}} b \in NF_{\text{stable}}(S_{15})$.

In S_{16} we observe that whenever variables instantiated with some value, they are instantiated with this value for the entire scope of the rule.

S_{16} : [1] $a \mathrel{=\!\!\rightharpoondown} c$
[2] `default` : $f(X) \mathrel{=\!\!\rightharpoondown} b$
[3] $X \neq c \Rightarrow f(X) \mathrel{=\!\!\rightharpoondown} d$

We find that $S_{16} : f(a) \Rightarrow_{\text{stable}} f(c) \Rightarrow_{\text{stable}} b$, because the condition $c \neq c$ of rule [3] fails. The variable `X` gets the value c after stable reduction of a, therefore the condition of rule [3] fails.

S_{17} : [1] $a \mathrel{=\!\!\rightharpoondown} c$
[2] `default` : $f(X) \mathrel{=\!\!\rightharpoondown} b$
[3] $X \neq e \Rightarrow f(X) \mathrel{=\!\!\rightharpoondown} d$
[4] $e \mathrel{=\!\!\rightharpoondown} e$

Now $f(a) \notin HNF_{\text{stable}}(S_{17})$ because refuting [3] diverges on the normalisation of e. If we change S_{17} to S_{18} by making [3] default instead of [2] then we observe that $a \in HNF_{\text{stable}}(S_{18})$ because [2] has higher priority and [3] need not be refuted in this case.

Assignment Conditions So-called *assignment conditions* have to find unifying substitutions. In our mathematical notation a new operator is introduced to indicate that in the right-hand side of such a positive condition new variables are introduced. Note that a general unification mechanism is not needed because new variables occur only at one side of the condition (recall that they may not occur in both sides of a condition simultaneously, see Section 2). Consider the following μASF specification

```
module muS19
signature
  c; f(_); g(_); h(_); e(_)
rules
  [1] f(X) == g(Z) ==> h(X) = e(Z);
  [2] f(X) = g(c)
```

The variable `Z` in the condition of rule `[1]` is a new variable. In our mathematical notation this specification looks like:

$$\begin{array}{lll} \mathrm{S}_{19}: & \texttt{[1]} & f(X) =: g(Z) \Rightarrow h(X) \mathrel{\overline{\rightharpoondown}} e(Z) \\ & \texttt{[2]} & f(X) \mathrel{\overline{\rightharpoondown}} g(c) \end{array}$$

When normalising $h(a)$ we find: $\mathrm{S}_{19} : f(a) \Rightarrow_{\text{stable}} g(c) \in \mathrm{NF}_{\text{stable}}(\mathrm{S}_{19})$ and the match $Z = c$ is found. This leads to an assignment of c to Z that can be used subsequently (in the same reduction). So, $\mathrm{S}_{19} : h(a) \Rightarrow_{\text{stable}} e(c) \in \mathrm{NF}_{\text{stable}}(\mathrm{S}_{19})$.

$$\begin{array}{lll} \mathrm{S}_{20}: & \texttt{[1]} & f(X) =: g(h(Z)) \Rightarrow g(X) \mathrel{\overline{\rightharpoondown}} e(Z) \\ & \texttt{[2]} & f(X) \mathrel{\overline{\rightharpoondown}} g(c) \\ & \texttt{[3]} & \texttt{default} : g(X) \mathrel{\overline{\rightharpoondown}} c \end{array}$$

During the rewriting of $g(a)$ we have to rewrite $f(a)$, the left hand side of the condition of rule `[1]`, by means of rule `[2]`. The right-hand side of an assignment condition will not be normalized, because we do not reduce open terms. For the reduction of $g(c)$ the condition of `[1]` must be evaluated again, so divergence. Thus $g(a)$ has no stable normal form.

$$\begin{array}{lll} \mathrm{S}_{21}: & \texttt{[1]} & X =: f(Y) \,{}_{s}\!\wedge\, h(Y) = h(Y) \Rightarrow g(X) \mathrel{\overline{\rightharpoondown}} c \\ & \texttt{[2]} & a \mathrel{\overline{\rightharpoondown}} b \\ & \texttt{[3]} & h(b) \mathrel{\overline{\rightharpoondown}} h(b) \end{array}$$

$g(f(a))$ has no normal form because Y will be bound to b, but the test $h(b) = h(b)$ does not succeed because the normalisation of $h(b)$ diverges. If `[3]` is replaced by

$$\begin{array}{ll} \texttt{[3]} & h(b) \mathrel{\overline{\rightharpoondown}} d_1 \\ \texttt{[4]} & h(b) \mathrel{\overline{\rightharpoondown}} d_2 \end{array}$$

then the absence of stability causes a problem, which in turn implies that $g(f(a))$ has no stable normal form.

4.5 Liberal Rewriting

The notion of stable rewriting provides a well-defined semantics, however this semantics is too restricted to be useful in case of μASF. We will relax some restrictions of stable rewriting and introduce the notion of liberal rewriting. Liberal rewriting allows application of a rule if its left-hand side matches and all conditions are satisfied (and, in case it is a default, if no ordinary rule applies). So, it is allowed to have more than one rewrite rule at the same priority level that matches a term. Consider the following example:

$$\begin{array}{lll} \mathrm{S}_{22}: & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & a \Rrightarrow c \end{array}$$

The term a has a liberal rewrite to both b and c. These reductions are not stable in the sense of Section 4.4.

$$\begin{array}{lll} \mathrm{S}_{23}: & \texttt{[1]} & a = b \Rightarrow c \Rrightarrow c \\ & \texttt{[2]} & \texttt{default}: c \Rrightarrow e \\ & \texttt{[3]} & \texttt{default}: c \Rrightarrow f \end{array}$$

In S_{23} c has a liberal rewrite to both e and f, because the condition $a = b$ fails and this prevents the divergence of [1].

$$\begin{array}{lll} \mathrm{S}_{24}: & \texttt{[1]} & a = b \Rightarrow c \Rrightarrow d \\ & \texttt{[2]} & a \Rrightarrow a \\ & \texttt{[3]} & c \Rrightarrow e \end{array}$$

In S_{24} c has a liberal rewrite to e. Obtain S_{25} by changing [3] into a default rule. Then in S_{25} c has no liberal rewrite to e because the default [3] can only be applied if [1] has been rejected first (and this causes a divergence).

We write $S : t \Rightarrow_{\text{liberal}} t'$. It should be noticed that the notion of a normal form itself depends on whether liberal or stable rewriting is used. Consider

$$\begin{array}{lll} \mathrm{S}_{26}: & \texttt{[1]} & a = b \Rightarrow c \Rrightarrow d \\ & \texttt{[2]} & a \Rrightarrow b_1 \\ & \texttt{[3]} & a \Rrightarrow b_2 \\ & \texttt{[4]} & b_1 \Rrightarrow e \\ & \texttt{[5]} & b_2 \Rrightarrow e \end{array}$$

Here using liberal rewriting a normalizes to e which differs from b. Therefore [1] does not apply and c is a normal form. It follows that we do not have $\mathrm{S}_{26} : c \Rightarrow_{\text{liberal}} d$ and thus $c \in \mathrm{NF}_{\text{liberal}}(\mathrm{S}_{26})$. a can be rewritten to b_1 and b_2, thus a has no stable normal form.

The use of variables in the left-hand side of rewrite rules may be a source of unstable rewriting. Instability often has the following form:

$$\begin{array}{lll} \mathrm{S}_{27}: & \texttt{[1]} & f(a) \Rrightarrow b \\ & \texttt{[2]} & f(X) \Rrightarrow c \end{array}$$

thus $f(a) \notin \mathrm{HNF}_{\text{stable}}(\mathrm{S}_{27})$. In case of liberal rewriting the specification is not problematic.

Liberal Normal Form Stable rewriting is too restricted to serve as semantic basis for μASF, but the notion of a stable normal form is well-defined. Whereas the notion of a liberal normal form is not obvious. Consider the following specification:

$$\begin{array}{lll} \mathrm{S}_{28}: & \texttt{[1]} & a \Rrightarrow b \\ & \texttt{[2]} & a \Rrightarrow c \\ & \texttt{[3]} & a = b \Rightarrow e \Rrightarrow f \end{array}$$

Is e a liberal normal form in S_{28}? If we evaluate $a = b$ by reducing a to c by [2] then the condition fails and e emerges as a normal form. But if [1] is chosen then the condition succeeds and e is not a normal form. So e is both a normal form and not.

In liberal rewriting a term t has a normal form when all rules have been tried (respecting defaults) and all conditions have been refuted in (some!) liberal way. We write $\mathrm{S} : t \Rightarrow_{\text{liberal}} t' \in \mathrm{NF}_{\text{liberal}}(\mathrm{S})$ if a liberal rewrite of t to t' has been found and a liberal proof that t' is normalized. We notice that $\mathrm{S} : t \Rightarrow_{\text{liberal}} t' \in \mathrm{NF}_{\text{liberal}}(\mathrm{S})$ does not exclude $\mathrm{S} : t' \Rightarrow_{\text{liberal}} t'' \in \mathrm{NF}_{\text{liberal}}(\mathrm{S})$ (cf. S_{28}).

Consider the following example:

$$\begin{array}{lll} \mathrm{S}_{29}: & \texttt{[1]} & a \rightharpoonup b \\ & \texttt{[2]} & a \rightharpoonup c \\ & \texttt{[3]} & a = b \Rightarrow e \rightharpoonup f \\ & \texttt{[4]} & a \neq b \Rightarrow f \rightharpoonup e \end{array}$$

Now $\mathrm{S}_{29} : e \Rightarrow_{\text{liberal}} f \in \mathrm{NF}_{\text{liberal}}(\mathrm{S}_{29})$. This is found if [3] is applied to e with a normalised via [1]. Rewrite rule [4] will fail if [1] is used again to rewrite a. Similarly we have $\mathrm{S}_{29} : f \Rightarrow_{\text{liberal}} e \in \mathrm{NF}_{\text{liberal}}(\mathrm{S}_{29})$ (by choosing [2] to evaluate the left-hand side of the condtions).

Fortunately, if $t \in \mathrm{NF}_{\text{stable}}(\mathrm{S})$ then $t \in \mathrm{NF}_{\text{liberal}}(\mathrm{S})$ and for no $t' \neq t : t \Rightarrow_{\text{liberal}} t'$ (so liberal rewriting cannot perform another step). So stable normal forms are detected by liberal rewriting without false positives.

Our example S_{29} explains our preference for stable normal forms as the semantic foundation, however. In any case, without further precautions the notion of liberal normal form leads to circularities.

4.6 Semi-Correctness of Liberal Rewriting

Uniqueness of stable rewriting: if $\mathrm{S} : t \Rightarrow_{\text{stable}} t'$ and $\mathrm{S} : t \Rightarrow_{\text{stable}} t''$ then $t' \equiv t''$.

If $\mathrm{S} : t \Rightarrow_{\text{stable}} t'$ and $t' \in \mathrm{NF}(\mathrm{S})$ then (a) $\mathrm{S} : t \Rightarrow_{\text{liberal}} t'$ and (b) if $\mathrm{S} : t \Rightarrow_{\text{liberal}} t''$ and $t'' \in \mathrm{NF}(\mathrm{S})$ then $t' \equiv t''$.

We may infer that liberal normalisation will never miss a stable normal form if it exists.

4.7 Stability Conditions

We define a number of stability conditions on a specification S to improve (ensure) the correctness of a specification with respect to liberal normalizations with stable normal forms:

- Non-overlapping property.
- Left to right condition filtering property.
- Syntactic condition filtering property.

A stability condition is a condition which implies that all liberal normalisations are in fact stable.

The first stability condition is the absence of top level unifications for left-hand sides of rules. So, if for each pair of rules R_i, R_j $(i \neq j)$ of the same priority $R_i : C_i \Rightarrow l_i \rightharpoonup r_i$ and $R_j : C_j \Rightarrow l_j \rightharpoonup r_j$ the left-hand sides l_i and l_j cannot be unified, we call such a system non-overlapping. Notice that for μASF the *non-overlapping property* is weaker in general, because for innermost rewriting only root overlaps play a role.

Of course conditions can also play a role, e.g.:

$$\begin{array}{l} eq(X, h(Y)) = true \Rightarrow g(X, Y) \rightharpoonup 0 \\ eq(X, h(Y)) = false \Rightarrow g(X, Y) \rightharpoonup f(Y, X) \end{array}$$

Suppose the left-hand sides of two rules with the same priority R_i and R_j are identical. In that case the conditions must be used to discard at least one of them. We say that S satisfies *left to right syntactic condition filtering* if for all such pairs R_i, R_j the following holds. Let

$$\begin{array}{l} R_i : C_0^i \mathbin{\&\!\wedge} \ldots \mathbin{\&\!\wedge} C_p^i \Rightarrow l_i \rightharpoonup r_i \\ R_j : C_0^j \mathbin{\&\!\wedge} \ldots \mathbin{\&\!\wedge} C_q^j \Rightarrow l_j \rightharpoonup r_j \end{array}$$

then for some $m \leq p, q$ $C^i_0 \equiv C^j_0, \ldots, C^i_m \equiv C^j_m$ and C^i_{m+1} is the negation of C^j_{m+1}. In more detail: for some t_1, t_2: $C^i_{m+1} \equiv t_1 = t_2$ and $C^i_{m+1} \equiv t_1 \neq t_2$ or $C^i_{m+1} \equiv t_1 \neq t_2$ and $C^i_{m+1} \equiv t_1 = t_2$.

A slightly more liberal condition is obtained if one may first permute the conditions (as long as this is syntactically correct) as well as the internal order ($t_1 = t_2$ to $t_2 = t_1$ and $t_1 \neq t_2$ to $t_2 \neq t_1$) of the non-matching conditions. Note that $f(X) =: f(Y) \mathbin{{}_{\circ}\!\wedge} h(Y) = u(X) \Rightarrow l(X) \rightrightarrows r(X)$ is syntactically correct whereas $h(Y) = u(X) \mathbin{{}_{\circ}\!\wedge} f(X) =: f(Y) \Rightarrow l(X) \rightrightarrows r(X)$ is not. When permuting conditions it is not allowed to place assignment conditions before conditions that use variables introduced by that assignment condition. We call this constraint on S the *syntactic condition filtering property.*

The non-overlapping property, the left to right syntactic condition filtering property and the syntactic condition filtering property are the three stability conditions that we have found to be useful in practice.

4.8 Weak and Strong Normalisation for Liberal Rewriting

First notice that for stable reduction weak and strong normalisation coincide. We say that $t \in \mathrm{SN}_{\mathrm{liberal}}(S)$ if all liberal reduction paths of t terminate with some liberal normal form.

If some liberal reduction(s) of t lead(s) to a liberal normal form we write $t \in \mathrm{WN}_{\mathrm{liberal}}(S)$. Here WN stands for weakly normalising. Of course $\mathrm{WN}_{\mathrm{liberal}}(S) = \mathrm{HNF}_{\mathrm{liberal}}(S)$ holds. Recall that HNF stands for "has normal form". The intuitions behind the distinction between $\mathrm{SN}_{\mathrm{liberal}}(S)$ and $\mathrm{WN}_{\mathrm{liberal}}(S)$ is now clarified by means of examples S_{30} and S_{31}.

$\mathrm{S}_{30}:$ [1] $a \rightrightarrows b$
[2] $a \rightrightarrows a$
[3] $b \rightrightarrows c$

$a \in \mathrm{WN}_{\mathrm{liberal}}(\mathrm{S}_{30})$ and $a \notin \mathrm{SN}_{\mathrm{liberal}}(\mathrm{S}_{30})$ because a can be normalized to c (via b) but its reduction may also loop (by repeating a).

$\mathrm{S}_{31}:$ [1] $a \rightrightarrows b$
[2] $a \rightrightarrows c$
[3] $e = f \Rightarrow c \rightrightarrows d$
[4] $f \rightrightarrows f$
[5] $f \rightrightarrows g$

$a \in \mathrm{WN}_{\mathrm{liberal}}(\mathrm{S}_{31})$ because $\mathrm{S}_{31} : a \Rightarrow_{\mathrm{liberal}} b$. But $a \notin \mathrm{SN}_{\mathrm{liberal}}(\mathrm{S}_{31})$ because $\mathrm{S}_{31} : a \Rightarrow_{\mathrm{liberal}} c$. Then normalization of c invokes normalization of e and f (condition of [3]). The normalisation of f may (but need not) diverge.

5. List Matching

One of the most powerful features of ASF+SDF is list matching. It allows very concise specifications, e.g., removing multiple elements from a set can be specified in only one equation. Consider the following μASF specification:

```
module Sets
signature
  "{list}"(_) {external};
  conc(_,_) {external};
  set(_)
rules
  [s-1] set("{list}"(conc(*Els1,conc(El1,conc(*Els2,conc(El2,*Els3)))))) =
          set("{list}"(conc(*Els1,conc(El1,conc(*Els2,*Els3)))))
```

Another example of the use of list matching in ASF+SDF is given in Figure 2. List matching may involve backtracking. However, backtracking is restricted to the scope of the rewrite rule in which the list matching occurs. So, if the right-hand side of a rewrite rule containing a list matching pattern is executed, backtracking to find another match is no longer possible.

5.1 List Matching in More Detail

In the left-hand side of a context-free grammar rule in ASF+SDF the sorts $S*$ and $S+$ can be used if sort S is defined. The sorts $S*$ and $S+$ can only be applied as source but not as target in a context-free grammar rule. This simplifies the type structure at hand, but this restriction is not essential. If one is interested in, e.g., function $f : D \Rightarrow S$ (with D of the form $U_1 \times \ldots \times U_n$ and S of the form S_i* or S_i+) it is possible introduce a sort LS, a function $list : S* \Rightarrow LS$ and specify $lf : T \Rightarrow LS$ with $lf(\vec{X}) = list(f(\vec{X}))$. In further specifications lf is used instead of f.

Variables may be defined for lists and non-empty lists ($*$-lists or $+$-lists). We use the "," as a separator for lists elements instead of the concatenation function `conc` used in μASF. Equation `[1-1]` of the μASF specification of the type environments presented in Section 3.2 would in our mathematical notation become:

$$\texttt{[1-1]}\, lookup(I, type\text{-}env(list(*P1, pair(I,T), *P2))) \rightrightarrows T$$

Consider now the rule $f(list(*X, a, f(Z), +Y)) \rightrightarrows g(list(+Y, b, *X), Z)$ with $*X : S*, +Y : S+$. We consider various matches. Let

$$t_1 = f(list(b,b,a,f(d),c,c))$$
$$t_2 = f(list(a,f(d),b,a,f(d)))$$
$$t_3 = f(list(a,f(d),b,a,f(d),b))$$

We notice that t_1, t_2, and t_3 all match with the pattern $list(*X, a, f(Z), +Y)$. The term t_3 matches with $list(*X, a, f(Z), +Y)$ in two different ways:

- $*X = \varepsilon$, $Z = d$, and $+Y = b, a, f(d), b$
- $*X = a, f(d), b$, $Z = d$, and $+Y = b$

This kind of matching is called associative list matching. (The operator , is associative.)

Given a match of a term t with a left-hand side L of a rewrite rule one finds a binding of L's variables to (closed) subexpressions of t just as in the case of μASF terms without lists. Notice, however, that assignment conditions may introduce list matching as well.

We are faced with two tasks:

1. To extend notions of stable and liberal rewriting to deal with list matching in μASF.
2. To extend the three stability criteria.

5.2 Stable and Liberal Rewriting with Lists

Let $list(\vec{X})$ be a term with list variables, and suppose it matches t in at least two different ways. Then there is an ordering between these matches. Let $X_1, X_2, \ldots, X_k$ be a listing of the list variables of $list(\vec{X})$ in the order of first appearance in $list(\vec{X})$ reading from left to right. A match between $list(\vec{X})$ and t can be characterised by the length of the list that is found for each of the X_i. (Of course this length must exceed 0 if X_i is a $+$-variable).

We then represent each match with a sequence of non-negative integers: our ordering is just the lexicographic ordering on these sequences.

First, during the evaluation of conditions the existence of multiple matches in an assignment condition need not be a problem: the rewriting strategy must take the lexicographically first match, other matches are not allowed. Next, we consider the case that for a rule $C_1 \,\mathring{\wedge}\, \ldots \,\mathring{\wedge}\, C_n \Rightarrow l \rightrightarrows r$ there are multiple matches between the left-hand side l and term t containing a list. Here our rewriting strategy must look for the lexicographically first list match that satisfies the conditions. If one of the conditions can not be satisfied, we backtrack to the last list matching pattern in the conditions or left-hand side of the rewrite rule and try the next lexicographically first match. If all possibilities for all list matching patterns are tried in this way, the rule fails. This backtracking discipline in combination with the lexicographically first match leads to a deterministic behaviour of list matching[1].

So, for a list match to be accepted not only must *all* conditions succeed but it must be the lexicographically first match for which they succeed. This is a matter of strategy unrelated to the stability issue. Rewriting in a liberal way one can also look for the first match. The outcome of that search however may depend on decisions taken during rewriting. Consider, e.g., the following specification:

$$
\begin{array}{lll}
\mathrm{S}_{32}: & \texttt{[1]} & g(X) = Y \Rightarrow f(\mathit{list}(X, Y, *Z)) \rightrightarrows \mathit{list}(Y) \\
 & \texttt{[2]} & g(a) \rightrightarrows b \\
 & \texttt{[3]} & g(a) \rightrightarrows c
\end{array}
$$

where the variables X and Y are variables of sort S and $*Z$ is a list variable of sort $S*$. Given the term $t = f(\mathit{list}(a, b, c))$ the normal form is either $\mathit{list}(c)$ when rule `[2]` was used to normalize $g(a)$ or $f(\mathit{list}(a, b, c))$ when rule `[3]` was used instead. In both cases the lexicographically first match was chosen.

5.3 Stability Conditions

Now stability is just as in the case without lists. Given a term there must not be two different applicable rules (of the same priority) leading to different results. The case of stable versus liberal rewriting is just the same in the list matching case, assuming that the strategy looks strictly for the lexicographically first match that satisfies the conditions.

It is essential that liberal rewriting looks for the lexicographically first match, otherwise one cannot guarantee the soundness and single-valuedness of liberal rewriting in case a stable normal form exists. This is obvious because stable rewriting uses the search for the lexicographically first match.

If we drop the search for a lexicographically first match we must define stable rewriting by requiring that at most one match (satisfying the conditions as well) can be found. This is a very strict requirement however that unnecessarily restricts the number of specifications that exhibit stable rewriting.

We consider an example:

$$
\begin{array}{lll}
\mathrm{S}_{33}: & \texttt{[1]} & f(\mathit{list}(*X, a, b, *Y, a)) \rightrightarrows \mathit{list}(b, *Y) \\
 & \texttt{[2]} & f(\mathit{list}(*X, b, b, *Y)) \rightrightarrows \mathit{list}(*Y, a) \\
 & \texttt{[3]} & \texttt{default}: f(\mathit{list}(*X, a, *Y, a)) \rightrightarrows \mathit{list}(a, *Y)
\end{array}
$$

where the variables $*X$ and $*Y$ are list variables of sort $S*$. We consider the terms

$$
\begin{array}{l}
t_1 = f(\mathit{list}(a, b, a, b, a, a)) \\
t_2 = f(\mathit{list}(a, b, b, a)) \\
t_3 = f(\mathit{list}(a, a, a))
\end{array}
$$

$\mathrm{S}_{33} : t_1 \Rightarrow_{\mathrm{stable}} \mathit{list}(b, a, b, a) \in \mathrm{NF}(\mathrm{S}_{33})$. (There are two possible matches: $*X = \varepsilon$, $*Y = a, b, a$ and $*X = a, b$, $*Y = a$. The first match has been selected).
$\mathrm{S}_{33} : t_2 \not\Rightarrow_{\mathrm{stable}}$ because both [1] and [2] apply.
$\mathrm{S}_{33} : t_2 \Rightarrow_{\mathrm{liberal}} \mathit{list}(b) \in \mathrm{NF}(\mathrm{S}_{33})$ via [1].
$\mathrm{S}_{33} : t_2 \Rightarrow_{\mathrm{liberal}} \mathit{list}(a, a) \in \mathrm{NF}(\mathrm{S}_{33})$ via [2].
$\mathrm{S}_{33} : t_3 \Rightarrow_{\mathrm{stable}} \mathit{list}(a * a) \in \mathrm{NF}(\mathrm{S}_{33})$, using the default rule [3] with $X = \varepsilon$ and $Y = a$.

[1]In a previous version list matching was non-deterministic [DHK96, p. 20]

6. Outermost versus Innermost Rewriting

Sofar we have restricted ourselves to innermost rewriting, because μASF and ASF+SDF are based on innermost rewriting. However, the evaluation of "if-then-else"-like rewrite rules may be more efficient if an outermost evaluation strategy is applied. The interpreter in the ASF+SDF Meta-Environment [Kli93] recognizes these types of rewrite rules and switches to an outermost evaluation strategy. In order to trigger the outermost evaluation in μASF an attribute `delaying` is introduced. The delaying attribute for function arguments prevents the innermost rewriting of the delayed arguments when such a function is called. During its evaluation only the delayed arguments actual needed are rewritten in an innermost way, see below for an example. Given this delaying mechanism it is possible to overrule the innermost strategy in a straightforward way. This can be used, e.g., to define an "if-then-else"-like rewrite rule in such a way that depending on the result of the evaluation of the expression either the "then"-part or the "else"-part is evaluated. Consider the following μASF specification:

```
module muCond1
signature
  t; f;
  if-then-else(_,_,_) {delaying(2),delaying(3)};
  f(_);
  g(_);
  h(_);
  a; b
rules
  [1] if-then-else(t,X,Y) = X;
  [2] if-then-else(f,X,Y) = Y;
  [3] g(X) = g(X);
  [4] f(X) = if-then-else(t,h(X),g(X));
  [5] h(X) = b
```

The attributes `delaying(2)` and `delaying(3)` indicate that the second and third argument of the `if-then-else` function are delayed. The normal form of `f(a)` is `b`, since rewrite rule `[3]` is never evaluated because the third argument of the `if-then-else` function is delayed. The `delaying` attributes are very helpful to reduce the number of rewrite steps or to prevent non-termination.

The combination of delayed arguments and non left-linear rules may lead to unexpected results. Consider the following specification

```
module muCond2
signature
  t; f;
  fff(_,_,_,_) {delaying(2),delaying(3),delaying(4)};
  f(_);
  g(_);
  h(_);
  a; b
rules
  [1] fff(t,X,Y,Z) = X;
  [2] fff(B,Y,Y,Z) = Y;
  [3] default: fff(B,X,Y,Z) = Z;
  [4] g(X) = a;
  [5] h(X) = a;
  [6] f(X) = fff(f,h(X),g(X),b)
```

Non left-linear rules are transformed into left-linear rules by replacing one of the identical variable names by a unique name and by introducing a condition to check the equality of the terms contained by these variables. So, rule `[2]` is transformed into:

```
  [2] Y == Y1 ==> fff(B,Y,Y1,Z) = Y;
```

However, the result of this transformation is that `Y` and `Y1` in the condition are no longer delayed and will be normalized. So, in the example above the normal form of `f(a)` will be `a` instead of the expected `b`.

7. Implementation Requirements

In order to execute ASF+SDF specifications, among other things an implementation of ASF+SDF should provide term rewriting. The requirements on term rewriting are described in this section.

Let IMP be an implementation of S and let t be a closed term from $\Sigma(S)$. We will formulate five implementation requirements on IMP. We view IMP as a partial operator which takes a specification S and a closed term t over the signature $\Sigma(S)$, which either returns some t' (we write $\text{IMP}_S(t) \Rightarrow t'$) or is undefined (we write $\text{IMP}_S(t) \Rightarrow D$). The five conditions are:

1. Completeness for stable reduction (CSR):
   If $t \in \text{HNF}_{\text{stable}}(\text{S})$ then for some t' over $\Sigma(S)$ $\text{IMP}_S(t) \Rightarrow t'$.
2. Soundness for stable reduction (SSR):
   If $t \in \text{HNF}_{\text{stable}}(\text{S})$ and $\text{IMP}_S(t) \Rightarrow t'$ then $\text{S} : t \Rightarrow_{\text{stable}} t' \in \text{NF}_{\text{stable}}(\text{S})$.
3. Completeness for strong termination with respect to liberal reduction (CSNLR):
   If $t \in \text{SN}_{\text{liberal}}(\text{S})$ then for some t' over $\Sigma(S)$ $\text{IMP}_S(t) \Rightarrow t'$.
4. Soundness for liberal rewriting (SLR):
   If $t \in \text{IMP}_S(t) \Rightarrow t'$ then t' is a closed term over $\Sigma(S)$ and $\text{S} : t \Rightarrow_{\text{liberal}} t' \in \text{NF}_{\text{liberal}}(\text{S})$.
5. Completeness for liberal rewriting (CLR):
   If $t \in \text{HNF}_{\text{liberal}}(\text{S})$ then $\text{IMP}_S(t)$ is defined.

Each of these requirements is about the algorithm of the implementation, disregarding space and time limitations. If IMP statifies CSR and SSR then it allows one to work safely with specifications that lead to stable rewriting. In practice this will cover many cases.

Moreover it is plausible to require that IMP satisfies CSNLR and SLR. This will be helpful in practice as well. It is not reasonable to expect that IMP satisfies CLR. There is no reasonable rewriting strategy that guarantees that a liberal normal form is found.

We notice that μASF has not been designed for process control applications and that there is little use for diverging computations.

In practice we expect an implementation to perform liberal rewriting because the stability check can be quite expensive. Furthermore it is often possible to use a specification format which guarantees that liberal and stable rewriting coincide.

7.1 The ASF+SDF *Compiler*

The current ASF+SDF compiler [BHKO00] implements a liberal rewriting strategy. Given the μASF representation of an ASF+SDF specification it generates C code. For each ASF+SDF function a separate C function is generated. The left-hand sides and possible conditions of the corresponding rewrite rules are transformed into a matching automaton. In case of instability caused by overlapping left-hand sides of rewrite rules the compiler disambiguates by means of a syntactic specificity ordering. The right-hand sides of the rewrite rules are directly translated into function calls except of course for constructor functions. The C code dealing with the default equations is, if necessary, executed after the code generated for the ordinary rewrite rules.

If during rewriting a left-hand side of a function matches and all conditions are satisfied, the "right-hand side" of the rewrite rule is executed by calling the functions except for constructor functions. Furthermore, divergence in the specification leads to non-termination during rewriting.

7.2 Consequences for the Specification Writer

What are the consequences of the preceeding considerations when writing ASF+SDF specifications? Stable rewriting is the best way to ensure correctness of your specification, but it is a very restricted way of writing specifications. There are hardly any ASF+SDF specifications which have no non-overlapping left-hand sides. In the case of liberal rewriting the best way to ensure correctness is to:

- Have as few overlapping rules as possible. If rules have to overlap conditions should be used to disambiguate them:
  - If two or more rules have exactly the same left-hand side then for each of these rewrite rules with a positive condition there should also be a rewrite rule with a negative condition and vice versa.
  - A default rule must be used to catch all missing cases.
- Ensure that the default rule is the most general rule, so be aware that default rules with matching constraints on the arguments or conditions may fail. Having more than one default rule for a function may also be a cause of unexpected errors.
- Provide constructor information whenever it is possible.
- The arguments of the outermost function in the left-hand side of rewrite rules should be constructor terms.
- Rewrite rules with the same outermost function symbol should be grouped in *one* module.
- Never use delaying arguments in combination with non linear left-hand sides. It is advisable not to have matching patterns on the delaying argument positions.

ACKNOWLEDGEMENTS
We thank Jan Heering, Paul Klint, and Pieter Olivier for the fruitful discussions on this subject and for reading draft versions of this paper.

References

[BBKW89] J. C. M. Baeten, J. A. Bergstra, J. W. Klop, and W. P. Weijland. Term rewriting systems with rule priorities. *Theoretical Computer Science*, 67:283–301, 1989.

[BBR95] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical Logics*, 5:199–217, 1995.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[BHKO00] M.G.J. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: the ASF+SDF compiler. Technical Report SEN-R0014, CWI, 2000.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

[Dik89] C. H. S. Dik. A fast implementation of the Algebraic Specification Formalism. Master's thesis, University of Amsterdam, Programming Research Group, 1989.

[FKW98] W. J. Fokkink, J. F. Th. Kamperman, and H. R. Walters. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Trans. Program. Lang. Syst.*, 20:679–706, 1998.

[Hen91] P. R. H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Most recent version available at URL: ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

[Kam96] J.F.Th. Kamperman. *Compilation of Term Rewriting Systems.* PhD thesis, University of Amsterdam, 1996.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. ftp://ftp.cwi.nl/pub/gipe/reports/KW93.ps.Z.

[Moo94] L. Moonen. A Virtual Assembler for an Abstract Machine – Design and Implementation of an Incremental and Retargetable Code Generator for Term Rewriting Systems. Master's thesis, Hogeschool Eindhoven, Department of Computer Science, 1994.

[Wal91] H. R. Walters. *On Equal Terms — Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

1. μASF Syntax Definition

The syntax definition of μASF consists of two parts. First the lexical syntax of the various identifiers, such as function names, variables, tag names, and module names, is defined and then the overall structure of a specification is given.

1.1 μLiterals

The module `μLiterals` defines the lexical syntax of μASF. First, it defines which characters should be considered as layout and the structure of comments. Comments are also considered as layout in SDF. Furthermore the lexical structure of the various identifiers occuring in μASF is defined, such as function names, `FunId`, variables names, `VarId`, module names `ModId`, and the names occurring in equation tags, `TagId`.

```
exports
  lexical syntax
    [␣\t\n]               → LAYOUT
    "%%"~[\n]*[\n]        → LAYOUT
    "%"~[\n%]+"%"         → LAYOUT
exports
  sorts  FunId ModId VarId TagId
  lexical syntax
    "\"~□                      → EscChar
    "\"[01][0-7][0-7]          → EscChar

    ~[\000-\037"\]             → QChar
    EscChar                    → QChar

    [a-z][\-_A-Za-z0-9′]*      → FunId
    "\""QChar*"\""             → FunId
    [′][!\-~]                  → FunId
    [\][0-2][0-9][0-9]         → FunId

    [A-Z][a-zA-Z0-9′_\−]*      → VarId

    [A-Z][a-zA-Z0-9.\−]*       → ModId

    [a-zA-Z0-9′_\−]*           → TagId
```

1.2 μASF

The syntax definition of μASF has a simple structure consisting of a module name, a signature, and a collection of rewrite rules.

```
imports MuLiterals Integers
exports
  sorts  FuncDef FunArg AttributeOpt Attribute
```

The structure of function definitions The signature of μASF consists of a set of operator or function declarations. Each function declaration consists of a function name and a list of arguments to indicate how many arguments a function has.

context-free syntax

FunId AttributeOpt	→ FuncDef
FunId “(” {FunArg “,”}+ “)” AttributeOpt	→ FuncDef
	→ AttributeOpt
“{” {Attribute “,”}+ “}”	→ AttributeOpt

The definition of arguments μASF is single-sorted, so there is no need to give the type of the arguments. We use `"_"` to represent a function argument in its declaration. A numerical value could also be used to give the arity of a function. A function can be declared “local” or “external”. The latter means that the corresponding rewrite rules are defined somewhere else.

context-free syntax

“_” → FunArg

The definition of attributes All functions are by default local. If a function is externally defined the attribute `external` *must* be added. The attribute `delaying` followed by a number (≥ 1) indicates that the specified argument is protected against innermost rewriting. See Section 6 for more details. The attribute `constructor` indicates that the corresponding function is a free constructor in the sense that it may not be used as outermost function symbol in the left-hand side of a rewrite rule. The constructor information can be automatically derived given a non-modular specification. However, in the case of ASF+SDF modules it is more complicated to derive. It is therefore possible to indicate explicitly which functions are constructors.

context-free syntax

“external”	→ Attribute
“delaying” “(” Int “)”	→ Attribute
“constructor”	→ Attribute

The definition of terms in μASF Variables prefixed with a star or plus are so-called list variables, they represent lists. For their usage we refer to Section 3.2.

sorts Var Term

context-free syntax

VarId	→ Var
“$*$” VarId	→ Var
“+” VarId	→ Var
Var	→ Term
FunId	→ Term
FunId “(” {Term “,”}+ “)”	→ Term

The definition of the conditions in μASF We distinguish two different types of conditions.

- Positive conditions, e.g., `X == Y`. This means that if X equals Y this condition succeeds.
- Negative conditions, e.g., `X != Y`. This means that if X equals Y this condition fails.

In negative conditions it is not allowed to introduce variables, that do not already occur in the left-hand side of the equation or in one of the preceeding conditions. In positive conditions, variables that do not already occur in the left-hand side of the equation or in one of the preceeding conditions, may only occur at one side of the condition.

exports

sorts Cond

context-free syntax

Term “$\equiv$” Term → Cond

Term “$\neq$” Term → Cond

*The definition of the rules in μ*ASF The language supports so-called default rules. These are applied when all other rules with the same outermost function symbol in the left-hand side have been tried. Hence, the default mechanism is used to introduce a priority level in the specification.

sorts Rule OptionalTag
context-free syntax

OptionalTag Term "=" Term	→ Rule
OptionalTag {Cond "&"}+ "⟹" Term "=" Term	→ Rule
OptionalTag "default:" Term "=" Term	→ Rule
OptionalTag "default:" {Cond "&"}+ "⟹" Term "=" Term	→ Rule
	→ OptionalTag
"[" TagId "]"	→ OptionalTag

*The definition of the complete module structure of μ*ASF A μASF specification consists of a module name, followed by a signature declaration, and a set of equations or rewrite rules. Variables are not explicitly declared because μASF is single-sorted. Variable names should always start with a capital.

sorts Module ModuleList RulesOpt SignatureOpt
context-free syntax

	→ SignatureOpt
"signature" {FuncDef ";"}+	→ SignatureOpt
	→ RulesOpt
"rules" {Rule ";"}+	→ RulesOpt
"module" ModId SignatureOpt RulesOpt	→ Module