

---

# ASF+SDF by example

2007-10-19 14:28:14 +0200 (Fri, 19 Oct 2007)

## Table of Contents

A simple evaluation function .....	1
Symbolic Differentiation .....	1
Sorting .....	2
Code Generation .....	3
Larger ASF+SDF Specifications .....	8

### Warning

This document is work in progress. It is in transition from The Meta-environment V1.5 to V2.0.

### Note

Here are some small examples of ASF+SDF specifications, which are selected to illustrate specific features of the formalism. Larger examples can be found in the online specifications and the ASF+SDF library.

## A simple evaluation function

Suppose we want to define a very simple evaluation function for the expressions as defined in [\link\\*{Simple Expressions}\[ Figure~\Ref\]{CODE:simple-prio}](#). The specification below shows how to do this.

### Example 1.1. ASF+SDF specification of a very simple eval function

```
module Eval

imports SimpleExpr
imports basic/Integers

exports
  context-free syntax
  eval(E) -> Integer

hiddens
  variables
    "E"[0-9]* -> E
    "N"[0-9]* -> NatCon

equations
  [e1] eval(E1 + E2) = eval(E1) + eval(E2)
  [e2] eval(E1 * E2) = eval(E1) * eval(E2)
  [e3] eval(N) = N
```

## Symbolic Differentiation

Computing the derivative of an expression with respect to some variable is a classical problem. Computing the derivative of  $X * (X + Y + Z)$  with respect to  $X$  gives:

$$d(X * (X + Y + Z)) / dX = X + Y + Z + X$$

Differentiation is defined in several stages in the specification below. First, the sorts `Nat` (natural numbers), `Var` (variables), and `Exp` (expressions) are introduced. Next, a differentiation operator of the form  $d E / d V$  is defined. Then, the differentiation rules are defined (equations [1]-[5]). Finally, some rules for simplifying expressions are given. As the above example shows, further simplification rules could have been added to collect multiple occurrences of a variable (giving  $2 * X + Y + Z$ ) or to compute constant expressions.

### Example 1.2. ASF+SDF specification for differentiation

```

module Diff

imports basic/Whitespace
imports basic/NatCon

exports
  context-free start-symbols Exp
  sorts Var Exp

  lexical syntax
    [XYZ]    -> Var

  context-free syntax
    NatCon          -> Exp
    Var             -> Exp
    Exp "+" Exp     -> Exp {left}
    Exp "*" Exp     -> Exp {left}
    "(" Exp ")"    -> Exp {bracket}
    "d" Exp "/" "d" Var -> Exp

  context-free priorities
    Exp "*" Exp -> Exp > Exp "+" Exp -> Exp

hiddens
  variables
    "N"      -> NatCon
    "V"[0-9]* -> Var
    "E"[0-9]* -> Exp

equations
  [ 1] dN/dV = 0
  [ 2] dV/dV = 1
  [ 3] V1 != V2
      =====>
      dV1/dV2 = 0
  [ 4] d(E1+E2)/dV = dE1/dV + dE2/dV
  [ 5] d(E1*E2)/dV = dE1/dV * E2 + E1 * dE2/dV
  [ 6] E + 0 = E
  [ 7] 0 + E = E
  [ 8] E * 1 = E
  [ 9] 1 * E = E
  [10] 0 * E = 0
  [11] E * 0 = 0

```

## Sorting

The use of list structures is illustrated by the specification of the *Dutch National Flag* problem presented below: given an arbitrary list of the colours red, white and blue, sort them in the order as they appear in the Dutch National Flag. We want:

```
{white blue red blue red white red} ->
{red red red white white blue blue}
```

In this specification, the list variables `Cs1` and `Cs2` permit a succinct formulation of the search for adjacent colours that are in the wrong order.

### Example 1.3. ASF+SDF specification for sorting

```
module Flag

imports basic/Whitespace

exports
  context-free start-symbols Flag
  sorts Color Flag

  context-free syntax
    "red"          -> Color
    "white"        -> Color
    "blue"         -> Color
    "{" Color+ "}" -> Flag

  hiddens
    variables
      "Cs"[0-9]* -> Color*
      "C"[0-9]*  -> Color

  equations

    [1] {Cs1 white red Cs2} = {Cs1 red white Cs2}

    [2] {Cs1 blue white Cs2} = {Cs1 white blue Cs2}

    [3] {Cs1 blue red Cs2}   = {Cs1 red blue Cs2}
```

## Code Generation

Consider a simple statement language (with assignment, if-statement and while-statement) and suppose we want to compile this language to the following stack machine code:

**Table 1.1. A simple stack language**

Operator>	Description
push $N$	Push the number $N$
rvalue $LOC$	Push the contents of data location $LOC$
lvalue $LOC$	Push the address of data location $LOC$
pop	Remove the top of the stack
copy	Push a copy of the top value on the stack
assign	The r-value on top of the stack is stored in the l-value below it and both are popped
add, sub, mul	Replace the two values on top of the stack by their sum (difference, product)
label $Lab$	Place a label (target of jumps)
goto $Lab$	Next instruction is taken from statement following label $Lab$
gotrue $Lab$	Pop the top value; jump if it is nonzero
gofalse $Lab$	Pop the top value; jump if it is zero

The statement:

```
while a do a := a - 1; b := a * c od
```

will now be translated to the following instruction sequence:

```
label xx ;
rvalue a ;
gofalse xxx ;
lvalue a ;
rvalue a ;
push 1 ;
sub ;
assign ;
lvalue b ;
rvalue a ;
rvalue c ;
mul ;
assign ;
goto xx ;
label xxx
```

Defining a code generator now proceeds in several steps that we describe in more detail.

**Basic Notions.** The specification below defines the sorts `Nat` (numbers) and `Id` (identifiers).

#### Example 1.4. ASF+SDF specification for BasicNotions

```
module BasicNotions
exports
  context-free start-symbols Nat Id
  sorts Nat Id

  lexical syntax
    [0-9]+      -> Nat
    [a-z][a-z0-9]* -> Id
```

**Expressions and Statements.** Given these basic notions the expressions and statements of our little source language are defined.

**Example 1.5. ASF+SDF specification for Expressions**

```

module Expressions

imports BasicNotions

exports
  context-free start-symbols Exp
  sorts Exp

  context-free syntax
    Nat          -> Exp
    Id           -> Exp
    Exp "+" Exp -> Exp {left}
    Exp "-" Exp -> Exp {left}
    Exp "*" Exp -> Exp {left}

  context-free priorities
    Exp "*" Exp -> Exp > {left: Exp "+" Exp -> Exp
                          Exp "-" Exp -> Exp}

```

**Example 1.6. ASF+SDF specification for Statements**

```

module Statements

imports Expressions

exports
  context-free start-symbols Stats
  sorts Stat Stats

  context-free syntax
    Id "!=" Exp          -> Stat
    "if" Exp "then" Stats "fi" -> Stat
    "while" Exp "do" Stats "od" -> Stat
    {Stat ";" }+        -> Stats

```

**Assembly Language.** The instructions of the assembly language for the stack machine are defined below.

**Example 1.7. ASF+SDF specification for AssemblyLanguage**

```

module AssemblyLanguage

imports BasicNotions
exports
  context-free start-symbols Instrs
  sorts Label Instr Instrs

lexical syntax
  [a-z0-9]+ -> Label
context-free syntax
  "push" Nat      -> Instr
  "rvalue" Id     -> Instr
  "lvalue" Id     -> Instr
  "assign"       -> Instr
  "add"          -> Instr
  "sub"          -> Instr
  "mul"          -> Instr
  "label" Label  -> Instr
  "goto" Label   -> Instr
  "gotrue" Label -> Instr
  "gofalse" Label -> Instr
  {Instr ";" }+  -> Instrs

```

**Label generation.** Next, we define a function to construct a next label given the previous one as follows. It is defined on the lexical notion of labels (`Label`). The scheme of appending the character `x` to the previous label is, of course, naive and will in real life be replaced by a more sophisticated one.

**Example 1.8. ASF+SDF specification for NextLabel**

```

module NextLabel

imports AssemblyLanguage

exports
  context-free syntax
    "nextlabel" "(" Label ")" -> Label

hiddens
  lexical variables
    "Char*" [0-9]* -> [a-z0-9]+

equations

[1] nextlabel(label(Char*)) = label(Char* x)

```

**Codegenerator.** It remains to define a function `tr` that translates statements into instructions. During code generation we should generate new label names for the translation of if- and while-statements. This is an instance of a frequently occurring problem: how do we maintain global information (in this case: the last label name generated)? A standard solution is to introduce an auxiliary sort (`Instrs-Lab`) that contains both the generated instruction sequence and the last label name generated so far. The SDF part and the ASF part of module `CodeGenerator` define the actual translation function.

**Example 1.9. ASF+SDF specification for CodeGenerator (SDF part)**

```
module CodeGenerator

imports Statements AssemblyLanguage NextLabel

exports
  context-free start-symbols Instrs

  context-free syntax
    tr(Stats) -> Instrs

hiddens
  sorts Instrs-lab
  context-free syntax
    Instrs # Label -> Instrs-lab

  context-free syntax
    tr(Stats, Label) -> Instrs-lab
    tr(Exp)          -> Instrs

hiddens
  variables
    "Exp"[0-9\']*      -> Exp
    "Id"[0-9\']*      -> Id
    "Instr"[0-9\']*   -> Instr
    "Instr-list"[0-9\']* -> {Instr ";" }+
    "Label"[0-9\']*   -> Label
    "Nat"[0-9\']*     -> Nat
    "Stat"[0-9\']*    -> Stat
    "Stat+"[0-9\']*  -> {Stat ";" }+
```

**Example 1.10. ASF+SDF specification for CodeGenerator (ASF part)**

```

equations

[1] <Instr-list, Label> := tr(Stat-list, x)
====>
tr(Stat-list) = Instr-list

[2] <Instr-list1, Label'> := tr(Stat, Label),
<Instr-list2, Label''> := tr(Stat-list, Label')
====>
tr(Stat ; Stat-list, Label) = <Instr-list1 ; Instr-list2, Label''>

[3] Instr-list := tr(Exp)
====>
tr(Id := Exp, Label) = <lvalue Id; Instr-list; assign, Label>

[4] Instr-list1 := tr(Exp),
<Instr-list2, Label'> := tr(Stat-list, Label),
Label'' := nextlabel(Label')
====>
tr(if Exp then Stat-list fi, Label) =
<Instr-list1; gofalse Label''; Instr-list2; label Label'', Label''>

[5] Instr-list1 := tr(Exp),
<Instr-list2, Label'> := tr(Stat-list, Label),
Label'' := nextlabel(Label'),
Label''' := nextlabel(Label'')
====>
tr(while Exp do Stat-list od, Label) =
<label Label''; Instr-list1; gofalse Label'''; Instr-list2;
goto Label''; label Label''', Label''>

[6] Instr-list1 := tr(Exp1),
Instr-list2 := tr(Exp2)
====>
tr(Exp1 + Exp2) = Instr-list1; Instr-list2; add

[7] Instr-list1 := tr(Exp1),
Instr-list2 := tr(Exp2)
====>
tr(Exp1 - Exp2) = Instr-list1; Instr-list2; sub

[8] Instr-list1 := tr(Exp1),
Instr-list2 := tr(Exp2)
====>
tr(Exp1 * Exp2) = Instr-list1; Instr-list2; mul

[9] tr(Nat) = push Nat

[10] tr(Id) = rvalue Id

```

## Larger ASF+SDF Specifications

There are a quite a few very large ASF+SDF specifications around that are good subjects of study:

- The asf2c compiler.

- Part of the parse table generator for SDF.
- The SDF checker.
- Various transformation system for improving COBOL programs.
- A system for detecting code smells in Java refactoring.
- Tooling for Action Semantics.
- Tooling for CASL.
- Tooling for ELAN.
- Analysis of C code (ASML).
- A compiler from UML diagrams to various target languages (Progress, Java, DB2).
- The syntax and type checking of a domain specific language for describing financial products.