
An explanation of error messages of ASF

Paul Klint
Jurgen Vinju

2007-10-18 23:00:33 +0200 (Thu, 18 Oct 2007)

Table of Contents

Introduction	1
See also	1
Static warnings and errors	1
Type check warnings for ASF+SDF	1
Type check errors for ASF+SDF	2
Common Syntax Errors in Specifications or Input Terms	3
Dynamic warnings and errors	6
Common Errors when Executing Specifications	6

Warning

This document is unfinished.

Introduction

This document describes the error messages that the ASF checker and the ASF interpreter or compiler may report. The first errors that ASF users run into are syntax errors. Since the syntax of most of an ASF program is defined by the user, the cause can always come from two ways. Either the definition of the syntax is wrong, or the use of the syntax is wrong. Being aware of this is the first lesson to learn when debugging ASF programs.

There is a static ASF checker that warns the programmer of the most common mistakes. Each warning or error is discussed in this document. A warning may be ignored, but most warnings tend to point to actual errors. An error indicates that an ASF program is guaranteed to not run correctly.

See also

Static warnings and errors

Type check warnings for ASF+SDF

- `Exported variables` section: it is good practice (but not enforced) to declare variables in a `hiddens` section.
- `Kernel syntax construction`: full SDF distinguishes three syntax sections: lexical (no layout may appear between symbols in a production), context-free (optional layout between all

symbols in a production), and kernel (layout between is explicitly indicated by the user). ASF+SDF does not support kernel syntax.

- Production renamings not supported: SDF does not only allow the renaming of sort names but also of complete productions. ASF+SDF does not support this.
- Not supported symbol: ASF+SDF imposes some restrictions on the symbols that are allowed. In particular, the symbols `XXX` are not allowed.
- Lexical probably intended to be a variable
- Deprecated condition syntax `"="`: in previous versions, the equality symbol in conditions was written as `=` instead of `==`.
- Constructor not expected as outermost function symbol of left hand side: a function defined with the `constructor` attribute is supposed to be irreducible and should not appear as outermost symbol on the left-hand side of an equation.

Type check errors for ASF+SDF

- Traversal attributes in non-prefix function: traversal functions can only be prefix functions.
- Illegal traversal attribute.
- Missing bottom-up or top-down attribute.
- Missing break or continue attribute.
- Missing trafo and/or accu attribute.
- Accu should return accumulated type.
- Trafo should return traversed type.
- Accutrafo should return tuple of correct types.
- Inconsistent arguments of traversal productions.
- Inconsistent traversal attributes.
- Asf equation sort must not be used.
- Charclasses not allowed in context-free syntax.
- Equations contain ambiguities: one or more equations can be parsed in more than one way. Use the graphical parse viewer to understand what is going on.
- Uninstantiated variable occurrence: a variable occurs in the right-hand side of a condition or of a complete equation but is has not been introduced earlier on in the equation.
- Negative condition introduces variable(s).
- Uninstantiated variables in both sides of condition.
- Uninstantiated variables in equality condition.
- Right-hand side of matching condition introduces variables.

- Matching condition does not introduce new variables.
- Strange condition encountered.
- Left hand side is contained in a list.
- No variables may be introduced in left hand side of test.

Common Syntax Errors in Specifications or Input Terms

Syntax errors are unavoidable when writing any program or specification. Here we have collected some case that are typical for ASF+SDF:

- Parse error after equations keyword.
- Equation tag does not parse correctly.
- Parse error in equation just after = sign.
- Parse error in conditions.
- Common syntax errors in input terms.
- Ambiguities in equations.

Parse Error after Equations Keyword

Consider the specification in `BlackOrWhite.sdf` and `BlackOrWhite.asf`. They define a sort `BlackOrWhite` which contains the values `black` and `white` and a `flip` function to switch colors.

Example 1.1. `BlackOrWhite.sdf`

```
module BlackOrWhite
exports
  sorts BlackOrWhite
  context-free syntax
    "white" -> BlackOrWhite
    "black" -> BlackOrWhite

  flip(BlackOrWhite) -> BlackOrWhite
```

Example 1.2. `BlackOrWhite.asf`

```
equations
[1] flip(black) = white
[2] flip(white) = black
```

After carefully designing this specification, you save it. To your distress, you get an error message that resembles the following:

```
character ' ' expected, line 2, column 0
```

What is going on here? Well, actually two things:

- The message tries to tell you that it could not parse the newline symbol at the end of the first line (but the actual newline is not properly displayed in the message).

- You have forgotten to include `basic/Comments` in your module.

This is easily corrected as shown in `BlackOrWhiteCorrected.sdf` and `BlackOrWhiteCorrected.asf`.

Example 1.3. `BlackOrWhiteCorrected.sdf`

```
module BlackOrWhiteCorrected
exports
  imports basic/Comments
  sorts BlackOrWhite
  context-free syntax
  "white" -> BlackOrWhite
  "black" -> BlackOrWhite

  flip(BlackOrWhite) -> BlackOrWhite
```

Note:

- ❶ The missing import is inserted here.

Example 1.4. `BlackOrWhiteCorrected.asf`

```
equations
[1] flip(black) = white
[2] flip(white) = black
```

Tag of Equation does not parse

Now let's embellish the previous example further by giving better names to the equations. See `BlackOrWhiteWithTags.asf`.

Example 1.5. `BlackOrWhiteWithTags.asf`

```
equations
[flip] flip(black) = white
[flip] flip(white) = black
```

Unfortunately, this leads to an error message in the following spirit:

```
character ']' unexpected, line 2, column 6
```

The explanation is that the tag of an equation (here: `flip`) and a function name in the specification (here also `flip`) interfere with each other. The simple solution is shown in `BlackOrWhiteWithCorrectedTags.asf`. Whether you change `[flip]` to `[flip1]`, `[flip-1]`, `[flipa]`, or something else is not important as long as you avoid the tag `[flip]` itself.

Example 1.6. `BlackOrWhiteWithCorrectedTags.asf`

```
equations
[flip-1] flip(black) = white
[flip-2] flip(white) = black
```

Parse Error in Equation just after = Sign

It is not unusual to equate the Boolean values `true` and `false` with the respective integers 1 and 0. You specify this in `BoolAsInt.sdf` and `BoolAsInt.asf`.

Example 1.7. BoolAsInt.sdf

```
module BoolAsInt
  exports
    imports basic/Comments
```

Example 1.8. BoolAsInt.asf

```
equations
[t1] true  = 1
[f0] false = 0
```

You are confident that you did a good job: a proper tag, a Boolean constant on the left-hand side and Integer constant on the right-hand side of each equation. Unfortunately, this experiment ends prematurely in the following error message:

```
character '1' unexpected, line 2, column 14
```

The story behind this error is as follows. The implementors of The Meta-Environment are clever (maybe too clever?) and they use the parser to do the type checking of equations as well. In this example, the left-hand side is of type Boolean and the right-hand side of type Integer, a clear type error. Unfortunately, the system reports this type error as a parse error. A likely solution for this problem is to introduce a new sort `BoolOrInt` that contains both Booleans and Integers. Now, the above equations can be parsed as equations over this new sort.

Syntax Errors in Conditions

It is sometimes hard to find syntax errors in conditional equations with many, complex, conditions. Here are the most likely causes:

- A textual error in one of the sides of a condition. Remedy: correct it.
- A parse error just before the right-hand side of a condition. Probably a type check error disguised as a parse error (see Parse Error in Equation just after = Sign). Remedy: check the sorts of the arguments and the results of the functions on both sides of the condition and correct the type error.
- Conditions are separated by commas and, unfortunately, these commas *have* to be there. They are easily forgotten. Remedy: add the missing comma.

Common Syntax Errors in Input Terms

When you have completed your specification you want to write some input terms and parse and reduce them in order to validate your specification. As in ordinary programming when you are hit by a syntax error detected by the compiler, you can also get a parse error after entering an input term. There are three possible explanations for this:

- You simply made a textual error in the input term. Remedy: correct the textual error and try again.
- You made an error in your syntax definition; the input can impossibly be parsed according to the given syntax definition. Remedy: adjust the syntax definition and try again.
- None of the above seems to apply. In this case it is likely that you want to parse an input term for some sort, say S , but a start symbols declaration for the sort S is missing. Remedy: add a start symbols declaration for S and try again. *Add cross ref to start symbols.*

Ambiguities in Equations

Can we give some hints here?

Dynamic warnings and errors

Common Errors when Executing Specifications

The examples in this section have not yet been checked.

Apart from the warnings and errors that are detected before execution, various errors remain that are only discovered during execution. We will use the factorial function as running example, see `Fac.sdf` and `Fac.asf`.

Example 1.9. `Fac.sdf`

```
module Fac
exports
  imports Multiplier ❶
  context-free syntax
  fac(NUM) -> NUM
```

Notes:

- ❶ We re-use here the module `Multiplier` defined earlier. Observe that `Numerals.sdf` exports two variables `X` and `Y` that we are now happy to re-use. Note however, that it is in general considered bad practice to export variables from a module.

Example 1.10. `Fac.asf`

```
equations
[fac0] fac(0)          = succ(0)
[fac1] fac(succ(0)) = succ(0)
[fac2] X != 0 ==> fac(succ(X)) = mul(X, fac(X))
```

Missing Equations

If the normal form of a term still contains function symbols that should have been removed during rewriting, you probably have

- *forgotten* one or more equations that define the function,
- made an *error in one of the conditions* that prevents one of the equations from being applied in some cases.

A typical example of a forgotten equation is shown in `FacError1.asf`.

Example 1.11. `FacError1.asf`

```
equations
[fac1] fac(succ(0)) = succ(0)
[fac2] X != 0 ==> fac(succ(X)) = mul(X, fac(X))
```

Trying to reduce `fac(0)` will now yield `fac(0)` instead of `succ(0)`.

Execution depends on Ordering of Equations

If the left-hand sides of two equations can match the same term, then two reductions are possible and the outcome of rewriting becomes uncertain. Consider the example in `FacError2.asf` where the condition in equation `[fac2]` is forgotten. Now the left-hand sides of both `[fac1]` and `[fac2]` can

match the term $\text{fac}(\text{succ}(0))$ and lead to different outcomes depending on the implementation. Always make sure that such overlapping left-hand sides are guarded by a condition that determines which equation to apply.

Example 1.12. FacError2.asf

```
equations
[fac0] fac(0)          = succ(0)
[fac1] fac(succ(0)) = succ(0)
[fac2] fac(succ(X)) = mul(X, fac(X))
```

Incorrect Inductive Definition

As in any language, if the equations that describe the induction over a given structure are wrong, this may lead to infinite recursion. Consider the erroneous definition of the factorial function in FacError3.asf.

Example 1.13. FacError3.asf

```
equations
[fac0] fac(0)          = succ(0)
[fac1] fac(succ(0)) = succ(0)
[fac2] X != 0 ==> fac(succ(X)) = mul(X, fac(succ(X)))
```

Important

Don't try this at home :-)

Currently, The ASF+SDF Meta-Environment does not have good support for recovering from non-terminating specifications. In some cases the system can recover gracefully, for instance, when a stack overflow is discovered, in other cases you have to restart the system.

Non-termination due to Commutative Equations

Some operators are inherently commutative, i.e., it does not matter in which order the arguments occur. It is tempting to express this in a specification.

Consider the following, extended, definition of addition in AdderError.asf. Mathematically, this is a fine specification. However, executing *may* lead to non-termination. The careful reader will observe that equation [0] also overlaps with equations [1] and [2], therefore the outcome is uncertain as explained above.

Example 1.14. AdderError.asf

```
equations
[0] add(X, Y) = add(Y, X) ❶
[1] add(0, X) = X
[2] add(succ(X), Y) = succ(add(X, Y))
```

Notes:

- ❶ New equation that expresses commutativity of addition.

Non-termination due to Commutative List Equations

Commutative equations may also occur in the guise of an equation containing list matching.

The specification of sets in `ItemSet.sdf` and `ItemSet.asf` illustrates a specification that leads to non-termination since equation [2], which expresses that two elements in a set may be exchanged, will lead to an infinite rewriting loop.

Example 1.15. `ItemSet.sdf`

```
module ItemSet

imports basic/Whitespace

exports
  context-free start-symbols Set
  sorts Item Set

  lexical syntax
    [a-z]+ -> Item

  context-free syntax
    Set[Item] -> Set

hiddens
  variables
    "I"[0-9]* -> Item
    "L"[0-9]* -> {Item " ",""}*
```

Example 1.16. `ItemSet.asf`

```
equations
[1] {L1, I, L2, I, L3} = {L1, I, L2, L3}
[2] {L1, I1, L2, I2, L3} = {L1, I2, L2, I1, L3}
```

Erroneous Conditions

Add examples here.

There are a few issues to be aware of when writing conditions:

- When using the inequality operator `!=` in a condition, no new variables may be introduced in either side of the inequality.
- Be careful when a condition contains both instantiated and uninstantiated variables.