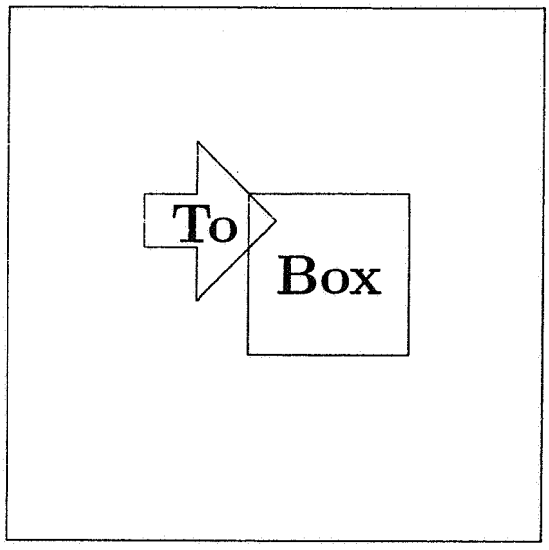

A rule-driven front-end for prettyprinting AsFix

S. L. Barth

Abstract

We describe a formatter for ASF+SDF. Since the syntax of an ASF+SDF specification is highly flexible, specifications to be formatted should first be translated to the fixed format AsFix. The formatter defines a mapping from AsFix to Box. This mapping can be fine-tuned by the user with linear non-duplicating term reduction rules. Termination of the formatter can be shown using induction with respect to the structure of the AsFix term to be rewritten, provided no rules are given whose lefthand side is a single variable. It turns out that formatting AsFix cannot be handled properly by context-free rules, and special functions have been defined to circumvent this problem.

Acknowledgements

This thesis is the result of my graduation project carried out at the Programming Research Group of the Faculty of Mathematics, Computer Science, Physics and Astronomy, University of Amsterdam.

First of all, I would like to express my thanks to my supervisors M. van den Brand and E. Visser, and to prof. dr. P. Klint. The many insights and helpful comments of the former two especially have been of invaluable aid.

Secondly, I would like to thank my parents and friends, among whom most notable Eduard Lohmann. I could not have done this without their support.

Finally, I would like to thank my dog Speedy, who still holds that no academic research can be as important as taking a walk.

Serge L. Barth
August 22, 1996

ToBox: a rule-driven front-end for prettyprinting AsFix

Serge L. Barth

August 22, 1996

Contents

1 Introduction

1.1 Prettyprinting as translation

A prettyprinter is a program that has a syntax tree of some language L as its input, and a syntactically correct text in L as its output. A prettyprinter does not require input during its execution; it can be viewed as a system to translate syntax trees into specific text forms. Therefore, prettyprinters can be specified using term rewriting, if a target language is provided that can be represented by terms. This paper discusses the front-end of such a translation, the translation of AsFix [Kli94] into Box [BV94b]. Box is a formatting language. A formatting language is essentially a formal language for the purpose of describing how text should be formatted. People familiar with formatting languages will discover some "familiar faces" in Box; for example, they will find similarities with PPML [PPML86]. In our system, Box is meant to be an *interlingua*: an intermediate representation. AsFix is translated to Box; Box can be translated to a number of "devices", like LaTeX or HTML. The interlingua approach has the advantage of compositionality: every language only needs a back-end and a front-end for the interlingua. It has the drawback of loosing information: even an interlingua cannot be assumed to keep track of every aspect of a particular language. For example, consider the sequence LaTeX→ Box → HTML → Box → LaTeX. (This is a purely hypothetical sequence, for illustration purposes only). Although it starts with LaTeX and ends with LaTeX, the mapping is not guaranteed to yield the identity function, even though that would be desirable.

1.2 Introduction to the ToBox system

The ToBox system is an executable ASF+SDF [BHK89] specification which translates AsFix [Kli94] to Box [BV94b]. We will now give a brief introduction to ASF+SDF, AsFix, and Box.

ASF+SDF is a formalism for defining programming languages. It allows the user to define a syntax and give it semantics in the form of an equational specification. The ASF+SDF formalism is supported by the ASF+SDF Meta-environment [Kli93]. ("Meta" because it helps to generate programming environments). In the Meta-environment, the equational part of an ASF+SDF specification is interpreted as a TRS (Term Rewriting System). Now ASF+SDF allows users to define part of the syntax of their rewriting rules. This flexibility is a great strength, but also has some shortcomings. Writing meta-level applications -like prettyprinters- for ASF+SDF becomes problematic, and that's where AsFix comes into play.

AsFix is a fixed format for ASF+SDF. It has the same expressive power but a fixed syntax. In other words, for every ASF+SDF specification there is a representation in AsFix. This greatly simplifies developing meta-applications for ASF+SDF: just develop them for AsFix instead.

Finally, we arrive at explaining a little about Box. Box is a formatting language. It serves as an interlingua, that is, an intermediate representation during translation. Informally, an interlingua is what is exchanged between a front-end and a back-end.

The place of the ToBox system can be seen in Fig. 1:

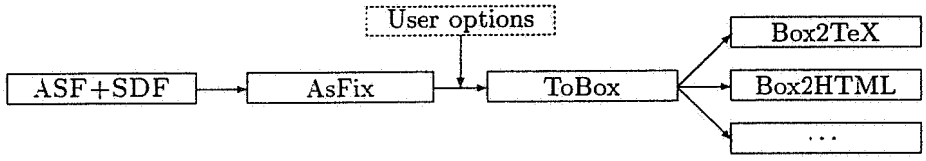

Figure 1: ToBox as part of a larger mapping

In this schema, the mappings are shown as arrows. Note that the ToBox system has two sets of inputs: the AsFix specification to be formatted and the "user options". The latter are an optional set of user-defined prettyprinting rules. These allow the user to influence the formatting.

1.3 ToBox as an exercise in self-application

Term rewriting can be described in terms of term rewriting. Phrased more clearly, it is possible to build a TRS that implements a term rewriting engine. An example of such a TRS is presented in [Deur94].

The ToBox system is such a system. It is written as a TRS, using the ASF+SDF formalism. It *implements* a term rewriting engine that rewrites (closed) AsFix-terms to (closed) Box-terms [1].

What's new is that the ToBox system allows a user to add his or her own rewriting rules to the system. These rules are bound to some restrictions to prevent abuse of the prettyprinter and to guarantee termination of the system. This is described in further detail in section 5.4.

For clarity, we will from here onward make a difference between "definable rules" and "ASF+SDF rules". The former are the parameters for the prettyprinting process. The latter are the set of rewriting rules that make up the specification of the ToBox system. Further distinctions between specific kinds of rules will be made later on.

Previously, we have discussed information loss during translation processes. This is not a problem when translating ASF+SDF to AsFix. No piece of information is lost in this particular translation scheme, since an AsFix representation of a module retains the source text of the specification. This is important,

[1] Strictly speaking, this is term rewriting with respect to the language defined by the union of the signatures of AsFix and Box.

because the source text, in turn, contains the comments which are lost in the syntax tree. As AsFix is mostly concerned with providing a fixed format for a syntax tree, comments are lost in all other parts of the AsFix representation of a specification. Otherwise, however. AsFix has all the expressive power of ASF+SDF, in a more easily processable and less easily legible format.

1.4 The main question

Having discussed the preliminaries, we now get to the main question of this thesis. Suppose one is writing a paper which involves one or more ASF+SDF specifications, developed with the Meta-environment. Naturally. one wants to add these specifications to the paper. In addition, one wants these specifications to be typeset, if this can be done without too much effort.

In order to add legible specifications to an article. a specification in the Meta-environment can be typeset using the ToLaTeX system. ToLaTeX generates LaTeX representations of ASF+SDF modules. Alternatively. one can use a prettyprinter generator [Bra93b]. The prettyprinter generator takes SDF files as input, and outputs a prettyprinter. Then, the equational part of a specification (the ASF part) can be prettyprinted using the generated prettyprinter. Thus we have a two-step process, which is a drawback. In addition, if one doesn't like the result the output of the generator must still be modified. To remedy the problem, it was proposed to make a translation system that accepted production rules as parameters. This thesis is about the ToBox system, which is meant to replace ToLaTeX. It is a system that defines a mapping between AsFix and Box. Users can influence the mapping by parametrizing it with special production rules. But they don't *have* to: the parameterization is completely optional.

2 ASF+SDF: rapid prototyping of programming languages

In this and the next section, we shall have a more "intimate" look at ASF+SDF and AsFix. This section is devoted to the ASF+SDF formalism and the ASF+SDF Meta-environment. In the next section, we shall explore AsFix. At first glance, ASF+SDF and AsFix seem quite different. We will see that AsFix, despite its illegible appearance, is just ASF+SDF in disguise.

Before continuing, one more point should be addressed. "TRS" is the abbreviation of "Term Rewriting System" or "Term Reduction System". To avoid confusion, I will refer to the set of rewriting rules as the TRS. I will refer to the (virtual) device that performs these rewritings as the "term rewriter" or "term rewriting engine".

2.1 Overview

ASF+SDF [BHK89] is an algebraic specification formalism. It should not be confused with the ASF+SDF Meta-environment, which is a tool supporting the development of ASF+SDF specifications.

The formalism results from the combination of SDF [HHKR92] and ASF [BHK89]. SDF, or "Syntax Definition Formalism", is a formalism for defining lexical and context-free syntax. ASF, or "Algebraic Specification Formalism". is a specification formalism based upon many-sorted algebras.

In ASF+SDF the two formalisms are combined. For example, the nonterminals of SDF become the sorts of ASF. Thus TRS-es can be written using a user-defined syntax. Intuitively, but not *quite* correct, one could think of this as a parser generator which parses a text (the equations in the user-defined syntax), and hands their parse trees to a term rewriting engine. More specific, the abstract syntax of a language is mapped to a many-sorted signature.

The Meta-environment is the implementation of ASF+SDF. It accepts (executable) ASF+SDF specifications and interprets them as term- rewriting systems. Figure 2 is an example of such an (executable) specification.

At the core of the ASF+SDF system is a term rewriting engine and a parser generator. The parser generator accepts a syntax definition in SDF format. The term rewriter accepts conditional rewriting rules, where the terms are written in the given syntax. The rewriting rules are reduced using an leftmost-innermost reduction strategy.

An ASF+SDF specification consists of modules. This is for the same reason that many programming languages support writing programs in modules: it eases the creative effort involved, and it allows one to re-use previous work (which is usually easier in ASF+SDF since it is a specification language and therefore avoids complex constructs). In addition, modularity can simplify proofs of correctness, which is an important consideration for specification lan-

imports Layout
exports
sorts BOX-BOOL BOX-BOOL-LIST
context-free syntax

true	→ BOX-BOOL	
false	→ BOX-BOOL	
BOX-BOOL "|" BOX-BOOL	→ BOX-BOOL	{**left**}
BOX-BOOL or BOX-BOOL	→ BOX-BOOL	{**left**}
BOX-BOOL "&" BOX-BOOL	→ BOX-BOOL	{**left**}
BOX-BOOL and BOX-BOOL	→ BOX-BOOL	{**left**}
not BOX-BOOL	→ BOX-BOOL	
"(" BOX-BOOL ")"	→ BOX-BOOL	{**bracket**}
"{" {BOX-BOOL ","}* "}"	→ BOX-BOOL-LIST	

variables
Bool [*0-9'*]* → BOX-BOOL
priorities

{BOX-BOOL "|"BOX-BOOL → BOX-BOOL,
BOX-BOOL or BOX-BOOL → BOX-BOOL} <
{BOX-BOOL "&"BOX-BOOL → BOX-BOOL,
BOX-BOOL and BOX-BOOL → BOX-BOOL} <
not BOX-BOOL → BOX-BOOL

equations

[1]true | *Bool* = true [2]false | *Bool* = *Bool*
[3]true & *Bool* = *Bool* [4]false & *Bool* = false
[5]not false = true [6]not true = false

and and **or** are synonyms for & and |, respectively.

[6] $Bool_1$ or $Bool_2$ = $Bool_1$ | $Bool_2$
[7] $Bool_1$ and $Bool_2$ = $Bool_1$ & $Bool_2$

Figure 2: Example ASF+SDF specification

guages. The reader is reminded that ASF+SDF is based on a term rewriter, and that of confluence and strong normalization, only the former property is modular. Strong normalization is a modular property only when certain conditions are met; the details can be found in [BKM89]. Another aspect that should be mentioned is that the Meta-environment allows circular imports; it will issue a warning but not an error.

We have already seen a concrete ASF+SDF module; the reader might be interested in a more abstract representation. An ASF+SDF module looks like this:

imports *Names-of-imported-modules*
section*
priorities
    Priority-relations-between-context-free-rules
equations
    Conditional-equations

The **section*** represents zero or more sections. Each section has the following format:

exports-or-**hiddens**
sorts *Names-of-sorts*
lexical syntax
    Rules-of-the-lexical-syntax
context-free syntax
    Rules-of-the-context-free-syntax
variables
    Names-of-variables

If the section starts with "**hiddens**", then the language constructs defined there will be visible only in the module in which it is declared. Otherwise, they will be visible in every module that imports this module.

The conditional equations in the **equations**-part are written in the syntax defined by the corresponding SDF module and the exported language constructs of the imported modules. That is, the equations are written in the language defined by the user, with some auxiliary constructs (in the **hiddens**-sections).

Obviously, the ASF+SDF system has a very flexible syntax. The grammar to be defined must be context-free but is not otherwise restricted. The price of this is ambiguity. Three ambiguation tools are available for the specifier: brackets, priorities and attributes. All of these are described in the section about SDF.

2.2 About SDF

There are some features of the SDF system, *as it is implemented in the Meta-environment*, that should be mentioned. These are layout, lists, chain functions, and disambiguation.

2.2.1 Layout and comments

The sort names **"LAYOUT"**, **"REJECT"** and **"IGNORE"** are predefined. When a lexical construct is defined to be a member of these sorts, it will be ignored by the parser. Thus it is possible to separate lexical tokens by whitespace (tabs and blanks) by defining, for instance,

[\t] → **LAYOUT**.

This is the appropriate way of defining comments in a language. For example, when defining the syntax of C, one would have a statement like:

"/*"~["*/"]"*/"→ **LAYOUT**

2.2.2 Lists

A special feature of ASF+SDF are lists.

A list is an iteration of elements of the same sort, sometimes separated by a special separator symbol. This is often the format of a group of statements in an imperative programming language (cf. Pascal, where the separator is the semicolon (";")). Similarly, the syntax of variable declarations usually lends itself well to being described by lists.

Lists occur in two formats: the "*" and the "+", corresponding to their well-known analogons for regular expressions. In addition, the user may specify separators. This makes for four distinct cases:

1. $SORT* \rightarrow SORT'$: Terms of $SORT'$ are zero or more terms of sort $SORT$. There is *no* top function symbol to combine them, so syntactically this is an extension of standard term rewriting. The same holds for the other three cases.

2. $SORT+ \rightarrow SORT'$: As above, but for an element to be of sort $SORT'$, it must consist of at least *one* element of sort $SORT$.

3. $\{SORT\ "SEP"\}* \rightarrow SORT'$: Terms of sort $SORT'$ consist of zero or more terms of sort $SORT$, separated by SEP.

4. $\{SORT\ "SEP"\}+ \rightarrow SORT'$: Terms of sort $SORT'$ consist of *one* or more terms of sort $SORT$, separated by SEP.

2.2.3 Chain functions

A chain function is a sort directly injected into another sort. In other words, anything of the form $SORT \rightarrow SORT'$ is a chain function. Chain functions may contain list sorts, and this happens quite often. The list sorts may not occur on the righthand side of a function definition, though. So $SORT1* \rightarrow SORT2$ is legal, but $SORT1 \rightarrow SORT2*$ is not. In other words, a single sort cannot be forced into being a list.

Chain functions are the appropriate way of implementing context-free chain rules in SDF[2]. This construction is used to abstract away from lower levels of syntax definition. It can also be used as an abbreviation mechanism:

{**statement** ";"}*	→ **stats**
"program" **decls stats** "end"	→ **program**

2.2.4 Disambiguation

ASF+SDF allows a user to define any context-free language (s)he desires. Since ambiguity of context-free languages is not in general decidable, the user should have a means of disambiguation. ASF+SDF offers three methods: associativity attributes, brackets, and priorities.

An associativity attribute can be added to a context-free grammar rule, provided it is of the form

$SORT$ "*operator*" $SORT$.

Four attributes are available. The user should select the appropriate one and at it to his rule, like this:

$SORT$ "*operator*" $SORT \rightarrow SORT${**attribute**}

1. **left**: the operator associates to the left.
2. **right**: the operator associates to the right.
3. **assoc**: the operator is associative.
4. **non-assoc**: the operator is not associative.

Brackets can be added in a similar manner. Given a function

<*lhs-bracket*><*sort-name*><*rhs-bracket*>,

the user should add the attribute **{ bracket }** .

The third way of disambiguation is by priorities. Suppose we have the well-known grammar for arithmetic expressions (see Figure 3).

Note that it has four nonterminals, and it isn't even complete: < *Number* > was not developed further since it has it's intuitive meaning anyway. In Figure 4, we have another version with priorities.

[2]A chain rule is a context-free grammar rule of the form <A>→<B>.

```
exports
  sorts EXPR TERM FACTOR
  context-free syntax
    EXPR "+" TERM    → EXPR
    EXPR "−" TERM    → EXPR
    TERM             → EXPR

    FACTOR "*" TERM → TERM
    FACTOR "/" TERM → TERM
    FACTOR           → TERM

    "(" FACTOR ")"   → EXPR
```

Figure 3: The "classical" grammar for arithmetic expressions.

```
exports
  sorts EXP
  context-free syntax
    EXP "*" EXP  → EXP  {right}
    EXP "/" EXP  → EXP  {right}
    EXP "+" EXP → EXP  {right}
    EXP "−" EXP → EXP  {right}
    "(" EXP ")"  → EXP  {bracket}
  priorities

    {EXP "*"EXP → EXP,                    EXP "/"EXP → EXP} >
    {EXP "+"EXP → EXP, EXP "−"EXP → EXP}
```

Figure 4: A grammar for arithmetic expression, using priorities.

Notice that the second version, thanks to its use of priorities, has less non-terminals. One may use the "<"-sign to declare a function to be of *less* priority than another. One may thus write increasing or decreasing sequences of priorities; obviously, the >- and <- symbols should not be used in a single chain. As a final remark, the associativity attributes *are* allowed in a priority chain. They were left out in the example only to enhance legibility.

2.3 About ASF

"ASF" stands for "Algebraic Specification Formalism". Without SDF, an ASF module is simply an equational specification over a many-sorted algebra. With SDF, it becomes an equational specification with a context-free grammar for the signature. These specifications can be interpreted as TRS-es. The Meta-environment interprets the "=" symbols as "$\rightarrow$" when considering the specification as a TRS. The ASF+SDF also system supports *default rules*. When rewriting a term, the default rules are the *last* rules that the system tries to match. It can be considered an abbreviation mechanism. It can be used to avoid spellng out each and every condition of an equation. The cases in which a particular condition fails must usually be addressed anyway. So the specifier defines equations for these cases. Then (s)he marks the original rule as "default". The system is spared from re-calculating conditions for every rule it encounters. If it gets to the default rule, all normal rules with conditions have failed. This is not meant to say that a default rule will always succeed; a default rule may still have conditions of its own, and it may still fail to match a term. In that case, this particular part of the reduction simply doesn't reduce.

The ASF+SDF system allows a user to write any TRS (s)he wants. Therefore it cannot guarantee strong normalization or even confluence. This is the responsibility of the author of the specification. Theoretically, the TRS-es written in ASF+SDF don't have to be complete. But in practice, this is considered the most elegant way of specification using term rewriting.

The Meta-environment uses a leftmost-innermost reduction strategy. This is of particular importance for those writing a regular TRS, since for regular TRS-es this strategy will always find the non-terminating rewriting in a TRS [BKM89].

2.4 Some practical aspects of the Meta-environment

In order for the user to further visualize the Meta-environment, I shall highlight some if its practical aspects. Users only interested in the theory may skip this subsection.

There are to kinds of editors in the Meta-environment: module-editors and term-editors. The difference is in purpose, not in the way the actual text editing is performed. Module-editors are used to edit ASF+SDF modules, whereas term-editors are used to edit terms over the languages defined by these modules.

Module editors consist of two parts: one part contains the syntax of a specification, the other part contains the equations. The syntax part corresponds to the SDF part of the specification; the equations part contains the ASF part.

When a user wishes to rewrite a term, (s)he can edit it with a term editor. The term can be edited in an ordinary text editor as well, and then loaded in the term editor. A term editor is bound to a module; the term editor for a module M will only accept terms over the language defined in M. This is one way in which the Meta-environment supports modularity of specifications.

A term editor can be extended with *buttons*. These will evaluate a function in the specification or a part of LeLisp[3] code. For example, suppose one has designed a programming language P. Suppose we also have a typechecker for P. One could make it easy for the user to type-check a term over the P-language by making a button that calls the type-checker.

Even with a specification language, the result of our "program" (our executable specification) is not always what we expect. That's why the Meta-environment offers a debugger. The equation debugger traces the execution of a term reduction, allowing a user to pinpoint the source of an irregularity.

[3]LeLisp is (*-yet-*) the language in which the Meta-environment is written.

3 ToLaTeX: the original ASF+SDF typesetter

The ASF+SDF Meta-environment introduced in the last section provides a typesetting mechanism for ASF+SDF specifications. This is the ToLaTeX [VK94] system, which translates an ASF+SDF specification to LaTeX code.

3.1 A brief history of pretty printing ASF+SDF specifications

Default pretty printer: the default pretty printer is part of the ASF+SDF Meta-environment. It is used for pretty printing within the Meta-environment. The most obvious part of its work is pretty printing the normal form of a reduced term.

It has seen two incarnations. The old version was written in LeLisp code, and translated an ASF+SDF term to text. It used the language PPML (described in [PPML86]), for intermediate representation.

The new version is also written in LeLisp, but no longer uses PPML. Instead, it maps an ASF+SDF term to a term over Box (described in section 4.3. This intermediate representation is then used to generate the textual output.

ToLaTeX: The ToLaTeX [VK94, Vis94] tool is a typesetting facility for ASF+SDF specifications. Like the default pretty printer, it is written in LeLisp. ToLaTeX is used to format ASF+SDF specifications for use with LaTeX. It is described in more detail below.

AsFix to Box: This is the original system for translating AsFix (described in section 3.3) to Box. The built-in formatting rules of the ToBox system are direct translations of the rules in this specification.

Pretty printer generator: The pretty printer generator [Bra93b] takes an ASF+SDF specification as its input. It outputs a specification to translate terms over the defined language to Box. The generator is incremental: if a user extends the original input specification, new rules are added in separate modules. There is a drawback to the use of the prettyprinter generator: it is a two-step process. One must first generate a prettyprinter. and *then* apply it to the specification that is to be pretty printed.

3.2 Usage

The Meta-environment has a button to activate the ToLaTeX tool. When activated, the system will generate several files for each module - unless the module hasn't been changed since the last time the tool was used. ToLaTeX supports literate specification. Literate specification is what is comparable to literate

programming. applied to specification languages rather than imperative programming languages.

The ToLaTeX tool is very useful for extensive documentation. The idea behind literate programming is that a program contains its own explanation. Since ASF+SDF modules are often used in courses, this property is very desirable.

In order to add comments to a specification in LaTeX, a user just has to add "%%"→ **LAYOUT** to the SDF part of an ASF+SDF specification. Anything that is put behind the "%%" mark in the specification will then be considered comment by the ASF+SDF Meta-environment tools, such as the term rewriter and the parser. ToLaTeX, however, will process these "comments" and interpret them as LaTeX text.

The ToLaTeX system uses a file "ASF+SDF-options.sty" containing a number of options. For example, there is the "\ModuleIs" macro. It determines the kind of header for modules in the specification that is being typeset. Figure 5

```
%\ModuleIs{chapter}
%\ModuleIs{section}
%\ModuleIs{subsection}
%\ModuleIs{subsubsection}
```

Figure 5: Options for the "\ModuleIs" macro.

shows the code to define the "\ModuleIs" macro. Note that all commands start with a "%" sign, forcing LaTeX to ignore the remainder of the line. By removing one of these "%"-signs, the line will no longer be ignored and the command will be executed.

Choosing the font in which the specification is to be shown is done in almost the same manner. The "default" font is "\ComputerModernStyle" (CMR font family). One may replace it with "\SansSerifStyle" or "\TypeWriterStyle".

There are many more options, they can be found in [Vis94].

3.3 Shortcomings

The ASF+SDF typesetters, formatters and prettyprinters have evolved into ToLaTeX and some other products, but the evolution is not yet complete.

ToLaTeX cannot be parametrized; it can only be slightly influenced. For example, the "=" signs can be turned into ⇒ signs. Also, all comments are considered LaTeX text, unless they are preceded by "%%%" instead of "%%". And there are restrictions on comments on the LaTeXlevel in the SDF part. (Remember that only the ASF part of a module has a user-defined syntax).

The prettyprinter generator on the other hand allows some modification of the prettyprinting rules used. But this is a two-step process. First the prettyprinter must be generated, then it must be debugged. This debugging is tedious and error-prone.

4 AsFix: A fixed format for ASF+SDF

AsFix [Kli94] is a fixed format for ASF+SDF (hence the name: **Asf+Sdf-FIX**ed format). AsFix has the same expressive power that ASF+SDF has, but its syntax is fixed. Thus, it becomes easier to design meta-specifications and prove theorems about them; it was used thus in, for example, [Rom95]. However, AsFix is not designed to be read by human beings.

AsFix is built upon special terms called "ATerms". One could think of ATerms as atoms or toy bricks from which larger components of AsFix are built.

4.1 ATerms

ATerms are defined by the following syntax (in the SDF formalism).

imports Literals
exports
  sorts ATerm
  context-free syntax
    Literal → ATerm
    "[" ATerm ATerm "]" → ATerm
    ATerm ";" ATerm → ATerm {**right**}
    nil → ATerm
    ATerm "/" ATerm → ATerm {**left**}
    "(" ATerm ")" → ATerm {**bracket**}
  variables
    T [*0-9'*]* → ATerm
    Ts [*0-9'*]* → {ATerm ","}*
    Ts "+"[*0-9'*]* → {ATerm ","}+
    L [*0-9'*]* → Literal
  priorities
    ATerm "/"ATerm → ATerm > ATerm ";"ATerm → ATerm

Note that variables whose names begin with an L are literals, whereas variables whose names begin with a T may represent any ATerm.

This syntax is interpreted as follows.

$[T_0\ T_1]$: This is a very frequently used operation in AsFix, representing function application. It should be read as "T_0 applied to T_1".

$T_0;T_1$ **and** *nil*: These two functions are used to construct lists. The *nil* represents an empty list of ATerms. In $T_0;T_1$, T_0 is the head of the list and T_1 is the tail.

T_0/T_1: This operator is used for *annotation*. The T_1 is an annotation at ATerm T_0, for internal use by applications.

4.2 Format of a specification

An AsFix specification of an ASF+SDF module has the following format: ["*Module*" $L1; L2; T1; T2; T3; T4$]. Here $L1$ is the module name, and $L2$ is the full module name (i.e., the module name including the directory path). The other for ATerms are of the form

["*SourceSyntax*" T_a]
["*PrefixSyntax*" T_b]
["*SourceEquations*" T_c]
["*PrefixEquations*" T_d]

respectively. They are described in more detail below.

4.2.1 SourceSyntax and SourceEquations

The ["*SourceSyntax*" T_i] and ["*SourceEquations*" T_j] parts of a module in AsFix format keep the original ASF+SDF text present in an AsFix representation of an ASF+SDF module. The text is kept in the sub-ATerm (that is, T_i or T_j, in this case). The text has the form of a list of literals: that is, a set of literals separated by the AsFix list constructor, a semicolon (";").

The purpose of maintaining the original code is to keep comments. Comments are removed in the prefix-forms of syntax and equations, but must be restored when prettyprinting. Note that comments are *not* inserted by the prettyprinting rules we give; this is done by an algorithm described in [Bra93a].

4.2.2 PrefixSyntax

The *PrefixSyntax* section of an AsFix specification gives the SDF part in prefix format. It closely follows the format of the syntax part of an ASF+SDF module. The *PrefixSyntax* part consists of zero or more sections. Here a section is ["*Imports*"...], ["*Exports*"...], or ["*Hiddens*"...].

The ["*Imports*"...] Section has as its arguments a list of identifiers; obviously these represent the imported modules.
Example: ["*Imports*"["*Id*" "*Booleans*"];["*Id*" "*Ints*"]].

The ["*Exports*"...] and ["*Hiddens*"...] sections both consist of zero or more "SyntaxSections". This can be a sorts declaration, a "LexicalSyntax" or "ContextFreeSyntax" section, a set of priorities or the definitions of the variables. An example is shown in Figure 6. Note that, syntactically, it is perfectly acceptably to have a section like:
["*Imports*"["*Sorts*"*nil*];["*Sorts*"["*Id*" "*Bool*"]]]

```
["Exports"
    ["Sorts"...];
    ["LexicalSyntax"...];
    ["ContextFreeSyntax"...];
    ["Priorities"...]
]
```

Figure 6: Example "*Exports*"-part of an AsFix PrefixSyntax section.

We will now briefly discuss the precise construction of these SyntaxSections; the reader is referred to [Kli94] for a complete description. Alternatively, one could examine the AsFix specification of a well-understood and sufficiently rich ASF+SDF module.

- Sorts: a list of sorts. Sorts in AsFix have the form ["*Sort*" "*sortname*"], and this is no exception.
- LexicalSyntax: a list of zero or more lexical functions. These have the form ["*LexicalFunction*" *LexElems*; ["*Sort*" *Result*]]. *Res* is the result sort of the function: The *LexElems* are zero or more lexical elements; e.g. literals and quoted literals.
- ContextFreeSyntax: a list of CfFunctions. These have the form ["*Cffunction*"["*CfElems*"T_a]; ["*Sort*"T_b]; ["*Attributes*"T_c]]
  Here T_a is a list of terminals and nonterminals; T_b is the name of the result sort of the function; and T_c yields the attributes ("bracket", "assoc", etc).

  CfFunctions appear within the function application operation:
  [["*CfFunction*"["*CfElems*"T_a]; ["*Sort*"T_b]; ["*Attributes*"T_c]]T_d]
  Here T_d contains the instantiations of the nonterminals in T_a.
- Priorities: this may be an increasing priority chain (i.e. of the form $f < g < h$, a decreasing priority chain (i.e., of the form $h > g > f$), or a list of functions that have the same priority (i.e., g and h in $f > \{g, h\} > t$. In ASF+SDF, priorities may be expressed with only the function name or with full representation. In other words, one may write "*" > "+" or *INT*"*"*INT* → *INT* > *INT*"+"*INT* → *INT*. In the former case. the function is represented in AsFix using the "Abbrev" function. (i.e., ["*Abbrev*"...]).
- Variables: as its name implies, this syntax section contains the variable declarations.

4.2.3 PrefixEquations

The PrefixEquations-part of an AsFix specification is simply a list of zero or more conditional equations in prefix format. It has the form:

```
["PrefixEquations"
      ["CondEquation" ...];
            ⋮
      ["CondEquation" ...]
]
```

The format of an equation in AsFix is
["*CondEquation*" "*Implies*"; "*Tag*"; "*Lhs*"; "*Rhs*"; "*Conditions*"] Here *Tag* is the tag (name) of the equation. *Lhs* and *Rhs* are ATerms, usually of the form ["*CfFunction*"...]. Occasionaly they are something else. Examples of this are iterations and meta-variables.

The *Conditions* is ["*Conditions*" "*List*"], where *List* is a list of conditions. An example of a condition is ["*Condition*" "*Neq*"; T_0; T_1]. *Neq* is an abbreviation of "not equal to". It could also be "Eq".

Let's finish this part by an example of a complete conditional equation. We shall use an axiom of process algebra concerning abstraction. For clarity, irrelevant subterms have been replaced with their textual counterparts. The < and > have been used to denote this.

The axiom is $\tau_I(a) = \tau if a \epsilon I$. Let's assume this is represented in an ASF+SDF specification as "T(I,a)=i when (a is-member-of I)=true".

```
["CondEquation" "When";"TI-2";
    [["CfFunction" <T>] <I>;<a>];
    [["CfFunction" <i>]];
    ["Conditions"
        ["Condition"
            "Eq";
            [["CfFunction" <is-member-of>] <a>;<I>];
            [["CfFunction" <true>] nil ]
        ]
    ] ]
```

4.2.4 CfFunctions

The ["*CfFunction*" T_i] function deserves some special attention. It has the format ["*CfFunction*" ["*CfElems*" *LIST*]; ["*Sort*" "*Res*"]; ["*Attributes*" *T*]].

CfElems: the list following the keyword "CfElems" represents zero or more members of the context-free function, be they terminals or nonterminals. Terminals are represented by Literals or a function application ["*QLit*" *Literal*], for "Quoted Literal". Nonterminals are simply represented by the names of the corresponding sort: ["*Sort*" "*subterm − sort*"].

Res: this is the result sort of the equation.

Attributes: this is a list of ASF+SDF attributes, as described in 1.4.

Textually, this corresponds to something like f(<sub-expression>). This in turn corresponds to a context-free syntax rule
$< Res > \leftarrow$ "f" "(" $< sub - expression >$ ")"

In the prefix-syntax part of a specification in AsFix format, these are all the constructs that can be used. In the prefix-equations part, the signature of a function is subsequently applied to a list of parameters, which instantiate the nonterminals. Note that these instantiations may themselves be CfFunctions. Thus, in the prefix-equations part, we get CfFunctions that are closed with respect to instantiation of nonterminals.

4.3 About AsFixGram

The correctness of AsFix is to a certain extent the responsibility of the user. It is possible to make a syntactically correct but none the less quite nonsensical ATerm. The "AsFixness" of an ATerm can be checked using an executable specification described in [Kli94]. There is also another approach, which is called AsFixGram [Kli94]. As the name suggests, this is a finer-grained grammar for AsFix. It has separate nonterminals for sorts, conditions, metavariables... all in all, it has over 50 nonterminals. This makes it unfit for writing general applications [Kli94]. For this reason, it is not used.

5 About Box

Box is a formatting language. A sentence in Box is a recursive structure. It is either an "atomic" box, or a box built from smaller boxes. This description neatly fits a context-free grammar rule or a term in a TRS, and therefore it naturally translates into ASF+SDF. Since sentences in the language Box are so close to prefix terms, the words "Box-term" and "Box-sentence" will be used for the same thing: a sentence in the language Box. The kind of box at the top of a term describes how the children are placed relative to each other. In spite of this, some boxes are context-sensitive. We are mostly interested in the box-terms, and not in their interpretation. Interpreting a term over the Box language is the task of back-ends like Box2TEX [BV94a]. Some of these back-ends will be reviewed for completeness.
Here's an example of a sentence in the box language.

$V\ vs = 1\ [$"*Boxing :*" "*the noble art*" $HV\ [$"*of*" "*formatting*"]]

Here V and HV are operators. Their arguments are the strings between the square brackets. The $vs = 1$ statement is a user option to customize the output. These customization options will be discussed later in this section.

5.1 Box

A term over the Box language has the form

Op options [...].

Now the "..." represent boxes, so we get a nested structure like
$Op_0\ options_0\ [\ Op_1\ options_1\ [...]\ Op_2\ options_2\ [...]]$.
The lowest level of the recursion is a simple string. So we get a recursive definition for a sentence in the Box language.

A Box is:

- A string, or
- $H[B_0 \ldots B_n]$, where $0 \leq i \leq n$. "H" means horizontal composition. The sub-boxes are aligned next to each other, regardless of the screen size. If the screen is not wide enough, boxes beyond the righthand side of the screen will be invisible.
- $V[B_0 \ldots B_n]$, where $0 \leq i \leq n$. This is vertical composition: the sub-boxes are all placed below each other.
- $HV[B_0 \ldots B_n]$, where $0 \leq i \leq n$. This should be read as "horizontal and vertical composition". The system tries to fit all of its boxes on a row. When the right margin is encountered, a new line is inserted.
- $HOV[B_0 \ldots B_n]$, where $0 \leq i \leq n$. This should be read as "horizontal or vertical composition". This operator will act as a $V[B_0 \ldots B_n]$ operator if

its arguments consist of more than one line. This happens if the operator has a V-subterm or a list of boxes that don't fit on a single line.

- $I[B]$: this is the Indentation operator. It prints its argument boxes with a few spaces indented.

- $WD[B]$: this operator takes the width of it's argument box and yields that much white space. For example, WD ["Boo"] = " "

The LST box is not a standard operator, but should be mentioned for its importance in the ToBox system.

- $LST[B_0 \ldots B_n]$: this operator is mostly a syntactic affair. It converts its arguments into an ASF+SDF list of boxes. Intuitively, $LST[B_0 \ldots B_n] = B_0 \ldots B_n$. While not a proper formatting operator, the LST box has turned out to be quite useful.

Now we get to parametrizing the system. The formatting of the Box-terms can be influenced with "Space-options". Let's get back to our example.
$V\ vs = 1$ ["*Boxing :*" "*the noble art* " HV ["*of*" "*formatting*"]]
The $vs = 1$ is a space-option. There are three such options, all set between the name of the box and the start of its argument list.

1. $vs = n, n \geq 0$: this parameter is for use with V, HOV and HV operators. "vs" is an abbreviation of "vertical space". "$vs = n$" instructs the system to print n blank lines between the argument boxes.

2. $hs = n, n \geq 0$: this parameter is the horizontal equivalent of vs. It instructs the system to leave n spaces between the argument boxes.

3. $is = n, n \geq 0$: this parameter addresses the indentation. I-boxes with the $is = n$ parameter must be indented n positions.

The attentive reader will have noticed that for all space-options, $n \geq 0$. This implies the possibilities of having 0 spaces or lines between boxes, or 0 indentation. This comes in very useful, as the default values for hs, is and vs are usually set to 1, 2 and 0, respectively.

Notice that Box strongly resembles PPML (a prettyprinting formalism described in Section 7.1), but it has been extended with operators not found in that language. The free format of ASF+SDF allows users to define boxes of their own (although implementation in the back-ends may prove a problem). I will discuss a few of these extensions. Many others exist in other modules; all. however, have the basic format
$BOX - NAME\ options * [BOX \ldots BOX]$

- $KW[Box]$: the argument is a keyword, and must be typeset accordingly. In LaTeX and most other systems, this means printing the string in a **bold** font.

- *MATH*[*Box*]: the argument of the box is a mathematical expression and must be typeset accordingly. This will be quite familiar to users of LaTeX.
- *O s – options* [*Box Box – String Box*]: this is slightly more complicated. The three arguments must be set above each other, and centered with respect to the Box-String. This is used for drawing the horizontal line in a conditional equation.

KW and *MATH* are defined in the module "Fonts" of [BV94b], together with several other extensions. The "O"-box is defined in the module "Over" [BV94b], which defines only this operator (since *O*[...] is a highly specialized operation).

5.2 Life after Box: Box2Text

Box2Text [BV94b, BV94a] is the simplest back-end for interpreting Box. It converts a Box term to a list of strings. The Box2Text system implements the basic semantics of the Box language, thus yielding the fundamentals for other back-ends. The Box2Text back-end was developed using the ASF+SDF Meta-environment, meaning that it is an executable specification over ASF+SDF. It is designed as a mapping from Box to terms over a sort "Text", where a "Text" is a list of strings. It offers several options and operators to typeset the text, to ease the effort of translating. It has been extended with "Text-Laws", a module which models the interactions of the operators defined in the module "Text". A full listing of the system can be found in [BV94b, BV94a].

5.3 Life after Box: Box2TeX

As the name implies, Box2TeX [BV94b, BV94a] is a back-end to translate Box to TeX. The output can also be used with LaTeX [BV94a]. Readers familiar with TeX will at first glance suppose that this is not too hard, since TeX like Box, is a box-language. Unfortunately, life is rarely that simple. For starters, TeX has a maximum nesting depth of 40 boxes.

That's why the module Box-Laws [BV94b, BV94a] was written. It is an executable specification in ASF+SDF. The module rewrites a Box-term as much as possible, aiming at minimizing the amount of typesetting work that the Box-term requires. The name Box-Laws is a bit ambitious: while most of the laws are simple to understand, some have not yet been proven[4]. This is the case for two reasons. First, some of the operators (like *I*[...]) are context-sensitive, making the writing of a proof prone to error. Second, the semantics of Box are dependent upon the back-end used to interpret the Box-term. There is not yet a model that captures the essence, and *only* the essence, of the Box operators.

[4]The Box-to-Text-Alg ASF+SDF specification yields the semantics for the Box-Laws; a law is correct if and only if, for all terms over box, the same text results before and after application of the law.

5.4 Life after Box: Box2HTML

The Box2HTML back-end is the subject of another Master's thesis and is yet under development. Like ToBox, however, it will be an algebraic specification developed using the ASF+SDF Meta-environment. See Section 8.5 for a note about hypertext links.

6 The ToBox system

ToBox translates AsFix to Box, using user-defined rules. In this section, we describe the basic functionality, and take a look at its components and their interaction.

6.1 General overview

The ToBox system consists of a set of modules. Their hierarchy is shown in Figure 7.

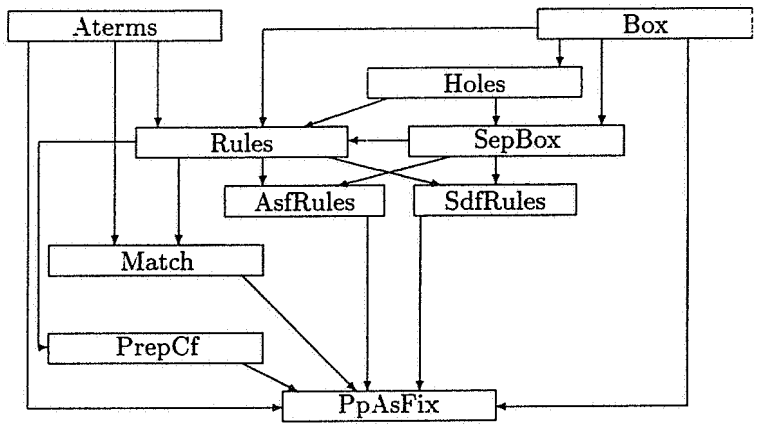

Figure 7: Import graph of the ToBox system.

The core of the system is the module "PpAsFix" (described in detail in Section 6.2.1). "PpAsFix" is offered an ASF+SDF specification in AsFix format and a set of user-defined formatting rules.

In order to apply the formatting rules to the AsFix specification, "PpAsFix" calls "Match" (described in section 6.2.2). "Match" tries to match the specification in AsFix to the lefthand sides of the rules. Failing this, built-in default rules take care of the "offending" parts.

The module "Rules" (described in section 6.3) describes the syntax of the user-definable formatting rules. It has a sister module "Rule-correct" which is not part of the system, but can be used to test if the rules are semantically correct. (Some of the correctness issues of the rules are context-sensitive, and therefore cannot be properly described in a context-free syntax: for example, formatting rules must be left-linear). The modules "AsfRules" (described in section 6.3.1) and "SdfRules" (described in section 6.3.1 give the "default" rules for formatting AsFix. The modules "SepBox" and "PrepCf" define auxiliary functions of the same names, respectively. They are needed to circumvent the context-free nature of the formatting rules.

The module "Holes" is strictly syntactic. It defines the holes (ToBox-specific variables) in a term over the Box-language. In other words, a term over Box is open if it contains a hole.

6.1.1 Taxonomy of rules

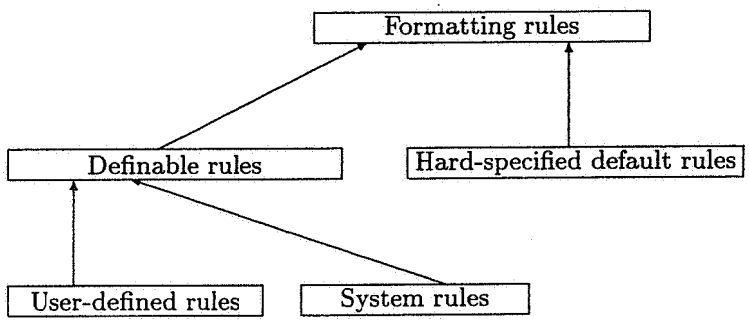

Figure 8: Taxonomy of rules in the ToBox system

There are several different kinds of rules in the system; it is vital to understand the differences between them in order to understand the remainder of this chapter. Their interaction is shown in Figure 8.

Definable rules Definable rules are the rules that the ToBox system takes as parameters. Intuitively, these are the rules of the form $< ATerm, Box >$. They are called "definable" because they can be defined by the user and the implementor (e.g, the module AsfRules is a set of implementor-defined definable rules).

User defined rules User defined rules, also called user rules, are definable rules that the user has provided as parameters. These are the rules that the user may use to fine-tune the ToBox output. An example of such a rule is $< [\text{"}Sort\text{"}\ N[0]], KW[\ [0]\ ] >$ which causes the system to print all sort-names as keywords.

System rules System rules are definable rules built into the specification. These are the "standard" rules for formatting ASF+SDF. They come into play when the user rules fail to address a case. Thus, the user is saved from having to specify every case that may occur in a specification, allowing him/her to concentrate on the details. In addition, as the system rules are definable, they can be altered to supply support for modifications or extensions of AsFix, without having to change essential parts of the system. Note that the only difference between a user defined rule and

a system rule is the location: a user defined rule is a parameter, whereas a system rule is part of the module. In other words, a user rule is user-defined, and a system rule is implementor-defined.

Hard-specified default rules Hard-specified default rules are ASF+SDF rules, used as defaults for translating AsFix to Box. They come into play when the formatting rules fail to cover a case. The difference between these rules and the system rules is that these are "hard-specified" (hard-coded). The difference in purpose is that the system rules are strictly for AsFix, whereas the hard-specified default rules are defaults for *any* ATerm.

The hard-specified default rules should never be activated, with one exception: the rule that translates a Literal in ATerm to a String in Box. This transformation takes place at such a low level of implementation that it cannot be described by the definable rules.

Formatting rules Finally, formatting rules are all the rules that directly influence the translation of AsFix to Box. They consist of the user defined rules, the system rules, and the hard-specified rules.

6.2 The core of the rewriting system

We can now look at the core of the system. At the heart of the system is a top-down recursive tree traversal process . (The "tree" is the ATerm to be formatted; remember that an AsFix ATerm is the syntax tree of an ASF+SDF specification). At each node, the system tries to match the current node to the given rules. If a match succeeds, the system recursively formats the subtrees. If all matches fail, a set of hard-specified default-rules is activated. Once their work is done, the system recursively calls the main loop for formatting the subterms.

6.2.1 PpAsFix and substitute

The module "PpAsFix" has a number of entry points: PpAsf, PpSdf. PpAsFix, and PpATerm. Which one is used depends on the user's situation. For example, if one is only interested in formatting the ASF section of an ASF+SDF specification, one calls PpSdf. All these entry points, however, eventually lead to a call of the function "DoPpATerm".

DoPpATerm is the function that implements the outer recursive loop. It accepts two parameters: a list of user defined rules and an ATerm to be formatted. The user defined rules are added to the system rules, yielding a complete list of the definable rules to be used. The user defined rules form the front of this list. Thus it is guaranteed that they will be considered before the system rules; this implements a simple scheme for maintaining priority.

The eventual call of *DoPpATerm* has as its arguments an ATerm and two identical lists of formatting rules:

$DoPpATerm(Rules, Rules, T)$

The first list of rules is a "master copy". The second list is scanned element after element to find a matching rule. We will see this in more detail below.

DoPpATerm checks the list of definable rules to match the head of the given ATerm. As soon as a match is found, it is applied. This happens in equations [DoPpATerm-0] and [DoPpATerm-2], which are listed below.

[DoPpATerm-0]

$M = match(AT_0, AT)$,
$M \neq NoMatch$

$DoPpATerm(SystemRules, < AT_0, B_0 >; PPrule - list, AT) =$
$substitute\ (SystemRules, M)\ in\ B_0$

[DoPpATerm-1]

$M = match(AT_0, AT)$,
$M = NoMatch$

$DoPpATerm(SystemRules, < AT_0, B_0 >; PPrule - list, AT) =$
$DoPpATerm(SystemRules, PPrule - list, AT)$

There is a third equation in the main loop, which is responsible for the call of the "prep" function. This will be detailed in section 6.5.1; it is listed here for completeness.

[DoPpATerm-2]

$AT = [$["CfFunction"
["CfElems" AT_1]; ["Sort" AT_2]; ["Attributes" AT_4]
$]\ AT_4]$,
$AT_4 \neq nil$

$DoPpATerm(SystemRules, PPrule - list, AT) =$
$DoPpATerm(SystemRules, PPrule - list, prep(AT))$

Thus, the first matching rule in the list of definable rules is always used. The precise operation of matching is explained in section 6.2.2; for now, all we need to know is that it returns either "NoMatch" (=matching failed), "EmptyMatch" (=matching succesful with 0 bindings), or a non-empty list of bindings. At first glance, it seems illogical to have an "EmptyMatch" instead of an empty list of bindings. The main reason for doing so is elegance; a condition "$B* = EmptyMatch$" is more legible than "$B* = $ ".

On a succesful match, DoPpATerm calls the function substitute with the result of the matching operations, and the complete set of definable rules.

"substitute" now has the complete list of rules, and a set of ATerms that are bound to nonterminals. These ATerms are recursively translated to Box. This is where the aforementioned "master copy" of the rule list comes into play. It is this *complete* list of rules, not the partially emptied list, that must be handed over to lower levels of recursion.

If no rule matches succesfully, the hard-specified default rules come into play. Without these, it would be possible for the system to terminate with a partially unformatted output. The default rules guarantee that every ATerm is eventually translated to a proper Box-term.

The hard-specified default-rules are[5]:
$[DoPpATerm-4]\ DoPpATerm(SystemRules, , L) = L2S(L)$
The $L2S$ function translates a character string of sort *Literal* to an equivalent character string of sort *Box-String*.

$[DoPpATerm\text{-}5]\ DoPpATerm(SystemRules, , [AT_0\ AT_1]) =$
$V\ [\ DoPpATerm(SystemRules, SystemRules, AT_0),$
$DoPpATerm(SystemRules, SystemRules, AT_1)$
$]$
$[DoPpATerm\text{-}6]\ DoPpATerm(SystemRules, , AT_0; AT_1) =$
$LST[DoPpATerm(SystemRules, SystemRules, AT_0),$
$DoPpATerm(SystemRules, SystemRules, AT_1)$
$]$
$[DoPpATerm\text{-}7]\ DoPpATerm(SystemRules, , nil) = H[]$
$[DoPpATerm\text{-}8]\ DoPpATerm(SystemRules, , AT_0/AT_1) =$
$DoPpATerm(SystemRules, SystemRules, AT_0)$

6.2.2 Match

The module "Match" is an almost independent but vital element of the ToBox system. It implements only one function, named "match". "match" accepts two parameters: a closed ATerm (meaning an ATerm without Nonterminals) and an ATerm that may contain Nonterminals. The system then tries to match these two. If succesful, it returns the bindings (if any) of the Nonterminals in the open ATerm. If it is succesful but there were no Nonterminals (meaning, in practice, that the ATerms were syntactically equal) it returns "EmptyMatch". If no rule succesfully matches the two ATerms, the default rule comes into play. This rule returns "NoMatch". As the default rule is without conditions and doesn't look at the structure of the ATerms offered, it always succeeds, thus guaranteeing termination of the "match" function. Care was taken to make the executable sub-specification "Match" confluent. "match" makes recursive calls: if even one subterm fails to match, the "NoMatch" result is propagated upwards. In addition, "EmptyMatch" may only be returned if there is no binding at all; it doesn't

[5]Attentive readers may notice that there is no equation "DoPpATerm-3". This is a result of the systems development history; it is not the case that equations have been omitted.

make sense to have "EmptyMatch" as part of a list of bindings. When combining bindings, "EmptyMatch" is always removed. When two "EmptyMatch" results are combined, the result is still an "EmptyMatch". (The same, of course, applies to "NoMatch"; but then "NoMatch" combined with any result will yield "NoMatch", as a match can only be achieved when every subterm succesfully matches). Note that the match-function does not look for consistent bindings! This would needlessly complicate the function, since only left-linear definable rules are allowed. Even checking the offered ATerms is beyond the scope of this module; the module Rule-correct sees to that part. Incorporating such a check into "Match" would mean it was carried out every time the "match" function was called, putting an undesired strain on the time performance of the system.

6.3 Rules

This section takes a closer look at the definable rules, as described in the module "Rules". The module defines a new kind of ATerm, the Nonterminal, and a new kind of Box, the Hole. A rule is then defined as a 2-tuple $< T, B >$, where T is an ATerm, and B is a term over Box. We shall use an example for the toy language Pico.
A Pico program has the format
`"begin"` *Decls Stats* `"end"`
A formatting rule for Pico (in pseudo-AsFix notation) would look like this:
< `"begin"`;N[0];N[1];`"end"`, V[`"begin"`HOV[[0]] [1]]`"end"`] >

Here `"begin"`;N[0];N[1];`"end"` is the lefthand side of the rule the part that must be matched to a closed ATerm.

V[`"begin"`HOV[[0]] [1]]`"end"`] is the righthand side of the rule. If the lefthand side succesfully matches an ATerm, then the righthand side tells how it is to be formatted. In this case, "N[0]" and "N[1]" are the variables in the lefthand side. The ATerms that are bound to them will be translated to a Box-term recursively; and these Box-terms are then filled in for the "[0]" and "[1]" on the righthand side. Simplified, "[0]" and "[1]" are the variables corresponding to "N[0]" and "N[1]", respectively.

6.3.1 AsfRules and SdfRules

These modules are based upon the previously written specifications "PpSyn" and "PpEqs" described briefly in [Bra95]. This approach was chosen to make the ToBox system correspond as much as possible to existing translation systems. However, the mapping between the original rules (in ASF+SDF) and our rules is not perfect. In the "PpSyn" and "PpEqs" specifications, variables can be of sort "*Literal*" or "*ATerm*" (note that *Literal* is a subset of *ATerm*). Our system, on the other hand, only has variables of the sort *ATerm*.

Another issue is the separation of formatting rules for ASF and SDF. The reason is that SDF has a fixed syntax (as described in section 1.4). There

is much agreement over how the SDF-part of a module should be formatted. Therefore, the rules for formatting SDF are more sophisticated, and larger in number, than the rules for formatting ASF.

6.4 Rule-correct

The module Rule-correct is separate from the ToBox system itself. It tests a given set of rules for semantical errors. There are four such errors:

1. A nonterminal as the only part of the left-hand side of a rewriting rule. This is forbidden in all TRS-es. If one allows this kind of rule, every term always matches this rule. Therefore there are no normal forms anymore, and hence no termination. In addition, in ToBox, the strong normalization of the formatting rules is based upon the stepwise descent of an ATerm. Rules which have only a nonterminal on the lefthand side do not descend the term to be rewritten.

2. A rule has the same nonterminal more than once on the lefthand side. In other words, the system is not left-linear. This is forbidden for efficiency reasons. If the system allowed such rules, the matching system would have to keep track of the consistency of bindings, which causes overhead. The price paid for this is that it is not possible to emphasize repetitions (e.g, multiple occurences of the same variable in a single condition). Another reason is that allowing rules that are not left-linear allows the user to (ab)use the system for program transformations. Here's an example of such an abusive rule.
   <"if" N[0] "then" N[1] "else" N[1], [1] >

3. A rule has the same nonterminal more than once on the righthand side. This, too, is forbidden because it might be (ab)used for program transformations instead of rewriting. Here's an example of a rule that does some program transformation instead of formatting. For example, consider the rule
   <"for" $i = 1$ "to" 2"do" N[0]";". V[[0][0]] >
   This is erroneous if N[0] equals, for example, " `write(i);` ".

4. A rule has nonterminals on the righthand side that do not occur on the lefthand side. This, too, is forbidden in all term rewriting systems. The reason is obvious: every variable on the righthand side must have something to be instantiated with.

One more thing should be said about program transformations. It would seem that using formatting rules for program transformations is not inherently wrong. A computer scientist of sufficient skill could avoid simple errors like the one mentioned above. But the formatter is not meant for this. Anyone wanting to do program transformations should use a proper program transformation

tool, since it will alert him/her to errors and refuse to perform some erroneous transformations. The ToBox system, on the other hand, is not at all concerned with the semantics of the specification to be formatted and will not issue such warnings. The most secure way of enforcing this is to make the "transforming rules" illegal.

6.5 Additions to the system

Thus far, the system has been straightforward. There are two complications in the systems' operation though. The one concerns the formatting of context-free functions in the "equations" section of a specification. The other complication concerns separators between boxes. Both complications are introduced as a result of the restrictions on the expressive power of the rewriting rules.

The problem with context-free functions is that an AsFix "CfFunction" is represented in AsFix by a signature first, and its actual arguments following. Simply put, we get (in pseudo-AsFix):
[*some-function* (*sort1*, *sort2*, *sort3*) *term1*; *term2*; *term3*]
This is inherently context-sensitive. Since the rules are context-free, they cannot handle this kind of function unless the arity of *some-function* is known beforehand.

The problem with separators is best illustrated by the formatting of so-called priority chains. Syntactically, a "priority chain" is a list of function names separated by ">" or "<" (the same separator is applied throughout a chain). The problem is that priority chains are represented in AsFix as
[*type-of-chain*$T_0 \ldots T_n$]
The type of chain ("IncrChain" or "<", versus "DecrChain" or ">") is known only when processing the entire term. When processing the subterms (i.e, $T_0 \ldots T_n$), this information is no longer available. Therefore, the system doesn't know whether the subterm should be affixed with an ">", an "<", or not affixed at all! (The latter is the case for the last element of the chain).

Both problems can be eliminated by allowing the system to pass context-sensitive information to lower levels of recursion. This would require the ASF+SDF rules to have an additional parameter to inform the system about the current context. An example of such a parameter would be a simple tag which could take the values of symbolic constants. The constants would have expressive names like "in_a_list" and "not_in_a_list". This will result in a more beautiful but almost certainly slower system, since another argument must be matched at every rewriting step.

6.5.1 PrepCf

The function "PrepCf" comes into play when a context-free function with arguments is to be formatted. These have the form
[["*CfFunction*" ["*CfElems*" T_0]; ["*Sort*" T_1]; ["*Attributes*" T_2]]T_3]

This reads as "Apply the declaration" (["CfFunction" ...]) "to the list of arguments given in T_3". It is the application to T_3 that gives the problem. T_3 is a list of terms that may contain any finite number of elements. So we have the declaration of a context-free function, followed by its instantiation. Obviously, in the output of the formatter, we want the instantiations substituted at the proper places in the signature.

This is where PrepCf comes into play. As soon as the system encounters a "CfFunction", it calls the auxiliary function "prep". "prep", being an ASF+SDF function, has full rewriting power. It substitutes the elements of T_3 in T_0 (or, substitutes the instantiations in the signature).

So, if we go back to our earlier pseudo-AsFix example, here is the input to *prep*:
[*some-function*(*sort1*, *sort2*, *sort3*)*term1*; *term2*; *term3*]
The function "prep" will translate this into:
[*some-function*(*term1*, *term2*, *term3*) *nil*]

Note the "nil" in the output of *prep*. It represents an empty list of arguments, keeping the ATerm as correct as possible. (Remember that the square brackets are interpreted as function application, and therefore always have *two* ATerms between them. Putting a "nil" there is the only proper solution).

Since this function is called only when a CfFunction is encountered, the rules-defining user does not have to worry about it; everything about AsFix will remain as it was until the system is no longer able to stall the preparation. Even then, only one level of substitution is performed. Thus, deeper sub-CfFunctions retain the same basic structure that they had before PrepCf starts to work. Should the formatter get to these parts, it will call "prep" to rewrite them just before it starts the formatting process. This makes the system more orthogonal, and prevents "prep" from preparing sub-ATerms that will never be formatted in the first place. Note that "prep" will be called whether we're formatting AsFix representations of ASF or of SDF, since ToBox doesn't know the difference. All it knows is that a "CfFunction" should be prepared by "prep". However, as T_3 always equals nil in a context-free function declaration (i.e., a CfFunction in the AsFix representation of SDF), T_3 will immediately terminate in this case.

6.5.2 SepBox

As was noted, the SepBox module was implemented for the same reason that PrepCf was: because there are things the system cannot do without passing context-sensitive information to lower levels of recursion. The SepBox is a new kind of Box-term, which has for its subterms a BOX-LIST and a BOX. We will refer to the latter box as the "separator". When the separator-box is a closed boxterm, it will be rewritten into a standard boxterm: a BOX-LIST, where the BOX that was given as an argument is used to separate the elements. There are three separator-boxes: SEP-I, SEP-A and A-SEP. The intuitive meaning of

these names is, respectively, "separate by insertion", "separate by appending behind", and "separate by appending before". "separate by insertion" is the naive way of separation: it simply changes the BOX-LIST by inserting the separator between any two boxes. This will result in the following structure when dealing with an H-box:

BOX sep BOX sep BOX

And the following structure when dealing with a V-box:

BOX
sep
BOX
sep
BOX

It should be clear that the latter result is not always desired. For instance, the conditions in an equations section of AsFix are one thing for which we'd like to use SepBox, with "," as separator. But, we want the comma to come directly after each condition. This is where SEP-A comes in useful: it will generate H-boxes, with the elements of the BOX-LIST (the conditions in our example) followed by a comma as subterms. Thus, we get:
SEP-A ["," *LST*[*Cond1 Cond2 Cond3*]] →
LST [*H*[*Cond1* ","] *H*[*Cond2* ","] *H*[*Cond3*]]
If we had used A-SEP, we would have gotten the commas before the conditions, like:
LST[*H*[*Cond1*]*H*["," *Cond2*]*H*["," *Cond3*]].
Note that in none of these cases the separator is the start or end of a BOX-LIST. It is, after all, a separator.

With this functionality of SEP-A in mind, it should be clear why there is no "I-SEP" box as a counterpart to SEP-I. The two boxes would have the same functionality.

Some words need being said about the conditions under which SepBox will rewrite into a LST. When given an open boxterm (meaning a boxterm containing a hole), the SepBox shouldn't rewrite. To see this, assume the hole is the separator. Now, since in this counterexample the open box-term could be rewritten, the rule itself would be modified. The system would rewrite *SEP*["*X*"[0]] into [0]. Obviously, this is not the intended result.

7 Creating your own formatting rules

A set of formatting rules is a set of rewrite rules, separated by semicolons. The formatting rules consist of an ATerm and a BOX-term each. So a template of a set of formatting rules is

$< T_0, B_0 >$;
$< T_1, B_1 >$;
$< T_8, B_3 >$

Of course, this doesn't suffice to create generic formatting rules. Therefore both ATerms and BOX-terms have been expanded with operators usable in ToBox rules *only*.

For ATerms, we have nonterminals. A Nonterminal has the form "N[n]", where n is a natural number or zero. It represents an unknown sub-ATerm. During the rewriting process, any (closed) sub-ATerm may be bound to a nonterminal.

For BOX-terms, we have so-called "holes". A hole has the form [n], where n is again a natural number or zero. It represents the place to substitute a BOX.

When the rewriting engine encounters an ATerm that matches the ATerm in a rule, the nonterminals of the latter ATerm are bound. (This sounds complicated, but it's really standard term rewriting). Now the system descends the BOX-term, keeping these bound nonterminals and their bindings with it. Every time we encounter a hole, we look for the corresponding nonterminal. So, if we encounter [3], we look for the binding of N[3]. This binding is recursively translated and substituted for the hole.

A small example should clarify things. Suppose we have the rules
`< ["Example" N[2]], H["e.g." [2]] > ;`
`< N[0] / N[2], [0] >`
and the term
`["Example" ["look" "here"] / "this is an example"].`

The system will first try to match the first rule. (We will see later that rules are always tried out in the order in which they appear). This rule matches the to-be-rewritten ATerm, yielding the binding.

N[2] = `["look" "here"]/"this is an example"`

Now the system recursively rewrites N[2]. I will explain below how this works, but for the moment, just accept that it yields V [`"look" "here"`]. Therefore, the box V`["look" "here"]` is substituted for the [2] in H[“e.g.” [2]], yielding

H [`"e.g."` V `["look" "here"]`]

Now we come to how the box `["look" "here"]` was obtained. At first, the system tries again to match the first rule, this time to the term `["look"`

"here"]/"this is an example". This match fails, and the next rule is tried. The next rule *does* match, yielding the bindings

N[0] = ["look" "here"] N[2] = "this is an example"

Note that the N[2] of the second rule has no bearing whatsoever upon the N[2] of the first rule; nonterminals are like variables in PROLOG, they have their own meaning in every rule. In other words: the *scope* of a nonterminal is limited to the rule in which it appears. Since the matching was succesful, we may now substitute. The system once again descends the BOX-part of the appropriate rule, which is simply [0]. This means that we have only the translation of N[0] to substitute; N[2] is ignored, since no corresponding hole appears in the BOX-term! This is not a bad thing; you will often want the formatter to ignore parts of an ATerm.

So, the system must now translate ["look" "here"]. Since no rules are provided, the system is on its own. Fortunately, there are default formatting rules in the system for just this kind of thing. The rules for ATerms of the form [T0 T1] tell the system to build the box V [B0 B1], where B0 and B1 are the translation of T0 and T1, respectively. The translation of T0 and T1 will take in account all the rules the user supplied. In our case, the literals "look" and "here" will be translated to the strings "look" and "here", which is also the work of a default rule. So the system has translated ["look" "here"] to V["look" "here"].

The end result of our rewriting session is therefore H ["e.g." V["look" "here"]]

which is prettyprinted as

e.g. look
here

7.1 Summary

- The scope of a number (nonterminal or hole) is a single rule.
- The left hand side of a rule can never be a single nonterminal.
- The numbers of the holes in the right handside of a rule should be a subset (not necessarily a proper subset) of the numbers of the nonterminals on the lefthand side.
- The same nonterminal shouldn't appear twice in the same rule.
- The same hole shouldn't appear twice in the same rule.

8 Related work

8.1 Oppen

Oppen [Opp80] was the one of the first papers to discuss an implementation of language-independent prettyprinting. Oppen's algorithm breaks a text into blocks. It has two interacting procedures, *scan* and *print*. (Oppen calls them "functions", but this is not correct since they do not return data; worse yet, if they were functions, they would have serious side effects).

The procedure *scan* receives tokens from an auxiliary function *receive*. These tokens are stored in a FIFO buffer named *stream*. The tokens are used to build blocks; parallel to *stream* is a buffer of integers called *size*, containing the sizes of these blocks. (These blocks basically behave like our $H[\ldots]$ boxes). The size information in *size* is used by *print* to decide if a block can be printed on the current line; the procedure does, of course, have to keep track of the remaining space on a line. A complete description of the algorithm, together with a listing in the Pascal-like language MESA, can be found in [Opp80].

The system is remarkable because it performs on-line prettyprinting without intervention of a parser. A very simple on-line lexical analysis is performed to determine which symbols constitute a string.

8.2 Blaschek & Sametinger

Blaschek and Sametinger [BS89] describe a more or less general prettyprinter for Pascal-like languages.

There is a language-dependent front-end that parses the program, yielding a token stream. This token stream, which contains embedded control information, is then prettyprinted using an extended version of Oppen's algorithm.

The system is parametrized by a set of variables like "NewLineThen" (which causes the "then" of an if-then-else statement to be printed on a new line if it is set to "true") and "DclCommentTab" (to control the amount of tabulation for comments in declarations. The user may set it to 0 if (s)he doesn't desire tabulation). These parameters can be found in a separate parameter file (a 'user profile'). Since the set of variables may vary slightly depending upon the language, this approach is only useful for languages whose grammar is fixed.

Comments in [BS89] are treated as if they concern the last statement processed. The previously mentioned 'DclCommentTab' allows for special treatment of comments in a declaration environment. The system has a nice feature with respect to comments and "**END**" commands, allowing to append auxiliary comments to the end of a block. (e.g, "**END (* FOR *)**" for the end of a FOR-loop; [BS89] doesn't mention if procedures are appended with their name, the appropriate keyword, or neither). Innermost blocks are not appended this way, since doing so would impair, rather than enhance, the legibility.

Rather than using a nested structure of formatting operators like the Box-language, their pretty printer uses the less radical approach of inserting a special symbol in the token stream where line breaks are allowed.

The extensions to Oppen's algorithm are symbols tab, beginMargin, and endMargin. tab has it's intuitive meaning, whereas text between beginMargin and endMargin has extra indentation.

8.3 PPML

The first default pretty printer of the ASF+SDF Meta-environment used PPML [PPML86], and the basic constructs of Box are straightforward adaptations of similar PPML constructs. (E.g. the $V[\ldots]$-box is the Box counterpart of PPML's $[v\ldots]$).

An example of a PPML rule is:

```
if(*cond, *stat1, *stat2)

-> [v [hv "if" *cond][hv "then" *stat1][hv "else" *stat2]];
```

In this example (the source of which is [PPML86]), the *cond, *stat1 and *stat2 are the meta-variables.

PPML also has a construct similar to the separator-boxes: the *seplist*$(\ldots,\ldots)$ operator.

In a strange way, the ToBox system completes a circle. PPML also used the ordering of the pretty printing rules to determine priority. However, in order to deal with ambiguities, the PPML system has conditional rules. (By contrast, AsFix does not suffer from ambiguities and therefore does not need conditional rules). Another example from [PPML86] will illustrate this. It is part of a specification for prettyprinting mathematical expressions in a Pascal-like language, and determines whether or not a sub-expression should be put between brackets.

```
plus(*a,*b) -> [hv *a "+" if *b in {plus} then "(" *b ")" else *b end if ];
```

A similar construction is the case command for context-propagation, which is used to solve the dangling-else problem.

PPML is stronger than ToBox when it comes to comments: in PPML, one can add rules concerning how to deal with comments. One uses ordinary PPML rules for this purpose. These annotations are "mounted" in so-called frames. and a list of frames we want to see is added to the call of the pretty printer. Thus the PPML user can choose to ignore comments.

The final difference is that PPML also has display options as part of its specification: it does not use it's box language as an interlingua, but rather as a data structure.

8.4 Arnon, Attali & Franchi-Zannettacci

The system described in [AAFZ92] is used to convert documents between LaTeX and Tioga, using a syntax tree as an interlingua. It is mentioned here because there was a formal approach. The authors went as far as to define a class "Article". Two abstract syntaxes were given: one for the contents, and one for the physical structure of an article. The first is concerned with sections, subsections, citations etc. The second is defined in in terms of words, lines, paragraphs and other physical layout concepts. PPML is used for unparsing.

Concrete syntaxes were constructed for both LaTeX and Tioga. An example of a grammar rule for LaTeX is:

```
<abstract>::= "\begin{abstract}" <text> "\end{abstract}";
abstract(<text>)
```

The second line in the example refers to the place this rule has in the signature of the "Article" class.

Finally, a mapping is defined between the logical structure (section, subsection, paragraph...) and the physical structure (page, line, word...). A set of conditional rules governs this translation.

8.5 Garlan

Garlan [Gar85] does not address pretty printing, but the related problem of unparsing[6]. There is a complication in his work since the unparser must work in a structure editing environment, which means that

1. The program to be unparsed changes dynamically.
2. He has to print the nonterminals of the language.

In addition, it is possible for a single parse tree to be displayed more than once, and a change in one instance of the tree requires an update of all others.

Garlan [Gar85] introduces "VIZ" and "UAL". "VIZ" is an unparse specification language. "UAL" is the implementation of an unparser that uses VIZ. The VIZ/UAL system contains various features, including a user-extendable library of formatting environments.

In a VIZ specification, a set of unparse descriptors is given for each view ("window")[7]. These unparse descriptors are condition-scheme pairs. (As the conditions may involve the parent node of the nonterminal to be unparsed, context-sensitivity is introduced). In case of conflict, the first condition-scheme pair whose conditions evaluate to "true" is used. Note that different views may have different unparse descriptors.

[6] The difference is mainly that an unparser has access to a parse tree, wheras a pretty printer often uses a lexical analysis tool with language-specific information.

[7] There is the possiblity of having a default window, and of assigning the same set of unparse descriptors to multiple views.

An example of a VIZ unparse scheme is

@myhighlight(If <bool-part> —— @+(Then <then-part>))

This example shows us that VIZ is not a box-language. The "——" represents a newline. VIZ allows allows for *optional* newlines, using the related command "!!".

While these directives are more primitive then boxes, the expressive power of VIZ is strongly enhanced by the use of *conditional attributes* which affect the style of a document. Examples are *horiz-vert*, which behaves more or less like a $HOV[\ldots]$ box, and *columnize*, which formats a list as a table.

The interesting part of the UAL system is that, for reasons of efficiency, it is incremental. When a node is changed in the program being displayed, all nodes whose representation depends on it are updated. These changes are then propragated to the "U-trees" (unparse trees), by means of a procedure called the "U-machine". These U-trees are similar to concrete syntax trees. (By comparison, the node that was changed is part of an abstract syntax tree representation).

The other half of the algorithm is the D-machine, or "display machine". This system translates the U-trees to display directives. Finally, these directives are carried out: command sequences are issued to printers, bitmaps are altered, etc.

9 Conclusions and future work

This finishes the description of ToBox. All essential parts of the system have been included in Appendix A. It has been shown that a "configurable" translation of AsFix to Box is feasible. In addition, it has been shown that a non-confluent set of rewriting rules can be useful in a deterministic everyday tool, when an easily comprehensible rewriting strategy is used.

As is the way with projects like this, there is still room for beautifying the system. Anybody who would involve him- or herself with such a project is therefore referred to the following summary. These are the points that I feel should be adressed first.

- Currently, the user can translate AsFix to Box, but has nothing to say about comments. This is a result of comments not usually being part the prefix- part of an AsFix specification. The comments are added later by an algorithm described in [Bra93a].

- The PrepCf-function and the SEP-boxes are the ugly part of the specification. The specification would be more orthogonal, and therefore prettier, if these were discarded in favor of a top-down parameter passing system. Such a system would tell the lower levels of recursion in what kind of context they found themselves. While this is a form of further context-sensitivity, it does not threaten termination as long as the rules are *strictly* top-down. Note that termination of the system can be shown by induction on ATerms [8].

- Currently, the ToBox system maps AsFix to Box. This means that a user has to know AsFix to write rules. There are two solutions to this problem, but both boil down to write the left hand sides of the rules in SDF.

  1. Write a preprocessor to translate SDF lefthand sides to AsFix.
  2. Rewrite the system to accept SDF, instead of AsFix, as lefthand sides. Note that SDF has a fixed format; it is the ASF part of an ASF+SDF specification that causes the problems for prettyprinting.

  For either solution, the implementor should extend SDF with a "nonterminal" symbol, like the Nonterminal-sort that is injected into the sort ATerms. Note that the end-user must retain the freedom to design any context-free language (s)he wants. This slightly complicates the issue, since such nonterminals should be kept separate from user-defined symbols (sort-names and literal text).

[8] Provided there are no rules that have only a variable on their lefthand side. This is not as strong a condition as it seems since this is forbidden in all TRS-es, for *just* this reason

- AsFix has no explicit operation to represent cross-references, although Box does (see the *LBL*- and *REF*-boxes in [BV94b], under module "Fonts"). It is possible to pry the cross-reference marks from the *SourceSyntax* and *SourceEquations* parts of an AsFix specification. This takes extra time, though, and is not entirely reliable. Rather than making an "opinionated" system, it was decided to wait until AsFix has explicit cross-references in the *PrefixSyntax* and *PrefixEquations* parts of a specification. It is the author's opinion that not having cross-references is better than unreliable cross-references. This is even more so as cross-references will probably be translated to hypertext links in the Box2HTML back-end.

  A more elegant approach would be to insert rules using the logical names of the modules. This causes two problems. The first problem is caused by the use of the *LBL*-box. A rule for this box would look like:
  $< [\texttt{"Module"} N[0]; N[1]; N[2]; N[3]; N[4]; N[5], V[\texttt{"Module"}[0] LBL[0] \ldots] >$
  Obviously, this violates the restriction that non-duplicating rules are not allowed in the ToBox system. This restriction can be easily removed, however.

  The second problem is caused by the *REF*-box. A list of imported may have any finite length. Therefore we cannot simply give a rule like:
  $< [\texttt{"Imports"}[\texttt{"Id"} N[0]]; [\texttt{"Id"} N[1]]; [\texttt{"Id"} N[2]],$
  $HV[H[[0] REF[[0]] H[[1] REF[1]]] H[[2] REF[2]]]] >$
  We would have to define such a rule for every number of imported modules that occur. (Note that the rules are duplicating, as well). One would like to pass on the context-sensitive information that we are in a "referencing environment" to lower levels of recursion. Failing that, a few ASF+SDF rules defining a *REFLIST* box may help a user out:
  REFLIST[["Id" T_0];T_1] = LST[REF[T_0] REFLIST[T_1]]
  REFLIST[*nil*] = $H[]$

- At the release of this paper, a new AsFix, named AsFix2, is out already. One of its features is a consistent use of the ["*QLit*"T] function. "QLit" is an abbreviation of "Quoted Literal", a literal between double-quotes (" and ").

  In AsFix, how "QLit" should be interpreted depends upon the context. If one is in the *PrefixSyntax* section of a specification in AsFix format, ["*QLit*" "*item*"] should be interpreted as "<*item*>". If one is in the *PrefixEquations*, it should be interpreted as *item*.

  In AsFix2, this is no longer the case. An *QLit* is *always* to be interpreted as "something between double-quotes". Thus, with respect to double-quotes, we don't have to keep track of context anymore.

A The Maintenance Manual

A.1 Extensions of AsFix

Before we start with extending the system, a word of warning: the system itself has already extended ATerms with a new sort: the Nonterminal. *All nonterminals are of the form N[i], and of sort "Nonterminal"*. The sort "Nonterminal" is exported, since it is needed to create prettyprinting rules.

If ATerms are extended, the ToBox system should be extended correspondingly. Now the new ATerm operator may combine zero or more sub-ATerms. In the examples, we will assume the new operator is $\oplus(...)$. If AsFix is changed but ATerms retain the same structure, you only need look into the module PrepCf. This is especially so for changes in the approach towards CfFunctions.

The following rules should be added:

- In module PpAsFix, a new hard-specified default rule should be added. It should have the form

  $$\begin{array}{l} DoPpATerm(SystemRules, , \oplus(AT_0 ... AT_n) = \\ \quad X\ [\ \ DoPpATerm(SystemRules, SystemRules, AT_0) \\ \qquad \vdots \\ \qquad DoPpATerm(SystemRules, SystemRules, AT_n) \\ \quad] \end{array}$$

  Basically, the righthand side of the equation could be any box, with $DoPpATerm(AT_i)$ for the box-version of ATerm AT_i.

- In module Match, a new matching rule must be added. Matching rules follow the pattern

  $$\begin{array}{l} match(\ominus(T_{1a} ... T_{na}), \ominus(T_{1b} ... T_{nb})) = \\ \quad matchcat(match(T_{1a}, T_{1b}) ... match(T_{na}, T_{nb})) \end{array}$$

  If your new ATerm operator only yields constants (like Literal), things are easier. Equal constants always match, yielding *EmptyMatch*. Unequal constants are taken care of by a default-rule. Thus, when C is a variable ranging over the new constants: $match(C, C) = EmptyMatch$ is all you need to add.

- The module PrepCf substitutes the actual parameters of a "CfFunction" for the formal parameters. This is done recursively, yielding a complete expression tree. As such, this module may have to be looked into everytime AsFix is changed with respect to CfFunctions. If your new ATerm is in no way concerned with CfFunctions, all you have to do is to complete the recursion. $[prep - n]prep(\oplus(T_0, ..., T_n)) = \ominus(prep(T_0), ..., prep(T_n))$

However, your ATerm may deal with CfFunctions in a way you didn't expect. The square brackets (function application) and semicolons (lists) were used for CfFunctions and their arguments; your functions may also become part of it. So the rule above may have to be made into a default rule, with more specific rules for CfFunction-related purposes. Unfortunately, no general guidelines can be given for such specific rules since these changes to AsFix could be anything.

- In module Rule-correct, you should add a rule to count the nonterminals in an ATerm. These rules follow the pattern

$$\frac{\begin{array}{l}[nont - X] \\ IntList_0 = nonterminals(T_0, IntList), \\ IntList_1 = nonterminals(T_1, IntList_0), \\ \ldots \\ IntList_n = nonterminals(T_n, IntList_{n-1}\end{array}}{nonterminals(\oplus(T_0 \ldots T_n), IntList) = IntList_n}$$

  This looks a little complicated. Since most ATerm-operators work on only two arguments, though, there is an easy shorthand. $nonterminals(T_0 \oplus T_1, IntList) = nonterminals(T_1, nonterminals(T_0, IntList))$ The $IntList$ variables are bags (multisets). The $nonterminal(T, IntList)$ function extends the $IntList$ with all nonterminals found in T. When dealing with many ATerms, all ATerms have their nonterminals added to the bag, one after another. [9]

- You may wish to systematically adapt the rules the system uses for pretty-printing. The rules the system uses are found in modules AsfRules and SdfRules. These rules work exactly as user-defined rules do; the pretty-printing engine itself doesn't even know the difference. Don't forget to apply Rule-correct if you have changed AsfRules and SdfRules!

- If your extension of ATerms is specified in other modules than the module ATerms, you should import your new module in

  - PrepCf
  - Rules
  - PpAsFix - but this is done automatically since PpAsFix imports Rules
  - Rule-correct - once again, this is done automatically

[9] Since bags are not ordered by definition, the *order* in which the nonterminals are added doesn't matter. Just make sure all ATerms are considered.

- AsfRules and SdfRules, if they have rules using the new kind of ATerm - again done automatically by importing Rules.

Summarizing, all modules in the ToBox system that deal with ATerms are based on induction on the structure of the ATerm. Thus when ATerms are extended you should add rules to complete the induction.

A.2 Extensions of Box

In this subsection, we will assume that the user has created a new box
X xo* [B]. Here X is the name of the new box, and xo* are its options. The parameter B is a BOX or a BOX-LIST depending upon the situation; when it makes a difference, B represents a BOX, and Bl represents a BOX-LIST. Often it doens't matter because functions are overloaded to accept both.

Should a user desire to extend BOX, there are two possibilities.

1. The new BOX can be rewritten to an existing BOX. See the module SepBox for an example.

2. The new BOX cannot be rewritten to an existing BOX.

Even in the first case, we're not quite ready! Your new BOX may have sub-boxes that are holes. In this case, the process of rewriting your box may yield an unexpected result. An example of this is the SEP-box:
$SEP - A[","[0]] \rightarrow [0] \rightarrow LST["1" "2" "3"]$
vs.
$SEP - A[","[0]] \rightarrow SEP - A[","LST["1" "2" "3"]] \rightarrow SEP - A[","\ "1" "2" "3"]$
$\rightarrow LST["1" "," "2" "," "3"]$
We notice that the ASF-rules are *not* confluent! It is obvious that the first rewriting is the undesired one. It can be avoided very simply though: the system must be told that it may only expand *closed* SEP-boxes. The same may be the case for your new box. In any case, you should define "closed-boxterm" for your new box. If you don't need this predicate yourself, you can simply import your new box in the module "Holes" and define the predicate there. Otherwise, your module should import "Holes" instead.

If only the closed version of your BOX can be rewritten, you should also update the module Rule-correct. You should add a function
[holes-n] $holes(Xxo * [Bl], IntList) = holes(Bl, IntList)$

In the second case, if your BOX cannot be rewritten into an existing BOX. you should change some more. PpAsFix should be able to substitute in your new box. You should add the following lines to PpAsFix:

[substitute-n] $substitute(SystemRules, M)\ in\ Xxo * [B] =$
$Xxo * [substitute(Systemrules, M) in B]$

for boxes that have a BOX as parameter, and

[substitute-n] $substitute(SystemRules, M)\ in\ Xxo * [Bl] =$
$Xxo * [listsubstitute(SystemRules, M) in Bl]$

for boxes that have a BOX-LIST as parameter.

A.3 Summary

- define "closed-boxterm" for your box.
  - If your system doesn't use "closed-boxterm" itself, put the function in module "Holes". Add your module to the modules imported by "Holes".
  - If your system does use "closed-boxterm" itself, put the function in your own module and import "Holes"
- If only the closed version of your box can be rewritten, add a line to Rule-correct to calculate how many holes the box has.
- As an option, you may import your module in "Rules", as this will make your box available to every module that uses "Rules". These are just the modules that are likely to use your new box, and it's not a coincidence. After all, why have a new BOX if you can't use it in a prettyprinting rule?

  The point mentioned below are only of consequence if your new box cannot be rewritten into an existing box.
- Add a substitute-rule to module PpAsFix.

B ASF+SDF specification of The ToBox system

B.1 Literals

exports
sorts Literal
lexical syntax
[␣\t\n] → LAYOUT
"%%"~[\n]*"\n" → LAYOUT
"%"~[%\n]+"%" → LAYOUT

"\"~□ → EscChar
"\"[01][0-7][0-7] → EscChar

~[\000-\037"\] → L-Char
EscChar → L-Char
"\""L-Char*"\"" → Literal

B.2 ATerms

imports Literals
exports
sorts ATerm
context-free syntax
Literal → ATerm
"[" ATerm ATerm "]" → ATerm
ATerm ";" ATerm → ATerm {**right**}
nil → ATerm
ATerm "/" ATerm → ATerm {**left**}
"(" ATerm ")" → ATerm {**bracket**}
variables
T [*0-9'*]* → ATerm
Ts [*0-9'*]* → {ATerm ","}*
Ts "+"[*0-9'*]* → {ATerm ","}+
L [*0-9'*]* → Literal
priorities
ATerm "/"ATerm → ATerm > ATerm ";"ATerm → ATerm

B.3 Box

This module describes the most elementary box operators and their options. The box language can be extended in a very simple way. We will give several extensions.

imports Box-Strings Box-Ints

exports

sorts SPACE-SYMBOL S-OPTION S-OPTIONS

Space

options are used to adapt the amount of layout between the boxes. The horizontal, vertical, and indentation offset between boxes can be adapted. Not every combination of space option and box operator makes sense. E.g. the modification of the vertical offset in combination of the H operator does not make any sense.

context-free syntax

"hs" → SPACE-SYMBOL
"vs" → SPACE-SYMBOL
"is" → SPACE-SYMBOL
SPACE-SYMBOL "=" BOX-INT → S-OPTION
S-OPTION* → S-OPTIONS

variables

"*ss*"[*0-9'*]* → SPACE-SYMBOL
"*so*"[*0-9*]* → S-OPTION
"*so*" "*" [*0-9*]* → S-OPTION*
"*o*"[*0-9*]* → S-OPTION
"*o*" "*" [*0-9*]* → S-OPTION*

We distinguish 6 basic operators in the box language: H (horizontal composition), V (vertical composition), HV (horizontal and/or vertical composition), HOV (horizontal or vertical composition), I (indentation), and WD (invisible box with the same width as some visible box). The most elementary box is a plain string, enclosed by double quotes. A box term consists of a box operator, zero or more space options, and followed by zero or more boxes. The kernel box language is simple but can be easily extended. A specification writer can easily define a new module which imports this kernel box language. For each extension the text formatter and the tex formatter have to be adapted, but because of the strong modularity of ASF+SDF this can be done in a clean manner.

sorts BOX BOX-LIST

context-free syntax

BOX-STRING → BOX
BOX* → BOX-LIST

"H" S-OPTIONS "[" BOX-LIST "]" → BOX
"V" S-OPTIONS "[" BOX-LIST "]" → BOX
"HV" S-OPTIONS "[" BOX-LIST "]" → BOX
"HOV" S-OPTIONS "[" BOX-LIST "]" → BOX
"I" S-OPTIONS "[" BOX "]" → BOX
"WD" "[" BOX "]" → BOX

hiddens

variables

[*B-E*][*0-9*]* → BOX

[*B-E*]"+"[*0-9*]* → BOX+
[*B-E*]"*"[*0-9*]* → BOX*

exports
context-free syntax
BOX-LIST "++" BOX-LIST → BOX-LIST {**right**}
context-free syntax
"I*" S-OPTIONS "[" BOX-LIST "]" → BOX-LIST
"LST" "[" BOX-LIST "]" → BOX
equations

[CONC] B^* ++ C^* = B^* C^*

[0] I* so^*[] =
[0] I* so^*[B B^*] = Iso^* [B] ++ I* so^*[B^*]

[0] B^* LST[C^*] D^* = B^* C^* D^*

B.4 Rules

imports ATerms Box Box-Ints Box-Strings Holes Alignments Fonts Over SepBox
exports
sorts Nonterminal PPrule PPrule-list
context-free syntax
"N" "[" BOX-INT "]" → Nonterminal
Nonterminal → ATerm

"<" ATerm "," BOX ">" → PPrule
{PPrule ";"}* → PPrule-list
"rulelistcat" "(" PPrule-list "," PPrule-list ")" → PPrule-list

"dquote" → BOX-STRING
variables
PPrule-list [*0-9'*]* → {PPrule ";"}*
PPrule [*0-9'*]* → PPrule
"*c**" → CHAR*
equations

[listcat-0] rulelistcat(*PPrule-list*$_0$. *PPrule-list*$_1$) = *PPrule-list*$_0$; *PPrule-list*$_1$

[dquote-1] dquote = box-str-con("""")

B.5 PpAsFix

imports ATerms Alignments Box Box-Strings Over Fonts Match Rules
Box-Ints AsfRules SdfRules PrepCf

exports

context-free syntax

"PpAsFix" "(" ATerm ")" → BOX

"PpAsFix" "(" PPrule-list "," PPrule-list "," ATerm ")" → BOX

"PpATerm" "(" PPrule-list "," ATerm ")" → BOX

"PpAsf" "(" PPrule-list "," ATerm ")" → BOX

"PpSdf" "(" PPrule-list "," ATerm ")" → BOX

With all the above functions, the users rules take precedence over the built-in rules. Note that it is always possible to supply an empty set of rules.

hiddens

context-free syntax

"DoPpATerm" "(" PPrule-list "," PPrule-list "," ATerm ")" → BOX
"L2S" "(" Literal ")" → BOX-STRING

"substitute" "(" PPrule-list "," MatchResult ")" "in" BOX → BOX
"listsubstitute" "(" PPrule-list "," MatchResult ")" "in" BOX-LIST → BOX-LIST
"lookup" BOX-INT "in" MatchResult → ATerm

variables

"*B*"[*0-9*]* → BOX
"*B*" "*" "[*0-9*]* → BOX*
"*Bl*"[*0-9*]* → BOX-LIST
"*SystemRules*" → PPrule-list
"*c*+" → CHAR+
"*c*" → CHAR*
"AT"[*0-9'*]* → ATerm

equations

PpAsFix formats a module by calling PpSdf and PpAsf with 0 user-defined rules, and placing the results below each other.

[PpAsFix-0] PpAsFix(*AT*) = V [PpSdf(, *AT*) PpAsf(, *AT*)]

An alternative PpAsFix rule calls PpSdf and PpAsf WITH some user-defined rules.

[PpAsFix-1] PpAsFix(*PPrule-list*$_1$. *PPrule-list*$_2$, *AT*)
= V [PpSdf(*PPrule-list*$_1$, *AT*) PpAsf(*PPrule-list*$_2$, *AT*)]

PpAsf first concatenates the user-defined rules and the built-in rules for equations. Note that the user-defined rules come first in the resulting list, thus

implementing their precedence.

$$[\text{PpAsf-0}]\ \frac{\textit{PPrule-list}_1 = \text{rulelistcat}(\textit{PPrule-list}_0, \text{ASF-RULES})}{\text{PpAsf}(\textit{PPrule-list}_0, AT) = \text{DoPpATerm}(\textit{PPrule-list}_1, \textit{PPrule-list}_1, AT)}$$

The PpSdf function is analogous to PpAsf.

$$[\text{PpSdf-0}]\ \frac{\textit{PPrule-list}_1 = \text{rulelistcat}(\textit{PPrule-list}_0, \text{SDF-RULES})}{\text{PpSdf}(\textit{PPrule-list}_0, AT) = \text{DoPpATerm}(\textit{PPrule-list}_1, \textit{PPrule-list}_1, AT)}$$

$$[\text{PpATerm-0}]\ \text{PpATerm}(\textit{PPrule-list}, AT) = \text{DoPpATerm}(\textit{PPrule-list}, \textit{PPrule-list}, AT)$$

The function DoPpATerm is the heart of the prettyprinter. It has two copies of the rule-list. The second copy of the rule-list only contains the rules that haven't been used yet in the current rewriting process. The first copy contains all the rules in the system. When a rewriting operation is succesful, the full list of rules must be applied on the children of the to-be-rewritten ATerm. DoPpATerm takes the list of rules, and tries to match the current rule to the current ATerm. If succesful, the match-function returns a list of 0 or more bindings. These sub-ATerms are recursively rewritten and substituted in the box-part of the rule. The box, with all its holes filled with (recursively) rewritten ATerms, is the end result of the PpAsFix module. Note that the substitute-function recursively calls DoPpAsFix. Therefore, it needs the full list of rewriting rules (in SystemRules). This complete list is then passed on to DoPpAsFix during the recursive call, assuring all rules in the system can be applied on all levels of recursion.

$$[\text{DoPpATerm-0}]\ \frac{M = \text{match}(AT_0, AT),\ M \neq \text{NoMatch}}{\text{DoPpATerm}(\textit{SystemRules}, < AT_0, B_0 >; \textit{PPrule-list}, AT) = \text{substitute}(\textit{SystemRules}, M) \text{ in } B_0}\ \textbf{otherwise}$$

If the match between rule and ATerm fails (i.e. a NoMatch result), then the "offending" rule is removed from the list of untried rules. Then the system tries to match the ATerm to the next rule, using a recursive call. Note that "SystemRules" is the list of rules that remains intact, so that it may pass on its information to later generations, whereas "PPrule-list1" is the list of rules that haven't been tried yet. This latter list constantly shrinks during the rewriting of an ATerm.

$$[\text{DoPpATerm-1}]\ \frac{M = \text{match}(AT_0, AT),\ M = \text{NoMatch}}{\text{DoPpATerm}(\textit{SystemRules}, < AT_0, B_0 >; \textit{PPrule-list}, AT) = \text{DoPpATerm}(\textit{SystemRules}, \textit{PPrule-list}, AT)}\ \textbf{otherwise}$$

[DoPpATerm-2] $\dfrac{AT = [[\texttt{"CfFunction"}\ [\texttt{"CfElems"}\ AT_1];\ [\texttt{"Sort"}\ AT_2];\ [\texttt{"Attributes"}\ AT_3]]\ AT_4].\quad AT_4 \neq \text{nil}}{\begin{array}{l}\text{DoPpATerm}(\textit{SystemRules}, \textit{PPrule-list}, AT) = \\ \text{DoPpATerm}(\textit{SystemRules}, \textit{PPrule-list}, \text{prep}(AT))\end{array}}$

The equations DoPpATerm-4 to DoPpATerm-8 are the so-called "hard-specified default rules". (This prevents confusion with the built-in rules for rewriting ASF and SDF, since these are also sometimes referred to as "default rules"). These rules are used when no single rule has matched the current ATerm.

[DoPpATerm-4] DoPpATerm(*SystemRules*, . L) = L2S(L)

[DoPpATerm-5] DoPpATerm(*SystemRules*, , $[AT_0\ AT_1]$)
= V [DoPpATerm(*SystemRules*, *SystemRules*, AT_0)
DoPpATerm(*SystemRules*, *SystemRules*, AT_1)]

[DoPpATerm-6] DoPpATerm(*SystemRules*, , AT_0; AT_1)
= LST[DoPpATerm(*SystemRules*, *SystemRules*, AT_0)
DoPpATerm(*SystemRules*, *SystemRules*, AT_1)]

[DoPpATerm-7] DoPpATerm(*SystemRules*, , nil) = H []
[DoPpATerm-8] DoPpATerm(*SystemRules*, . AT_0 / AT_1) = DoPpATerm(*SystemRules*, *SystemRules*, AT

The substitute-function is rather straightforward. Note the difference between "substitute" and "listsubstitute". "substitute" results in a BOX, whereas "listsubstitute" results in a BOX-LIST. The substitute-function descends in a box, keeping in memory the set of rewriting rules and the to-be-substituted MatchResult. With an EmptyMatch result, of course, there is nothing to substitute in the first place.

[substitute-0] substitute(*SystemRules*, EmptyMatch) in $B = B$

When a hole is found, the corresponding binding is found, prettyprinted, and substituted.

[substitute-1] substitute(*SystemRules*, M) in [*Int*] = PpATerm(*SystemRules*, lookup *Int* in M)

Substitution in a string is substituting in a box without holes - there's nowhere to substitute something in.

[substitute-2] substitute(*SystemRules*, M) in *String* = *String*

The remainder of substitution functions implements the descent into the box-expression.

[substitute-3] substitute(*SystemRules*, M) in H $so^*[Bl_1]$

= H so^*[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-4] substitute(*SystemRules*, *M*) in V $so^*[Bl_1]$
= V so^*[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-5] substitute(*SystemRules*, *M*) in HV $so^*[Bl_1]$
= HV so^*[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-6] substitute(*SystemRules*, *M*) in HOV $so^*[Bl_1]$
= HOV so^*[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-7] substitute(*SystemRules*, *M*) in I $so^*[B]$
= I so^*[substitute(*SystemRules*, *M*) in *B*]

[substitute-8] substitute(*SystemRules*, *M*) in WD[*B*]
= WD[substitute(*SystemRules*, *M*) in *B*]

[substitute-9] substitute(*SystemRules*, *M*) in *f*[*B*]
= *f*[substitute(*SystemRules*, *M*) in *B*]

[substitute-10] substitute(*SystemRules*, *M*) in R[Bl_1]
= R[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-11] substitute(*SystemRules*, *M*) in A (*ao**) $so^*[Bl_1]$
= A (*ao**) so^*[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-12] substitute(*SystemRules*, *M*) in LST[Bl_1]
= LST[listsubstitute(*SystemRules*, *M*) in Bl_1]

[substitute-13] substitute(*SystemRules*, *M*) in O $so^*[B_0$ *String* $B_2]$
= O so^*[substitute(*SystemRules*, *M*) in B_0
String
substitute(*SystemRules*, *M*) in B_2]

[substitute-14] substitute(*SystemRules*, *M*) in SEP-I[*B Bl*]
= SEP-I[substitute(*SystemRules*, *M*) in *B*
listsubstitute(*SystemRules*, *M*) in *Bl*]

[substitute-15] substitute(*SystemRules*, *M*) in SEP-A[*B Bl*]
= SEP-A[substitute(*SystemRules*, *M*) in *B*
listsubstitute(*SystemRules*, *M*) in *Bl*]

[substitute-16] substitute(*SystemRules*, *M*) in A-SEP[*B Bl*]

$$= \text{A-SEP}[\text{substitute}(\mathit{SystemRules},\ M) \text{ in } B \\ \text{listsubstitute}(\mathit{SystemRules},\ M) \text{ in } Bl]$$

An empty list has no holes to substitute anything in. Therefore substituting in an empty list yields an empty list.

[listsubstitute-1] listsubstitute($\mathit{SystemRules}$, M) in $=$

Substituting in a non-empty list goes step by step: substitute in the head of the list, then substitute in the tail.

[listsubstitute-2] listsubstitute($\mathit{SystemRules}$, M) in $B\ B^*$
$= \text{substitute}(\mathit{SystemRules},\ M) \text{ in } B \mathbin{+\!\!+} \text{listsubstitute}(\mathit{SystemRules},\ M) \text{ in } B^*$

The function "lookup" yields binding Int from MatchResult M, provided the pretty-print rules were correct.

[lookup-0] lookup Int in $(\mathit{Int},\ AT)\text{O}\ M = AT$

$$\text{[lookup-1]}\ \frac{\mathit{Int} \neq \mathit{Int}_2}{\text{lookup } \mathit{Int} \text{ in } (\mathit{Int}_2,\ AT)\text{O}\ M\ =\ \text{lookup } \mathit{Int} \text{ in } M}$$

Note that equation lookup-2 only occurs when your rules weren't correct.

[lookup-2] lookup Int in $=$ ["PP-error" "Unknown␣nonterminal"]

L2S translates a literal into a string. The trick that was used works only on functions defined in the lexical syntax. That's why we use "box-str-con" instead of "box-string" or "string".

[L2S-0] L2S(literal(""" c^* """)) = box-str-con(""" c^* """)

B.6 Match

imports ATerms Box-Ints Rules
exports
  sorts MatchResult Binding
  context-free syntax
    "match" "(" ATerm "," ATerm ")" → MatchResult

    "NoMatch" → Binding

    "EmptyMatch" → Binding

    "(" BOX-INT "," ATerm ")" → Binding

{Binding "O"}* → MatchResult

variables

"*Bind*"[0-9]* → Binding

"*M*"[0-9]* → {Binding "O"}*

hiddens

context-free syntax

"matchcat" "(" MatchResult "," MatchResult ")" → MatchResult

equations

The purpose of module "Match" is to find out if an ATerm and an "AContext" match. Two ATerms match when they are the same modulo instantiating nonterminals. The result of the matching operation is a list those ATerms that correspond to nonterminals. This may be an empty list (EmptyMatch), a single ATerm, or a list of ATerms (injected in the sort "Binding") separated by "O" (capital "o") symbols. Alternatively, if the matching process fails, it returns NoMatch. All of the above values are injected in the sort MatchResult. The following rule binds a nonterminal to an ATerm. Note that the Match- Result still has to keep track of the numberof the nonterminal.

[match-0] match(N[*Int*], *T*) = (*Int*, *T*)

The following two rules are trivial: two equal ATerms without nonterminals match, but don't have bindings.

[match-1] match(L, L) = EmptyMatch

[match-2] match(nil, nil) = EmptyMatch

The following three rules follow the same pattern. Note how the NoMatch result is propagated. If the lower levels have returned NoMatch, then the higher levels won't pass the conditions for further matching. Thus, the default rule remains the only rule that can be applied to a pair of matches with one or more NoMatch results. The default rule, however, yields NoMatch. Thus the NoMatch result is propagated to a higher level.

$$\text{[match-3]}\ \frac{\begin{array}{l} M_1 = \text{match}(T_1, T_3), \\ M_2 = \text{match}(T_2, T_4), \\ M_1 \neq \text{NoMatch}, \\ M_2 \neq \text{NoMatch} \end{array}}{\text{match}(T_1; T_2, T_3; T_4) = \text{matchcat}(M_1, M_2)}$$

$$\text{[match-4]}\ \frac{\begin{array}{l} M_1 = \text{match}(T_1, T_3), \\ M_2 = \text{match}(T_2, T_4), \\ M_1 \neq \text{NoMatch}, \\ M_2 \neq \text{NoMatch} \end{array}}{\text{match}([T_1\ T_2], [T_3\ T_4]) = \text{matchcat}(M_1, M_2)}$$

$$\text{[match-5]}\ \frac{\begin{array}{l} M_1 = \text{match}(T_1,\ T_3), \\ M_2 = \text{match}(T_2,\ T_4), \\ M_1 \neq \text{NoMatch}, \\ M_2 \neq \text{NoMatch} \end{array}}{\text{match}(T_1\ /\ T_2,\ T_3\ /\ T_4) = \text{matchcat}(M_1,\ M_2)}$$

[match] match(T_0, T_1) = NoMatch **otherwise**

We are now ready to witness the non-propagation of EmptyMatch results. We have seen above that non-leaf ATerms always call "matchcat()" for combining their MatchResults. We have also seen that these MatchResults, when matchcat() is called, can no longer be NoMatches. Now there are four possibilities: (a) neither the left nor the right MatchResult is an EmptyMatch.

$$\text{[matchcat-0]}\ \frac{\begin{array}{l} M_1 \neq \text{EmptyMatch}, \\ M_2 \neq \text{EmptyMatch} \end{array}}{\text{matchcat}(M_1,\ M_2) = M_1 \text{O}\ M_2}$$

(b) The right MatchResult is an EmptyMatch.

[matchcat-1] matchcat(M, EmptyMatch) = M

(c) The left MatchResult is an EmptyMatch.

[matchcat-2] matchcat(EmptyMatch, M) = M

(d) Both MatchResults are EmptyMatch. In this case, we want matchcat() to return a single EmptyMatch. Both equation [matchcat-1] and equation [matchcat-2] will do just that when they must rewrite "matchcat(EmptyMatch, EmptyMatch)". This is why neither [matchcat-1] nor [matchcat-2] has conditions concerning the value of M.

B.7 Holes

Holes
imports Box Alignments Fonts Over
exports
  context-free syntax
    "[" BOX-INT "]" → BOX
    "closed-boxterm" "(" BOX ")" → BOX-BOOL
    "closed-boxterm" "(" BOX-LIST ")" → BOX-BOOL
hiddens
  variables
    "*B*"[*0-9'*]* → BOX

"*Bl*"[*0-9'*]* → BOX-LIST
"*B**"[*0-9'*]* → BOX*
equations

[cb-1] closed-boxterm([*Int*]) = false
[cb-2] closed-boxterm(*String*) = true
[cb-3] closed-boxterm(H *so**[*Bl*]) = closed-boxterm(*Bl*)
[cb-4] closed-boxterm(V *so**[*Bl*]) = closed-boxterm(*Bl*)
[cb-5] closed-boxterm(HV *so**[*Bl*]) = closed-boxterm(*Bl*)
[cb-6] closed-boxterm(HOV *so**[*Bl*]) = closed-boxterm(*Bl*)
[cb-7] closed-boxterm(I *so**[*B*]) = closed-boxterm(*B*)
[cb-8] closed-boxterm(WD[*B*]) = closed-boxterm(*B*)
[cb-9] closed-boxterm(*f*[*B*]) = closed-boxterm(*B*)
[cb-10] closed-boxterm(R[*Bl*]) = closed-boxterm(*Bl*)
[cb-11] closed-boxterm(A (*ao**) *so**[*Bl*]) = closed-boxterm(*Bl*)
[cb-12] closed-boxterm(LST[*Bl*]) = closed-boxterm(*Bl*)
[cb-13] closed-boxterm(O *so**[B_0 *String* B_2]) = closed-boxterm(B_0) & closed-boxterm(B_2)

$$\text{[cb-20]}\ \frac{Bl\ =\ B\ B^*}{\text{closed-boxterm}(Bl)\ =\ \text{closed-boxterm}(B)\ \&\ \text{closed-boxterm}(B^*)}$$

$$\text{[cb-21]}\ \frac{Bl\ =}{\text{closed-boxterm}(Bl)\ =\ \text{true}}$$

B.8 PrepCf

PrepCf
imports Rules Box-Booleans
exports
  context-free syntax
    "prep" "(" ATerm ")" → ATerm
hiddens
  context-free syntax
    "prepare" "(" ATerm "with" ATerm ")" → ATerm
    "is-list" "(" ATerm ")" → BOX-BOOL
  variables
    "*L*"[*0-9'*]* → Literal
    "*T*"[*0-9'*]* → ATerm
equations

[prep-1] prep([["CfFunction" ["CfElems" T_1]; ["Sort" T_2]; ["Attributes" T_3]] T_4])

$= [[\texttt{"CfFunction"}\ [\texttt{"CfElems"}\ \text{prepare}(T_1\ \text{with}\ T_4)];\ [\texttt{"Sort"}\ T_2];\ [\texttt{"Attributes"}\ T_3]]\ \text{nil}]$

[prep-2] $\text{prep}(L) = L$
[prep-3] $\text{prep}(T_0;\ T_1) = \text{prep}(T_0);\ \text{prep}(T_1)$
[prep-4] $\text{prep}(\text{nil}) = \text{nil}$
[prep-5] $\text{prep}(T_0\ /\ T_1) = \text{prep}(T_0)\ /\ \text{prep}(T_1)$
[prep-6] $\text{prep}([T_0\ T_1]) = [\text{prep}(T_0)\ \text{prep}(T_1)]$ **otherwise**

Note that "prepare (["Sort" T1] with T3;T4) = T3;T4" has the same pattern as [prepare-1]. So whether T3 = T4;T5 or not, the result remains the same in [prepare-1].

[prepare-1] $\text{prepare}([\texttt{"Sort"}\ T_1]\ \text{with}\ T_3) = T_3$
[prepare-2] $\text{prepare}([\texttt{"Sort"}\ T_1];\ T_2\ \text{with}\ T_3;\ T_4) = T_3;\ \text{prepare}(T_2\ \text{with}\ T_4)$

[prepare-3] $$\frac{\text{is-list}(T_3) = \text{false}}{\text{prepare}([\texttt{"Sort"}\ T_1];\ T_2\ \text{with}\ T_3) = T_3;\ T_2}$$

For [prepare-4]. same reasoning as for [prepare-1].

[prepare-4] $\text{prepare}([\texttt{"Iter"}\ T_1]\ \text{with}\ T_3) = T_3$
[prepare-5] $\text{prepare}([\texttt{"Iter"}\ T_1];\ T_2\ \text{with}\ T_3;\ T_4) = T_3;\ \text{prepare}(T_2\ \text{with}\ T_4)$

[prepare-6] $$\frac{\text{is-list}(T_3) = \text{false}}{\text{prepare}([\texttt{"Iter"}\ T_1];\ T_2\ \text{with}\ T_3) = T_3;\ T_2}$$

[prepare-7] $\text{prepare}(T_1;\ T_2\ \text{with}\ T_3) = T_1;\ \text{prepare}(T_2\ \text{with}\ T_3)$ **otherwise**
[prepare-8] $\text{prepare}(T_1\ \text{with nil}) = T_1$ **otherwise**

[is-list-0] $\text{is-list}(T_0;\ T_1) = \text{true}$
[is-list-1] $\text{is-list}(T) = \text{false}$ **otherwise**

B.9 SepBox

imports Box Holes
exports
context-free syntax
"SEP-I" "[" BOX BOX-LIST "]" → BOX
"SEP-A" "[" BOX BOX-LIST "]" → BOX

"A-SEP" "[" BOX BOX-LIST "]" → BOX

hiddens

variables

"*separator*" → BOX
"*B*"[*0-9*]* → BOX
"*B**"[*0-9*]* → BOX*

equations

The following rewritings transform a SEP-box into a LST-box. However the boxes within the SEP-box must be closed terms, or the following problem may occur: SEP-I ["," [1]] -¿ [1] We get [1], instead of a list bound to [1]. Only when the binding is fulfilled we are allowed to do rewriting. Since SEP is an expansion of BOX, we must first define closed SEP-boxes. This will also help us prevent the undesired rewriting mentioned above.

[cb-14] closed-boxterm(SEP-I[$B\ B^*$]) = closed-boxterm(B) & closed-boxterm(B^*)
[cb-15] closed-boxterm(SEP-A[$B\ B^*$]) = closed-boxterm(B) & closed-boxterm(B^*)
[cb-16] closed-boxterm(A-SEP[$B\ B^*$]) = closed-boxterm(B) & closed-boxterm(B^*)

Equations for the Insert-Separator, SEP-I Notice that an empty box-term is automatically closed.

$$\text{[sepI-0]}\ \frac{B^* = }{\text{SEP-I}[\textit{separator}\ B^*] = }$$

$$\text{[sepI-1]}\ \frac{B^* = B,\quad \text{closed-boxterm}(B) = \text{true}}{\text{SEP-I}[\textit{separator}\ B^*] = B}$$

$$\text{[sepI-2]}\ \frac{B^* = B\,B_1^*,\quad B_1^* \neq ,\quad \text{closed-boxterm}(B\ B_1^*) = \text{true}}{\text{SEP-I}[\textit{separator}\ B^*] = \text{LST}[B\ \textit{separator}\ \text{SEP-I}[\textit{separator}\ B_1^*]]}$$

Equations for the post-append separator, SEP-A Once again, an empty box-term is always closed.

$$\text{[sepA-0]}\ \frac{B^* = }{\text{SEP-A}[\textit{separator}\ B^*] = }$$

Note that SEP-A yields just "B" for a singleton list, not "B separator" - there is nothing to SEPARATE.

$$\text{[sepA-1]}\ \frac{B^* = B,\quad \text{closed-boxterm}(B) = \text{true}}{\text{SEP-A}[\textit{separator}\ B^*] = B}$$

$$[\text{sepA-2}]\ \frac{\begin{array}{r} B^* = B\,B_1^*, \\ B_1^* \neq \ , \\ \text{closed-boxterm}(B\,B_1^*) = \text{true} \end{array}}{\text{SEP-A}[\textit{separator}\ B^*] = \text{LST}[\text{H}\ [B\ \textit{separator}]\ \text{SEP-A}[\textit{separator}\ B_1^*]]}$$

Equations for the pre-separator A-SEP. See comment at [sepA-1] and [sepI-1].

$$[\text{Asep-0}]\ \frac{B^* =}{\text{A-SEP}[\textit{separator}\ B^*] =}$$

See comment at [sepA-1].

$$[\text{Asep-1}]\ \frac{\begin{array}{r} B^* = B, \\ \text{closed-boxterm}(B) = \text{true} \end{array}}{\text{A-SEP}[\textit{separator}\ B^*] = B}$$

$$[\text{Asep-2}]\ \frac{\begin{array}{r} B^* = B\,B_1^*, \\ B_1^* \neq \ , \\ \text{closed-boxterm}(B\,B_1^*) = \text{true} \end{array}}{\text{A-SEP}[\textit{separator}\ B^*] = \text{LST}[\text{H}\ [B\ \textit{separator}]\ \text{A-SEP}[\textit{separator}\ B_1^*]]}$$

B.10 AsfRules

AsfRules
imports Rules Fonts Over SepBox
exports
context-free syntax
"ASF-RULES" → PPrule-list
equations

Below are the built-in rules for prettyprinting ASF. They are translated from dr. van den Brands module "PpEqs" as far as possible. Unfortunately, there is no perfect correspondence.

[asfrules] ASF-RULES

```
= < ["Module" N[0]; N[1]; N[2]; N[3]; N[4]; N[5]], V [[5]] >;
  < nil, H [] >;
  < ["PrefixEquations" nil], H [] >;
  < ["PrefixEquations" N[0]], V vs = 2[KW["equations"] I [V vs = 2[[0]]]] >;
  < ["CondEquation" "Implies"; N[0]; N[1]; N[2]; ["Conditions" nil]],
    H ["[" [0] "]" HV [H [HV [[1]] "="] HV is = 0[[2]]]] >;
  < ["CondEquation" "Implies"; N[0]; N[1]; N[2]; ["Conditions" N[3]]],
    H ["[" [0] "]" O [HOV [SEP-A["," [3]]] "=" HV [H [HV [[1]] "="] HV is = 0[[2]]]]] >;
  < ["CondEquation" "When"; N[0]; N[1]; N[2]; ["Conditions" N[3]]],
    H ["[" [0] "]" HOV [HV [H [HV [[1]] "="] HV is = 0[[2]]
                                KW["when"] HV [SEP-A["," [3]]]]] >;
  < [["CfFunction" ["CfElems" N[0]]; ["Sort" N[1]]; ["Attributes" N[2]]] N[3]], [0] >;
  < ["Sort" N[0]], [0] >;
  < [["Iter" N[0]; N[1]; N[5]] nil], H [] >;
  < [["Iter" N[0]; ""; N[5]] N[2]; N[3]], LST[[2] [3]] >;
  < [["Iter" N[0]; ["QLit" N[1]]; N[5]] N[3]], SEP-I[[1] [3]] >;
  < [["Iter" N[0]; N[1]; N[5]] N[2]], [2] >;
  < ["Var" N[0]; N[1]], [0] >;
  < ["Lex" N[0]; ["Sort" "BOX-STR-CON"]], H hs = 0[dquote [0] dquote] >;
  < ["Lex" N[0]; N[1]], [0] >;
  < ["MetaVar" N[0]; N[1]], [0] >;
  < ["Condition" "Eq"; N[0]; N[1]], H [HV [[0]] "=" HV [[1]]] >;
  < ["Condition" "Neq"; N[0]; N[1]], H [HV [[0]] "!=" HV [[1]]] >;
  < ["QLit" N[0]], [0] >
```

B.11 SdfRules

SdfRules
imports Rules Fonts Alignments SepBox
exports
context-free syntax
"SDF-RULES" → PPrule-list
equations

The rules below are the translation of dr. van den Brands "PpSyn" module for prettyprinting SDF. Once again there is no perfect correspondence between these rules and "PpSyn".

[sdfrules] SDF-RULES

```
= < ["Module" N[0]; N[1]; N[2]; N[3]; N[4]; N[5]], V [H [KW["Module"] [0]] [3]] >;
  < nil, H [] >;
  < N[0] / N[1], [0] >;
  < ["PrefixSyntax" N[0]], [0] >;
  < ["Imports" N[0]], H [KW["imports"] HV is = 0[[0]]] >;
  < ["Exports" N[0]], V [KW["exports"] I [V [[0]]]] >;
  < ["Hiddens" N[0]], V [KW["hiddens"] I [V [[0]]]] >;
  < ["Sort" N[0]], [0] >;
  < ["Id" N[0]], [0] >;
  < ["QLit" N[0]], H hs = 0[[0]] >;
  < "left", KW["left"] >;
  < "right", KW["right"] >;
  < "non-assoc", KW["non-assoc"] >;
  < "assoc", KW["assoc"] >;
  < "bracket", KW["bracket"] >;
  < ["CharClass" N[0]], [0] >;
  < ["Neg" N[0]], H hs = 0["~" [0]] >;
  < ["Iter" N[0]; ""; "*"], H hs = 0[[0] "*"] >;
  < ["Iter" N[0]; ""; "+"], H hs = 0[[0] "+"] >;
  < ["Iter" N[0]; ["QLit" N[1]]; "*"], H hs = 0["{" H [[0] dquote [1] dquote] "}" "*"] >;
  < ["Iter" N[0]; ["QLit" N[1]]; "+"], H hs = 0["{" H [[0] dquote [1] dquote] "}" "+"] >;
  < ["Sorts" N[0]], H [KW["sorts"] HV is = 0[[0]]] >;
  < ["LexicalSyntax" N[0]], V [H [KW["lexical"] KW["syntax"]] I [A (l , c , l ) [[0]]]] >;
  < ["ContextFreeSyntax" N[0]],
    V [H [KW["context-free"] KW["syntax"]] I [A (l , c , l ) [[0]]]] >;
  < ["Priorities" N[0]], V [KW["priorities"] I [V [SEP-I["," [0]]]]] >;
  < ["Variables" N[0]], V ["" KW["variables"] I [A (l , c , l ) [[0]]]] >;
  < ["CfElems" nil], "" >;
  < ["CfElems" N[0]], [0] >;
  < ["LexElems" N[0]], [0] >;
  < ["LexIter" N[0]; "*"], H hs = 0[[0] "*"] >;
  < ["LexIter" N[0]; "+"], H hs = 0[[0] "+"] >;
  < ["IncrChain" ["FunctionList" N[0]; N[1]]],
    H hs = 0["{" KW[[0]] ":" SEP-A["," [1]] "}"] >;
  < ["IncrChain" N[0]; N[1]],
    V [SEP-A["<" LST[[0] [1]]]] >;
  < ["DecrChain" ["FunctionList" N[0]; N[1]]],
    H hs = 0["{" KW[[0]] ":" SEP-A["," [1]] "}"] >;
  < ["DecrChain" N[0]; N[1]],
    V [SEP-A[">" LST[[0] [1]]]] >;
  < ["FunctionList" ""; N[1]], H hs = 0["{" [1] "}"] >;
  < ["FunctionList" N[0]; N[1]], H hs = 0["{" KW[[0]] H [":" [1]] "}"] >;
  < ["FunctionList" N[0]], [0] >;
  < ["Abbrevs" N[0]], HV is = 0[SEP-I["," [0]]] >;
  < ["Abbrev" N[0]; ["Sort" ""]; N[1]], HV is = 0[[0]] >;
  < ["Abbrev" N[0]; N[1]; N[2]], H [[0] ESC["->"] [1]] >;
  < ["MetaVar" N[0]; N[1]], H [[0]] >;
  < ["LexicalFunction" N[0]; N[1]],
    R[H [[0]] ESC["->"] [1]] >;
  < ["CfFunction" N[0]; N[1]; ["Attributes" nil]],
    R[H [[0]] ESC["->"] [1]] >;
  < ["CfFunction" N[0]; N[1]; ["Attributes" N[2]]],
    R[H [[0]] ESC["->"] H [[1] H hs = 0["{" SEP-I["," [2]] "}"]]] >;
  < ["Variable" N[0]; N[1]],
    R[H hs = 0[[0]] ESC["->"] [1]] >
```

References

[AAFZ92] D.S. Arnon, I. Attali and P. Franchi-Zannettacci, *Language-based Document Processing*, Rapports de Recherche No 1731, Programme 2, Calcul symbolique, Programmation et Génie logiciel.

[BHK89] J.A. Bergstra, J. Heering and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering and P. Klint, editors, *Algebraic Specification.* ACM Press Frontier Series, pages 1-66. The ACM press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BKM89] J.A. Bergstra, J.W. Klop and A. Middeldorp. *Termherschrijfsystemen.* (in Dutch). Kluwer Bedrijfswetenschappen, 1989.

[Bra93a] M.G.J. van den Brand, *Prettyprinting without Loosing Comments*, Programming Research Group, 1993.

[Bra93b] M.G.J. van den Brand, *Generation of Language-Independent Prettyprinters*, Programming Research Group, 1993.

[Bra95] M.G.J. van den Brand, *Pretty Printing in the* ASF+SDF *Meta-environment: Past, Present, and Future.* Technical Report Programming Research Group, University of Amsterdam.

[BS89] G. Blaschek and J. Sametinger, *User-adaptable Prettyprinting*, in *Software-practice and experience*, vol. 19(7), 687-702, July 1989

[BV94a] M. G. J. van den Brand, E. Visser, *From Box to TEX: An Algebraic Approach to the Construction of Documentation Tools*, Programming Research Group, 1994.

[BV94b] M.G.J. van den Brand, E. Visser, *BOX: Language, Laws & Formatters.* Programming Research Group, 1994.

[Deur94] A. van Deursen, *An Algebraic Specification of Term Rewriting and Origin Tracking*, CWI, 1994.

[Gar85] D. Garlan, *Flexible Unparsing in a Structure Editing Environment*, CMU-CS-85-129, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, Pa. 15213

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint and J. Rekers. *The Syntax Definition Formalism SDF, reference manual*, 1992.

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

[Kli94] P. Klint, *Writing meta-level specifications with* ASF+SDF

[Opp80] D.C. Oppen, *Prettyprinting*, ACM Transactions on Programming Languages and Systems, Vol. 2. No 4. October 1980. Pages 465-483

[PPML86] Morcos-Chounet, E., Conchon, A., *PPML: A general formalism to specify prettyprinting*

[Rom95] J. M. T. Romijn, *Automatic Analysis of Term Rewriting Systems*, Master's thesis, technical report P9505, Programming Research Group, University of Amsterdam, 1995.

[Vis94] E. Visser, ASF+SDF *to LaTeX, The User Manual.* Programming Research Group, 1994.

[VK94] P. Klint and E. Visser, *ToLaTeX the Program*, Programming Research Group, University of Amsterdam, 1994.