From Box to TEX: An algebraic approach to the construction of documentation tools

Mark van den Brand*†
Eelco Visser*‡

June 30, 1994

Abstract

We define a translation from an intermediate box language for pretty printing to TEX. This translation can be used as a back-end for pretty printers in documentation tools for programming languages. The translation is formulated in an executable algebraic specification formalism. An important aspect of the translation is the transformation of boxes according to a set of equations. These equations preserve the text formatting semantics of boxes which is also defined algebraically. New in this approach is that algebraic transformations of box terms are used to circumvent the limitations of the typesetter.

The TEX generator, which translates the box language to TEX code, is a component of documentation tools generated for the programming environments developed with the ASF+SDF meta-environment, but can also be used as a separate tool. As a case study, the construction of a typesetter for the process specification formalism PSF is shown.

Keywords: typesetting, pretty printing, box language, programming language, software documentation, transformation, algebraic specification, tool generation, term rewriting

---

*Programming Research Group, University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam, email: {markvdb,visser}@fwi.uva.nl

†Partial support received from ESPRIT project 2177 (GIPE-II Generation of Interactive Programming Environments)

‡Partial support received from NWO project 612-317-420

1 Introduction

Software documentation consists of (partial) listing of a program along with explanations of its goals and functionality. There are essentially two ways to list programs in a document: either including a verbatim listing or a typeset listing of the program as illustrated in Figure 1. The first method is easy and ensures that the documentation is up-to-date with the program, but long verbatim listings do not encourage reading. However, typesetting a listing manually is time consuming and error-prone. Moreover, if the program is changed, its documentation must be updated, either by entirely retypesetting the program or by mirroring the changes in the program in its documentation.

A documentation tool aids the programmer in producing up-to-date documentation of a program. It is as easy to use as verbatim listings, but provides the elegance of typeset listings.

In this paper we will describe how the language dependent parts of a text oriented pretty printer can be reused by a generator of typesetting tools for programming languages. The text formatter that forms the back-end of the text oriented pretty printers is replaced by a back-end that produces TEX[1] [Knu84a]. The back-end can be used to typeset programs but can also be embedded in more sophisticated documentation tools.

2 Pretty Printing

A pretty printer for a programming language rearranges the layout between the tokens of a program to achieve a better readable program. It does this by putting linebreaks in appropriate places and indenting subconstructs to emphasize their embedding in their surrounding context.

A pretty printer can be formulated as a translation (L → Display) from abstract syntax trees[2] over some language L to some displaying device. This translation can be decomposed into a translation (L → Box) from abstract syntax trees to box terms and a translation (Box → Display) from box terms to a displaying device like a screen or a printer. The box language acts as an intermediate representation that is independent of both source language and target device.

Box Language A box is a simple data structure built from strings as basic boxes and a number of box composition operators. Associated with each composition are its subboxes and the spacing between those boxes. A box can also contain information about non-structural aspects of a text, such as fonts and colors.

For instance, from the basic boxes $\boxed{a}$ and $\boxed{b}$ we can build the horizontal composition $\boxed{\boxed{a}\boxed{b}}$ and the vertical composition $\boxed{\begin{array}{c}\boxed{a}\\ \boxed{b}\end{array}}$.

To express all possible linebreakings the box language has compositions other than horizontal and vertical composition. These compositions are illustrated by considering all possible arrangements of the boxes $\boxed{A}$, $\boxed{B}$, $\boxed{C}$:

(a) (b) (c) (d)

[1]We will speak about TEX, not excluding LATEX. The code that will be generated by our formatter can be used both in plain TEX and in LATEX documents.

[2]We assume the existence of an abstract syntax tree; it can be the result of parsing a text with (Text → L) or it can be built up in a syntax directed editor in which case no textual origin of the tree might exist. Furthermore, we will identify a language L with its abstract syntax.

```
begin
  declare
    input: natural,
    output: natural,
    repnr: natural,
    rep: natural;

    input := 3;
    output := 1;
    while  input - 1
    do
      rep := output;
      repnr := input;
      while repnr - 1
      do
        output := output + rep;
        repnr := repnr - 1
      od;
      input := input - 1
    od
end
```

begin
  declare
    $input$: natural,
    $output$: natural,
    $repnr$: natural, rep : natural;
  $input := 3$; $output := 1$;
  while $input - 1$ **do**
    $rep := output$; $repnr := input$;
    while $repnr - 1$ **do**
      $output := output + rep$;
      $repnr := repnr - 1$
    od; $input := input - 1$
  od
end

Figure 1: Verbatim and typeset listing of a Pico program that computes the factorial of the variable *input*.

In case (a) the boxes are placed horizontally, in case (b) there is a linebreak after $\boxed{B}$, in case (c) there is a linebreak after $\boxed{A}$ and in case (d) the boxes are arranged completely vertically. The structural operators can be characterized by indicating which of these cases they cover. A horizontal composition (H) can only set boxes as in case (a). A vertical composition (V) can only produce case (d). The composition horizontal *or* vertical (HOV) can produce either (a) or (d): (a) if they fit on one line and (d) otherwise. Finally, composition horizontal *and* vertical (HV) can produce all four cases: boxes are put on the same line as long as that can be done, and a linebreak is inserted when the line is full.

Box Construction The front-end of a pretty printer, the *box constructor*, can be considered as a compositional, syntax-directed mapping from abstract syntax trees to box expressions. For example, consider the rules

$$pp((\bullet)) \quad = \quad \boxed{\boxed{(}\; pp(\bullet)\; \boxed{)}} \qquad \text{and} \qquad pp(\bullet_1 + \bullet_2) \quad = \quad \boxed{\begin{array}{l} pp(\bullet_1) \\ \boxed{\boxed{+}\; pp(\bullet_2)} \end{array}}$$

for tree nodes of the form $(\bullet)$ and $\bullet + \bullet$, respectively. They specify that brackets should be wrapped around their argument with a horizontal composition and that a '+' node should be displayed as a vertical composition of the first argument and a horizontal composition of '+' with the second argument. Applying these rules to the tree $(a + b)$ gives

$$pp((a+b)) \quad = \quad \boxed{\boxed{(}\; pp(a+b)\; \boxed{)}} \quad = \quad \boxed{\boxed{(}\boxed{\begin{array}{l} \boxed{a} \\ \boxed{\boxed{+}\boxed{b}} \end{array}}\boxed{)}}$$

Text Formatting The box construction rules for a language declare in an abstract way how its constructs should be displayed. The interpretation of a box is done by a *text formatter* that translates the high level box instructions to low level display instructions such as 'print a string' or 'go to new line'. The formatter has to make decisions like setting a string on the current line or continuing on the next one if the current line is too full.

For instance, the formatter (Box → Text) may produce a list of strings of ASCII characters, where each string corresponds to a line of text, that can be displayed by an editor or printer. For the box $pp((a + b))$ it would produce the strings `"(a"` and `" +b)"`, i.e., the text

```
(a
 +b)
```

Assuming that text is displayed with non-proportional fonts and that the width of the display is fixed, spacing issues like indentation and interword space can be dealt with by counting the characters in strings.

If the output of a text formatter has to be used as input for typecheckers, compilers or syntax directed editors—or any other tool that first parses its input with (Text → L)—the composition

$$(\mathrm{L} \rightarrow \mathrm{Box}) \circ (\mathrm{Box} \rightarrow \mathrm{Text}) \circ (\mathrm{Text} \rightarrow \mathrm{L})$$

is usually required to be the identity function on L. A pretty printer should not change the abstract syntax of a program—let alone its meaning.

Other Formatters The decomposition of pretty printing into box construction and formatting creates a potential for software reuse. A formatter for some display device can be used for any language for which a box constructor exists. By writing a formatter for a new display device all existing pretty printers can produce code for that device. For instance, if we have the box constructor (Pascal → Box) and the formatter (Box → Text), we can make the pretty printer (Pascal → Text). If we make a box constructor (Cobol → Box) we can reuse the existing formatter to get (Cobol → Text). If we write a new formatter (Box → TEX), it can be used as back-end for both box constructors.

We must ensure that the meaning of the box construction rules for the new formatter and for the text formatter is the same. If this would not be the case it would not make sense to reuse the pretty printer front-ends. If two boxes have the same result on some text display, they should give similar results on any other display.

3 TEX as Formatter

Since TEX is a box language itself it would seem trivial to translate our box language to TEX directly. Unfortunately, this is not the case, because (1) our notion of horizontal and vertical composition is not the same as TEX's `\hbox` and `\vbox`, (2) TEX's nesting capacity is limited, and (3) we need to control linebreaking.

Horizontal and Vertical Composition The semantics of the horizontal and vertical composition in our box language is incompatible with the semantics of TEX's `\hbox` and `\vbox`. For instance, consider the composition

$$\mathsf{H}[\ (\ \mathsf{V}[\ a\ \mathsf{H}[+b]]\ )\ ]\ .$$

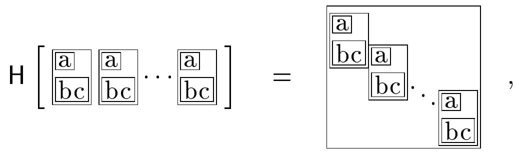
A direct translation to TEX would be

```
\hbox{(\vbox{\hbox{a}\hbox{+b}})}
```

or

```
\hbox{(\vtop{\hbox{a}\hbox{+b}})}
```

which result in $\begin{array}{l}\text{a}\\ \text{(+b)}\end{array}$ and $\begin{array}{l}\text{(a)}\\ \text{+b}\end{array}$, respectively, since a `\vbox` aligns at its bottom line and a `\vtop` aligns at its top line in a horizontal composition. The desired result, however, is $\begin{array}{l}\text{(a}\\ \text{ +b)}\end{array}$. In general, our notion of horizontal composition is:

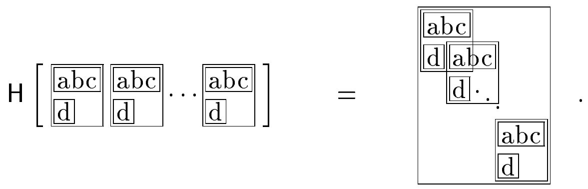


where a box in a horizontal composition aligns with its top at the bottom of the previous box and with its bottom at the top of the next box, where top (bottom) corresponds to the top (bottom) text line in a box.

The next problem that we face is illustrated by the composition

$$\mathsf{H}[\ (\ \mathsf{V}[\ \mathsf{H}\ [\ a+\ ]\ b]\ )\ ]\ .$$

With this semantics we get $\begin{array}{l}\text{(a+}\\ \text{ b)}\end{array}$ whereas we want $\begin{array}{l}\text{(a+}\\ \text{ b)}\end{array}$, which requires that a box in a horizontal composition be placed immediately after the end of the last line of the previous box, i.e.,



In Section 8.2 we will define formatter independent transformations on boxes that simplify boxes in such a way that they can be translated to TEX in a straightforward manner.

Nesting Nesting is needed since box terms are built up compositionally from a syntax tree; the box expression has at least as much—and probably more—nesting than the original tree. This presents a problem when translating to TEX, since TEX's 'semantic nest size' is limited to 40 (in a standard configuration).

This problem can be solved by applying transformations on box expressions that reduce nesting as much as possible. These transformations reduce horizontal boxes to consist only of basic boxes. In other words, a nested box expression is transformed into a flat list of lines.

Linebreaking When we translate boxes to TEX we are unable to precompute where the linebreaks in HOV and HV boxes should be placed, because we have no information about the width of the display or the sizes of characters. Therefore, the TEX code that we generate has to know where it may continue on the next line. This can not be solved by transformations, but has to be done by clever macros.

4 Tool Generation from Algebraic Specifications

4.1 ASF+SDF

ASF+SDF is a modular algebraic specification formalism developed for the description of language definitions. The formalism is supported by an integrated programming environment generator, the ASF+SDF meta-environment [Kli93], which supports the interactive development of a specification, as well as the generation of the associated programming environment. Such a programming environment contains several implicitly derived tools.

The ASF+SDF formalism is a combination of the formalisms SDF (Syntax Definition Formalism) [HHKR93] and ASF (Algebraic Specification Formalism) [BHK89]. SDF provides a formalism for the definition of lexical and context-free syntax. Given an SDF definition a structured editor and a parser are derived by the ASF+SDF meta-environment.

ASF is used to define the semantics of a language. An ASF specification consists of a set of conditional equations over the language defined in the corresponding SDF part. The equations of an ASF specification are interpreted as rewrite rules in a term rewrite system. In terms of functional programming this is similar to a strict first order functional programming language, where the left-hand sides of function definitions can be arbitrary patterns and functions do not have to be defined for all cases. There is actually no distinction between functions and datatypes.

The only non-standard feature of ASF+SDF are associative lists.

ASF+SDF is not only used for defining programming languages and their semantics, but also for the development of tools. The typical architecture of these tools is: given the languages A and B, the tool defines the mapping (A $\rightarrow$ B) from A to B, where this mapping can be decomposed into several submappings.

$$(A \rightarrow A_1) \circ (A_1 \rightarrow A_2) \circ (A_2 \rightarrow A_3) \circ \cdots \circ (A_n \rightarrow B)$$

4.2 Architecture of text pretty printers

A text pretty printer can be decomposed as

$$(\text{Text} \rightarrow \text{L}) \circ (\text{L} \rightarrow \text{Box}) \circ (\text{Box} \rightarrow \text{Text})$$

The mapping (Text $\rightarrow$ L) is a parser for L generated from the SDF part of a language definition. The mapping (L $\rightarrow$ Box) is a box constructor for a language L that can be generated from the context-free grammar G_L for L [Bra93a]. Basically this generator defines for each production in G_L a default equation defining the function *pp* that maps the abstract syntax tree node that corresponds with the production to a box composition of its arguments.

Various modifications are possible ranging from simple rearrangements of boxes to the introduction of columns and user options to influence pretty printing. The default box construction rules are based on the assumption that the source language is a Pascal-like language. Therefore, more adaptation is needed for a more remote language.

The pretty print generator is completely specified in ASF+SDF. It has been used to generate pretty printers for, among others, Cobol, Lisp, and Action Semantics.

The mapping (Box $\rightarrow$ Text) is a text formatter that produces plain ASCII text. During the last two years several versions of this text formatter, all specified in ASF+SDF, have been developed ranging from inefficient ones to a very efficient one. The compiled version of this efficient text formatter is currently being used in the ASF+SDF meta-environment itself. In Sections 7, 8, and 9 we present the most algebraic version.

We study the specification of a text formatter to understand the semantics of the box language. With that knowledge we can derive transformations on boxes that enable us to give a direct translation of boxes to TEX.

4.3 Design of TEX back-end

The TEX typesetter for a language L can be decomposed as

$$(\text{Text} \rightarrow \text{L}) \circ (\text{L} \rightarrow \text{Box}) \circ (\text{Box} \rightarrow \text{Box}) \circ (\text{Box} \rightarrow \text{\TeX}) \circ (\text{\TeX} \rightarrow \text{DVI})$$

The first two components, the parser (Text → L) and the box constructor (L → Box) can be reused from a text pretty printer for L. The component (TEX → DVI) is the program `tex` or `latex`.

The mapping (Box → Box) transforms boxes so that they can be translated easily to TEX, preserving the formatting semantics of (Box → Text), i.e.,

$$(\text{Box} \rightarrow \text{Box}) \circ (\text{Box} \rightarrow \text{Text}) = (\text{Box} \rightarrow \text{Text})$$

The box transformations may not influence the text produced by the text formatter.

The mapping (Box → TEX) translates boxes in normal form to TEX code. Box operators are represented by macros. We provide a the definitions of these macros in TEX.

In Section 10 we present a specification in ASF+SDF of a translation from boxes to TEX code; that is, we will specify the components (Box → Box) and (Box → TEX). In Section 11 we present an application of this back-end in a typesetter for the specification language PSF.

5 Box Language

The syntax of our box language [Bra93a, Bra93b], has been influenced by the syntactic representation of PPML [MCC86] boxes. This paper concentrates on the core of the language that describes structural compositions. In appendix B we extend this core with fonts.

A box is built by application of a box operator to a list of boxes. Basic boxes are strings. Each application can have any number of space options that govern the spacing between the boxes in the composition. We will use the letters A, B, C and D (possibly indexed by a number) to denote boxes. Box lists of zero (one) or more boxes are denoted by a box variable with a * ($^+$).

imports Strings$^{(6.1)}$
exports
  sorts BOX BOX-LIST
  context-free syntax

STRING	→ BOX	
OPERATOR SPACE-OPTIONS "[" BOX-LIST "]"	→ BOX	
BOX*	→ BOX-LIST	
"(" BOX ")"	→ BOX	{**bracket**}

  variables

[*A-D*][*0-9′*]*	→ BOX
[*A-D*]"*"[*0-9′*]*	→ BOX*
[*A-D*]"+"[*0-9′*]*	→ BOX+

Basic Operators The basic structural operators of the box language are H (horizontal composition), V (vertical composition), I (indentation), HOV (horizontal or vertical composition), HV (horizontal and/or vertical composition), and WD (invisible box with same width as some visible box). An arbitrary operator is denoted by O.

sorts OPERATOR
context-free syntax
"H" → OPERATOR
"V" → OPERATOR
"I" → OPERATOR
"HOV" → OPERATOR
"HV" → OPERATOR
"WD" → OPERATOR
variables
$[O][0\text{-}9']*$ → OPERATOR

Space Spacing between boxes in a list can be adapted by instantiating the space options of a composition. We allow horizontal (hs), vertical (vs) and indentation (is) space. The amount of space is indicated by a number, which is an integer or real. The amount of space this number represents depends on the formatter, but a good agreement to eliminate surprises would be to interpret a number n as n times the width of the letter M in the current font for horizontal and indentation space and n times the height of the letter X for vertical space.

The constants $\mathcal{H}$, $\mathcal{I}$ and $\mathcal{V}$ represent the default values for horizontal, indentation and vertical space, respectively. When no space options are given in a composition, these values are used.

imports Numbers-Syntax[(A.3)]
exports
sorts SPACE-OPERATOR SPACE-OPTION SPACE-OPTIONS
context-free syntax
hs → SPACE-OPERATOR
vs → SPACE-OPERATOR
is → SPACE-OPERATOR
SPACE-OPERATOR "=" NUM → SPACE-OPTION
SPACE-OPTION* → SPACE-OPTIONS
"$\mathcal{H}$" → NUM
"$\mathcal{I}$" → NUM
"$\mathcal{V}$" → NUM
variables
$[S][0\text{-}9']*$ → SPACE-OPERATOR
"so" $[0\text{-}9']*$ → SPACE-OPTION
"so" "$*$" $[0\text{-}9']*$ → SPACE-OPTION*
$[o][0\text{-}9']*$ → SPACE-OPTIONS

Examples With this syntax we can construct powerful boxes. In Section 2 we saw the box construction for $(a + b)$ by the box

H["(" V["a" H["+" "b"]] ")"]

This expression can be expressed better by the box

H["(" HOV["a" H["+" "b"]] ")"]

It will set the expression on one line if there is enough room, but will break before the "+" otherwise.

Another application of HOV is in typesetting an if-then-else construct

```
HOV ["if" I[H["a" "<" "b"]]
     "then" I[H["x" ":=" "17"]]
     "else" I[H["x" ":=" "18"]] ]
```

This box declares that, if the entire if-then-else fits on the rest of the line, it should be formatted as

```
if a < b then x := 17 else x := 18
```

but, if there is not enough room, it should be formatted as

```
if
  a < b
then
  x := 17
else
  x := 18
```

A more sophisticated way to typeset this construct, is by the box

```
HOV [HOV["if" I[H["a" "<" "b"]] "then"]
     I[H["x" ":=" "17"]]
     "else"
     I[H["x" ":=" "18"]] ]
```

This box should be set horizontally if there is room, but if there is no room, there will first be a test whether the if-then part fits on the rest of the line to get

```
if a < b then
  x := 17
else
  x := 18
```

If there is insufficient space, the result will be a totally vertical text as shown before.

An application of HV is typesetting a prefix function application like $f(a, b, c)$ by the box

H$hs = 0$["f(" HV[H$hs = 0$["a" ","] H$hs = 0$["b" ","] "c"] ")"]

This box can be set as one of the following texts:

```
f(a, b, c)        f(a, b,        f(a,
                    c)             b,
                                   c)
```

6 Text

We give formatting semantics to boxes by mapping them to text. A text is a list of strings where each one is interpreted as a line on a display.

6.1 Strings

A string is a list of arbitrary characters except for a newline enclosed in double quotes. A double quote can only occur in a string if it is preceded (escaped) by a backslash.

imports Layout$^{(A.1)}$ Numbers-Op$^{(A.4)}$

exports
  sorts STRING
  lexical syntax
    "\"" STR-CHR∗ "\"" → STRING
    "\\"" → STR-CHR
    ~["\n] → STR-CHR

We define the following operations on strings: ∗ (string concatenation), *width* (number of characters in a string) and *blank* (string with all characters of argument string replaced by blanks). Concatenation is right associative. An arbitrary string is denoted by a, ..., *e*

  context-free syntax
    STRING "∗" STRING → STRING {**right**}
    width(STRING) → INT
    blank(STRING) → STRING
    "(" STRING ")" → STRING {**bracket**}
  variables
    [*a-e*][*0-9'*]∗ → STRING
hiddens
  variables
    "*chr*"[*0-9'*]∗ → CHAR
    "*chr*" "∗"[*0-9'*]∗ → CHAR∗

$$(1) \quad \mathit{string}(\texttt{"""}\ \mathit{chr}_1^*\ \texttt{"""}) * \mathit{string}(\texttt{"""}\ \mathit{chr}_2^*\ \texttt{"""}) \ = \ \mathit{string}(\texttt{"""}\ \mathit{chr}_1^*\ \mathit{chr}_2^*\ \texttt{"""})$$

$$(2) \quad \mathit{width}(\texttt{""}) \ = \ 0$$
$$(3) \quad \mathit{width}(\mathit{string}(\texttt{"""}\ \mathit{chr}\ \mathit{chr}^*\ \texttt{"""})) \ = \ 1 + \mathit{width}(\mathit{string}(\texttt{"""}\ \mathit{chr}^*\ \texttt{"""}))$$

$$(4) \quad \mathit{blank}(\texttt{""}) \ = \ \texttt{""}$$
$$(5) \quad \mathit{blank}(\mathit{string}(\texttt{"""}\ \mathit{chr}\ \mathit{chr}^*\ \texttt{"""})) \ = \ \texttt{"␣"} * \mathit{blank}(\mathit{string}(\texttt{"""}\ \mathit{chr}^*\ \texttt{"""}))$$

6.2 Text

A text is simply a list of strings, where each string denotes a line on a display. We will use T as variable for texts.

imports Strings(6.1) Numbers-Op(A.4)
exports
  sorts TEXT
  context-free syntax
    STRING∗ → TEXT {**constructor**}
  variables
    [a-e]"∗"[*0-9'*]∗ → STRING∗
    [a-e]"+"[*0-9'*]∗ → STRING+
    [*T*][*0-9'*]∗ → TEXT

Horizontal and Vertical Composition The main text composing operations are // for vertical composition and || for horizontal composition. Additionally we define the operations ⊣ and | to prefix a text with a string. The first prefixes a string to every line of a text. The second prefixes a string a to the first line of a text and then prefixes a string of blanks that has the same width as a to all subsequent lines.

context-free syntax
TEXT "//" TEXT $\rightarrow$ TEXT {**right**}
TEXT "$\|$" TEXT $\rightarrow$ TEXT {**right**}
STRING "$\dashv$" TEXT $\rightarrow$ TEXT
STRING "$|$" TEXT $\rightarrow$ TEXT
"(" TEXT ")" $\rightarrow$ TEXT {**bracket**}
priorities
"$\dashv$" > "$|$" > "$\|$" > "//"

The operator // places the lines of a text vertically below another. This boils down to concatenating the lines of the texts.

(1) $a^* \;//\; b^* \;=\; a^*\; b^*$

The operator $\dashv$ prefixes each line of a text with a string.

(2) $a \dashv \quad =$

(3) $a \dashv b\; b^* \;=\; a * b \;//\; a \dashv b^*$

Note that if the text is empty the prefix is discarded.

The operator $|$ prefixes every line after the first line of a text with a blank string with the same width as its first string argument a and prefixes a to the first line of the text.

(4) $a \,| \quad = \; a$

(5) $a \,|\, b\; b^* \;=\; a * b \;//\; \mathit{blank}(a) \dashv b^*$

This is similar to the way this paragraph is typeset; the first line is prefixed with the string 'The operator | ' and the rest of the lines are prefixed (indented) by an empty string that has the same width as that string. This is called hanging indentation.

Finally, the operator $\|$ takes the last line of the first text and hangs the second text from it, composing the resulting text vertically with all but the last lines of the first text.

(6) $\|\; T \quad = \; T$

(7) $a^*\; a \,\|\, T \;=\; a^* \;//\; a \,|\, T$

Conditional Composition For each of the operators $\|$ and // there are left and right looking conditional compositions. For instance, the left looking horizontal composition $\underline{\;\|}$ only sets its second argument if its first (left) argument is not empty.

context-free syntax
TEXT "$\underline{\;//}$" TEXT $\rightarrow$ TEXT {**right**}
TEXT "$\underline{//\;}$" TEXT $\rightarrow$ TEXT {**right**}
TEXT "$\underline{\;\|}$" TEXT $\rightarrow$ TEXT {**right**}
TEXT "$\underline{\|\;}$" TEXT $\rightarrow$ TEXT {**right**}
priorities
{**right**: "$\|$", "$\underline{\;\|}$"}, {**right**: "//", "$\underline{\;//}$"}, "$\underline{\|\;}$" > "$\|$" > "$\underline{//\;}$" > "//"

(8) $\underline{\;\|}\; T \quad =$	(9) $\underline{\;//}\; T \quad =$
(10) $a^+ \underline{\;\|}\; T \;=\; a^+ \,\|\, T$	(11) $a^+ \underline{\;//}\; T \;=\; a^+ \;//\; T$
(12) $T \,\underline{\|\;} \quad =$	(13) $T \,\underline{//\;} \quad =$
(14) $T \,\underline{\|\;}\, a^+ \;=\; T \,\|\, a^+$	(15) $T \,\underline{//\;}\, a^+ \;=\; T \;//\; a^+$

These operators are very convenient for setting space between texts that should only appear if the trailing text is not empty.

Width Operations Finally, we define some operators for measuring texts and for creating texts consisting of white space; *twidth* gives the maximal width of a text, *hwidth* measures the horizontal width of a text, i.e., the width of its last line, *wd* creates a blank string that has the same width as the horizontal width of its argument text, *hskip* and *vskip* create blank texts of given width/height.

context-free syntax

$twidth(\text{TEXT}) \rightarrow \text{INT}$
$hwidth(\text{TEXT}) \rightarrow \text{INT}$
$hskip(\text{NUM}) \rightarrow \text{TEXT}$
$vskip(\text{NUM}) \rightarrow \text{TEXT}$
$wd(\text{TEXT}) \rightarrow \text{TEXT}$

hiddens

variables

$[r][0\text{-}9']* \rightarrow \text{REAL}$
$[n][0\text{-}9']* \rightarrow \text{INT}$
$[N][0\text{-}9']* \rightarrow \text{NUM}$

The text width (*twidth*) of a text is the maximum of the widths of its lines. If a text has text width n it has at least one line that extends n characters from the left margin. The horizontal width of a text (*hwidth*) is the width of its last line.

(16) $twidth() = 0$
(17) $twidth(\text{a a}^*) = max(width(\text{a}), twidth(\text{a}^*))$
(18) $hwidth() = 0$
(19) $hwidth(\text{a}^*\ \text{a}) = width(\text{a})$

$hskip(N)$ yields an empty string of width N and $vskip(N)$ builds a text of N empty lines.

(20) $hskip(N) = \quad$ **when** $0?(N) = \textbf{true}$
(21) $hskip(n) = \texttt{"␣"} \parallel hskip(n-1)$ **when** $n > 0 = \textbf{true}$
(22) $hskip(r) = hskip(\lfloor r \rfloor)$

(23) $vskip(N) = \quad$ **when** $0?(N) = \textbf{true}$
(24) $vskip(n) = \texttt{""} \mathbin{//} vskip(n-1)$ **when** $n > 0 = \textbf{true}$
(25) $vskip(r) = vskip(\lfloor r \rfloor)$

where $\lfloor r \rfloor$ maps a real number r to the smallest integer that is greater or equal than r.

The function *wd* creates an empty string with the same width as the last line of its argument text.

(26) $wd() =$
(27) $wd(\text{a}^*\ \text{a}) = blank(\text{a})$

7 Text Formatting

We will not give the formatting for all the box operators at once. We will first describe the text formatting using only the plain box operators H and V without space between boxes and then gradually improve this formatter.

7.1 Box to Text (A)

A first try at formatting boxes using only strings, H and V boxes. We consider a box as a function that, when applied to no arguments produces a text, i.e., box A is mapped to text by applying it as $A()$.

imports Box$^{(5)}$ Text$^{(6.2)}$
exports
  context-free syntax
    BOX "(" ")" → TEXT

A string yields itself. An empty box yields an empty text. A horizontal (vertical) composition is mapped to horizontal (vertical) text composition with $\|$ (//).

$$
\begin{array}{lll}
(1) & a() & = a \\
(2) & O[]() & = \\
(3) & \mathsf{H}[A\ B^*]() & = A() \parallel \mathsf{H}[B^*]() \\
(4) & \mathsf{V}[A\ B^*]() & = A() \mathbin{/\!/} \mathsf{V}[B^*]()
\end{array}
$$

Example The best way to understand this definition is by looking at an example. We follow how the formatting of our favourite box example proceeds.

$$
\begin{array}{rl}
\mathsf{H}[\texttt{"("}\ \mathsf{V}[\mathsf{H}[\texttt{"a"}\ \texttt{"+"}]\ \texttt{"b"}]\ \texttt{")"}]() & = \texttt{"("}() \parallel \mathsf{V}[\mathsf{H}[\texttt{"a"}\ \texttt{"+"}]\ \texttt{"b"}]() \parallel \texttt{")"}() \\
& = \texttt{"("} \parallel (\mathsf{H}[\texttt{"a"}\ \texttt{"+"}]() \mathbin{/\!/} \texttt{"b"}()) \parallel \texttt{")"} \\
& = \texttt{"("} \parallel (\texttt{"a"}() \parallel \texttt{"+"}() \mathbin{/\!/} \texttt{"b"}()) \parallel \texttt{")"} \\
& = \texttt{"("} \parallel (\texttt{"a"} \parallel \texttt{"+"} \mathbin{/\!/} \texttt{"b"}) \parallel \texttt{")"} \\
& = \texttt{"("} \parallel (\texttt{"a+"} \mathbin{/\!/} \texttt{"b"}) \parallel \texttt{")"} \\
& = \texttt{"("} \mid \texttt{"a+"}\ \texttt{"b"} \parallel \texttt{")"} \\
& = (\texttt{"(a+"} \mathbin{/\!/} \mathit{blank}(\texttt{"("}) \dashv \texttt{"b"}) \parallel \texttt{")"} \\
& = \texttt{"(a+"} \mathbin{/\!/} \texttt{"␣b"} \mid \texttt{")"} \\
& = \texttt{"(a+"} \\
& \quad\ \texttt{"␣b)"}
\end{array}
$$

7.2 Space Options

The box language allows the modification of the layout between boxes. Each operator may be accompanied by a number of options. Not every combination of operator and option makes sense. If for an operator no relevant option is specified a default value is used.

imports Box$^{(5)}$
exports
  context-free syntax
    SPACE-OPTIONS "(" OPERATOR ")" → NUM
hiddens
  variables
    $[hiv][0\text{-}9']*$ → NUM

For some list of space options o and operator O, the call $o(O)$ yields the amount of space specified by o in context O. To give the horizontal space (H) the function looks in the list of options from left to right, until it finds an option $hs = h$ in which case the number h is yielded [equations (2) and (3)]. If no hs option exists in the options the default value $\mathcal{H}$ is yielded [equation (1)].

$$
\begin{array}{lll}
(1) & (\mathsf{H}) & = \mathcal{H} \\
(2) & hs = h\ so^*(\mathsf{H}) & = h \\
(3) & S = h\ so^*(\mathsf{H}) & = so^*(\mathsf{H}) \quad \textbf{when} \quad S \neq hs
\end{array}
$$

For V and I the function works similarly.

$$
\begin{array}{lll}
(4) & (\mathsf{V}) & = \mathcal{V} \\
(5) & vs = v\ so^*(\mathsf{V}) & = v \\
(6) & S = v\ so^*(\mathsf{V}) & = so^*(\mathsf{V}) \quad \textbf{when} \quad S \neq vs \\
(7) & (\mathsf{I}) & = \mathcal{I} \\
(8) & is = i\ so^*(\mathsf{I}) & = i \\
(9) & S = h\ so^*(\mathsf{I}) & = so^*(\mathsf{I}) \quad \textbf{when} \quad S \neq is
\end{array}
$$

The values for default space are given by default equations. This implies that they can be overridden in an importing module.

$$
\begin{array}{lll}
(10) & \mathcal{H} = 1 & \textbf{otherwise} \\
(11) & \mathcal{I} = 2 & \textbf{otherwise} \\
(12) & \mathcal{V} = 0 & \textbf{otherwise}
\end{array}
$$

7.3 Box to Text (B)

We improve the previous attempt by separating boxes by space according to the space options of compositions.

imports Box$^{(5)}$ Text$^{(6.2)}$ Space-Options$^{(7.2)}$
exports
  context-free syntax
    BOX "(" ")" → TEXT

Formatting is defined as before, except that a *hskip* is used to separate two boxes in a horizontal compositions and a *vskip* is used to separate boxes in a vertical composition.

$$
\begin{array}{lll}
(1) & \mathrm{a}() & = \mathrm{a} \\
(2) & O_o[]() & = \\
(3) & \mathsf{H}_o[A\ B^*]() & = A() \parallel \mathit{hskip}(o(\mathsf{H})) \underline{\parallel} \mathsf{H}_o[B^*]() \\
(4) & \mathsf{V}_o[A\ B^*]() & = A() \mathbin{/\!/} \mathit{vskip}(o(\mathsf{V})) \underline{\mathbin{/\!/}} \mathsf{V}_o[B^*]()
\end{array}
$$

By using conditional compositions to attach the separating space between boxes we do not have to worry that box A is the last box in the composition.

[Note that we set the options (o) of boxes as a subscript of the operator in equations.]

8 Indentation

Extending the text formatter with the box operator I influences the formatting of both the H and V operator. The reason for this is that the I operator behaves differently when applied in a H or V operator. Given this nearly complete text formatter we are able to define the first set of box transformations in Section 8.2

8.1 Box to Text (C)

The formatting function now takes an operator as argument, because the formatting of the I operator is context sensitive. The term $A(O)$ should be read as: format box

A in an O context. We use box operators to indicate the context. $A(\mathsf{H})$ means that box A occurs inside a H box.

imports Box$^{(5)}$ Text$^{(6.2)}$ Space-Options$^{(7.2)}$

exports

context-free syntax

BOX "(" OPERATOR ")" $\rightarrow$ TEXT

Strings, H and V boxes are set as before, except that the context operator is passed as an argument. H and V are not context sensitive.

$$
\begin{array}{lll}
(1) & a(O) & = a \\
(2) & O_O[\,](O') & = \\
(3) & \mathsf{H}_O[A\ B^*](O) & = A(\mathsf{H}) \parallel \mathit{hskip}(o(\mathsf{H})) \,\underline{\parallel}\, \mathsf{H}_O[B^*](\mathsf{H}) \\
(4) & \mathsf{V}_O[A\ B^*](O) & = A(\mathsf{V}) \,/\!/\, \mathit{vskip}(o(\mathsf{V})) \,\underline{/\!/}\, \mathsf{V}_O[B^*](O)
\end{array}
$$

In a vertical context an I box works by *prefixing* indentation space before each of its containing boxes and composing the resulting texts horizontally. In a H context an I box is formatted as a H box.

$$
\begin{array}{lll}
(5) & \mathsf{I}_O[A\ B^*](\mathsf{V}) & = \mathit{hskip}(o(\mathsf{I})) \,\underline{\parallel}\, A(\mathsf{V}) \parallel \mathsf{I}[B^*](\mathsf{V}) \\
(6) & \mathsf{I}_O[A^*](\mathsf{H}) & = \mathsf{H}_O[A^*](\mathsf{H})
\end{array}
$$

Another new operator is WD. It yields a blank text that has the same width as its arguments.

$$
(7) \quad \mathsf{WD}_O[A\ B^*](O) = \mathit{wd}(A(O) \parallel \mathit{hskip}(o(\mathsf{H})) \,\underline{\parallel}\, \mathsf{WD}_O[B^*](O))
$$

8.2 Box Laws (C)

Some box transformations are needed to reduce the nesting level in box expressions and to conform to the semantics of horizontal and vertical composition of TEX. The laws preserve the text formatting semantics of boxes, i.e., a box law $A = B$ is valid if and only if $A(O) = B(O)$ for any O.

imports Box$^{(5)}$ Space-Options$^{(7.2)}$

hiddens

variables

$[hiv][0\text{-}9']* \rightarrow$ NUM

Redundant Options We saw that space options are not relevant in all situations. Therefore, we can throw away those options that are irrelevant. For instance, in a H box only horizontal space (indicated by the *hs* option) is relevant, so we can remove options for indentation and vertical space [equations (1) and (2)].

$$
\begin{array}{lll}
(1) & \mathsf{H} so_1^*\ is = i\ so_2^*[A^*] & = \mathsf{H} so_1^*\ so_2^*[A^*] \\
(2) & \mathsf{H} so_1^*\ vs = i\ so_2^*[A^*] & = \mathsf{H} so_1^*\ so_2^*[A^*] \\
(3) & \mathsf{V} so_1^*\ hs = i\ so_2^*[A^*] & = \mathsf{V} so_1^*\ so_2^*[A^*] \\
(4) & \mathsf{V} so_1^*\ is = i\ so_2^*[A^*] & = \mathsf{V} so_1^*\ so_2^*[A^*] \\
(5) & \mathsf{I} so_1^*\ vs = i\ so_2^*[A^*] & = \mathsf{I} so_1^*\ so_2^*[A^*]
\end{array}
$$

Furthermore, we saw that only the first option for some category is relevant for the function that looks up the space for some context in a list of options. Therefore, we can remove any extra options for the same category S.

$$
(6) \quad so_1^*\ S = h_1\ so_2^*\ S = h_2\ so_3^* = so_1^*\ S = h_1\ so_2^*\ so_3^*
$$

I in H is H According to equation (8.1.6) an I box in a H context is treated in the same way as a H box. Therefore, we can replace an I box in a H box by a H box.

(7) $\mathsf{H}o_1[A^*\ \mathsf{I}o_2[B^*]\ C^*]\ =\ \mathsf{H}o_1[A^*\ \mathsf{H}o_2[B^*]\ C^*]$

Note that the H that replaces the I inherits its space options. That is the reason that we can not remove a *hs* option of an I box, but only its *vs* option [equation (5)].

Empty boxes According to equation (8.1.2) an empty box of any kind yields no text. Therefore, we can remove it from any list of boxes.

(8) $A^*\ Oo[]\ B^*\ =\ A^*\ B^*$

Strings in H Strings can be concatenated in a H box when its horizontal space is zero.

(9) $\mathsf{H}o[A^*\ a\ b\ C^*]\ =\ \mathsf{H}o[A^*\ a*b\ C^*]$ **when** $0?(o(\mathsf{H})) = \textbf{true}$

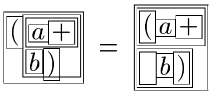
V lifting The following transformations are used to move the V operator out of the box expression, The result of these transformations is a box expression consisting of a list of H boxes which do not contain V operators any more.

(10)
$$\frac{B_1 \neq\ ,\ \ B_2 \neq}{\begin{array}{l}\mathsf{H}o_1[A^*\ \mathsf{V}o_2[B_1\ B^*\ B_2]\ C^*] = \\ \mathsf{V}o_2[\mathsf{H}o_1[A^*\ \mathsf{V}o_2[B_1]]\ \mathsf{H}o_1[\mathsf{WD}o_1[A^*]\ \mathsf{V}o_2[B^*\ \mathsf{H}o_1[\mathsf{V}o_2[B_2]\ C^*]]]]\end{array}}$$

The following reduction sequence shows how this equation—in combination with a couple of equations that we discuss below—works on our $(a + b)$ example.

```
H["(" V[H["a" "+"] H["b"]] ")"]
= V[H["(" H["a" "+"]] H[WD["("] V[H[V[H["b"]] ")"]]]]
= V[H["(" "a" "+"] H[WD["("] V[H[V[H["b"]] ")"]]]]
= V[H["(" "a" "+"] H[WD["("] V[H[H[ "b"] ")"]]]]
= V[H["(" "a" "+"] H[WD["("] V[H["b" ")"]]]]
= V[H["(" "a" "+"] H[WD["("] "b" ")"]]
```

If we draw the first and last box in this sequence as a picture we see how flattening of boxes takes place.



Note that the bracket in the left picture is actually in the outer horizontal box. We see that the maximal nesting is reduced from 4 in the original box to 3 in resulting box.

After applying this equation there are still V boxes inside H boxes, but they are always singleton V boxes. They are left around the boxes B_1 and B_2 to protect I boxes in them from the surrounding H box. Below we will show how these singleton V boxes can be lifted as well.

Lifting V boxes from I boxes is tricky, since we must ensure the I does not occur in a H box. We ensure this by the condition $O \neq \mathsf{H}$, that restricts the context to non-H boxes.

(11)
$$\frac{O \neq \mathsf{H},\ \ B \neq}{Oo_0[A^*\ \mathsf{I}o_1[\mathsf{V}o_2[B\ B^*]]\ C^*] = Oo_0[A^*\ \mathsf{V}o_2[\mathsf{I}o_1[B]\ \mathsf{I}o_1[\mathsf{V}o_2[B^*]]]\ C^*]}$$

Only the last box of a V box inside a WD box is relevant, because the function *wd* yields a blank line with the same width as the last line of its argument text [see equations (6.2.27) and (8.1.7)].

(12) $\mathsf{WD}o_1[\mathsf{V}o_2[A^*\ B]]\ =\ \mathsf{WD}o_1[B]$

Nested boxes of the same kind Nesting H, V and WD boxes makes only sense if different space options are attached to the outer and inner box. Otherwise, if the options are the same, the inner list can be lifted into the outer list.

(13) $\mathsf{V}o[A^*\ \mathsf{V}o[B^*]\ C^*]\ =\ \mathsf{V}o[A^*\ B^*\ C^*]$
(14) $\mathsf{H}o[A^*\ \mathsf{H}o[B^*]\ C^*]\ =\ \mathsf{H}o[A^*\ B^*\ C^*]$
(15) $\mathsf{WD}o[A^*\ \mathsf{WD}o[B^*]\ C^*]\ =\ \mathsf{WD}o[A^*\ B^*\ C^*]$

Furthermore, for H boxes we can move the trailing part of the outer horizontal list into the inner horizontal list.

(16) $\mathsf{H}o_1[A^*\ \mathsf{H}o_2[B^+\ B]\ C^+]\ =\ \mathsf{H}o_1[A^*\ \mathsf{H}o_2[B^+\ \mathsf{H}o_1[B\ C^+]]]$

An I box inside an I box is *not* meaningless, but indicates 'double' indentation. For the case of a singleton indented I box we can derive the following equation independent of the context.

(17) $\mathsf{I}o_1[\mathsf{I}o_2[A]]\ =\ \mathsf{I}\,is = o_1(\mathsf{I}) \oplus o_2(\mathsf{I})[A]$

[where $\oplus$ is addition and $\ominus$ is subtraction on NUM.]

Singleton Boxes It would seem that for any operator O, $O_O[A] = A$. However, in the first place this does not hold for $O = \mathsf{I}$ and in the second place, since A itself can be an I box, this equation might change the context of that I. Therefore, we have to consider carefully for each O and for each context O' whether $O'_{O'}[O_O[A]] = O'_{O'}[A]$. We must ensure that the context of a box is only changed if that is safe; in particular we have to ensure that an I box in a V box is protected from a possible H context of the V box.

A singleton string in a H or V box is just that string.

(18) $\mathsf{H}o[a]\ =\ a$
(19) $\mathsf{V}o[a]\ =\ a$

Singleton H boxes:

(20) $\mathsf{H}o_1[A^*\ \mathsf{H}o_2[B]\ C^*]\ =\ \mathsf{H}o_1[A^*\ B\ C^*]$
(21) $\mathsf{H}o_1[\mathsf{H}o_2[A^*]]\ =\ \mathsf{H}o_2[A^*]$

Singleton V boxes:

(22) $\mathsf{V}o_1[\mathsf{V}o_2[A^*]]\ =\ \mathsf{V}o_2[A^*]$
(23) $\mathsf{V}o_1[\mathsf{H}o_2[A^*]]\ =\ \mathsf{H}o_2[A^*]$

(24) $\mathsf{V}o_1[A^*\ \mathsf{V}o_2[B]\ C^*]\ =\ \mathsf{V}o_1[A^*\ B\ C^*]$
(25) $\mathsf{H}o_1[A^*\ \mathsf{V}o_2[\mathsf{H}o_3[B^*]]\ C^*]\ =\ \mathsf{H}o_1[A^*\ \mathsf{H}o_3[B^*]\ C^*]$
(26) $\mathsf{V}o_1[\mathsf{H}o_2[\mathsf{V}o_3[A]]]\ =\ \mathsf{V}o_1[A]$

H in WD

(27) $\mathsf{WD}o[\mathsf{H}o[A^*]]\ =\ \mathsf{WD}o[A^*]$

Since Is are protected in Is we can flatten a singleton H or V box in a I box.

(28) $\mathsf{I}o_1[\mathsf{H}o_2[A]] \;=\; \mathsf{I}o_1[A]$

(29) $\mathsf{I}o_1[\mathsf{V}o_2[A]] \;=\; \mathsf{I}o_1[A]$

(30) $\mathsf{H}o_1[\mathsf{V}o_2[\mathsf{I}o_3[B^*\ B]]\ C^+] \;=\; \mathsf{V}o_2[\mathsf{I}o_3[B^*\ \mathsf{H}o_1[B\ C^+]]]$

(31) $\mathsf{H}o_1[A^+\ \mathsf{V}o_2[\mathsf{I}o_3[B_1\ B^*\ B_2]]\ C^*] \;=\; \mathsf{V}o_2[\mathsf{I}is=0[\mathsf{H}o_1[A^+\ \texttt{""}]\ \mathsf{I}o_3[B^*\ \mathsf{H}o_1[B_2\ C^*]]]]$

9 Linebreaking

In this section we give the semantics of the two remaining box operators HOV and HV. We first give a non-deterministic specification of these operators in terms of the H, V and I operators.

A HOV can either be formatted as if it were a H box or as if it were a V box.

(1) $\mathsf{HOV}_o[A^*] \;=\; \mathsf{H}_o[A^*]$

(2) $\mathsf{HOV}_o[A^*] \;=\; \mathsf{V}_o[A^*]$

The formatter prefers the first possibility and only uses the second if there is not enough room left on the current line to put all boxes horizontally. We present a deterministic definition of formatting HOV in section 9.1.

A HV can be set either as a H box or as a V box. In the latter case the list of arguments is broken into three arbitrary, but non-empty, sublists. The first and the last are new, but smaller HVs and the middle one is set as an indented H box.

(3) $\mathsf{HV}_o[A^*] \;=\; \mathsf{H}_o[A^*]$

(4) $\mathsf{HV}_o[A^+\ B^+\ C^+] \;=\; \mathsf{V}_o[\mathsf{HV}_o[A^+]\ \mathsf{I}_o[\mathsf{H}_o[B^+]]\ \mathsf{HV}_o[C^+]]$

We present a deterministic definition of formatting HV in section 9.3.

9.1 Box to Text (D = C + HOV)

The deterministic formatting of HOV boxes is performed in a rather brute force manner. First the HOV box is formatted as a H box, if the resulting text does not fit on the remaining space *or* consists of more than one line the box is reformatted as a V box. The formatting function is extended with an extra argument which denotes the remaining space on a line.

imports Box$^{(5)}$ Text$^{(6.2)}$ Space-Options$^{(7.2)}$
exports
  context-free syntax
    BOX "(" OPERATOR "," NUM ")" → TEXT
    "*if*HOV"(TEXT, BOX, OPERATOR, NUM) → TEXT
hiddens
  variables
    [*hiv*][*0-9'*]* → NUM
    [*ms*][*0-9'*]* → NUM

Strings and V boxes are set as before, except that the context operator and the remaining space on the current line are passed to inner boxes.

(1) $a(O, m) \;=\; a$

(2) $Oo[](O', m) \;=$

(3) $\mathsf{V}o[A\ B^*](O, m) \;=\; A(\mathsf{V}, m)\ /\!/\ vskip(o(\mathsf{V}))\ \underline{/\!/}\ \mathsf{V}o[B^*](O, m)$

In formatting H, I and WD boxes we have to keep track of how much space remains after setting a box.

$$(4)\quad \frac{A(\mathsf{H}, m) = T,\ \ hwidth(T) = s,\ \ h = o(\mathsf{H})}{\mathsf{H}o[A\ B^*](O, m) = T \parallel hskip(h) \,\|\!\!\_\, \mathsf{H}o[B^*](\mathsf{H}, m \ominus s \ominus h)}$$

$$(5)\quad \mathsf{I}o[A^*](\mathsf{H}, m) = \mathsf{H}o[A^*](\mathsf{H}, m)$$

$$(6)\quad \frac{i = o(\mathsf{I}),\ \ m' = m \ominus i,\ \ A(\mathsf{V}, m') = T,\ \ hwidth(T) = s}{\mathsf{I}o[A\ B^*](\mathsf{V}, m) = hskip(i) \,\|\!\!\_\, T \parallel \mathsf{I}[B^*](\mathsf{V}, m' \ominus s)}$$

$$(7)\quad \frac{A(O, m) = T,\ \ hwidth(T) = s,\ \ h = o(\mathsf{H})}{\mathsf{WD}o[A\ B^*](O, m) = wd(T \parallel hskip(h) \,\|\!\!\_\, \mathsf{WD}o[B^*](O, m \ominus s \ominus h))}$$

Formatting of HOVs is implemented by the auxilary function *if*HOV which takes the formatted text and decides whether this text can be used or whether the box should be reformatted vertically.

$$(8)\quad \mathsf{HOV}o[A^*](O, m) \ = \ \mathit{if}\mathsf{HOV}(\mathsf{H}o[A^*](O, m), \mathsf{V}o[A^*]\ , O, m)$$

$$\begin{array}{llll}
(9) & \mathit{if}\mathsf{HOV}(, B, O, m) & = & \\
(10) & \mathit{if}\mathsf{HOV}(\mathrm{a}, B, O, m) & = \mathrm{a} & \textbf{when} \quad width(\mathrm{a}) \leq_N m = \textbf{true} \\
(11) & \mathit{if}\mathsf{HOV}(\mathrm{a}, B, O, m) & = B(O, m) & \textbf{when} \quad width(\mathrm{a}) >_N m = \textbf{true} \\
(12) & \mathit{if}\mathsf{HOV}(\mathrm{a}\ \mathrm{b}^+, B, O, m) & = B(O, m) &
\end{array}$$

9.2 Box Laws (D = C + HOV)

The formatting of the HOV operator has also impact on the box transformation rules. The text formatting semantics of the HOV operator prescribes that if the text produced by an HOV box consists of more than one line, the HOV box should be formatted in a vertical way.

imports Box-Laws-C$^{(8.2)}$

The trailing part of a horizontal composition after a HOV will always be placed horizontally after the last element of the HOV.

$$\begin{array}{ll}
(1) & \mathsf{H}so^*[A^*\ \mathsf{HOV}o_2[B^+\ B]\ C^+] = \\
& \quad \mathsf{H}so^*[A^*\ \mathsf{HOV}o_2[B^+\ \mathsf{I}is = 0\ hs = 0[B\ \mathsf{H}so^*[\texttt{""}\ C^+]]]]
\end{array}$$

The strange construction in the right hand side is needed to protect a possible I in box B, but keeping the possibility open for eliminating the I if the HOV is set horizontally.

A singleton V box containg a HOV is that HOV.

$$(2)\quad \mathsf{V}o_2[\mathsf{HOV}o_3[B^*]] \ = \ \mathsf{HOV}o_3[B^*]$$

According to equation (9.1.12) a HOV box containing a V box with more than one box is always formatted vertically.

$$\begin{array}{llll}
(3) & \mathsf{HOV}o_1[A^*\ \mathsf{V}o_2[B\ B^+]\ C^*] & = \ \mathsf{V}o_1[A^*\ \mathsf{V}o_2[B\ B^+]\ C^*] & \textbf{when} \quad B\ B^+ \neq \\
(4) & \mathsf{HOV}o_1[A^*\ \mathsf{V}o_2[B]\ C^*] & = \ \mathsf{HOV}o_1[A^*\ B\ C^*] &
\end{array}$$

$$(5)\quad \frac{B_1 \neq\ ,\ \ B_2 \neq}{\mathsf{HOV}o_0[A^*\ \mathsf{I}o_1[\mathsf{V}o_2[B_1\ B^*\ B_2]]\ C^*] = \mathsf{V}o_0[A^*\ \mathsf{V}o_2[\mathsf{I}o_1[B_1]\ \mathsf{I}o_1[\mathsf{V}o_2[B^*\ B_2]]]\ C^*]}$$

9.3 Box to Text (E = D + HV)

We extend the formatter of Section 9.1 with rules for formatting HV boxes. These are the most complicated ones, because for each argument box it must be checked whether it fits on the remainder of the current line or should be placed at the start of a new line.

imports Box-to-Text-D$^{(9.1)}$
hiddens
variables
$[hiv][0\text{-}9']* \rightarrow$ NUM
$[nms][0\text{-}9']* \rightarrow$ NUM

The first box is formatted without paying attention to the amount of space this box uses.

(1) $\mathsf{HV}o[A](O, m) = A(\mathsf{H}, m)$

(2) $$\frac{A(\mathsf{H}, m) = T}{\mathsf{HV}o[A\ B^+](O, m) = \mathsf{HV}\ o[T, m \ominus \mathit{hwidth}(T) \mid B^+](m)}$$

The formatting of the remaining boxes depends on the amount of space left on the current line. The boxes are formatted one by one but as soon as a box runs over the right margin the produced text will be shifted to the next line. Unless the text uses more space than the orginal space or consists of more than one line. These cases ask for a reformatting of the complete box.

We use the following auxiliary function

context-free syntax
"HV" SPACE-OPTIONS "[" TEXT "," NUM "|" BOX-LIST "]"
"(" NUM ")" $\rightarrow$ TEXT

A term $\mathsf{HV}o[T, s|B^*](O, m)$ declares that on this line we have already set text T, that has width s. We still have to set boxes B^*. From the start of a new line we have m space until the right margin. If there are no more boxes left we are ready and yield text T (3). If there is at least one box A left in the list we set that box in a horizontal context with s space to go.

(3) $\mathsf{HV}\ o[T, s \mid\ ](m) = T$

(4) $\mathsf{HV}\ o[T, s \mid A\ B^*](m) = \mathsf{HV}\ o[T, s \rhd A(\mathsf{H}, s), A \lhd B^*](m)$

context-free syntax
"HV" SPACE-OPTIONS "[" TEXT "," NUM
"$\rhd$" TEXT "," BOX "$\lhd$" BOX-LIST "]" "(" NUM ")" $\rightarrow$ TEXT

Now we have to deal with the situation $\mathsf{HV}o[T, s \rhd T', A \lhd B^*](m)$, i.e., we have mapped box A to text and that resulted in text T'. Note that we remember box A. Now we have to decide where we are going to put T'.

Box A is mapped to the single line of text a and a fits on the current line because its width n is smaller than the amount s of space left.

(5) $$\frac{o(\mathsf{H}) \oplus \mathit{width}(a) = n,\ \ n \leq_N s = \mathbf{true}}{\mathsf{HV}\ o[T, s \rhd a, A \lhd B^*](m) = \mathsf{HV}\ o[T \,\|\, \mathit{hskip}(o(\mathsf{H})) \parallel a, s \ominus n \mid B^*](m)}$$

Text a does not fit on the current line but it fits on the next line.

(6) $$\frac{\mathit{width}(a) = n,\ \ o(\mathsf{H}) \oplus n >_N s = \mathbf{true},\ \ o(\mathsf{I}) \oplus n \leq_N m = \mathbf{true}}{\mathsf{HV}\ o[T, s \rhd a, A \lhd B^*](m) = T \,/\!/\, \mathit{vskip}(o(\mathsf{V})) \,/\!/\, \mathsf{HV}\ o[\mathit{hskip}(o(\mathsf{I})) \parallel a, m \ominus n \ominus o(\mathsf{I}) \mid B^*](m)}$$

Text a does not fit on the current line and neither on the next.

$$(7)\quad \frac{\begin{array}{c} \mathit{width}(a) = n,\ o(\mathsf{H}) \oplus n >_N s = \mathbf{true},\ o(\mathsf{I}) \oplus n >_N m = \mathbf{true}, \\ A(\mathsf{H},\, m \ominus o(\mathsf{I})) = T',\ m \ominus o(\mathsf{I}) \ominus \mathit{hwidth}(T') = s' \end{array}}{\begin{array}{l} \mathsf{HV}\ o[T,\, s \rhd a,\, A \lhd B^*](m) = T \mathbin{/\!/} \mathit{vskip}(o(\mathsf{V})) \\ \qquad\qquad\qquad\qquad \mathbin{\underline{/\!/}} \mathsf{HV}\ o[\mathit{hskip}(o(\mathsf{I})) \parallel T',\, s' \mid B^*](m) \end{array}}$$

The text resulting from A consists of multiple lines. In this case the box is reformatted after starting on a new line.

$$(8)\quad \frac{T' = \mathit{hskip}(o(\mathsf{I})) \parallel A(\mathsf{H},\, m \ominus o(\mathsf{I}))}{\begin{array}{l} \mathsf{HV}\ o[T,\, s \rhd a^+\ a,\, A \lhd B^*](m) = T \mathbin{/\!/} \mathit{vskip}(o(\mathsf{V})) \\ \qquad\qquad\qquad\qquad \mathbin{\underline{/\!/}} \mathsf{HV}\ o[T',\, \mathit{hwidth}(T') \mid B^*](m) \end{array}}$$

Reformatting is necessary because A was formatted for the remaining space s. It might be the case that, if it is formatted with the (larger) linewidth m of the HV, the resulting text is different.

9.4 Box Laws (E = D + HV)

The introduction of the HV operator results also in an extension of the set of box transformation rules.

imports Box-Laws-D$^{(9.2)}$

I in HV

(1) $\mathsf{HV}o_1[A^*\ \mathsf{I}o_2[B^*]\ C^*] \ = \ \mathsf{HV}o_1[A^*\ \mathsf{H}o_2[B^*]\ C^*]$

HV in H

(2) $\mathsf{H}o_1[A^*\ \mathsf{HV}o_2[B^+\ B]\ C^+] \ = \ \mathsf{H}o_1[A^*\ \mathsf{HV}o_2[B^+\ \mathsf{H}o_1[B\ C^+]]]$ **when** $B \neq$

Singleton boxes

(3) $\mathsf{HV}o[A] \ = \ \mathsf{H}o[A]$

(4) $\mathsf{H}o_1[\mathsf{HV}o_2[A^*]] \ = \ \mathsf{HV}o_2[A^*]$

(5) $\mathsf{V}o_2[\mathsf{HV}o_3[A^*\ A]] \ = \ \mathsf{HV}o_3[A^*\ A]$

V lifting in case of HV is almost identical to the V lifting of H. The result is a V box with only HV argument boxes which do not contain V operators.

(6) $\mathsf{HV}o_1[A^*\ \mathsf{V}o_2[B^+\ B]\ C^*] \ = \ \mathsf{V}o_2[\mathsf{HV}o_1[A^*]\ B^+\ \mathsf{HV}o_1[B\ C^*]]$

(7) $\mathsf{V}o_1[\mathsf{HV}o_2[A^*]] \ = \ \mathsf{HV}o_2[A^*]$

10 TEX Formatting

After defining the syntax of TEX we can translate boxes to TEX.

10.1 TEX

Since we only want to generate code, we use a simple syntax for TEX code. The TEX code we generate consists of a list of strings, numbers, command sequences and braces { and }. To be able to translate a nested box, a token can also be a list of tokens.

imports Strings$^{(6.1)}$ Identifiers$^{(A.5)}$
exports
  sorts CS TEX
  lexical syntax
    "\\"ID → CS
  context-free syntax
    "(" TEX* ")" → TEX
    STRING → TEX
    NUM → TEX
    CS → TEX
    "{" → TEX
    "}" → TEX
  variables
    [*t*][*0-9'*]* → TEX
    [*t*]"*"[*0-9'*]* → TEX*

Nested lists are flattened by

(1) $(t_1^* \; (t_2^*) \; t_3^*) \;=\; (t_1^* \; t_2^* \; t_3^*)$

This implies that any nested list we generate is reduced to a flat list.

10.2 Box to TEX

The function *tex* translates a box to TEX code. It makes use of the auxiliary functions *tex1*, *tex2* and *tex3* that map lists of boxes to TEX. The translation is based on the following assumptions:

- V boxes occur only at the toplevel
- H boxes contain only strings and possibly one HOV or HV box as last box.

That is, the translation works for any box, but the TEX implementation is only guaranteed to work under these assumptions.
imports Box-Laws-E$^{(9.4)}$ TeX$^{(10.1)}$
exports
  context-free syntax
    tex(BOX) → TEX
    texs1(BOX-LIST) → TEX
    texs2(TEX, BOX-LIST) → TEX
    texs3(TEX, BOX-LIST) → TEX
hiddens
  variables
    [*hiv*] → NUM
    "*chr**" → CHAR*

Lists of Boxes We define three functions to translate a list of boxes. The first one simply concatenates the translations of the boxes in a list. The second one separates code for boxes by a piece of code. The third one prefixes each box with a piece of code t and puts braces around the code for the box. Here t is intended to be a macro that takes a piece of code as argument.

(1) $\mathit{texs1}() \;=\; ()$
(2) $\mathit{texs1}(A \; A^*) \;=\; (\mathit{tex}(A) \; \mathit{texs1}(A^*))$

$$
\begin{array}{lll}
(3) & \mathit{texs2}(t, \,) & = () \\
(4) & \mathit{texs2}(t,\, A) & = \mathit{tex}(A) \\
(5) & \mathit{texs2}(t,\, A\ A^{+}) & = (\mathit{tex}(A)\ t\ \mathit{texs2}(t,\, A^{+}))
\end{array}
$$

$$
\begin{array}{lll}
(6) & \mathit{texs3}(t, \,) & = () \\
(7) & \mathit{texs3}(t,\, A\ A^{*}) & = (t\ \{\ \mathit{tex}(A)\ \}\ \mathit{texs3}(t,\, A^{*}))
\end{array}
$$

Strings Strings are translated by removing their doublequotes.

$$(8) \quad \mathit{tex}(\mathit{string}(\texttt{"\char`\\""}\ \mathit{chr}^{*}\ \texttt{"\char`\\""})) \ = \ (\texttt{\char`\\S}\ \{\ \mathit{string}(\mathit{chr}^{*})\ \})$$

[This is not very decent because it implies that the resulting code does not conform to the syntax of TEX as we defined it above. Since we do not plan to analyse the generated code this solution is better than trying to remove the quotes in TEX.]

Boxes We translate boxes to code. Each box operator is mapped to a corresponding macro. For instance, H is mapped to `\H` and V to `\V`. The pieces of code that correspond to the subboxes of a box are separated by skips that take care of spacing. The skips are parameterized by the relevant space option values.

$$
\begin{array}{lll}
(9) & \mathit{tex}(\mathsf{H}o[A^{*}]) & = (\texttt{\char`\\H}\{\mathit{texs2}((\texttt{\char`\\HSKIP}\{o(\mathsf{H})\}),A^{*})\}) \\
(10) & \mathit{tex}(\mathsf{V}o[A^{*}]) & = (\texttt{\char`\\V}\{\mathit{texs2}((\texttt{\char`\\VSKIP}\{o(\mathsf{V})\}),A^{*})\}) \\
(11) & \mathit{tex}(\mathsf{I}o[A^{*}]) & = (\texttt{\char`\\I}\{\texttt{\char`\\ISKIP}\{o(\mathsf{I})\}\{0\}\mathit{texs2}((\texttt{\char`\\ISKIP}\{o(\mathsf{I})\}\{o(\mathsf{H})\}),A^{*})\}) \\
(12) & \mathit{tex}(\mathsf{WD}o[A^{*}]) & = (\texttt{\char`\\WD}\{\mathit{texs2}((\texttt{\char`\\HSKIP}\{o(\mathsf{H})\}),A^{*})\}) \\
(13) & \mathit{tex}(\mathsf{HOV}o[A^{*}]) & = (\texttt{\char`\\HOV}\{\mathit{texs2}((\texttt{\char`\\HOVSKIP}\{o(\mathsf{H})\}\{o(\mathsf{V})\}),A^{*})\}) \\
(14) & \mathit{tex}(\mathsf{HV}o[AA^{*}]) & = (\texttt{\char`\\HV}\{\mathit{tex}(A)\mathit{texs3}((\texttt{\char`\\HVSKIP}\{o(\mathsf{H})\}\{o(\mathsf{I})\}\{o(\mathsf{V})\}),A^{*})\})
\end{array}
$$

Note that `\HVSKIP`s take the next box as argument. As we will see below in the macro definitions it handles the placing of its argument by comparing its width with the amount of space left on the current line.

We can reduce the size of the generated code by removing irrelevant skips. A zero `\HSKIP`, `\ISKIP`, `\VSKIP` or `\HOVSKIP` has no effect.

$$
\begin{array}{llll}
(15) & (t_1^{*}\texttt{\char`\\HSKIP}\{h\}t_2^{*}) & = (t_1^{*}t_2^{*}) & \textbf{when} \quad \mathit{0?}(h) = \textbf{true} \\
(16) & (t_1^{*}\texttt{\char`\\VSKIP}\{v\}t_2^{*}) & = (t_1^{*}t_2^{*}) & \textbf{when} \quad \mathit{0?}(v) = \textbf{true} \\
(17) & (t_1^{*}\texttt{\char`\\ISKIP}\{i\}\{h\}t_2^{*}) & = (t_1^{*}t_2^{*}) & \textbf{when} \quad \mathit{0?}(i)\textbf{and}\mathit{0?}(h) = \textbf{true} \\
(18) & (t_1^{*}\texttt{\char`\\HOVSKIP}\{h\}\{v\}t_2^{*}) & = (t_1^{*}t_2^{*}) & \textbf{when} \quad \mathit{0?}(h)\textbf{and}\mathit{0?}(v) = \textbf{true}
\end{array}
$$

Example To see the effect of the translation we translate our $(a+b)$ box to TEX by evaluating the term

```
tex( H["(" V[H["a" "+"] "b"] ")"] )
```

By application of the box transformations this first reduces to

```
tex( V[H["(" "a" "+"] H[WD["("] "b" ")"]] )
```

and then the translation yields

```
\V{\H{\S{(}\HSKIP{1}\S{a}\HSKIP{1}\S{+}}\H%
{\WD{\S{(}}\HSKIP{1}\S{b}\HSKIP{1}\S{)}}}%
```

which results in $\begin{matrix}(\ \mathrm{a}\ + \\ \mathrm{b}\ )\end{matrix}$. A variant of the previous box does not put space after (and before):

```
tex( H hs = 0 ["(" V[H["a" "+"] "b"] ")"] )
```

results in the code

```
\V{\H{\S{(}\H{\S{a}\HSKIP{1}\S{+}}}\H{\WD%
{\S{(}}\S{b)}}}%
```

Note that it does not contain code of the form \HSKIP{0}. The result of this code is: $\begin{matrix}(\mathrm{a}\ + \\ \mathrm{b})\end{matrix}$.

10.3 TEX macros

We give macro definitions for the macros used in code generated by the function *tex*.

Preliminaries Command sequence \RM represents the amount of space until right margin is reached. Its default value is `\hsize`.

```
\edef\RM{\hsize}
```

\RMdecr decreases the value of \RM by the amount given in its argument.

```
\def\RMdecr#1{\dimen1=#1\dimen0=\RM\advance\dimen0 by-\dimen1\edef\RM{\dimen0}}
```

We introduce two conditionals that indicate whether or not we are inside a \V or \HOV.

```
\newif\ifVMODE\newif\ifHVMODE\VMODEfalse\HVMODEfalse
```

Boxes in external list We pack the definitions of the box macros in the body of a macro, to ensure that they are not redefined. Furthermore, we have two sets of definitions, internal and external. The external definitions apply to boxes at the outermost level. The boxes contained in these are internal boxes.

An external box is either directly at the top level or is contained in a \V box at the top level. Therefore, each external box starts a new line: see the `\par\noindent` at the beginning of the bodies.

```
\def\EXTERN@L{%
```

\S, \H and \I are \hbox's the contents of which are processed with internal definitions. The boxes in a \H are not in vertical mode, \I does not change the modes.

```
\def\S##1{\par\noindent\hbox{\INTERN@L##1}}%
\def\H##1{\par\noindent\hbox{\INTERN@L\HVMODEfalse\VMODEfalse ##1}}%
\def\I##1{\par\noindent\hbox{\INTERN@L##1}}%
\def\WD##1{\par\noindent\setbox0=\hbox{\INTERN@L##1}\hbox{\hskip\wd0}}%
```

\HV's and \HOV's are set as internal \HV's and \HOV's after starting a new line.

```
\def\HV##1{\par\noindent{\INTERN@L\HV{##1}}}%
\def\HOV##1{\par\noindent{\INTERN@L\HOV{##1}}}%
```

A \V only sets \VMODE.

```
\def\V##1{\VMODEtrue##1}%
}% end of \EXTERN@L
```

Boxes in internal list

```
\def\INTERN@L{%
```

As in the external definition \S, \H and \I are \hbox's. The boxes in a \H are not in vertical mode, \I does not change the modes. The difference is that the width of the boxes is measured (\wd0) and used to decrease the space left on the current line (\RMdecr{\wd0}).

```
\def\S##1{\setbox0=\hbox{##1}\copy0\RMdecr{\wd0}}%
\def\H##1{\setbox0=\hbox{\HVMODEfalse\VMODEfalse ##1}\copy0\RMdecr{\wd0}}%
\def\I##1{\setbox0=\hbox{##1}\copy0\RMdecr{\wd0}}%
\def\WD##1{\setbox0=\hbox{##1}\RMdecr{\wd0}\hbox{\hskip\wd0}}%
```

An internal \V box can not occur according to the assumptions of the translation. To be prepared for the worst we issue a warning to the user and set the code with a \vtop.

```
\def\V##1{\typeout{WARNING (Box to TeX): internal V box}%
\vtop{\EXTERN@L\HVMODEfalse\VMODEtrue ##1}}%
```

A \HOV first tries to set its contents completely horizontally. Then it tests whether the resulting box fits on the line by comparing its width with \RM. If there it does not fit the contents are set in a vertical box that aligns at the top, otherwise the horizontal box is placed.

```
\def\HOV##1{\setbox0=\hbox{\hor ##1}%
\ifnum\wd0>\RM\vtop{\EXTERN@L\HVMODEfalse\VMODEtrue##1}%
\else\copy0\RMdecr{\wd0}\fi}%
```

For a \HV the same procedure is followed. The difference is that the vertical box has a fixed width of \RM.

```
\def\HV##1{\setbox0=\hbox{\hor ##1}%
\ifnum\wd0>\RM\setbox0=\vtop{\hsize=\RM\VMODEfalse\HVMODEtrue\noindent ##1}%
\fi\copy0\RMdecr{\wd0}}%
}% end of \INTERN@L
```

In the implementation of \HOV and \HV we used the macro \hor to force possibly non-horizontal boxes and skips inside these boxes to behave like \H boxes. Its definition is

```
\def\hor{\VMODEfalse%
\def\HV##1{\H{##1}}%
\def\HOV##1{\H{##1}}%
\def\ISKIP##1##2{\HSKIP{##2}}%
\def\HVSKIP##1##2##3##4{\HSKIP{##1}##4}}
```

Space Units Units for horizontal, indentation and vertical space can be used to increase or decrease the spacing in the documentation uniformly.

```
\newdimen\UH\UH=.3em% unit for horizontal space
\newdimen\UI\UI=.5em% unit for indentation space
\newdimen\UV\UV=1ex%  unit for vertical space
```

Skips A `\HSKIP` produces a horizontal skip if not in `VMODE`, an `\ISKIP` produces a horizontal skip only in `VMODE` and a `\VSKIP` produces a vertical skip in `VMODE`.

```
\def\HSKIP#1{\ifVMODE\else\hbox{\hskip#1\UH}\RMdecr{#1\UH}\fi}
\def\ISKIP#1#2{\ifVMODE\hbox{\hskip#1\UI}\RMdecr{#1\UI}\else%
\hbox{\hskip#2\UH}\RMdecr{#2\UH}\fi}
\def\VSKIP#1{\ifVMODE\ifdim#1\UV>0pt\par\vskip-\lastskip\vskip#1\UV\fi\fi}
```

A `\HOVSKIP` is a `\VSKIP` in vertical mode and a `\HSKIP` otherwise.

```
\def\HOVSKIP#1#2{\ifVMODE\VSKIP{#2}\else\HSKIP{#1}\fi}
```

A `\HVSKIP` takes a box as last argument and decides whether to set it on this line or on the next. If a completely horizontal setting of the box (with `\hor`) yields a box that is less wide than `\RM` it places the box. Otherwise a new line is started and the width of the box is compared to the width of the enclosing `\HV`. If the box is still to wide it is set non-horizontally, i.e., possible `\HOV`'s and `\HV` inside the box are allowed to be set vertically.

```
\def\HVSKIP#1#2#3#4{%
\setbox0=\hbox{\hor\HSKIP{#1} #4}%
\ifnum\wd0>\RM\par\noindent\dimen0=\hsize\edef\RM{\dimen0}%
\setbox0=\hbox{\VMODEtrue\ISKIP{#2}{0}\hor #4}%
\ifnum\wd0>\RM\setbox0=\hbox{\ISKIP{#2}{0}#4}\fi\fi%
\copy0\RMdecr{\wd0}%
}
```

Initialization To use the generated code in a document several initializations have to be done. These initializations are performed by `\boxterm` or by `\freeterm`. The first one puts the entire code inside a minipage, so that it can be used in figures. The second only does initialization and then sets the code in the surrounding vertical list. Both macros have two arguments. The first argument is the size of the display and the second argument is the generated code.

```
\long\def\boxterm#1#2{{%
\dimen0=#1\edef\RM{\dimen0}\dimen0=#1\edef\HSIZE{\dimen0}%
\fbox{\begin{minipage}[t]{#1}\EXTERN@L\VMODEtrue #2\end{minipage}}}}

\long\def\freeterm#1#2{{
\dimen0=#1\edef\RM{\dimen0}%
\dimen0=#1\edef\HSIZE{\dimen0}\EXTERN@L\VMODEtrue #2}\par}
```

11 Case Study: Typesetting PSF

The process specification formalism PSF [MV93] is an algebraic specification formalism based on process algebra for describing concurrent systems like communication protocols. It is a formal machine-readable language that can be written as ASCII text. There are several tools for PSF such as a typechecker, a term rewrite system compiler for data modules and an interactive simulator for processes.

PSF is a typical example of a document language. Its main purpose is not to be compiled into executable software, but to capture a system with a formal, high-level description that serves as a standard for many implementations. A documentation tool for this language is therefore very useful.

With the techniques described in this paper we constructed a typesetter for PSF. In this section we give an overview of the construction process of the typesetter and give an example of a typeset specification.

Syntax of PSF The syntax of PSF is defined in [MV93] as set of BNF rules. We translated these BNF rules to an SDF definition—using a specified BNF to SDF translator.

Generation and tuning of (PSF → Box) From the syntax definition of PSF a box constructor is generated. The default rules were adapted to the preference of [MV93]. The context-free grammar of PSF is too big to be presented here, therefore we have selected a context-free grammar rule and give the generated as well as the adapted pretty print rules for this rule.

The following SDF rule describes the process declaration list in a PSF specification:

```
"processes" Process-decl-list+  -> Processes
```

A piece of PSF code described by this context-free grammar rule is:

```
processes
  CLUSTER : PORT
  KICK-OFF : PORT
```

The pretty print generator derives the following default pretty print rules for this context-free grammar.

```
[default-48]
  ppProcesses(processes var_Process-decl-list_+0) =
    H ["processes" ppProcess-decl-list+(var_Process-decl-list_+0)]

[default-211]
  ppProcess-decl-list+(var_Process-decl-list_+0) =
    V [ppProcess-decl-list+1(var_Process-decl-list_+0)]
```

Independent of the number of elements in the `Process-decl-list` the pretty printed text will always have the same format. For example, if these rules are applied to the PSF code above the result will be:

```
processes CLUSTER : PORT
          KICK-OFF : PORT
```

If we want to have a more flexible pretty print result we have to adapt these generated rules. First of all we mark the keywords, so that the TEX back-end can put them in a special font, for example bold. Secondly we change the generated pretty print rule in such a way that when the list consists of only one element this element is printed immediately after the keyword `processes`, but when the list consists of more than one element the complete list is printed on a new line with extra indentation. This is simply achieved by replacing the H operator by the HOV operator in the right hand side of equation `[48]` and indenting the declaration list with `I`.

```
[48] ppProcesses(processes var_Process-decl-list_+0) =
       HOV [#KW ["processes"]
            I [ppProcess-decl-list+(var_Process-decl-list_+0)]]
```

The result of this adaptation can be found in process module TokenRing below where a `Process-decl-list` with one element as well as two elements appear.

Using (PSF → TEX) A PSF specification can be developed with the syntax directed editors in the PSF programming environment generated by the ASF+SDF meta-environment. These editors have a "TEX" button for the automatic translation of the PSF specification into TEX code. The generated TEX file can be included in a TEX document.

Two Complete Modules As a final example we show two PSF modules from [MV93]. The first is a data module specifying the datatype queue. The second is a process module that is part of a specification of tokenrings. Note the usage of fonts: keywords are set in bold type, module names, equation tags, sorts and set identifiers in roman and other identifiers in italic. Note also that pagebreaks can occur in specifications.

data module Queues
begin

  parameters
    Queue-parameter
    begin
      sorts Q-ELEMENT
      functions
        default-q-element : → Q-ELEMENT
    end Queue-parameter

  exports
  begin
    sorts QUEUE
    functions
      empty-queue : → QUEUE
      enqueue : Q-ELEMENT # QUEUE → QUEUE
      serve : QUEUE → Q-ELEMENT
      dequeue : QUEUE → QUEUE
      length : QUEUE → NATURAL
  end

  imports Naturals

  functions
    _*_ : Q-ELEMENT # QUEUE → QUEUE

  variables
    e, *e'* : → Q-ELEMENT *q* : → QUEUE

  equations
    [01] *enqueue*(*e*, *q*) = *e* ∗ *q*
    [02] *serve*(*empty-queue*) = *default-q-element*
    [03] *serve*(*e* ∗ *empty-queue*) = *e*
    [04] *serve*(*e* ∗ (*e'* ∗ *q*)) = *serve*(*e'* ∗ *q*)
    [05] *dequeue*(*empty-queue*) = *empty-queue*
    [06] *dequeue*(*e* ∗ *empty-queue*) = *empty-queue*
    [07] *dequeue*(*e* ∗ (*e'* ∗ *q*)) = *e* ∗ *dequeue*(*e'* ∗ *q*)
    [08] *length*(*empty-queue*) = *zero*
    [09] *length*(*e* ∗ *q*) = *s*(*length*(*q*))

end Queues

process module TokenRing
begin

  exports
    begin
      processes *TOKENRING*
    end

  imports Ports, Ring-Interfaces, Buffer, Token-Transmission, Utilities

processes
  CLUSTER : PORT
  KICK-OFF : PORT

sets
  of atoms
  C = $\{r(p, d), s(p, d)$
    $\mid p$ **in** PORT,
     d **in** OCTET$\}$
  H = $\{$*get-pdu-queued*(p, b), *put-pdu-queued*(p, b)
    $\mid p$ **in** PORT,
     b **in** BOOLEAN$\}$
   + $\{$*get-buf-req*(p, b), *put-buf-req*(p, b)
     $\mid p$ **in** PORT,
      b **in** OCTET$\}$

variables
  $p : \rightarrow$ PORT

definitions
  TOKENRING = **encaps**(C, **merge**(p **in** PORT, *CLUSTER*(p)))

  CLUSTER(p) =
   ([*is-monitor*(p) = *true*] $\rightarrow$ *TX-TOKEN*(p) $\cdot$ *KICK-OFF*(p)
   + [*is-monitor*(p) = *false*] $\rightarrow$ *KICK-OFF*(p))

  KICK-OFF(p) = **encaps**(H, *RI*(p) $\|$ *Buffer*(p))

end TokenRing

12 Discussion

12.1 Related Work

Pretty printing with boxes One of the first thorough studies of pretty printing is made by Oppen [Opp80]. He introduces the notion of blocks, which are comparable to boxes and the notion of markings for possible line breaks. He distinguishes two different kinds of possible line breaks, viz. consistent and flexible ones. The consistent ones have the same functionality as the HOV operators and the flexible ones resemble the HV operators.

PPML [MCC86] (inspiration for the box language presented in Section 5) is a programming language for constructing boxes. PPML supports many more features than our box language. Among these are computational capabilities that are similar to rewriting. We do not need such power in our box language since computation can be performed by the conditional equations. In PPML there is a confusion between the translation of one language to another and pretty printing a language. PPML specifications can also be processed by the formatting machine FIGUE [Has92], which offers the options for changinng fontsizes and colors.

To the best of our knowledge, the way we transform boxes is new.

Pretty printing to TEX Literate programming tools WEB [Knu84b], CWEB, noweb, fweb, nuweb. Some of these tools do pretty printing.

Spider [Ram89a, Ram89b] is a generator for `WEB` tools. The language independent parts of Knuth's original `WEB` tools are separated from the language independent parts such as the pretty printer. The pretty printer can be generated by an `AWK` script from a so called pretty print grammar. For a new language a pretty print grammar can be obtained by rewriting a grammar for a language that is similar in appearance.

MathPad [Mat] is an almost complete WYSIWYG editor for mathematical formulas working with TEX as background process, it has been developed at the Technical University of Eindhoven.

Other documentation tools There exist several language specific documentation tools, for example for Pascal, C, Eiffel [Mey92], Lotos.

The documentation tool (ASF+SDF → LATEX) [Vis94] that translates ASF+SDF specifications to LATEX was used to typeset the specifications in this paper. Improvement of this tool was the original motivation for the work that we described.

12.2 Applications

The combination of a powerful pretty printer generator with the (Box → TEX) backend offers us the possibility of constructing for almost any language a software documentation tool. Given a context-free grammar in SDF the amount of time needed to develop such a documentation tool for the language depends heavily on how sophisticated the documentation tool must be. After having generated the pretty printer the documentation tool is also available, but it must be tuned to satisfy the users of the documentation tool.

We have used two languages as case studies. The toy language Pico is a standard ASF+SDF example. It has a Pascal-like syntax for expressions and statements. An example of a Pico program which has been typeset can be found in Figure 1. In Section 11 we presented a typesetter for PSF [MV93].

In the near future we plan to construct documentation tools for a couple of languages. We discuss a few of these plans.

Typesetting ASF+SDF specifications is currently done by a lisp program. We plan to replace this program using (Box → TEX).

Another application, is the construction of literate programming tools from language definitions. The WEB approach to literate programming can be applied to arbitrary languages. In fact, a WEB language can be derived automatically from the context-free grammar of a language L by providing extra syntax for chunk definitions and quotations. By generating a typesetter for the new WEB language, which incorporates the typesetter for L, we have a new L-WEAVE. By generating a specification of substitution of chunk names by their bodies we have a new L-TANGLE. Syntax directed editing, parsing and other tools already specified for L can be extended automatically to L-WEB.

An application outside the domain of programming languages is typesetting of long proofs (see for instance [Lam93]). If a syntax for proofs based on a syntax of formulas is written, proofs can be written in ASCII and checked for syntactic correctness and even for other properties. These ASCII proofs can then be typeset automatically by the typesetter generated from the proof syntax, guaranteeing at least syntactic correctness of the proofs in a publication.

12.3 Future Work

12.3.1 Extensions

In this paper we have defined the core of a typesetting tool for programming languages. Although this core is powerful enough to cope with most typesetting problems for programming language constructs, users will never be satisfied with these possibilities. Therefore, a uniform way of extending the capabilities of generated typesetters, typesetter generation and the box language is needed.

User extensions Different types of users can be distinguishd. First of all there are users who develop a context-free grammar for a (new) language and want to generate a documentation tool for this language as well. They must be able to modify the generated pretty printer and if they feel the need to 'extend' the box language as well as the typesetter. The modification of the generated pretty printer is simple and well supported. The extension of the box language and typesetter is more tedious and asks more experience in both writing specifications and adapting TEX macros.

The second group, the users of a programming environment, should not have to adapt pretty print rules when not satisfied with the pretty printing of their programs. Instead user configuration should be done by setting a few flags that modify the behaviour. The language specifier should design these flags. A simple way to take such options into account in the pretty printer, is by adding an extra argument that contains a list of options to each box construction function and making the box construction rules conditional where appropriate: if an option governs the style of a certain construct, then for each value the option can take there is a rule that constructs the appropriate box under the condition that the option has that value.

Formatter extensions The introduction of new features in the typesetters or box language can be tackled by defining a new box language which incorporates the core box language and the new operator(s). Examples of adaptations that can be made are:

Fonts An obvious extension of the box language is declaration which fonts to use for the tokens of a language. As an example of extending the box language we describe in Appendix B the font extension.

Crossreferences and indexing are an important means for making documentation more accessable. The box language should be extended with declarations of crossreferences and indexed tokens. These declarations should not depend on the typesetter used by the back-end.

Comments should be handled in the same way as in the current implementation of (ASF+SDF $\rightarrow$ LATEX).

Alignments allow arranging the tokens in several lines of text in columns. This is a useful feature for formatting, for instance, variable declarations or BNF rules.

Other structural compositions The box language should offer a uniform way to be extended with new structural compositions, such as sub- and superscripts, over, fractions, etc.

For example, if the documentation writer wants to use the mathematical formula $\frac{\sqrt{x^n}}{300 \cdot k}$ it is much easier for him to type `sqrt x ^ n / (300 * k)` instead of `$\frac{\sqrt{x^n}}{300 \cdot k}$`.

Using a formal language, defined by a context-free grammar, to write mathematics offers more possibilities for consistency checking than the TEX source. The pretty printer can produce a clean translation to a typesetting language like TEX if formula's, expressions, proofs and other mathematical texts are written in such a formal (ASCII) language.

There is no need to introduce general computational aspects into the box language. All computation should be dealt with during box construction or during formatting.

12.3.2 Correctness of box transformations

Another point of future research is not concerned with typesetting but with the specification techniques used in this paper.

The correctness of box transformations should be proven. We derived equations on boxes by considering which boxes are treated in the same way by the text formatter. Another approach to derive transformations for boxes would be to take the text formatter as starting point and eliminating the assumptions that it uses about texts. For instance, the text formatter can compute the width of a text by assuming that all characters in strings and space have the same width. If we drop this assumption, i.e., leave the functions *width*, *hwidth* and *twidth* undefined, terms like $hskip(width(" + "))$ can no longer be computed. In general, we would drop all assumptions that depend on the text model. By abstract interpretation of the text formatter we get a text term that is not completely reduced to a list of strings. An implementation of the text operators in TEX—or any other typesetting language–would then give the desired result.

13 Conclusions

We have shown in this paper that by means of algebraic specifications we were able to reuse a rather simple front-end for pretty printing programs in a language L to build a documentation tool for this language. ASF+SDF proved not only to be powerful enough to specify all components of the pretty printer generator, but it inspired us to connect the pretty printer front-end to the TEX back-end.

Given a rather simple but powerful box language we developed a number of specifications which translate box expressions to TEX code. Although TEX is also based on a kind of boxes this translation could not be done in a straightforward manner. It was necessary to define a number of text formatting semantics preserving box transformation rules. ASF+SDF allowed us to define these transformations in a simple and elegant way. Given these transformation rules we were able to transform box expressions so that they can be translated easily to TEX code. The box transformations were essential in order to reduce the nesting level in the box expression and to lift V boxes to the outermost level

We have used this technique succesfully to construct documentation tools for the toy language Pico and the specification language PSF. The development of a documentation tool boils down to the generation of a box constructor, and connect it to a TEX code producing back-end. To obtain a good documentation tool it is necessary to tune the generated box constructor so that it produces a satisfactory code.

Acknowledgements

We thank Susan Üsküdarlı and Paul Klint for reading and commenting on drafts of this paper. We thank Sjouke Mauw for providing an ample supply of PSF specifications and for critically commenting on the output of our PSF typesetter. The fast production of the PSF typesetter would not have been possible without Wilco Koorns BNF to SDF translator.

References

[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.

[Bra93a] M. G. J. van den Brand. Generation of language independent prettyprinters. Technical Report P9327, programming Research Group, University of Amsterdam, Amsterdam, October 1993.

[Bra93b] M. G. J. van den Brand. Prettyprinting without losing comments. Report P9315, Programming Research Group, University of Amsterdam, July 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Bra93.ps.Z.

[Deu92] Arie van Deursen. A numerical package for ASF+SDF. Technical report, CWI, Amsterdam, 1992. Unpublished report.

[Has92] L. Hascoet. *FIGUE An Incremental Graphic Formatter User's manual for Version 1.* INRIA, Sophia-Antipolis, 1992.

[HHKR93] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF – Reference Manual,* version May 27, 1993. Earlier version in *SIGPLAN Notices*, 24(11):43–75, 1989. Available by *ftp* from ftp.cwi.nl:/pub/gipe as SDFmanual.ps.Z.

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

[Knu84a] Donald E. Knuth. *The TEXbook*, volume A of *Computers & Typesetting.* Addison-Wesley, 1984. (Ninth printing, revised, October 1989).

[Knu84b] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984. Reprinted as Chapter 4 of [Knu92].

[Knu92] Donald E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[Lam93] Leslie Lamport. How to write a long formula. Technical Report 119, DEC Systems Research Center, Palo Alto, California, December 25 1993. Minor correction: January 18, 1994.

[Mat] MathPad manual pages. (more information from `mathpad@win.tue.nl`).

[MCC86] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice Hall Object-Oriented Series. Prentice Hall, 1992.

[MV93] S. Mauw and G. J. Veltink, editors. *Algebraic Specification of Communication Protocols*, volume 36 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.

[Opp80] D. C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[Ram89a] Norman Ramsey. Literate programming: Weaving a language-independent `WEB`. *Communications of the ACM*, 32(9):1051–1055, September 1989.

[Ram89b] Norman Ramsey. A spider user's guide. Technical report, Department of Computer Science, Princeton University, July 1989. Available by anonymous ftp from `ftp.tex.ac.uk` in directory `/pub/spiderweb` as part of the distribution of Spider.

[Vis94] Eelco Visser. Writing course notes with ASF+SDF to LaTeX. In T. B. Dinesh and Susan M. Üsküdarlı, editors, *Using the ASF+SDF environment for teaching computer science*, chapter 6. 1994. To be presented at the Workshop on Teaching Formal Methods.

A Library Modules

In this section we give the signatures (syntax parts) of the library modules we used in this paper. Module Layout defines layout in specifications. Module Booleans defines the datatype Booleans with operations **true**, **false**, **or**, **and** and **not**. Modules Numbers-Syntax and Numbers-Op define the datatype NUM. A NUM is an integer or a real. The datatype is built on top of a standard numerical package for ASF+SDF [Deu92].

We also give the syntax of identifiers and the syntax and equations of Strings.

A.1 Layout

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

A.2 Booleans

imports Layout$^{(A.1)}$
exports
  sorts BOOL
  context-free syntax
    true → BOOL
    false → BOOL
    BOOL **or** BOOL → BOOL {**assoc**}
    BOOL **and** BOOL → BOOL {**assoc**}
    not BOOL → BOOL
    "(" BOOL ")" → BOOL {**bracket**}
  priorities
    or < **and** < **not**
  variables
    Bool [0-9′]* → BOOL

A.3 Numbers-Syntax

imports Int-con$^{(A)}$ Mant-real-con$^{(A)}$ Real-op$^{(A)}$
exports
  sorts NUM
  context-free syntax
    INT → NUM
    REAL → NUM
    NUM "⊕" NUM → NUM {**left**}
    NUM "⊖" NUM → NUM {**left**}
    NUM "$<_N$" NUM → BOOL
    NUM "$\leq_N$" NUM → BOOL
    NUM "$>_N$" NUM → BOOL
    NUM "$\geq_N$" NUM → BOOL
    "*0?*" (NUM) → BOOL
hiddens
  variables
    [*r*][0-9′]* → REAL
    [*hijkmn*][0-9′]* → INT
    [*N*][0-9′]* → NUM

A.4 Numbers-Op

imports Numbers-Syntax$^{(A.3)}$ Real-op$^{(A)}$ Deal$^{(A)}$ Dint$^{(A)}$
exports
context-free syntax
"$\lfloor$" NUM "$\rfloor$" → INT
hiddens
variables
N [*0-9*′]∗ → NUM
n[*0-9*′]∗ → INT
r[*0-9*′]∗ → REAL
dr[*0-9*′]∗ → DOT-REAL

A.5 Identifiers

imports Layout$^{(A.1)}$
exports
sorts ID
lexical syntax
[*A-Za-z*]+ → ID
variables
"*Id*" [*0-9*′]∗ → ID

B Fonts

In this appendix we describe the extension of the box language with font operators.

B.1 Box Fonts

The operators in the kernel box language determine the structure of the box. Font operators do not influence the structure of a box but determine the appearance of basic boxes, i.e., strings.

The argument of a font operator specifies its shape, f.i., italic, roman. It is conceivable that more aspects of a font might be specified such as size, family and boldness (see the new font selection scheme of TeX for specification of fonts); this has not only relevance for a TeX backend.

imports Box$^{(5)}$ Identifiers$^{(A.5)}$
exports
sorts FONT-OPERATOR FONT-PARAM FONT-OPTION FONT-OPTIONS
context-free syntax

FONT-PARAM "=" ID	→ FONT-OPTION
FONT-PARAM "=" NAT	→ FONT-OPTION
FONT-OPTION∗	→ FONT-OPTIONS
"#F" FONT-OPTIONS	→ FONT-OPERATOR
FONT-OPERATOR "[" BOX "]"	→ BOX
FONT-OPERATOR "(" BOX-LIST ")"	→ BOX-LIST

variables
"*fp*" [*0-9*′]∗ → FONT-PARAM
"*fo*" [*0-9*′]∗ → FONT-OPTION
"*fo*" "∗" [*0-9*′]∗ → FONT-OPTION∗
[*F*][*0-9*′]∗ → FONT-OPERATOR

Font Parameters A font is characterized by a series parameters. *fn* font name; abbreviation for other parameters except size *fm* (family), *se* (series), *sh* (shape), *sz* (size) and *cl* (color)

context-free syntax

$fn \rightarrow$ FONT-PARAM
$fm \rightarrow$ FONT-PARAM
$se \rightarrow$ FONT-PARAM
$sh \rightarrow$ FONT-PARAM
$sz \rightarrow$ FONT-PARAM
$cl \rightarrow$ FONT-PARAM

Semantics of Font Operators We show how font operators have no effect on the structural aspects of a box expression and how font operator interact with other font operators. Font operators distribute over all structural operators.

$$
\begin{array}{lll}
(1) & F[O[A^*]] & = \; O[F(A^*)] \\
(2) & F() & = \\
(3) & F(A\ A^*) & = \; F[A]\ B^* \quad \textbf{when} \quad B^* = F(A^*)
\end{array}
$$

A font operator without any font changing options is meaningless.

$$(4) \quad \#\mathsf{F}\ [A] \; = \; A$$

Inner font operators inherit font options for a parameter *fp* from outer font operators if do not already have an option for that parameter.

$$(5) \quad \#\mathsf{F}\ fo_1^*\ fp = Id_1\ fo_2^*[\#\mathsf{F}\ fo_3^*\ fp = Id_2\ fo_4^*[A]] = \#\mathsf{F}\ fo_1^*\ fo_2^*[\#\mathsf{F}\ fo_3^*\ fp = Id_2\ fo_4^*[A]]$$

$$(6) \quad \#\mathsf{F}\ fo_1^*[\#\mathsf{F}\ fo_2^*[A]] = \#\mathsf{F}\ fo_1^*\ fo_2^*[A] \qquad \textbf{otherwise}$$

Font Operator Abbreviations In many applications it is useful to abstract from the actual font that is given to a box. We only want to distinguish 'logical fonts'. We give some font operators that are useful in most programming languages; They distinguish keywords, variables, numbers and math items (such as brackets and other mathematical symbols). The formatting backend can give a suitable definition for these operators.

context-free syntax

"#KW" $\rightarrow$ FONT-OPERATOR
"#VAR" $\rightarrow$ FONT-OPERATOR
"#NUM" $\rightarrow$ FONT-OPERATOR
"#MATH" $\rightarrow$ FONT-OPERATOR
"#ESC" $\rightarrow$ FONT-OPERATOR

B.2 Box-Fonts to Text

The text formatter we defined has a plain ASCII display that can not handle fonts as target. Formatting fonts for these displays is therefore simple. A font operator applied to some string is just thrown away.

imports Box-to-Text-E$^{(9.3)}$ Box-Fonts$^{(B.1)}$

hiddens

variables

$m \rightarrow$ NUM

$$(1) \quad \#\mathsf{F}\ fo^*[\mathrm{a}](O, m) \; = \; \mathrm{a}(O, m)$$

B.3 Box-Fonts to $\TeX$

imports Box-to-TeX$^{(10.2)}$ Box-Fonts$^{(B.1)}$ Box-Laws-F$^{(??)}$
hiddens
variables
"*chr**"[0-9′]* → CHAR*

Font Operators A fontname is translated to a command sequence that calls the font.

(1) $tex(\#\mathsf{F}\ fn = id(chr^*)[string(\texttt{"""}\ chr_2^*\ \texttt{"""})]) = (\texttt{\textbackslash S}\ \{\ cs(\texttt{"\textbackslash"}\ chr^*)\ \{\ \}\ string(chr_2^*)\ \})$

We do not yet give a translation for the other font options. These were designed for use with the New Font Selection Scheme of LaTeX, and will be supported in the near future.

The abbreviations might for instance be defined, using LaTeX's fontnames, as

(2) #KW = #F fn = `bf`
(3) #VAR = #F fn = `sl`
(4) #NUM = #F fn = `rm`
(5) #MATH[a] = `"$"` * a * `"$"`

Several (combinations of) ASCII characters can be mapped to mathematical symbols. To indicate that this is desired the font operator `#ESC` should be used in the box construction rules.

```
(6)  #ESC["{"]  = "$\{$"        (7)  #ESC["}"]  = "$\}$"
(8)  #ESC["["]  = "$[$"         (9)  #ESC["]"]  = "$]$"
(10) #ESC["("]  = "$($"         (11) #ESC[")"]  = "$)$"
(12) #ESC["#"]  = "\#"          (13) #ESC["->"] = "$\to$"
(14) #ESC["^"]  = "\HAT"        (15) #ESC["="]  = "$=$"
(16) #ESC["||"] = "$||$"        (17) #ESC["_"]  = "\_"
(18) #ESC["."]  = "$\cdot$"     (19) #ESC["|"]  = "$|$"
(20) #ESC[">"]  = "$>$"         (21) #ESC[">="] = "$\geq$"
(22) #ESC["<"]  = "$<$"         (23) #ESC["<="] = "$\leq$"
(24) #ESC[":"]  = "$:$"
(25) #ESC["-"]  = "$-$"         (26) #ESC["+"]  = "$+$"
(27) #ESC["/"]  = "$/$"         (28) #ESC["*"]  = "$*$"
```

Contents