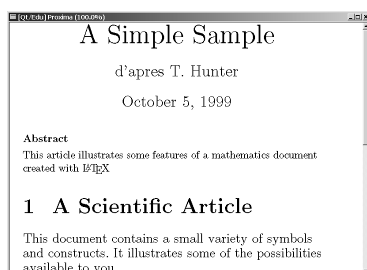
Chapter 6

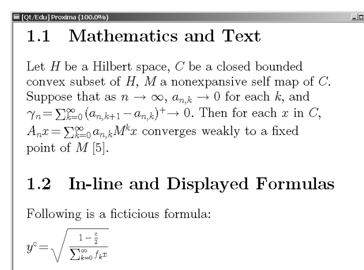
Presenting structured documents with XPREZ

In this section, we introduce the presentation language of Proxima: XPREZ. The language is an extended version of the presentation level that was discussed in Section 3.1.3. XPREZ has been implemented in Haskell. Although the language is not yet fully developed, it is already powerful enough to cover much of the TEX math typesetting as described in [33].

Although there are many presentation languages for structured documents (e.g. [1, 3, 11, 56, 71]), not a single one seems to have sufficient expressiveness and abstraction mechanisms to specify the presentations of the use cases from Chapter 2. Pretty-printing libraries (e.g. [14, 36, 41, 69, 84]) do offer expressiveness, abstraction mechanisms, but these libraries are mainly text-oriented.

Below are two screenshots of example XPREZ presentations containing ligatures (e.g. the “fi” in “Scientific”) and several mathematical constructs. An editor for these presentations can already be instantiated with the Proxima prototype, but in order to support pleasant editing, the prototype requires a few more extensions.





The XPREZ language consists of a set of Haskell combinators that can be used to define a presentation. A presentation is tree-structured and represents an attribute-grammar tree. The attributes are presentation attributes such as font size and color. A special combinator can be used for modifying presentation attributes.

Section 6.1 discusses several existing presentation languages and states a number of requirements for a presentation language. This is followed by an informal description of the XPREZ presentation language in Section 6.2. Section 6.3 concludes with an overview of future research.

6.1 Presentation languages

In this section, we discuss five presentation languages for structured documents. XSL [1] is a presentation language for XML documents, whereas CCSS [3] and PSL [56] are presentation languages for HTML. CSS 2.0 [11] can be used to present both XML and HTML documents. Finally, the language P [71] is the presentation language of the Thot editor toolkit (see Section 2.3.3).

In Section 2.2.3, we mentioned that a presentation language consists of two parts. One part is the *presentation target language*, which describes the components (strings, boxes, rows, etc.) of a presentation. The other part is the *presentation specification language*, in which we specify how a document is mapped onto an element of the target language. Not every presentation language explicitly identifies its presentation target language. Moreover, if the target language supports abstraction, a presentation may contain functions, which makes it difficult to clearly separate the two languages.

In the remainder of this chapter, we focus on presentation target languages, because XPREZ is the presentation target language of Proxima. The presentation specification language of Proxima is an attribute-grammar formalism, but the details of using this formalism in Proxima still require further research. Section 7.2 provides more information on the presentation attribute grammar, as well as a few example presentation specifications.

In most presentation languages, a presentation is a tree structure, in which the leaves are atomic presentations and the nodes are composite presentations. An atomic presentation is a string or a simple graphical objects such as a line, a box, or an image. On the other hand, a composite presentation specifies for a number of child presentations how these are put together. We distinguish three different layout models for composite presentations: a box, a matrix, and a flow layout.

Box layout. In a box layout, child presentations are positioned relative to each other, for example horizontally in a row, or vertically in a column. The exact positioning may be specified for the entire list of children (e.g. by using a row or column combinator), or for each child itself, by specifying how it should be positioned relative to its sibling. A box model may provide facilities for aligning and stretching child presentations.

Matrix layout. A matrix model is similar to a box model, but aligns its children both horizontally and vertically. Because it is more general than a row or a column, a matrix may be used to create a box layout.

Flow layout. A flow layout is used for line- and page-breaking. Child presentations are placed next to or below each other, until the remaining space is too small to fit the next presentation, in which case a new line or page is started. Thus, unlike in a box or matrix layout, the structure of the presentation as it is finally rendered is only partially determined by the structure of the presentation definition.

Besides constructs for specifying the structure of the presentation, presentation languages also have a notion of presentation attributes (sometimes called properties). A presentation attribute affects the style, size, or position of a presentation, but not its structure. Examples of presentation attributes are font size, background color, alignment information, etc.

6.1.1 Existing presentation languages

Of the five presentation languages we examined, only XSL regards its target language as a language in its own right, with a separate syntax. The other languages only describe how presentation attributes of the elements of the target presentation tree can be set, but do not treat a presentation as an actual value in the language.

CSS 2.0. Cascading Style Sheets, level 2 [11] is an example of a simple presentation language. Its target language is almost invisible to the style sheet designer. A presentation is a tree structure, in which the nodes specify presentation attributes such as font size or color, and the leaves are document content. A presentation attribute may be specified either absolutely or as a percentage. The meaning of a percentage depends on the attribute. For instance, a percentage value for the *font-size* attribute refers to the font size of the parent element, but a percentage for the *line-height* attribute refers to the font size of the element itself. It is not possible to let the value of a presentation attribute depend on arbitrary presentation attributes of the parent or siblings in the presentation tree.

CSS 2.0 supports a flow layout and a table format for a matrix layout. However, the control over alignment in the matrix model is rather weak.

CCSS. Constraint Cascading Style Sheets [3] is an extension of the CSS 2.0 standard that is based on constraints. The target language of CCSS closely resembles the CSS 2.0 target language, but presentation attributes for a child are specified using constraints instead of percentages of the parent's attribute values. Another difference is that the constraints may refer to global constraint variables and to left-siblings in the presentation tree as well as to the parent node. Similar to CSS 2.0, CCSS supports a flow and matrix layout, but no box layout.

XSL FO. On the other side of the spectrum is the XSL stylesheet language for XML. The design of the target language, XSL Formatting Objects, was based on the flow objects of

the DSSSL [39] presentation language for SGML. The formatting objects standard consists of a large collection of elements that can be used to specify page models, presentation attributes, and more complicated presentation aspects, such as hyphenation and counters. A presentation is a tree that consists of these formatting objects.

XSL FO offers strong control over the flow model, but a box model is not supported. The matrix (table) model for XSL FO has more control over alignment than CSS 2.0, but horizontal alignment is still poorly supported. Hence, a mathematical formula cannot be displayed elegantly in XSL FO.

PSL. The Proteus Stylesheet Language [56] is an attempt to combine the simplicity of CSS 2.0 with the power of XSL. PSL extends the CSS target language with a box model and graphical symbols. The value of a presentation attribute (which is called a property in PSL) can be expressed as a mathematical expression that refers to presentation attributes of nodes in the presentation tree. This mechanism is called *property propagation*.

PSL supports a flow model and a constraint-based box model, but lacks a matrix model. A presentation can specify its attributes for position and size relative to position and size attributes of other presentations in the tree. These other presentations can be addressed using a number of primitive functions for accessing siblings, parents, ancestors of a specific type, etc.

P. The language P is the presentation language of the Thot editor toolkit [71]. It has a target language that consists entirely of boxes, which may be composed according to a box, a flow, or a matrix model. P supports horizontal and vertical-reference lines for automatic alignment of boxes. Instead of having a large number of different presentation boxes, similar to XSL Formatting Objects, P has only three kinds of boxes with a large number of presentation attributes. In contrast to PSL, the box layout in P is not constraint-based.

Discussion

The languages discussed above are all declarative and domain-specific languages that vary in expressive power. The languages CSS 2.0, CCSS, and PSL allow simple presentations to be specified in a simple way, but cannot be used to specify more complex presentations, such as mathematical formulas. In contrast, XSL and P allow complex presentations to be specified, but due to the lack of abstraction, simple presentations also have rather elaborate specifications, especially in P.

Only P and PSL support a box model, but both models are of a rather object-oriented and imperative nature. Moreover, presentations are not first-class values. A box can specify its own position attributes relative to its parent or siblings, but it is not possible to state at parent-level that two child presentations should have their top and bottom aligned, or that two presentations should have the same widths.

Letting a child presentation specify its own layout makes it more difficult to understand a presentation. For example, to reverse the presentation of a horizontal list of children,

each child must specify that its right side must be aligned with the preceding child's left side. Moreover, if the order of the children depends on an attribute of the parent, then the presentation definition of each child needs to access this parent attribute and use its value to determine the alignment of the child.

If, on the other hand, child presentations are first-class, and abstraction mechanisms can be used to define combinators on them, a list of children may be reversed with a reverse function in the presentation definition of the parent. Another advantage of this approach is that the concepts of layout direction (horizontal or vertical) and order of the children are orthogonal now. The layout direction is determined by which combinator is applied to the list of children, whereas the order is determined by whether or not a reverse function is applied to the list. In the model of P and PSL, these concepts are intertwined, and reversing a horizontal list is conceptually different from reversing a vertical list.

Requirements

Based on the requirements from Chapter 2 and the previous discussion, we conclude that the presentation target language for Proxima, should meet the following requirements:

Proportional effort. It must be possible to specify complex presentations, but the specification of simple presentations should still be esy.

Declarative. In a declarative language, understanding a composite presentation is easier, because the computation of a presentation does not generate side effects. Another advantage is that the designer need not worry about the order of computation of presentations and attributes.

First-class presentations. A first-class presentation can be named and manipulated at the level of its parent, which in many cases is the natural place for such manipulations. At the same time, it is also possible to specify aspects of the presentation at the level of the child when this is more appropriate.

Box, matrix, and flow layout. All four layout models mentioned at the start of this section should be supported. The alignment of the box and matrix models must be powerful enough to specify complex presentations such as mathematical formulas.

Text, graphical, and widgets. It must be possible to specify text and graphical elements such as lines, boxes, and images. Moreover, the language must support user-interface widgets, such as buttons, selection lists, and menus.

Powerful abstraction mechanism. User-defined functions and variables help to reduce code duplication, facilitate code reuse, and increase transparency, because complex pieces of code may be replaced by functions with well-chosen names.

Domain-specific. The language should have syntax for presentation-specific constructs such as an **ex** (the height of the letter 'x' in the current font and size) and different measuring units such as pixels and inches.

```
data Inh = Inh { fontFamily :: String, fontSize :: Int,
                 textColor, lineColor, fillColor, bgColor :: Color }
data Syn = Syn { hRef, vRef, minWidth, minHeight :: Int,
                 hStretch, vStretch :: Bool}
```

Figure 6.1: The XPREZ presentation attributes

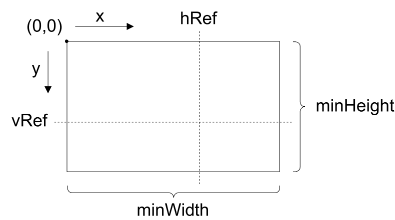
6.2 The XPREZ target language

With the requirements from the previous subsection in mind, we have developed the declarative presentation language XPREZ.

6.2.1 XPREZ presentation model

Similar to P and the document formatting languages TEX and Lout [45], XPREZ is a box language with support for flow layout. A presentation is a value of the abstract type `Xprez`, and is either an atomic box containing a text or a graphical object, or a composite box that contains a list of child presentation boxes. We construct `Xprez` values in the functional language Haskell, using a number of primitive functions that are described in Section 6.2.2.

A presentation box (from now on called presentation) has a number of attributes that describe its size and its appearance:



A presentation tree in Xprez represents an attribute grammar with inherited and synthesized attributes. Presentation attributes that are typically specified for an entire subpresentation, such as color and font size, are inherited attributes. On the other hand, presentation attributes that are set by a child and used by its parent, such as reference lines and size information, are synthesized attributes. Figure 6.1 shows the two Haskell records `Inh` and `Syn` that are used to model the inherited and synthesized presentation attributes. The figure also shows the type of each attribute.

The `hRef` and `vRef` attributes specify the reference lines that are used for aligning boxes horizontally and vertically when combined in composite presentations. Note that the

```
empty             :: Xprez
text              :: String -> Xprez
rect              :: Xprez
img               :: String -> Xprez
poly              :: [ (Float, Float) ] -> Xprez
row, col, overlay :: [ Xprez ] -> Xprez
rowR, colR        :: Int -> [ Xprez ] -> Xprez
matrix            :: [[ Xprez ]] -> Xprez
format            :: [ Xprez ] -> Xprez
```

Figure 6.2: The XPREZ primitives

vertical-reference line is in fact a horizontal line and vice versa. The term vertical-reference line stems from the fact that it is used for vertical alignment; modifying the vertical-reference line affects the vertical position of the presentation.

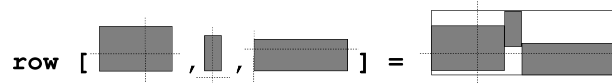
The boolean attributes `hStretch` and `vStretch` specify whether or not the presentation is allowed to stretch in horizontal or vertical direction. The remaining attributes are self-explanatory: `fontFamily`, `fontSize`, `textColor`, `lineColor`, `fillColor`, and `bgColor`. In the future, this set will be extended with other attributes such as line and font style, and attributes for modeling edit behavior (e.g. `onMouseClick :: EditCommand`).

6.2.2 XPREZ primitives

The first five combinators in Figure 6.2 specify atomic presentations. The `empty` combinator has a presentation that is invisible and takes up no space; it is the neutral element for various presentation compositions. A string is presented with combinator `text`, and a rectangle with combinator `rect`. The `poly` combinator takes a list of relative coordinates between (0.0, 0.0) and (1.0, 1.0) and produces a line figure that connects these points. The coordinates are relative because the final coordinates depend on the size of the `poly` presentation. Finally, `img` can be used to display external images. The argument is a string that contains the path to the image file. In a future version, an `img` term may also contain a reference to an image that is encoded as part of the document.

Except for `text`, the reference lines of an atomic presentation both have coordinate 0 (i.e. the north-west corner). For `text`, the vertical-reference line is the baseline of the text and the horizontal-reference line is at 0. By default, a simple presentation does not stretch.

The remaining primitives in Figure 6.2 specify composite presentations. The behavior of columns (`col`) is equal to that of rows (`row`) with the horizontal and vertical directions swapped. Hence, we only discuss the `row` primitive. In a row, each child presentation is placed immediately to the right of its predecessor, with their vertical-reference lines aligned. Horizontal-reference lines have no effect on the positioning in a row.



The bounding box of a row is the smallest rectangle that encloses all elements of the row. The vertical-reference line of the row is equal to the aligned reference lines of the children, whereas the horizontal-reference line is taken from the first child. In order to use the horizontal-reference line from one of the other children, we can use the `rowR` combinator. The integer argument of `rowR` specifies which child determines the horizontal-reference line for the row, with 0 denoting the first child.

By default, a row stretches in horizontal direction if one of its children does, and it stretches in vertical direction if all children stretch vertically. The defaults may be overridden by setting the stretch attributes with the method that is shown in the next section.

The `matrix` combinator can be used to describe a table layout, in which elements are aligned with elements to their left and right as well as with elements above and below them.

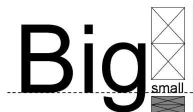
Because `row`, `column` and `matrix` do not allow their children to overlap, we need a special combinator for overlapping presentations. The `overlay` combinator places its children in front of each other, while aligning both the horizontal and vertical-reference lines. It can be used to create underlined text, for example. Because alignment takes place on both reference lines and hence all child reference lines overlap, no special `overlayR` combinator is needed.

A flow layout can be achieved with the `format` combinator for paragraph formatting. The combinator takes a list of presentations as argument and splits this list into rows based on the available horizontal space. The resulting rows are placed in a column. Because `Xprez` does not yet have a page model, only horizontal formatting is supported.

Here is an example XPREZ presentation that illustrates alignment and stretching in a row:

```
let cross     = poly [(0,0),(1,0),(1,1),(0,1),(0,0),(1,1),(0,1),(1,0)]
    greycross = cross ‘withStretch‘ True ‘withbgColor‘ grey
in  row [ text "Big" ‘withFontSize‘ 200
        , colR 2 [ cross, cross, text "small", greycross, greycross ] ]
```

The code produces the following image (the dashed line has been added to show vertical-reference line of the presentation):



The second element in the row is a column that takes the vertical-reference line from its third child. Therefore, the word “Big” is aligned with the word “small”. The `cross` object

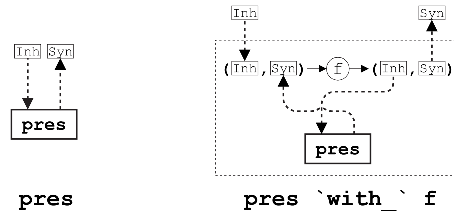


Figure 6.3: Data flow for `with_`.

is a line figure in the form of a rectangle with a cross. The `greycross` is a `cross`, which is made stretchable in both directions by `withStretch`, and which has a grey background.

Because the column contains presentations that stretch vertically, the column itself also stretches vertically. The column is created with a `colR` combinator with argument 2, which causes the third child (`text "small"`) to be the object from which the reference lines of the column are taken. The two stretching objects above the reference object are each assigned equal amounts of the remaining space above the vertical-reference line, and likewise, the objects underneath the reference object are assigned the remaining space below the reference line. If, on the other hand, the reference object itself is allowed to stretch, then the total amount of available space is distributed equally over all stretching objects. In this case, the reference object is not aligned.

6.2.3 Modifying presentation attributes

The presentation attributes of a presentation can be modified using the `with_` combinator. (The name has an underscore because “`with`” is a reserved word in Haskell.)

```
with_ :: Xprez -> ((Inh, Syn) -> (Inh, Syn)) -> Xprez
```

The combinator takes a single child presentation as argument, together with a function from attributes to attributes. The function is applied to the inherited attributes coming from the parent, and the synthesized attributes coming from the child. From the result of this application, the inherited attributes are passed to the child, whereas the synthesized attributes are passed to the parent. Thus, the `with_` combinator can be used to modify the inherited and synthesized attributes of a presentation.

Figure 6.3 shows the data flow for the attributes of `pres` and `pres ‘with_‘ f`. Because `f` may be an arbitrary function, the combinator may introduce cycles in the attribution. It is up to the designer of the presentation to ensure safety.

Because the inherited and synthesized attributes are modeled as Haskell records, we use the Haskell record syntax for accessing and updating presentation attribute values. Hence,

for a record of inherited attributes `inh :: Inh`, the expression `fontSize inh` denotes the value of the `fontSize` attribute in `inh`. Furthermore, `inh { fontSize = 10 }` denotes a copy of `inh` in which the `fontSize` field is updated to 10. Thus, we can define a `withFontSize` combinator:

```
withFontSize :: Xprez -> Int -> Xprez
withFontSize xp fs = xp `with_` \(inh, syn) -> (inh {fontSize = fs}, syn)
```

The function argument to `with_` introduces a considerable syntactic overhead to the presentation code. To reduce this overhead, we can define a library of combinators, such as `withFontSize`, for frequent applications of `with_`. Thus, most of the explicit applications of `with_` may be avoided.

Besides combinators that set an attribute value absolutely, we can also define combinators that take into account the original value of an attribute when setting its value. Consider the combinator `withFontSize_` defined below. Instead of an integer, it takes a function (`ffs :: Int -> Int`) as argument. Given the inherited font size, the function `ffs` specifies its new value.

```
withFontSize_ :: Xprez -> (Int -> Int) -> Xprez
withFontSize_ xp ffs =
  xp `with_` \(inh, syn) -> (inh { fontSize = ffs (fontSize inh) }, syn)
```

With ``pres `withFontSize_` (\fs -> 2*fs)`` we specify that `pres` gets a doubled font size. An application of `withFontSize_` has a function argument, but the function is considerably simpler than the function argument of `with_`.

The font-size combinators show how abstraction is used to meet the *proportional effort* requirement. For simple changes of the font size, the simple `withFontSize` combinator can be used, and only if more control is desired, it is necessary to use the more complicated `withFontSize_` or `with_` combinators.

A future version of XPREZ will support a domain-specific special syntax for `with_`. Thus, in order to specify that a presentation `pres` gets twice the font size of its parent, a red background color, and a height that is twice the height of the letter 'x' in the current font, we will be able to write something in the line of:

```
pres{ child.fontSize = 2*parent.fontSize, child.color = red, height = 1ex }
```

6.2.4 Advanced examples

Because a presentation in XPREZ is a first-class value, it is possible to manipulate a child presentation (e.g. change its position or modify the font size) at the level of its parent. This is illustrated in the presentation for a mathematical fraction:

```
frac e1 e2 = let numerator   = hAlignCenter (pad (shrink e1) )
                 bar         = hLine
                 denominator = hAlignCenter (pad (shrink e2) )
             in  colR 2 [ numerator, vSpace 2, bar
                        , vSpace 2, denominator ] ‘withHStretch‘ False

pad xp = row [ hSpace 2, xp, hSpace 2 ]

shrink e = e ‘withFontSize_‘ (\fs -> (70 ‘percent‘ fs) ‘max‘ 10)
```

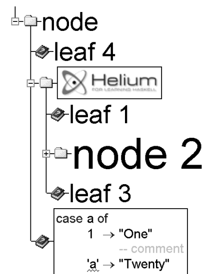
The non-primitive library function `hAlignCenter` centers its argument horizontally, and the `shrink` function reduces the font size to 70%, with a minimum of 10. The result of `(text "x" ‘frac‘ text "2") ‘frac‘ text "1 + y"` is:

$$\frac{\frac{x}{2}}{1 + y}$$

The `pad` and `shrink` functions illustrate the *first-class* and *abstraction* requirements. Because a presentation is a first-class value, the presentations of the numerator and the denominator can be resized and positioned in the presentation of the fraction itself. Furthermore, we can abstract over positioning and resizing by using the functions `pad` and `shrink`.

In contrast, child presentations in both P or PSL cannot be addressed at parent level. Hence, the numerator, the denominator, and even the fraction bar, each have to specify their own size and relative position. As a result, it is difficult to reuse parts of a presentation in another presentation, since all parts refer to each other. Furthermore, the manipulations on the appearance are harder to read, because no abstraction can be used.

The second example is a pair of combinators that can be used to create tree-browser presentations:



The image has been created with the `mkTreeLeaf` and `mkTreeNode` combinators, shown in Figure 6.4. Both combinators take an `Xprez` argument that is the presentation of the

```
mkTreeLeaf :: Bool -> Xprez -> Xprez
mkTreeLeaf isLast label =
  row [ leafHandle isLast, hLine `withWidth` 5, leafImg
      , hLine `withWidth` 5, refHalf label ]

mkTreeNode :: Bool -> Bool -> Xprez -> [Xprez] -> Xprez
mkTreeNode isExp isLast label children =
  rowR 1 [ hSpace 4, nodeHandle isExp isLast, hLine `withWidth` 5
         , col $ [ row [ col [nodeImg , if isExp then vLine else empty]
                       , hLine `withWidth` 5, refHalf label ] ]
                ++ (if isExp then children else [] ) ]

nodeHandle isExp isLast
 = colR 1 ([ vLine, handleImg isExp]++ if isLast then [] else [vLine])

leafHandle isLast = colR 1 ([vLine, empty]++ if isLast then [] else [vLine])

handleImg isExp = if isExp then minusImg else plusImg

nodeImg = img "folder.bmp" `withRef` (7,7)
leafImg = img "help.bmp"   `withRef` (7,6)
plusImg = img "plus.bmp"   `withRef` (4,4)
minusImg = img "minus.bmp" `withRef` (4,4)
```

Figure 6.4: XPREZ tree-browser combinators

label, and the tree node also takes a list of child presentations (which should be either nodes or leaves for a correct tree). A label is not restricted to text, but can be an arbitrary XPREZ presentation, as shown by the case statement at the bottom of the tree. The tree example shows that a complex and graphical presentation can be specified with relatively little effort.

6.3 Conclusions and further research

Current style sheet languages lack either the expressiveness or the abstraction mechanisms to specify complex presentations in a readable way. The declarative presentation language XPREZ, introduced in this chapter, combines a flow and box model with a powerful abstraction mechanism and first-class presentations. The language is well-suited for specifying a wide range of presentations, from tree browsers to WYSIWYG presentations of mathematical formulas, using concise and readable presentation code. An implementation of XPREZ is part of the Proxima prototype.

XPREZ can already describe a large variety of presentations, but the language does not yet meet all the requirements of Section 6.1.1. The language still needs a page model (required for vertical flow layout), support for user-interface widgets, and a domain-specific syntax.

Once XPREZ has a page model and vertical flow layout, it will be possible to support page-related concepts such as footnotes and page references. In order to support such presentations, an abstraction similar to the *galley* of Lout [45] may be added to XPREZ.

In a flow layout, spacing between two presentations should be visible when they end up on the same line or page, but not when there is a line or page break between them. Hence, XPREZ needs a primitive notion of padding and margins for a presentation, which can be left out when appropriate. Moreover, the horizontal and vertical formatting algorithms need support for optimal line- and page-breaking [48]. To support formatting during editing, we could use either a linear algorithm (e.g. [59]), or even an incremental algorithm (e.g. [40]).

Although support for primitive user-interface widgets to XPREZ will not have a large impact on the language, it is closely linked to the Proxima edit model. Hence, more experience with building editors in Proxima is needed in order to establish the right model.

Finally, a domain-specific syntax may be supported using syntax macros [50], for which a Haskell implementation has been developed. Until a domain-specific syntax is available, the syntax required for attribute modification may be reduced by using techniques similar to the property specification method of wxHaskell [51].

In XPREZ expressiveness and efficient evaluation are considered to be more important than safety. An XPREZ value is translated to an attribute grammar, in which the attribution may be influenced directly using the `with_` combinator. This results in an expressive presentation language that can be efficiently evaluated, but also makes it possible to define presentations that crash, or to create cycles in the attribution. A more restricted presentation language may guarantee safety, but will most likely not be able to specify the complex presentations from Chapter 2. Hence, such editors need to be built by hand, which makes it considerably harder to achieve safety.

It will be interesting to see whether XPREZ can be integrated with a constraint solver. Constraints provide an elegant way to specify presentations, and also offer more safety. An entirely constraint-based version of XPREZ, which made use of the Cassowary linear constraint solving algorithm [4], turned out to be too slow to be suitable for editing. However, it may be an option to use a combination of the two formalisms (AG and constraints) in which constraints are used only for certain subpresentations, while attribute grammars are used for the remaining layout.

Besides extensions to the XPREZ formalism, an extensive library of well-chosen combinators must be established to facilitate the specification of complex presentations. And finally, it is desirable to have an algebraic model for XPREZ presentations. With such a model, we can describe the exact behavior of the combinators with laws, rather than textual descriptions.