Chapter 7

The Proxima prototype

A prototype of the Proxima editor has been implemented in the functional language Haskell. Proxima is an editor generator, which means that given a document type definition and number of sheets, the system generates (or *instantiates*) an editor application. The sheets that need to be specified for the prototype are a presentation sheet and a parsing sheet, which are discussed in more detail in Section 7.2.

The architecture of the prototype corresponds to the architecture described in Chapter 3. The presentation target language is the XPREZ language from Chapter 6. The implementation is platform-independent and has been successfully tested on Windows, Linux, and MacOS X platforms.

The prototype is implemented entirely in Haskell and uses the wxHaskell [51] library for the implementation of the user interface. The presentation sheet is compiled by an attribute-grammar system [85], which is also used for the implementation of the arrangement layer. The parsing sheet is specified using a parser-combinator library [83].

In the near future, Proxima will support dynamic updates to the sheets, and hence make it possible to change the presentation of the document during editing. In theory, it is also possible to support dynamic updates to the document type, and thus eliminate the need for a generation step altogether. However, it is not clear yet whether the advantages of a dynamic document type justify the implementation effort and the efficiency penalty.

The prototype does not yet support incrementality. However, because about 90% of the execution time is taken up by the arrangement and rendering layers, and because editing is typically a local process, simple modifications to these two layers already yield substantial improvements. Experiments with such simple modifications have yielded an increase in execution speed of about 900%, which leads to an acceptable response time for documents of a few pages. For larger documents we need the underlying attribute-grammar compiler to support incremental evaluation.

Figure 7.1: An editor for Helium.

In Section 7.1, we show several example editors that have been instantiated with Proxima. Section 7.2 discusses the components that are required for instantiating an editor. In Section 7.3, we discuss the implementation aspects for each of the layers. Section 7.4 presents an overview of future work and concludes.

7.1 Instantiated editors

Three editors have been implemented with Proxima: a source editor for the functional language Helium [34]; an editor for presentation slides in the style of Microsoft PowerPoint; and a chess-board editor. Because the editors were implemented mainly for demonstration purposes, all three editors are integrated in a single editor instantiation.

7.1.1 A Helium source editor

A source editor has been implemented for a subset of the functional language Helium [34], which is a Haskell dialect designed for education. The editor provides most of the functionality described in the source-editor use case (see Section 2.1.1). In order to provide type information during editing, the editor is integrated with the Helium type checker. Figure 7.1 contains a screenshot of the Helium editor in action.

In the figure, we see an editable view of a program source, with the focus on function g in the definition of `s`. The right-hand side of `large` has been hidden and may be expanded by clicking on the dots. The definition of `f` shows that program constructs may have a graphical presentation.

The type signatures for the top-level declarations have been automatically derived, and furthermore, the editor provides type information for local expressions as well. At the top of the window we see the type for the expression focused on (if it has a type). At the bottom, the variables that are in scope at the focus are listed together with their types.

To the right of each type signature is a comment that shows the value of the declared identifier. The value changes dynamically when the source is being edited. Although not very useful in a source editor, since most declarations are functions, these computations provide an example of spreadsheet-like behavior; parts of the document that represent computations are interpreted dynamically, and the results are displayed in the presentation.

Mixed document- and presentation-oriented editing

Expressions can be edited structurally (or document-oriented) based on the Helium abstract syntax. Below is an example that shows how structure editing facilitates editing a list. When the 3*5 element is cut, the comma to the right of it automatically disappears. When the element is pasted at the end, a comma appears at the left.

```
list :: [Int] -- Value: [15,     ⇒  list :: [Int] -- Value: [3,   ⇒  list :: [Int] -- Value: [3,
list = [ 3*5, 1+2, 27 ];             list = [ 1+2, 27 ];              list = [ 1+2, 27, 3*5 ];
```

cut 3*5 *paste* 3*5

The editor also supports structure building with placeholders. By selecting constructs from a menu, an expression may be constructed:

```
list = [ 1+2, 27, 3*5 ];  ⇒  list = [ 1+2, 27, 3*5, {Exp}/{Exp} ];  ⇒  list = [ 1+2, 27, 3*5, {Exp}^{Exp}/{Exp} ];
```

insert FracExp *insert* PowerExp

Similar structure editing is supported by conventional syntax-directed editors, but Proxima has the advantage that the presentation can still be edited textually as well. Program fragments can be entered or modified textually without having to switch to a different view or mode. Below is a screenshot that shows a presentation-oriented cut operation. Although the selection that is cut does not make sense at document level, the cut is a valid edit operation at the presentation level.

```
list = [ 1+2, 27, 3*5, {Exp}^{Exp}/{Exp} ];  ⇒  list = [ 15, {Exp}^{Exp}/{Exp} ];
```

cut "+2, 27, 3*"

Type errors

The editor shows type errors by displaying an error message at the bottom and marking the location with a squiggly line in the source. This mechanism also works in the graphically presented parts of the program, as is shown by the screenshot fragment below.

```
f = λx → x²+2*x+(3+True)*(2+x)*1 / (x+1)² ;

Type error in constructor
 expression        : True
   type            : Bool
   expected type   : Int
```

A type error: 3+True.

The Helium type compiler is an interesting candidate for integration with Proxima because it has a sophisticated type checker. Besides the location of the error, the type checker can provide additional information about the other parts of the program that contribute to the error. Such information would be hard to show on a command-line, but can be displayed in a clear way by highlighting the relevant parts of the source code. Furthermore, for common errors, the Helium type checker is able to provide hints on how to repair them. A hint can be presented along with a button that performs the suggested reparation when clicked.

Beta reduction

A simple reduction engine can be applied to a term in the source. The screen-shots below show two steps in the reduction of the application `f 3`. First, the function `f` is replaced by its definition (see Figure 7.1). Then, a beta-reduction step is performed, and the argument 3 is substituted for all (free) occurrences of `x` in the fraction. The process can be continued by reducing the mathematical operators until we get the final value 16.

```
x :: Int         x :: Int -- Value: 16                            x :: Int -- Value: 16
x = f 3;    ⇒   x = (λx → x²+2*x+(3+x)*(2+x)*1 / (x+1)²) 3;   ⇒   x = (3²+2* 3+(3+ 3)*(2+ 3)*1 / ( 3+1)²);
```

"replace `f` by definition" "reduce lambda"

The reduction engine is implemented by an attribute grammar of about 150 lines, which can be reduced further to about 100 lines once the underlying attribute-grammar system supports default attribute declarations.

Similar to beta-reduction, we could implement other source-to-source transformation, such as refactoring operations [52]. Furthermore, by inserting the transformed term below the original, instead of replacing it, the editor can be used to semi-automatically create derivations, as in a proof editor (e.g. MathSpad [89]).

Editable list of top-level identifiers

The evaluation layer has not been completely implemented yet, but nevertheless a few experimental evaluation-layer features have been implemented. The list of top-level identifiers at the top of Figure 7.1 is similar to an editable table of contents. Editing a name

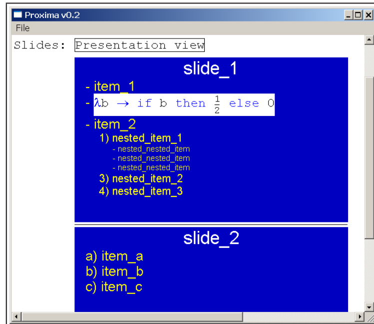


Figure 7.2: A slide editor.

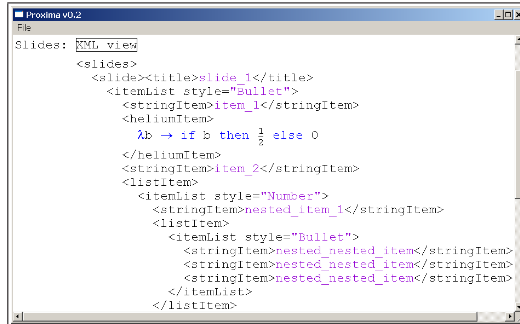
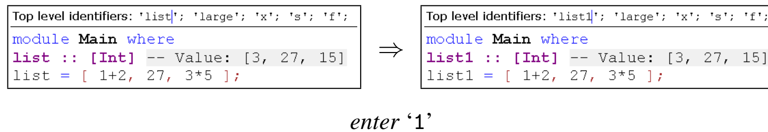


Figure 7.3: Slides as viewed as XML.

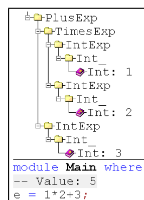
in the list causes an update to the identifier in the corresponding declaration, and moving an identifier moves the declaration. The screenshot shows how editing "`list`" in the identifier list results in an update to the declaration of `list` as well.



enter '1'

Tree view

A second experimental feature is a pre-defined tree presentation of the document. The tree is not fully editable yet.



A tree view of `1*2+3`.

7.1.2 A poor man's PowerPoint

Integrated with the Helium editor is a very basic slide editor in the style of Microsoft's PowerPoint. A slide presentation is a list of slides, each of which consists of a title and

```
module Main where

-- Value: <function>
increase = λx → x+1;

Slides: Presentation view
        slide_1
        - item_1
        - λx → increaze x
        - item_2
          1) nested_item_1
             - nested_nested_item
             - nested_nested_item
             - nested_nested_item
          3) nested_item_2
          4) nested_item_3

Undefined variable "increaze"
  Hint: Did you mean "increase" ?
```

Figure 7.4: Helium slides.

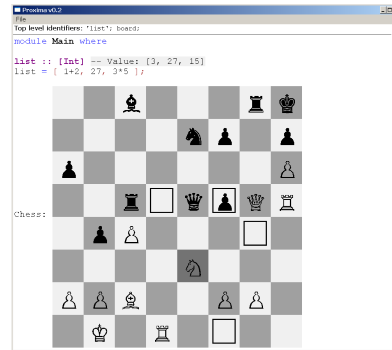


Figure 7.5: A game of chess: Ne3-g4?

a list of items. An item can be either a string, a Helium expression, or a nested item list. Figure 7.2 shows the slide editor for a slide presentation of two slides. An item list may choose from several display styles (bulleted, numbered, or enumerated with letters) and nested lists get a smaller font size. The entire slide editor is specified in about 200 lines of sheet code.

Because a WYSIWYG view is not always convenient, the editor also provides an XML source view, which is shown in Figure 7.3. The source view is only partially editable, but it is straightforward to turn it into a fully editable view.

Integration with Helium editor

The slide editor is fully integrated with the Helium editor: the list of slides is part of the program source, and, more interestingly, a slide may contain Helium code (which may again contain a list of slides, and so on). The Helium code may refer to declarations elsewhere in the source. Moreover, the edit functionality for Helium code in a slide is exactly the same as for code in the source editor. As an example, the screenshot in Figure 7.4 shows a slide with a Helium expression that refers to a non-existent identifier `increaze`.

7.1.3 Chess board

Although it may seem a unfamiliar application of structure editing, a chess board lends itself very well to be implemented with Proxima. Figure 7.5 shows the chess-board editor, which is integrated with the Helium editor similar to the slide editor (except that we cannot use Helium expressions as chess pieces). The editor is connected to a chess-move generator for computing possible moves. In total, not counting the generator, the sheets for the chess board add up to about 140 lines of code.

The chess-board editor highlights all squares that are reachable by the chess piece in focus. A piece may be moved by clicking one of these highlighted squares, or by using cut-and-paste operations. The editor does not yet support playing against the computer, but this can be implemented straightforwardly by connecting the editor to a chess program.

7.2 Instantiating an editor

In order to instantiate an editor in Proxima, three components need to be provided: a *document type definition*, a *presentation sheet*, and a *parsing sheet*. We will give a brief overview and a few example fragments of each of these three components. The *evaluation sheet*, *reduction sheet*, and *scanning sheet* that are mentioned in Section 3.3 are not yet fully supported and therefore not discussed here.

Because the sheet formalisms are still in an experimental stage, the example sheets do not yet contain much abstraction. Hence, for clarity, we will leave out certain details. A future version of Proxima will provide appropriate abstractions for these details.

7.2.1 Document type

As mentioned in Section 3.1.1, a Proxima document type consists of monomorphic data types and the list type. The type definition is similar to a Haskell type definition, but there are a few differences.

A constructor may have named children, which do not have to be globally unique. The name is optional and defaults to the name of the child type (with a number appended in case of more than one anonymous child). Thus, a binary tree type may be defined as:

```
data Tree = Bin left::Tree right::Tree | Leaf Int
```

Furthermore, each constructor needs to specify how many tokens are used for its presentation. This information could be deduced from a presentation sheet, but for the moment it has to be specified by hand. The tokens are specified by a list of identifiers, which is enclosed by braces: `{ident`$_1$ `...` `ident`$_n$`}`. This special syntax is necessary because the editor provides default behavior for these types. Each name gives rise to a child of type `IDP`, which represents the identity of the token. The identities are used when reusing tokens on presentation and parsing.

Figure 7.6 shows several fragments of the document type for the editors from Section 7.1. A document is a list of declarations, each of which can be either a Helium declaration, a chess board, or a slide presentation.

The `Decl` constructor has three tokens: a structural token for the type signature, and two tokens for "`=`" and "`;`" tokens. The other declarations (`BoardDecl` and `SlidesDecl`) only have a single token (for the keywords "`Chess`" and "`Slides`"). In the `Exp` type, we

see one token for a `PlusExp`, which is for the "+" operator, and one token for a `DivExp`, which is a structural token for the entire fraction "$\frac{e_1}{e_2}$". The `LambdaExp` has two tokens ("λ" and "→"), and the `BoolExp` and `IntExp` both have one. The rest of the types have structural presentations and thus contain no tokens.

The `layout` child of `Decl` is a boolean value that specifies whether or not automatic layout it turned on. The value is interpretation extra state, since it is not presented. `Decl` also has a boolean child `folded` specifying whether the right-hand side is visible or folded. Ideally, `folded` would be presentation extra state, but since the prototype does not yet support user-defined presentation extra state, the `folded` state is explicitly specified as part of the document type.

Besides some of the types needed for the Helium editor, the figure also contains the type definitions for the chess board. The board consists of eight rows, each consisting of eight squares. A square contains a chess piece, which is either one of the six kinds of chess pieces, or `Nothing`. (The editor does not yet have a `Maybe` type for optional values.)

For brevity, we do not show the definitions of the rest of the (implemented) Helium language types, nor the types for the slide presentations. However, all of these types are straightforward, and the entire type definition for the three examples together is about 60 lines of code.

7.2.2 Presentation sheet

The presentation sheet is an attribute grammar that specifies the presentation as a synthesized attribute `pres :: Xprez`. Besides the presentation, arbitrary inherited and synthesized attribute may be specified. Furthermore, for each document node, there are a number of predefined attributes, such as its path from the root and a default tree presentation.

As explained in Section 3, a presentation may be either *parsing* or *structural*, depending on whether or not it allows presentation-oriented editing. If a presentation supports presentation-oriented editing, this is specified in the presentation sheet with the combinator `parsing`. On the other hand, if the presentation does not support presentation-oriented editing, we use the combinator `structural`. The choice between parsing or structural presentations also affects the parsing sheet, as will be explained in the next section.

To support editing, the presentation should be constructed according to several guidelines, which are not enforced yet. Besides containing a `parsing` or `structural` combinator, a presentation must encode the document location, and modify the background color when it is in focus.

Although the functions for marking the location and presenting the focus are simple, they contain explicit references to attributes which are not of interest to this discussion. Because the attribute grammar compiler does not support first-class AG's, we cannot abstract over the location and focus functions yet. Hence, we will denote the two functions by *location* and *focus*.

```
data Root = Root decls::[Decl]

data Decl = Decl layout::Bool folded::Bool Ident Exp { idP0, idP1, idP2 }
          | BoardDecl Board                          { idP0 }
          | SlidesDecl Slides                        { idP0 }

data Exp = PlusExp exp1::Exp exp2::Exp               { idP0 }
         | DivExp exp1::Exp exp2::Exp                { idP0 }
         | LamExp param::Ident body::Exp             { idP0, idP1 }
         ...
         | LetExp [Decl] Exp                         { idP0, idP1 }
         | BoolExp Bool                              { idP0 }
         | IntExp Int                                { idP0 }

...

data Board    = Board    r1::BoardRow r2::BoardRow r3::BoardRow r4::BoardRow
                         r5::BoardRow r6::BoardRow r7::BoardRow r8::BoardRow { }

data BoardRow = BoardRow ca::Square cb::Square cc::Square cd::Square
                         ce::Square cf::Square cg::Square ch::Square { }

data Square = Square piece::Piece  { }

data Piece = King color::Bool { } | ... | Pawn color::Bool { } | Nothing { }
```

Figure 7.6: Fragments of Proxima document type definitions.

We discuss a number of examples from the presentation sheets of the editors in Section 7.1.

The presentation of a fraction

The first example is the presentation of a Helium fraction expression, which makes use of the `frac` combinator from Section 6.2.4. Each Helium expression needs to show a squiggly line when it is the location of a type error and, furthermore, it defines a popup menu for beta-reduction edit operations. We denote the two functions that implement this behavior by *squiggle* and *add_reduction_menu*, analogous to *location* and *focus*.

The presentation for the `DivExp` constructor of `Exp` is straightforward. The `SEM` keyword denotes the start of an attribution rule (also called a semantic function). The presentation is a synthesized attribute `pres`, which is denoted by `lhs.pres`. The fraction is presented as a `parsing` presentation consisting of a single structural token.

```
SEM Exp | DivExp exp1::Exp exp2::Exp { idP0 }
  lhs.pres = location . parsing . focus . squiggle . add_reduction_menu $
               tokens [ structToken @idP0 (frac @exp1.pres @exp2.pres)]
```

The presentation of a lambda expression

The second example is the presentation of a Helium lambda expression, such as λx→x+1. In the presentation we make use a function key to display a string in keyword color (i.e. the constant keyColor). The definition of key contains the application stringToken id str, which presents str as a token with identity id.

```
key ::  IDP -> String -> Xprez
key id str = stringToken id str `withColor` keyColor
```

Unlike the fraction, a lambda expression is a parsing presentation. This means that for the presentation of a lambda node, the tokens of its previous presentation must be reused. The presentation consists of two tokens ("λ" and "→"). For these tokens, @idP0 and @idP1 contain the identities of the tokens that were used by the parser to construct the node. In order to reuse the tokens, the identities are passed to key. If the node has not been parsed before, the presentation identities have a special value that causes the generation of a unique identity.

```
SEM Exp | LamExp param::Ident body::Exp { idP0, idP1 }
  lhs.pres = location . parsing . focus . squiggle . add_reduction_menu $
               tokens [ key @idP0 [lambdaSym] `withFont` "Symbol"
                      , @param.pres
                      , key @idP1 [arrowSym] `withFont` "Symbol"
                      , @body.pres ]
```

Note that the lambda and arrow symbols are presented as characters of the "Symbol" font.

The presentation of a chess-board square

A presentation sheet may also specify edit behavior. An example of this is found in the presentation of a square in the chess-board editor. When a square is reachable by the piece in focus, it displays a marker (see Figure 7.5) on top of itself. The marker specifies its own mouse-click behavior: on a mouse click, the piece in focus is moved.

We show only the interesting part of the presentation, which displays the marker and specifies its edit behaviour. This is done by a function pMove, which is applied to the rest of the presentation of the square (denoted by *square_presentation*).

The squares of a chess board have an inherited attribute @lhs.possibleMoves which is a list of possible destinations of the chess piece in focus. The local function pMove first checks whether the location of the presented square (@lhs.colNr, @lhs.rowNr) is in this list. If the square is not reachable then pMove returns pres unchanged. If the square is reachable, pMove returns an overlay with a marker (mrk) in front of pres. Furthermore, the marker associates the edit operation (move @pth @focus) with a mouse click. The move edit operation moves the piece from the square in focus to the presented square.

```
SEM Square | Square piece::Piece
  lhs.pres = location . structural $
              pMove square_presentation
              where pMove pres =
                      if (@lhs.colNr, @lhs.rowNr) `elem` @lhs.possibleMoves
                      then overlay [ mrk `withMouseDown` move @focus @pth
                                   , pres ]
                      else pres
```

Because the chess board has its own focus representation, there is no application of *focus*.

7.2.3 Parsing sheet

A parsing sheet is specified in Haskell, using a parser-combinator library [83]. A parser takes a presentation tree as input.

Because a presentation may consist of parsing and structural parts, which need to be treated differently, the parsing sheet consists of two different kinds of parsers. For a *structural* presentation, we need to specify a *recognizer*, which is a very basic parser that follows the structure of the presentation. On the other hand, for a *parsing* presentation, we specify a regular combinator parser.

If a document node has interpretation extra state, not all of its children are in the presentation. Hence, the parser will not get enough information to construct the node. In that case, we need to reuse the values of the missing children from the previous document. We use the location information from the parsed tokens to determine the document node of which they are the presentation. In case a node of the right type and constructor is found, we take the values of missing children from this node. If the tokens originate from different document nodes, the first node is used.

Because the syntax of reusing may be somewhat confusing without additional explanation, we use a special notation: if we wish to reuse the *i*-th child for a constructor `Constr`, this is denoted by:

reuse (Constr child_1 ... child_i ... child_n)

We briefly discuss the basic notation for the parsers in the examples. The `<*>` combinator composes two parsers sequentially, yielding a parser that succeeds only if both its component parsers succeed. To combine the results of a number of sequentially composed parsers, we use the `<$>` combinator, which takes a function and a parser and applies the function to the result of the parser. If `f <$>` is applied to a sequential composition of *n* parsers, the function `f` gets the results of these parsers as arguments. Thus, adopting the *reuse* syntax mentioned above, we can specify a parser for a constructor Constr $c_1 \ldots c_n$ :: T as:

```
        (\c1 ...  cn -> reuse (Constr c1...cn))
    <$> parser1 <*> parser1 <*> ...  <*> parsern
```

In general, a Proxima parser for type `T` consists of a number of alternative parsers (each for type `T`), which are combined using the choice combinator `<|>`. Often, there is one alternative parser for each constructor of `T`.

In reality, many other parser combinators exist, and the actual structure of the parsers does not have to be exactly as we explained, but the situation above resembles the example parsers. For more information about the parsing library, the reader is referred to Swierstra [83], Hutton [37], or Fokker [27].

The declaration parser

A declaration may be a Haskell declaration, a slide presentation, or a chess board. If it is a Haskell declaration, we distinguish a normal declaration from a collapsed one, which has "`...`" for the presentation of its right-hand side. Thus, we get four alternatives. Furthermore, a Haskell declaration may be preceded by a generated type signature, which is recognized by a function `recognizeTypeDecl`.

The declaration parser combines the four alternative parsers with the choice combinator `<|>`. For a collapsed function, the presentation does not contain a right-hand side `Exp`, which must therefore be reused.

```
parseDecl = (\tkSig idnt tk1 exp tk2 -> reuse (Decl tkSig tk1 tk2 idnt exp))
        <$> recognizeTypeDecl
        <*> parseIdent <*> pKey "=" <*> parseExp <*> pKey ";"
    <|>     (\tkSig idnt tk1 tk2 -> reuse (Decl tkSig tk1 tk2 idnt Exp))
        <$> recognizeTypeDecl
        <*> parseIdent <*> pKey "=" <*> pKey "..."
    <|>     (\tk board -> reuse (BoardDecl tk board))
        <$> pKey "Chess" <* pKey ":" <*> recognizeBoard
    <|>     (\tk slides -> reuse (SlidesDecl tk slides))
        <$> pKey "Slides" <* pKey ":" <*> recognizeSlides
```

The slide recognizer

A recognizer is specified as a parser that is transformed by a combinator `recognize`. The parser part consists of parsers for each constructor of the type, which are combined using `<|>` combinators. Each alternative consists of recognizers (or parsers) for the children of the constructor, and is preceded by a special combinator that recognizes the presentation of a specific constructor. For any constructor `Constr`, this special combinator is denoted by `pStructural ConstrNode`, (with `ConstrNode` being a generated constructor).

A slide contains a title string and a list of items. The title is parsed with the primitive `parseString`, whereas the item list is recognized by `recognizeItemList`.

```
recognizeSlide = recognize $
        (\str title itemList -> reuse (Slide title itemList))
    <$> pStructural SlideNode <*> parseString <*> recognizeItemList
```

Because recognizers exactly follow the structure of the presentation, a future version of Proxima will most likely have support for automatically generating them from the presentation sheet.

The chess-board recognizer

If all descendants of a node have a structural presentation, and thus may not be edited at the presentation-level, the recognizer for the node is simple. The presentation of a node will not have been modified, and instead of descending into the presentation structure, we may simply reuse all children.

Hence, the recognizer for the chess board is:

```
recognizeBoard = recognize $
        (\str -> reuse (Board BoardRow BoardRow ...  BoardRow)
   <$> pStructural BoardNode
```

7.3 Prototype implementation

The main components of the architecture are the five layers, together with a user-interface module. Each layer has a presentation and an interpretation component, which define two functions `present` and `interpret`. A special architecture module imports all component modules, and connects the `present` and `interpret` functions, thus hiding the data-flow patterns from the layer component modules. In total, the generic part of Proxima consists of about 15,000 lines of Haskell code.

7.3.1 Genericity

Internally, the document type is represented by a Haskell type. Because Haskell is not a generic language, this means that after changing the document type, the editor needs to be recompiled. It would also be possible to represent a document by an untyped tree structure, but we choose a typed implementation because it provides type-safety for the presentation and computation sheets, and also allows a more efficient implementation. Although it is an interesting feature to be able to dynamically change the document type, we do not consider this a main requirement for the editor.

All type-specific code is currently generated by a generator written in Haskell. Although the specification of generated code lacks transparency, this method does provide the flexibility that we need in this developmental stage of the Proxima project.

An alternative to the Haskell generator is the language Generic Haskell [53]. However, not all required functions and data types can be described elegantly in Generic Haskell yet. Furthermore, because we also need to generate AG code, switching to Generic Haskell will not eliminate the generator, until we also have a generic AG compiler.

7.3.2 User interface

The user interface of Proxima has been implemented with the wxHaskell library [51], which is an elegant and fast GUI library providing enough low-level support to implement the Proxima renderer. The library is based on the wxWidgets library, and parts of it are generated from a wxWidgets binding to the language Eiffel. As a result, keeping the wxHaskell library up to date with the latest developments of wxWidgets, requires relatively little effort.

Most Haskell GUI libraries are not suitable for Proxima because they either lack the required functionality or are no longer being maintained. There are several suitable libraries besides wxHaskell, but these are based either on the GTK library, which is still poorly supported on windows platforms, or on Tcl/Tk, which is portable but slow.

In Proxima, the dependency on the GUI library is limited to only four modules: the renderer module; a module for the type definitions of the renderer; a module for doing font queries; and a module that opens the main editor window and maps GUI events to Proxima edit gestures. Thus, the system can easily be ported to a different GUI library. In fact, the wxHaskell port was made only recently, after most of the prototype had already been implemented.

7.4 Future work and conclusions

Although still in a preliminary stage, the prototype already makes it possible to instantiate a relatively advanced editor with relatively little effort. Both the slide and the chess-board editors were implemented in only a few days. Nevertheless, the prototype was mainly implemented as a proof-of-concept, and hence requires further development in order to become a generally usable product.

In the remainder of this section we first discuss our experiences with Haskell as the implementation language, followed by an overview of the future development of Proxima. We make a distinction between straightforward versus more research-oriented issues.

7.4.1 Haskell

Haskell may not immediately seem the most logical candidate for implementing Proxima, because of statefulness of the architecture. Every layer has its own state, and moreover, different levels may point to each other. However, because the complex data-flow patterns

are confined to a single architecture module, each layer component only has to deal with its surrounding levels. The Proxima implementation consist of numerous algorithms over tree structures, which benefit greatly from Haskell's syntax for abstract data types and pattern-matching.

On the other hand, the typical Haskell feature of lazy evaluation has not been of significant importance (other than allowing for elegant programming), because of the overhead associated with lazy data structures. The code that is generated by the attribute grammar compiler depends on laziness, but this is also something that will most likely need to change in the future. Instead, the attribute grammar could be analyzed and partitioned into strict computations.

For the development of Proxima, probably the most important feature of Haskell is the way in which combinator languages can be defined and mixed with Haskell code. The combination of Haskell with XPREZ and parser combinators is especially useful for the experimental stage of the prototype. Standard patterns can be expressed elegantly using combinators, whereas experimental features can be coded explicitly. If these features turn out useful, they can be captured by an appropriate combinator. Thus, the behavior specified in the style sheets is highly customizable, while the code in the sheets remains concise and transparent.

7.4.2 Basic extensions to the prototype

The prototype is in an experimental stage, which means that there is an abundance of straightforward extensions to the system. Nevertheless, even though straightforward, the implementation of these extensions may still require a substantial amount of work.

Besides standard editor functionality (e.g. file handling, search facility, etc.) and basic updates to the system, we can identify several important issues that are local to a layer. We briefly discuss each layer separately. The evaluation layer is omitted from the discussion, because most of the future work on this layer is research-oriented.

Presentation

When the Proxima parser encounters an error, the entire parsed presentation is marked with a parse error. This behavior does not meet the modeless editing requirement from Chapter 2, since structure editing inside the region with the parse error is not possible until the error is corrected. Hence, the parser needs to be able to keep the error local and continue parsing the rest of the presentation. For parsed presentations that appear in a structural presentation, the parse error is already kept local. For example, a parse error in a Helium item of a slide only affects that item. Because the parser library that is used has support for error correction, local parse errors will most likely be relatively easy to support.

An extension of lower priority, but nonetheless straightforward, is adapting the presentation layer to support dynamically loaded presentation and parsing sheets. Because the

presentation and parsing modules are already clearly separated from the other layers, dynamic loading will not require any fundamental changes to the architecture.

Finally, many extensions to the Xprez formalism are desirable. Examples include support for windows, widgets, vertical formatters, and a page model. We mention these aspects here at the presentation layer because of their impact on the presentation level. Nevertheless, the implementation for these features will take place mainly at the arrangement and rendering layers, because these layers take care of computing the locations and sizes of XPREZ elements (arranging), as well as mapping them onto appropriate GUI commands (rendering).

Layout

The scanner component of the layout layer is a function that traverses the layout tree and tokenizes those parts of the tree that are marked for parsing. Because the specification of the tokens is hard-coded in the scanner definition, it is not straightforward to specify a scanner for a language that has different tokens. Instead, we need a parameterizable scanner, which takes its token specifications from a scanning sheet.

Arrangement

The arrangement layer needs a few updates to support editable formatters. Furthermore, because this layer performs the size and position computations for the presentation, extensions to XPREZ are implemented for a large part at the arrangement layer.

Rendering

The renderer will have to provide rendering support for the extensions to XPREZ.

7.4.3 Future research

Besides the straightforward extensions to the system, there is a multitude of possible areas for future research on Proxima. We mention a few of the important areas.

Incrementality. Probably the most important next step in the development of Proxima is support for incrementality, which has consequences for all layers. For the layout, arrangement and rendering layers, the presentation and interpretation mappings are mainly pre-defined, and hence these layers could provide built-in support for incrementality. Nevertheless, for handling larger documents, we also need incrementality on the evaluation and presentation layers. This will require extensions to the attribute-grammar compiler.

Other issues related to incrementality are the required support for change management on each of the data levels, as well as a mechanism for presenting and interpreting only the necessary parts of each level (e.g. only arrange the visible part of a presentation).

Evaluator layer. An evaluation layer must be implemented, together with language support for specifying the enriched document type. Because the enriched document is often similar to the document, it is a hassle to specify it from scratch. On the other hand, using the same type for both levels compromises safety. Instead, we need a formalism for specifying only those parts for which the enriched document type differs from the document type. The enriched document type definition can be generated from that specification.

Once this functionality is available, we can establish the formalisms for the evaluation and reduction sheets. A desirable aspect of these sheet formalisms would be the automatic specification of a reduction sheet, given an evaluation sheet.

AG presentation patterns. The presentation sheet contains many common patterns, over some of which the attribute-grammar formalism cannot abstract elegantly yet. Identifying these patterns and developing extensions to the formalism will help to make the presentation sheets more concise and transparent. A possible candidate for such an extension is support for first-class attribute grammars.

Extra state. More research is needed to identify the different forms of extra state, as well as language support for easily specifying extra state.

Focus model. Proxima has a concept of focus on the layout level as well as on the document level. Furthermore, once the evaluation layer is implemented, there will most likely also be a focus on the enriched document level. An integrated focus model must be developed for smoothly handling the translation of one kind of focus into another during editing.

Transformation formalism. Edit commands are still specified with basic cut-and-paste operations. We need a transformation formalism to easily specify type-safe transformations.

Graph support. Although a Proxima document is a tree, we could use cross-references between tree nodes to encode graph data. It would be interesting if this data could also be presented as a graph. Simple extensions to XPREZ will make it possible to create a graph presentation. For editing a graph, the arrangement layer needs to provide edit operations such as moving nodes and inserting and deleting edges. The result of these edit operations needs to be interpreted as a document update.

A first priority is the instantiation of more example editors. Especially a word-processing editor will be interesting, because this kind of application has not yet been investigated extensively with Proxima. The example editors will suggest more areas of future research, and allow us assign priorities to each area as well. Furthermore, by examining common patterns in the sheets of the example editors, we can determine the useful abstractions and libraries for these sheets.

Finally, creating editor instances for editing document type definitions, presentation sheets and parsing sheets is not only an interesting exercise by itself, but will also make editor instantiation easier.