

Proxima

A presentation-oriented editor for structured documents

Proxima
Een presentatiegerichte editor
voor gestructureerde documenten

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. W. H. Gispen,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 15 oktober 2004 des middags te 12.45 uur

door

Martijn Michiel Schrage

geboren op 4 juni 1973, te Veendam

promotores: Prof. Dr. S. D. Swierstra
Prof. Dr. J. Th. Jeuring
Prof. L. G. L. T. Meertens

Instituut voor Informatica en Informatiekunde, Universiteit Utrecht



The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 90-393-3803-5

Copyright © Martijn Schrage, 2004

Printed by Febodruk BV, Enschede

voor Joke

Contents

1	Introduction	1
1.1	Preliminaries	2
1.1.1	Structured documents	2
1.1.2	XML	3
1.1.3	Editing	3
1.1.4	Advantages of generic structure editors	6
1.1.5	Classes of structure editors	7
1.2	Proxima	10
1.3	Terminology	11
1.4	Outline of the thesis	12
2	Requirements for a structure editor	15
2.1	Use cases	15
2.1.1	A source editor for Haskell	16
2.1.2	A word processor	19
2.1.3	Equation editor/MathML	23
2.1.4	Non-primitive outline view/tree browser	25
2.1.5	Simple tax form/spreadsheet	27
2.2	Functional requirements	30
2.2.1	Genericity	30
2.2.2	Computation formalism	30
2.2.3	Presentation formalism	31

2.2.4	Editing power	32
2.2.5	Modeless editing	32
2.2.6	Extra state	33
2.2.7	Summary	34
2.3	Overview of structure editors	34
2.3.1	Syntax-directed editors	35
2.3.2	Syntax-recognizing editors	37
2.3.3	Editor toolkits	38
2.3.4	XML editors	39
2.4	Discussion	41
2.5	The Proxima editor	44
2.6	Conclusions	44
3	Architecture of the Proxima editor	47
3.1	The levels of Proxima	49
3.1.1	Document	50
3.1.2	Enriched document	50
3.1.3	Presentation	51
3.1.4	Layout	53
3.1.5	Arrangement	53
3.1.6	Rendering	54
3.2	Editing on different levels in Proxima	54
3.3	The layers of Proxima	56
3.4	Presentation process	59
3.4.1	Evaluation layer: Evaluator	60
3.4.2	Presentation layer: Presenter	60
3.4.3	Layout layer: Layouter	62
3.4.4	Arrangement layer: Arranger	62
3.4.5	Rendering layer: Renderer	63
3.5	Interpretation process	64

3.5.1	Rendering layer: Gesture interpreter	65
3.5.2	Arrangement layer: Unarranger	65
3.5.3	Layout layer: Scanner	66
3.5.4	Presentation layer: Parser	67
3.5.5	Evaluation layer: Reducer	69
3.6	The choice of layers in Proxima	71
3.7	Conclusions	72
4	Prelude to the specification	75
4.1	The editing process	75
4.2	Extra state	77
4.2.1	The presentation mapping	78
4.2.2	Extra-state nodes	80
4.2.3	Each intermediate level has two kinds of extra state	81
4.2.4	Reusing extra state	82
4.2.5	Safety of extra state	83
4.2.6	Conclusions	84
4.3	Duplicates in the presentation	84
4.4	Conclusions	86
5	Specifying a layered editor	87
5.1	A single layer	88
5.1.1	Editing	89
5.2	Extra state	93
5.2.1	Equivalence classes	94
5.2.2	An equivalence class for extra state	94
5.2.3	Presenting and interpreting	95
5.2.4	Editing	96
5.3	A wildcard representation for equivalence classes	101
5.3.1	Trees with wildcards	101
5.3.2	T^* induces an equivalence relation on T	103

5.3.3	Reuse on wildcard types	103
5.3.4	Reuse on equivalence classes	105
5.3.5	Improving reuse	110
5.4	A composite layer	111
5.4.1	Composite present _C and interpret _C	111
5.4.2	The INTERPRESENT requirement	115
5.4.3	An inductive definition of present _C and interpret _C	115
5.4.4	Editing	117
5.4.5	POSTCONDITION requirement	119
5.4.6	DOC-INERT requirement	120
5.4.7	PRES-INERT requirement	123
5.4.8	DOC-PRESERVE and INTENDED requirements	128
5.4.9	Conclusions	129
5.5	Duplicate presentations	129
5.5.1	Dealing with duplicates	130
5.5.2	Adapting the INTENDED requirement	130
5.5.3	Adapting the PRES-INERT requirement	132
5.5.4	The remaining requirements	133
5.5.5	Conclusions	134
5.6	Conclusions and related work	135
6	Presenting structured documents with XPREZ	137
6.1	Presentation languages	138
6.1.1	Existing presentation languages	139
6.2	The XPREZ target language	142
6.2.1	XPREZ presentation model	142
6.2.2	XPREZ primitives	143
6.2.3	Modifying presentation attributes	145
6.2.4	Advanced examples	146
6.3	Conclusions and further research	148

7 The Proxima prototype	151
7.1 Instantiated editors	152
7.1.1 A Helium source editor	152
7.1.2 A poor man's PowerPoint	155
7.1.3 Chess board	156
7.2 Instantiating an editor	157
7.2.1 Document type	157
7.2.2 Presentation sheet	158
7.2.3 Parsing sheet	161
7.3 Prototype implementation	163
7.3.1 Genericity	163
7.3.2 User interface	164
7.4 Future work and conclusions	164
7.4.1 Haskell	164
7.4.2 Basic extensions to the prototype	165
7.4.3 Future research	166
8 Conclusions and further research	169
8.1 Further research	170
8.2 Final remarks	171
Bibliography	173
Samenvatting	181
Dankwoord	185
Curriculum vitae	187
Titles in the IPA dissertation series	189

Introduction

Many software applications involve some form of editing: a user views a data structure and provides edit gestures in order to modify this data structure. Different kinds of documents require different ways of editing, and hence a multitude of editors exists, each having its own specific edit model and user-interface conventions. Moreover, since application designers have different ideas on what constitutes a pleasant edit model, even editors for the same document type may show significantly different edit behavior. Nevertheless, the core edit behavior, whether performed in a word-processor or a spreadsheet, is largely similar: document fragments may be copied and pasted, and new parts of the document may be constructed by selecting from menus or entering text.

An obvious research question is to abstract from the specific aspects of each editor and construct a generic system that can be instantiated to a specific editor application. Building an editor with such a system would require only a fraction of the amount of engineering required to build an editor from scratch. A generic editor enhances consistency between editors, because all instantiated editors share the same edit model, and, furthermore, it facilitates the integration of editors for different document types.

Especially in the nineteen-eighties, many research projects on structure editing were started. However, the editors developed were generally perceived as being overly restrictive, and attempts at developing less restrictive systems resulted mainly in text-only editors. Further, regardless of the restrictiveness of the edit model, the applicability of the generic editors was generally limited to source editors for programming languages and simple word-processing applications. In the years following, research interest in structure editing steadily declined, and many of the generic editors that were developed are now used only for educational purposes at the institute of origin.

In our opinion, the problem with most of these structure editors is that they either focus on editing the document structure, or the presentation (often just text). The document-

oriented editors may have a powerful presentation mechanism, but poor editing support in the presentation, which results in a restrictive edit model. On the other hand, purely presentation-oriented editors lack edit operations on the document, and have relatively weak presentation mechanisms.

With the increasing popularity of the XML format for representing structured documents, the advantages of a powerful generic editor are becoming even more apparent. Many XML document types are being developed, but support for editing documents of these types is still poor. There is a choice between using an expensive custom-made editor, or a generic XML editor, but the functionality of the latter does not come close to what a presentation-oriented (WYSIWYG) editor could potentially offer. It is, for instance, not possible to use any of the current XML editors as a convenient editor for a programming language or for mathematical equations.

In this thesis we investigate whether and how the advantages of structure editing and a powerful presentation formalism can be combined with a non-restrictive presentation-oriented edit model. The result of this research is the presentation-oriented structure editor Proxima. Before we introduce Proxima, we discuss the basic concepts that play a role in editing structured documents. In Section 1.2 we introduce the Proxima editor, followed by a summary of the introduced terminology in Section 1.3 and an overview of the thesis in Section 1.4.

1.1 Preliminaries

1.1.1 Structured documents

A *structured document* is a collection of logical entities between which a structural relation exists. Examples of structured documents are HTML pages, program sources, word processor documents, etc.

In this thesis, we restrict ourselves to structured documents that have a tree structure that can be described by an EBNF grammar. Although graphs can be viewed as structured documents as well, algorithms for performing computations over graphs are far more complex than tree algorithms. Furthermore, parsers for graphs are less well understood than parsers for trees, as well as computationally more expensive.

In cases where we explicitly want to describe the structure of a document fragment, we use monomorphic (i.e. parameter free) Haskell [70] data types together with the list type. Example document fragments are represented by Haskell terms. For example, a document representing the let expression “let $x = 1$; $y = 2$ in $x + y$ ” can be denoted in Haskell by:

```
Let [Decl (Ident “x”) (Int 1), Decl (Ident “y”) (Int 2)] (Sum (Ident “x”) (Ident “y”))
```

1.1.2 XML

The eXtensible Markup Language XML [16] is an increasingly popular standard for representing structured documents. The standard is a simplified descendant of SGML [38] (Standard Generalized Markup Language). An XML document is a sequence of characters that encodes a tree structure. The nodes of the tree are referred to as *elements*. The leaves of the tree are text or attributes (name-value pairs describing properties of an element). The structure of the tree is represented with opening and closing *tags*, and if these tags are nested correctly, the XML document is *well-formed*.

The let expression example of the previous section can be represented in XML by:

```
<Let><Decl><Ident>x</Ident><Int>1</Int></Decl>
  <Decl><Ident>y</Ident><Int>2</Int></Decl>
  <Sum><Ident>x</Ident><Ident>y</Ident></Sum></Let>
```

The type of an XML document can be specified in several formalisms. The *Document Type Definition* (DTD) is part of the XML specification, and basically describes an EBNF grammar over XML elements. A much more expressive formalism is XML Schema [87], which itself is a sublanguage of XML. Compared to the DTD formalism, the advantages of using a Schema definition include more control over textual content, as well as a form of inheritance. If an XML document conforms to a certain DTD or Schema, it is called *valid*. A third standard, which is not as common as DTDs or Schemas, is the Relax NG standard [23]. Relax NG is a combination of Relax [55] and TREX [22], and is based on regular expressions.

The number of standards for sublanguages of XML, also referred to as dialects, is rapidly growing. Besides the already mentioned XML Schema, we provide a few more examples.

The Mathematical Markup Language MathML [20] is a standard for describing mathematical equations and expressions. Technical documentation can be represented with the DocBook [92] standard, which exists for XML as well as for SGML. The standard can also be used for papers and books. Finally, we mention the XHTML [2] standard, which is an XML encoding of HTML. Although similar, an HTML document is not necessarily an XML document, since HTML is a dialect of SGML rather than XML.

1.1.3 Editing

While the term *editor* is usually only associated with plain-text editors such as Emacs [81] or the ubiquitous Microsoft Notepad, we will use the term in a much broader sense. We regard as an editor any application that presents a visual representation of an internal data structure to a user and allows the user to modify this structure. The internal data structure is referred to as the *document* and the visual representation is the *presentation*.

Obviously, word processors, image editors, and text editors are editors in this view, but there are also some less obvious examples. Take, for example, the preferences pane that is

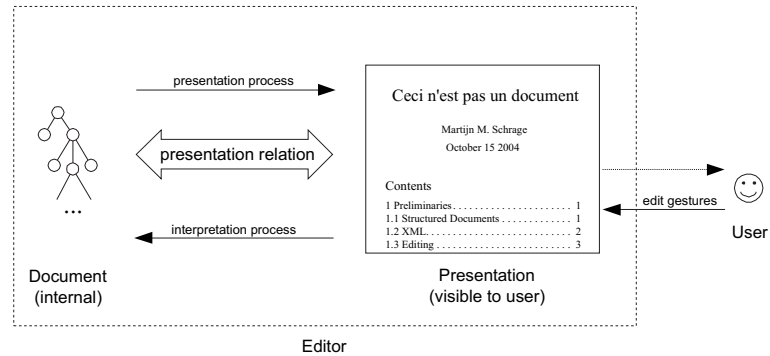


Figure 1.1: Schematic representation of an editor.

part of most window-based applications. The check boxes, selection lists, and text fields can be seen as a presentation of the preferences of the application. Another example of an application that is not usually regarded as an editor is a file browser. (For a description of a file browser as a text editor, see for example Fraser [28].)

Figure 1.1 contains a schematic representation of an editor. The main data structures in the editor (also referred to as *levels*) are the internal *document* on the left that is not visible to the user and the user-visible *presentation* on the right. The document should not be confused with a file, which is a representation of the document that is stored on a file system. Furthermore, we also do not consider an XML source to be a document, but rather a textual presentation of the internal document.

A *presentation*, or *view*, is the only thing a user sees of the document. A presentation may be textual, graphical, or a combination of both. We focus on static presentations only. Hence, we do not explicitly consider presentations containing sounds or animations, unless presented statically (e.g. as a textual link to a sound or video file). In the presentation, the editor shows the focus of attention, or, for brevity, just *focus*, which is a shared name for the selection as well the cursor (which is an empty selection). Several presentations of a single document may be shown simultaneously by the editor, each having its own focus. Finally, if a presentation closely mirrors the final physical appearance of the document when it is printed, it is referred to as a WYSIWYG presentation (What You See Is What You Get).

The relation between a document and its presentation is denoted by the term *presentation relation*, or *presentation mapping*. If, according to the presentation relation, the presentation shown to the user is a presentation of the document, we say that the *presentation invariant* holds. Computing the presentation of a document is called the *presentation process*, whereas computing a document from a presentation is called the *interpretation process*. Together, the two processes implement the presentation relation and maintain the presentation invariant if either side of the relation changes.

A presentation is *valid* if there exists a document, which, when presented, yields that presentation. A presentation for which there is no corresponding document is invalid. An invalid presentation may result from an editing the presentation level. Note the difference with the term valid document, which denotes a document that is well typed.

The presentation relation for an editor may be (partially) specified in a style sheet, or *presentation sheet*. A presentation sheet describes how elements of the document type are to be presented, and is a parameter of the presentation process. By modifying the sheet, a user may influence the appearance of the document without having to modify the editor itself. A presentation sheet can be regarded as a parameter to the interpretation process as well, since the interpretation depends on the presentation specified in the sheet. Examples of style-sheet formalisms are the Cascading Style Sheets (CSS) [11] for HTML as well as XML, and the Extensible Stylesheet Language (XSL) [1] for XML.

Generally speaking, editing consists of repeated interactive cycles of presenting and interpreting. The editor shows a presentation of the document together with the current focus to the user. The user then provides the editor with an *edit gesture*, such as a key press or a mouse movement, which is interpreted as an update on the document. The document is then re-presented and shown to the user. The process is repeated until the user quits the editor. Chapters 4 and 5 provide a more formal description of the editing process.

Document-oriented versus presentation-oriented editing

Because edit gestures may be targeted either at the document or the presentation, we distinguish two kinds of editing: *document-oriented* versus *presentation-oriented* editing.

On the one hand, we have *document-oriented editing*, which consists of edit operations (including navigation and selection) that are targeted at the structure of the document rather than at its presentation. Examples are swapping two chapters in a word processor, selecting an entire chapter, or navigating to a next section.

On the other hand, *presentation-oriented* editing consists of edit operations on the presentation, which do not necessarily make sense at the document level. If a presentation is textual, presentation-oriented editing amounts to freely editing the text. As an example, take the mathematical expression $(1 + 2) \times (3 + 4)$. Deleting the middle part $(1 + \overline{2}) \times (\overline{3} + 4)$ yields $(1 + 3 + 4)$ and is a presentation-oriented edit operation that does not directly correspond to a logical operation on the document level. Another example is navigating downwards in a formatted paragraph of a word processor, since the concept of lines in a paragraph only exists at the presentation level.

Section 3.5 provides a more thorough discussion of both document- and presentation-oriented editing. Furthermore, the section discusses editing at several other levels, which are introduced at the beginning of Chapter 3.

Different kinds of editors

The term *structure editor* is used to make explicit that an editor has document-oriented editing functionality (also including navigation). We do not make a sharp distinction

between plain-text editing and structure editing. Instead, we regard all editing as structure editing, but with a varying level of structure. A text editor can be seen as a structure editor with a very simple structure model: a string or a list of strings. Document-oriented and presentation-oriented editing coincide for a text editor.

An editor is a *generic editor* if it is not specifically built for a single document type but can be used to edit a whole class of document types. A generic editor may be *instantiated* to yield an editor for a specific document type. Genericity can be achieved with a single generic editor that edits documents of arbitrary types, but also with an editor generator. An *editor generator* is an environment that generates an editor application based on descriptions of the document type and its presentation. Although a generator is not as versatile as a single generic editor, we view both as generic editors.

For brevity, we will often adopt the common practice that the term structure editor implies genericity as well. Still, structure editors that are not generic are quite common. A few examples are: equation editors, bookmark editors in web browsers, and file browsers. On the other hand, a generic editor is always a structure editor since it knows about the type of the document.

In the context of generic editing, the term *user* is ambiguous. A user can either be an editor designer, who instantiates the generic editor for a specific domain, or a user who is editing a document. Unless explicitly stated otherwise, we use the term for the document-editing user.

Because it is difficult to give a precise definition of a generic structure editor and because such a definition might be restrictive, we will discuss a number of typical use cases to clarify what we mean by a generic structure editor. Section 2.1 presents these use cases.

1.1.4 Advantages of generic structure editors

An editor that knows about the structure of the edited document can offer interesting functionality. We list several potential advantages of generic structure editors. The first two advantages stem from the genericity of the editor, whereas the rest are mainly about the structural (document-oriented) abilities.

Uniform user interface/edit model. Rather than a separate editor application for each type of document, a single generic editor can be used for a range of document types. Thus, instead of having to cope with several slightly different interfaces, a user only needs to deal with a single uniform interface and edit model.

Integration of documents. Besides offering editors for different types of documents, a structure editor also facilitates the integration of different types of documents into a single editor instantiation. Thus, it is relatively easy to build an editor for a specific document type, with advanced functionality for the different kinds of edit. Examples are a word-processing editor with spreadsheet functionality, or an editor for slide shows that has syntax coloring and type checking for program code appearing in the slides.

Different Views on the Document. A structure editor may provide a user with several editable views on the document. The views can show the document in a different order, or with a varying amount of detail.

Graphical Views. A view may contain color and fonts in order to clarify document structure, but also use layout alignment, and graphical elements such as lines and boxes.

Derived Information in the Presentation. The editor can analyze the document during editing and display information computed from the document structure. Examples are the results of static analysis and type checking in source editors, but also chapter numbers or an automatically generated table of contents.

Structural Edit Operations. Certain edit operations, such as demoting a section with subsections to a subsection with subsections in a scientific article, are straightforward to specify at the structural level, but awkward at the presentation level.

Structural Navigation. Navigating over the document structure instead of its presentation can be very useful. In a source editor, when the focus is on an identifier, a user may easily navigate to its definition in the source. Furthermore, an outline view of the document can be shown in which a user can click to navigate to the corresponding position in the document.

Integration with Other Tools. A structure editor allows fine control over integration with other tools, such as spell checkers, program-transformation systems, and theorem provers. Furthermore, the editor may show the results coming from these tools at the appropriate position in the presentation, rather than as a list of messages with line numbers.

For document types with a textual presentation, such as program sources or XML documents, some of the advantages can be simulated with a text editor. Lexical analysis can be used on the edited text, and basic support for syntax coloring, auto-completion, and navigation can be provided. However, although simple and efficient, these solutions are very basic and prone to errors, because, in general, much of the structure of a document cannot be recognized at a purely lexical level.

1.1.5 Classes of structure editors

Three classes of structure editors are distinguished in the literature: *syntax-directed*, *syntax-recognizing*, and *hybrid* editors. Syntax-directed editors mainly support edit operations targeted at the document structure, whereas syntax-recognizing editors support edit operations on the presentation of the document. A hybrid editor combines syntax-directed with syntax-recognizing features, but the term is not used consistently.

Syntax-directed editors

The first structure editors that were developed are the *syntax-directed*, or *pure*, structure editors [6, 54, 77].

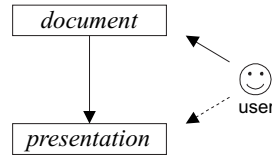


Figure 1.2: A syntax-directed editor.

Early syntax-directed editors show a textual presentation of the document (usually a program source) but exclusively offer edit operations targeted at the internal document structure, and not at the textual presentation. The original idea behind this was that if structural edit operations are available, a user would not need the textual edit operations anymore. Further, presentation-oriented edit operations would interfere with the user's structural model of the document and introduce errors. Hence they were prohibited altogether. Most editors for XML (see also Section 2.3.4), as well as editors for preferences panes, can be regarded as syntax-directed editors.

Figure 1.2 shows a schematic representation of a syntax-directed editor. The editor works by computing a presentation of the internal document structure, which is shown to the user together with a current focus of attention. The user provides an intended edit operation (edit gesture) on the document structure, from which a document update is computed. After the document is updated, a new presentation is computed, which is shown to the user.

If the editor supports clicking in the presentation to set the focus, the editor also needs to keep track of the origin in the document for each position in the presentation.

In the figure, the line between the user and the presentation is dotted because syntax-directed editors do not support edit operations on the presentation very well. Because the presentation is derived from the document, the editor needs to interpret the intended edit operation on the presentation as an edit operation on the document, which is difficult if the edit operation is not a logical operation on the document level.

A major problem with syntax-directed editors is the restrictiveness of the edit model (e.g. [62, 88]). New structures are easy to create, but not as easy to modify. For example, if a user wishes to change a while statement to an if statement, simply typing over the keyword is typically not supported.

Many later syntax-directed editors offer a form of presentation-oriented editing by providing a freely editable textual presentation of (part of) the document, and applying a parser to the edited text. Some publications [58, 86] refer to such editors as hybrid, but, as we will explain below, we still regard these editors as syntax-directed editors.

Unless the two forms of editing are completely integrated, the textual presentation forces a user to work in a different mode of editing, which is referred to as *mode switching*. Mode switching does not solve the problem of restrictiveness adequately. Often, a separate

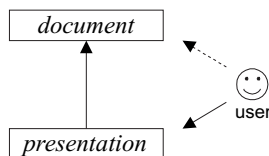


Figure 1.3: A syntax-recognizing editor.

window showing a text-only presentation is opened and before the mode can be switched back, the edited text has to be valid. Furthermore, separate modes require a user to be constantly aware of the current mode of the editor. The resulting increased cognitive burden has been shown to be a source of errors [79].

Syntax-recognizing editors

At the other end of the spectrum are the *syntax-recognizing* structure editors [7, 18]. A syntax-recognizing editor keeps track of the textual presentation of the document. The user can freely edit the text, and the editor tries to recognize the document structure by means of a parser. Once the text has been (partially) recognized, structural information (e.g. syntax-coloring or type information), navigation, and, in some editors, edit operations are available.

Figure 1.3 schematically shows a syntax-recognizing editor. The user's edit operations are targeted at the presentation, which can be edited freely. The document is derived by parsing (interpreting) the presentation; hence the reversed direction of the arrow, compared to Figure 1.2.

For each element in the document structure, the editor needs to keep track of what parts of the presentation it corresponds to, in order to show structural information in the presentation, as well as support structural navigation. When a document structure has been recognized, the presentation may show additional information using font and color changes, context-sensitive menus, tooltips, etc.

Similar to the syntax-directed editor, the picture of the syntax-recognizing editor in Figure 1.3 also contains a dotted arrow. In this case, because the document is derived from the presentation, structural edit operations on the document are difficult to support. A document-oriented edit operation has to be mapped onto an update on the presentation, in such a way that parsing the updated presentation returns the intended updated document. Presentation information that is not stored in the document tree, such as whitespace and comments, has to be related to the document tree in some way, in order to be put in the right place after a structural edit operation.

The main problem with syntax-recognizing editors lies in their limited applicability. Because the presentation needs to contain enough information to derive the document, interesting presentations that only show part of the document are hard to support. Furthermore,

graphical presentations, as well as presentations containing computed values and structures, do not fit the model, as these are difficult to parse. As a result, syntax-recognizing editors are mainly limited to text-oriented applications, such as program-source editors.

Hybrid editors

A *hybrid* editor supports structural as well as presentation-oriented edit operations. Figure 1.4 shows a hybrid editor. Because both levels can be edited, there are no dotted arrows in the figure. However, in order to offer this edit functionality, the editor must realize both the presentation and interpretation mappings. Hence the double arrow between the document and the presentation.

In some publications (e.g. [58, 86]), the term hybrid is used to refer to syntax-directed structure editors that have a limited form of syntax-recognizing functionality. As a consequence, most syntax-directed editors would qualify as hybrid editors, because most editors support some form of text parsing.

In contrast, other publications (e.g. [7, 49]) advocate that the term hybrid should be reserved for editors that support full textual editing of the document, as well as limited syntax-directed functionality, even if structural modifications on the document are not supported. According to this view, almost all syntax-recognizing editors would classify as hybrid editors, since most of these editors support a form of structural navigation.

Because of the confusion, and because most editors tend to be either primarily syntax-directed or syntax-recognizing, we will often use those terms, instead of the term hybrid.

1.2 Proxima

The subject of this thesis is the design of the presentation-oriented structure editor Proxima. Proxima is suitable for a wide range of applications, including word-processors and source editors, but also mathematical-equation editors and spreadsheets. An important aspect of Proxima is that the editor fully supports presentation-oriented as well as document-oriented editing. Thus, the editor classifies as a hybrid structure editor.

The implementation of the bidirectional mappings between the document and the presentation is facilitated by a layered architecture. The computation of the presentation is broken up in several stages, and each intermediate value in this computation corresponds to a *data level* (or just *level*). The interpretation process has the same intermediate values. Between two levels, there is a *layer*, which is a component that takes care of mapping values at one level onto values at another level, and thus implements a single stage of the presentation and interpretation processes.

The first step of the presentation process is that the *document* is enriched with computed values and structures, by the evaluator. The resulting *enriched document* is mapped onto a logical *presentation* in which positions and sizes are specified relatively. The layout layer adds whitespace that is not stored in the document to the presentation, yielding the *layout*

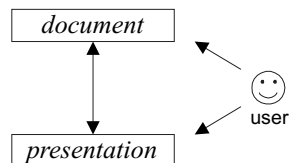


Figure 1.4: A hybrid editor.

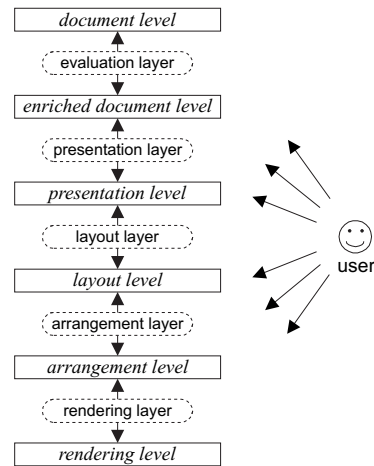


Figure 1.5: Proxima.

level. The layout level is mapped onto an *arrangement* level, which contains absolute positions and sizes. And finally, the renderer maps the arrangement onto a *rendering*, which is made visible to the user. A more detailed discussion of the presentation and interpretation processes is provided in Chapter 3.

Figure 1.5 shows the levels and layers of Proxima. Because the name *presentation* is reserved for one of the intermediate levels, the lowest level is referred to, more appropriately, as the *rendering* level. The figure shows multiple edit arrows coming from the user, because edit operations may be targeted at intermediate levels as well.

The layered architecture makes it possible to combine presentation-oriented editing with a powerful document-presentation mechanism that includes support for derived values and structures. A platform-independent Haskell prototype of Proxima has been implemented, and experiments with instantiated editors have yielded promising results.

1.3 Terminology

We give a brief summary of the terms that were introduced in the previous sections.

Editor: Application for creating and modifying documents. In this thesis, the term also used to refer to structure editors and generic editors (or generic structure editors).

Document: Internal data structure that represents the information that is edited.

Presentation: Term for a visible representation of the document (also called a *View*), as well as for one of the intermediate levels of the presentation process.

Presentation mapping/relation: The relation between the document and its presentation.

Presentation-oriented editing: Edit operation targeted at the presentation:
e.g. deleting “ + ” from “1 + 2”, yielding “12”.

Document-oriented editing/Structure editing: Edit operation targeted at the document:
e.g. swapping two sections in an article.

Presentation sheet: Parameter to the presentation/interpretation process.

Presentation process: Process of computing the presentation of a document.

Interpretation process: Process of computing a document from a presentation.

Level: Intermediate value of the presentation/interpretation process, including the document and the presentation.

Layer: Component that realizes the presentation and interpretation mappings between two levels.

Structure editor: An editor that has knowledge of the structure of the edited document.
Usually assumed to be a *generic editor* as well.

Generic editor: A structure editor suitable for editing documents of different types.

Syntax-directed editor: An editor that primarily supports document-oriented editing.

Syntax-recognizing editor: An editor that primarily supports presentation-oriented editing.

Valid document: A well-typed document, mainly used in the context of XML.

Valid presentation: A presentation that is the result of presenting some document.

Focus: Shared name for cursor and selection.

1.4 Outline of the thesis

The remainder of this thesis has the following structure:

Chapter 2 explores applications of generic structure editing by providing five use cases of real-world editors. With these use cases in mind, we formulate a number of functional requirements that in our view are important for a flexible non-restrictive structure editor. We evaluate a number of existing editors according to the requirements, and conclude with an overview of how the Proxima editor is designed to meet the requirements and be able handle all use cases.

The layered architecture of Proxima is introduced in Chapter 3. The chapter discusses the various data levels, as well as the layers that maintain the mappings between the

levels. The discussion is illustrated with examples of the presentation and interpretation processes.

In chapters 4 and 5, we develop a specification of the Proxima editor. Chapter 4 serves as an introduction to the specification and introduces our model of the edit process, as well as the concepts of extra state and duplicates in the presentation. In Chapter 5, we start by specifying a simple editor, to which extra state and multiple layers are added in subsequent sections. The chapter ends with an informal discussion on how to handle presentations that contain duplicates.

In Chapter 6, we discuss the XPRES presentation formalism of the Proxima. XPRES is a declarative presentation language, suited for specifying a wide range of presentations. We state a number of requirements for a presentation language for structured documents, and provide an informal overview of XPRES, using a series of examples.

A prototype that offers much of the functionality discussed in this thesis has been implemented in the functional language Haskell. Chapter 7 discusses the prototype as well as a number of editors that have been instantiated. The chapter also explains which components need to be provided to instantiate an editor.

Finally, Chapter 8 presents the conclusions and gives an overview of future research.

Requirements for a structure editor

In this chapter, we present five use cases of possible applications for a generic structure editor. The use cases will shed more light on the definition of the term editor from the previous chapter, and provide standard examples to explain and define edit behavior in the coming chapters. A design requirement of Proxima is that the editor is able to handle all five use cases.

Section 2.1 presents the five use cases, from which we formulate a set of functional requirements for a flexible non-restrictive structure editor, in Section 2.2. Existing editors are evaluated according to these requirements in sections 2.3 and 2.4, showing why none of these editors can handle all use cases. Finally, in Section 2.5 we discuss how the Proxima editor meets the requirements and thus will be able to implement all of the use cases.

2.1 Use cases

Some of the five example editors, presented in this section, are well-known applications of structure editors, but a few more exotic applications have been included as well. None of the current generic structure editors can handle all five use cases. It is important to note that although the use cases are discussed as separate applications, aspects of them can be combined in a single editor instance. The discussion of the edit behavior is illustrated with fictitious screenshots. Actual screenshots of the Proxima prototype can be found in Chapter 7.

2.1.1 A source editor for Haskell

As an example of a program-source editor, we take an editor for the functional programming language Haskell [70]. The editor supports an extended form of syntax highlighting, in-place display of syntactic and semantic errors, and a range of language-specific edit operations.

There exists evidence showing that syntax highlighting makes programs more readable [5, 68]. Our editor supports highlighting at a semantic rather than syntactic level. Hence, unlike most text editors, the editor can use different display styles for language constructs that are hard to recognize purely syntactically. The type declarations in the next screenshot are an example of such a construct. Although syntactically identical, identifiers in type expressions are colored differently from identifiers in ordinary expressions.

```

Haskell editor
File Edit View Help
module Main where

s :: (a -> b -> c) -> (a -> b) -> a -> c
s = \f -> \g -> \x -> f x (g x);

maybe :: a -> (b -> a) -> Maybe b -> a
maybe n j m = case m of
    Nothing -> n
    Just x -> j x

f :: a -> b -> c
g :: a -> b
s :: (a -> b -> c) -> (a -> b) a -> c

```

The Haskell source editor.

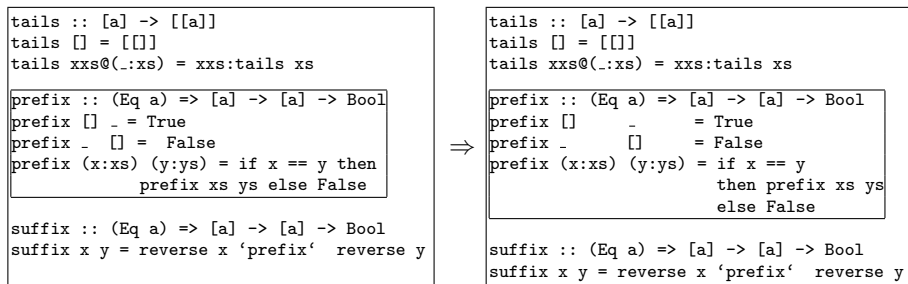
Haskell is a particularly interesting language for a source editor because, due to Haskell's rich type system, information about types is very useful during programming. Haskell programmers often experience that once type errors have been removed, a function is correct. Therefore, an environment that supports in-place display of type errors, as well as easy access to type information of variables in scope, will help rapid program development.

The upper pane of the editor in the screenshot shows a highlighted source of a simple Haskell module. The bottom pane shows the automatically derived types for identifiers that are in scope at the focused position, including the types of locally declared identifiers. A mouse click on an identifier in the list changes the focus to the definition of that identifier in the source pane.

Automatic layout/pretty printing

Some structure editors use an automatic layout scheme while editing program sources. The user then does not need to worry about layout issues, such as the alignment of parameters in functions with multiple clauses. However, for a Haskell editor this situation is not optimal because Haskell programs mainly consist of expressions, which are hard

to layout automatically. Therefore, rather than having automatic layout be continuously performed on the entire source, a user may request the editor to automatically lay out a selected part of the program. The specification of the layout of the program is part of a presentation sheet and may be adapted by the user. Of course, if desired, it is also possible to turn on continuous automatic layout.



apply layout

The screenshot shows the automatic-layout operation applied to the selected function `prefix`. The formatted function on the right-hand side is still freely editable, including its whitespace.

Structural edit operations

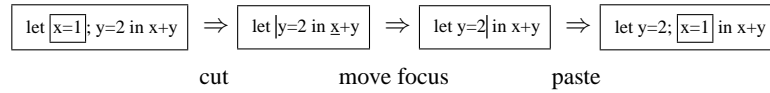
Because a program construct is represented by a contiguous area in the presentation, moving a program construct can usually be done in a straightforward way by moving its presentation. However, this is not always the case. Take, for example, the expression:

```
let x=1; y=2 in x+y
```

The expression consists of a list of declarations that are separated by semicolons and whitespace. Contrary to, for example, the language Java, the semicolon in Haskell acts as a separator, and not as a terminator. Unlike a terminator, which can be regarded as part of the presentation of a declaration, a separator belongs to the presentation of the list of declarations. As a result, semicolons may cause problems when declarations are moved.

Consider moving the first declaration `x = 1` to the end of the `let` expression. When the declaration is cut, the semicolon behind it must be deleted, and when the declaration is pasted, a semicolon with appropriate whitespace must be added. Similar issues apply to all list structures that are presented using separators, such as Haskell lists `[1, 2, 3]`, tuples `(1,2,3)`, or monadic `do` expressions: `do {a <- getChar ; putStr [a]}`.

If the structure of the edited list is taken into account, cut-and-paste on lists with separators can be handled elegantly. When the first declaration is selected, the editor recognizes it as an element of the `let` expression's declaration list, and when it is cut, the semicolon next to it disappears:

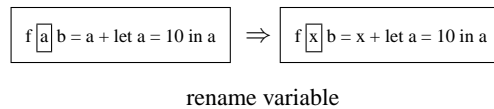


When the declaration is pasted, a semicolon is automatically placed in front of it. The whitespace from the semicolon is copied from the whitespace of the other semicolons in the presentation (or may come from a pretty-printing algorithm). If the list has an irregular layout (e.g. [1, 2, 3, 4]), the layout after the paste operation may not be what is expected. However, since list structures are usually layed out in a regular way, this need not be a problem.

If an edit operation can be performed both structurally as well as presentation-oriented, as is the case here, the editor gives preference to the structural edit operation. In order to perform the cut and paste operations from the example on the presentation (and thus leave the semicolon untouched), a modifier key may be pressed.

Rename within scope

A second example of an edit operation that takes the document structure into account is a rename operation on an identifier. In a regular text editor, occurrences of the identifier name need to be changed using search and replace. However, automatic search and replace does not always lead to the desired result because the identifier may be redeclared in inner scopes, or the identifier name may appear in a string.

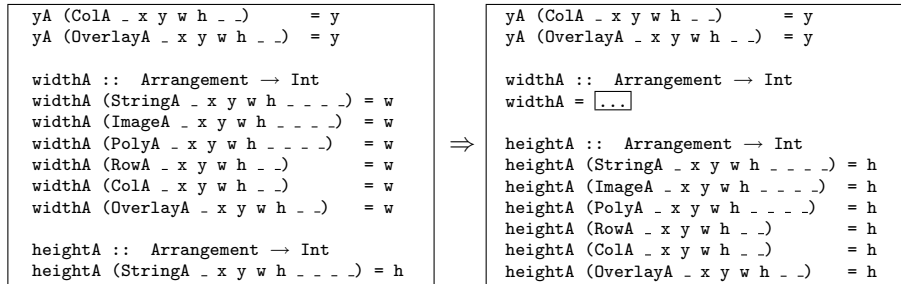


The rename operation takes account of the scoping rules of Haskell, and only changes the appearances that lie in scope of the updated identifier. If the new name is captured by an inner declaration, or if it shadows an identifier that is already declared, a warning is issued.

The rename operation is an example of a *refactoring* operation: a source-to-source program transformation that leaves the functionality of the program intact. The designers of the Haskell Refactorer [52] identify a number of such transformations.

Hide function definitions/folding

Function definitions in the source presentation may be collapsed, leaving only one left-hand side and the (possibly inferred) type declaration. This operation is often referred to as folding.



hide function definition

The two functions `widthA` and `heightA` have a large number of clauses. Hiding these clauses may improve the readability of the source. After applying the *hide function body* edit operation to the function `widthA`, only one clause remains with a collapsed right-hand side (`[...]`). The function is expanded again by clicking on the dots. A collapsed function may be renamed, deleted, or structurally moved around in the source. Editing the whitespace around the '=' sign would be ambiguous because of the hidden parameters names and the fact that a single collapsed function typically represents several clauses. Hence, editing this whitespace is not allowed.

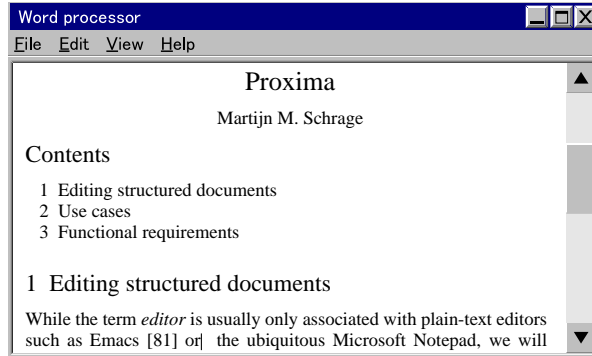
Requirements

The source editor generates a number of requirements, an important one being the possibility of freely editing the textual program source, including the layout. At the same time, it must be possible to let the layout be computed automatically as well. Furthermore, a formalism for specifying computations over the document is required for performing static analysis and type checking.

Freely editing the program source is not always regarded a requirement for a structure editor. Purists argue that text editing may introduce syntactic errors, and that it is not necessary for programming (e.g. [54, 86]). However, no clear consensus has been reached on the subject (e.g. [93] and reactions [66, 80], and [88]) and nowadays most syntax-directed editors support some form of free text editing. Furthermore, because up to now no pure syntax-directed editor has ever become popular with programmers, we believe that free textual editing is an essential requirement for a program-source editor.

2.1.2 A word processor

This section describes a WYSIWYG document editor with a user interface similar to word-processing applications such as Microsoft Word, but with a document model similar to the XML/SGML DocBook standard [92] and an output quality similar to \TeX [47]. Examples of editors with similar functionality are \TeX macs [63] and Lyx [25], but neither system is generic.

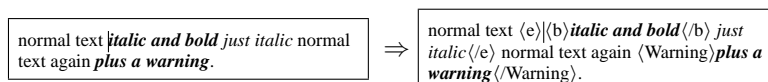


The word processor.

The document model consists of chapters, sections and subsections. The editor supports free editing in the WYSIWYG presentation with optimal line breaking, a derived table of contents, and an automatic bibliography. Cross-references, such as citations or references to figures, can be clicked to bring the referred part of the document into focus.

Structural view on the document

Although Microsoft Word is one of the most popular word-processing tools in the world, an often-heard complaint concerns its confusing document model. Sometimes edit operations are not allowed because of underlying document structure, but it is not obvious why this is the case. Furthermore, the reason why a document fragment looks the way it does is not always clear. The user may have set specific style attributes for a particular fragment, or the style may originate from the document's presentation rules. A more structural view on the document, such as WordPerfect's "underwater" screen, can help to clarify the situation, but is not supported by Word. Such a structural view is easily defined in a structure editor:



switch to structural presentation

The two screenshots show two presentations of the same document fragment. The left-hand presentation is the regular WYSIWYG presentation, whereas the right-hand one is a more structural presentation that shows the markup tags. The example document also contains a fictitious <Warning> element that is presented in a bold and italic style. Only in the structural presentation can the warning be distinguished from text that has been explicitly formatted as bold and italic.

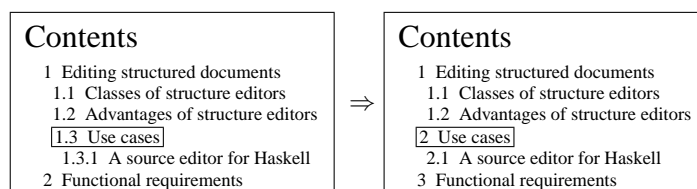
The structural view is also helpful for positioning the focus. In the left-hand presentation it is not clear whether the focus is in the regular text, the italic part, or the part that is both

italic and bold. The right-hand presentation, on the other hand, shows the exact position of the focus in the italic part. In order for the structural views to be helpful, the editor supports easy switching between views while preserving the current focus.

Structural edit operations

Edit operations that rearrange the document structure, such as promoting a subsection to section, are awkward to perform on a textually represented document, such as a \TeX source. All tags or \TeX commands that specify the subsection and its descendants need to be changed. This is a rather specific search/replace operation on only part of the document source, which is a hassle to automate.

A structure editor may be of some help here, because the structural similarities between sections and subsections are known to the editor and can be used to define edit operations for restructuring the document.



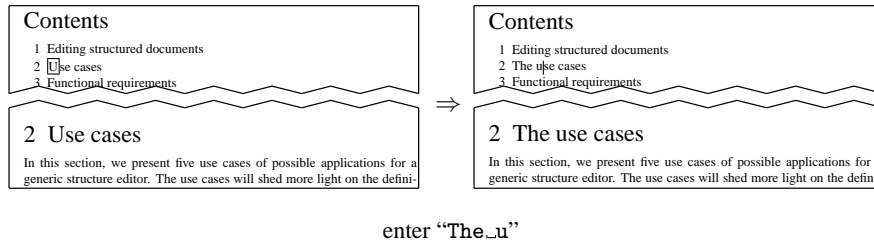
change to section

The screenshot shows the effect of the structural edit operation “change to section”. If the containing section (“Editing structured documents”) had had any subsections following the promoted subsection, these could have become subsections of the new “Use cases” section. However, different behavior for such sections may be specified by the user in a preferences window for the edit operation.

An operation that changes the level of a section or subsection is rather complex because it involves splitting and changing elements. Moreover, there are special cases to consider. For example, if a subsubsection is the deepest possible structural level, a warning needs to be issued when a section containing a subsubsection is demoted to subsection. Therefore, such an operation needs to be specified explicitly by the editor designer or user. Other document operations, however, such as splitting and joining elements of a list, may be derived automatically.

Editing a section title in the table of contents

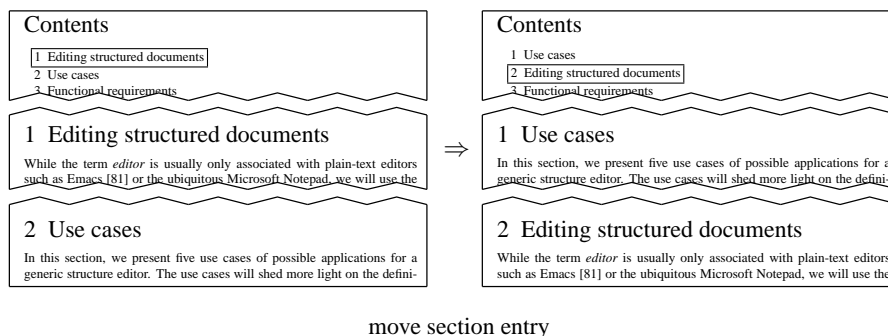
The word processor has support for the specification of a generated table of contents. From an entry in the table, a user can jump to the corresponding position in the document presentation. The presentation of the table of contents itself can be customized to match the style of the rest of the presentation. When the document is edited, the table of contents is updated accordingly. Moreover, editing an entry in the table of contents causes an update to the title of the corresponding chapter or section.



The screenshot shows a document with a table of contents. The second entry in the table of contents is edited by entering the text "The u" over the selected letter 'U' at the beginning of the title. The result is that the title in the table of contents as well as the title in the section is updated.

Moving a section in the table of contents

Besides textual edit operations, it is also possible to perform structural edit operations on derived structures. The screenshot shows a move operation on a section title in the table of contents, which has the result that the corresponding section is moved in the document.



The section entry for the "Editing structured documents" section is selected and dragged to its new location, just below the entry for the "Use cases" section. The result is an edit operation on the document structure that puts the first section after the second section. The section numbers switch because they are generated automatically. Whenever an edit operation on a derived structure is performed, the user may be signaled that the operation affects more than just the visible selection.

Although structure-changing operations on derived structures may not always make sense, it is important that they can be specified for the cases in which they do.

Requirements

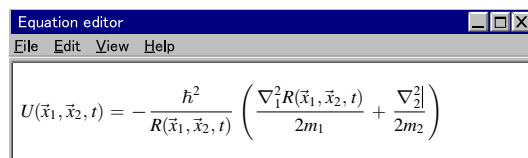
Compared to the source editor, the word processor requires a more powerful presentation formalism. Besides text in different fonts, sizes, and colors, the presentation also contains

images and basic graphical elements. Furthermore, support for optimal line and page breaking is needed for formatting paragraphs and pages.

Finally, in order to handle edit operations on the table of contents, the editor must support editing not only on presentation and document level, but also on the level of derived structures.

2.1.3 Equation editor/MathML

Because mathematical formulas have a high degree of structure, a mathematical equation editor is a good candidate for structure editing. Equation editing functionality is typically integrated with a word processor to support in-place editing of equations in a document. An example of an editor for mathematical documents is the MathSpad editor [89], which offers word-processing functionality as well. MathSpad also supports a form of genericity, but does not allow the document type to be specified. Furthermore, both the edit model and the presentation have a tendency towards documents of a mathematical nature.



The equation editor.

The screenshot shows a WYSIWYG equation editor with support for mathematical constructs such as fractions, roots, and integrals. A possible document type for the equation editor is the Mathematical Markup Language MathML [20].

Mathematical formulas are suitable for document-oriented edit operations, using menus and buttons for structure entry. Free presentation-oriented editing, on the other hand, is not as clearly defined on a formula as it is on a program source. For example, shrinking the 2 in the number 42 and moving it upwards a bit, could theoretically lead to recognition of the square 4^2 . However, this requires a complicated visual parsing scheme, the exact behavior of which is not clear. Therefore, the editor only allows free editing in the textual parts of a formula that can be parsed unambiguously.

Although such a rather restricted edit model is common even in the current generation of non-generic equation editors, we believe that a more sophisticated and flexible edit model is possible. The Proxima architecture does not prohibit such an edit model, but further research on parsing two-dimensional structures is required before it can be supported.

Drag and drop

Direct manipulation of parts of the formula is supported on a structural level. A proper subtree of the formula can be dragged to a different location.

$$\boxed{\frac{(x-1) \times \boxed{x+1}}{y + \{\text{Exp}\}}} \Rightarrow \boxed{\frac{(x-1) \times \{\text{Exp}\}}{y + \boxed{x+1}}}$$

move $x + 1$

The subformula $x + 1$ is dragged to its new location below the fraction bar, leaving a placeholder $\{\text{Exp}\}$ at its origin. Note that the parentheses disappear because the $+$ operator is associative.

Only proper subtrees in the document may be selected in the equation editor. This means that in the formula 2^{3^4} , the 2^3 part may not be selected because it is not a proper subtree (the power operator associates to the right).

In practice, we do not expect this restriction to be a major problem. A fragment of the presentation that does not correspond to a proper subtree does not actually represent a meaningful expression. Hence, the chance that the fragment is reused elsewhere or needs to be moved is small. An unlikely situation in which this might occur is when a user needs to build an expression that by chance has exactly the same presentation as some already present non-subtree selection.

Textual structure entry/parser

For quick and easy structure entry, the editor supports textual entry of mathematical structures without having to switch to a different mode.

$$\boxed{a_n b_n = \frac{4\theta_{ab}}{4\pi} = -1 + \frac{2\theta_{ab}}{\pi}} \Rightarrow \boxed{a_n b_n = \frac{4\theta_{ab} - 4(\pi - \theta_{ab})}{4\pi} = -1 + \frac{2\theta_{ab}}{\pi}}$$

enter “ $-4(\backslash\text{pi} - \backslash\text{theta}_{\{ab\}})$ ”

The entered text is parsed and causes the insertion of $-4(\pi - \theta_{ab})$, as shown on the right. It should be noted, however, that textual entry does not always lead to the desired result in a two-dimensional presentation. For example, when “ $2/4$ ” is entered, an intuitive result is the insertion of a fraction with the focus ending up below the fraction line to the right of the 4, yielding $\frac{2}{4}$. But now the expected result of entering “ $+6$ ” would be $\frac{2}{4+6}$, whereas the correct meaning of “ $2/4+6$ ” is $\frac{2}{4} + 6$.

If a complex subformula needs to be entered, or if the appearance of a formula needs to be fine tuned, the user may temporarily switch to a more structural presentation. This may be considered a mode switch, but since the structural presentation is in the same window as the graphical presentation and may be switched back at any time, it does not restrict the user.

Domain-specific transformations

Because the editor has knowledge about the exact structure of a document, rather than just about the structure of the presentation, it is possible to specify domain-specific mathematical transformations.

$$\boxed{x = a \times (b + c)} \Rightarrow \boxed{x = a \times b + a \times c}$$

distribute

The example shows the application of a distribution transformation to the selected sub-formula. Similar transformations, such as factorize or reverse, may be specified by the editor designer or the editor user. Furthermore, instead of updating the expression in-place, the editor may also insert the transformed expression below the original. This way, the editor can be used to construct derivations or proofs semi-automatically (or indeed fully automatically, if the editor is connected to a theorem prover).

Requirements

Presenting mathematics puts a heavy demand on the presentation formalism. Fine control over automatic alignment and resizing of presentation elements is needed for complex presentations such as integrals, square roots and fractions.

Editing mathematics requires basic document-oriented edit support (copy and paste), as well as drag and drop editing. Structure entry is also typically a document-oriented edit operation, because many expression structures, such as a quotient, a power expression, or a square root, have no presentation that can easily be entered with conventional editing methods.

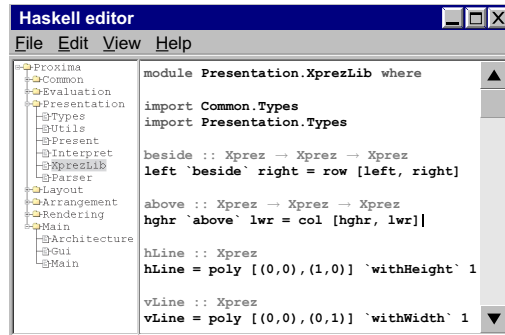
Because parts of an expression may be missing during its construction, the editor must be able to handle incomplete documents. Furthermore, for supporting domain-specific transformations, a formalism for specifying document-oriented edit operations is needed. Presentation-oriented editing on mathematical formulas is desirable, but is not a strict requirement, because of its still unclear nature.

2.1.4 Non-primitive outline view/tree browser

An outline view, or tree browser, is a hierarchical view on tree structures. It is found in the Java Swing GUI library and also forms the main navigation tool in Microsoft's Windows Explorer application.

Some editors, especially XML editors, provide tree-browser views on the document, but in almost all editors, the view is hard-coded. If an editor is sufficiently powerful to express a tree-browser view without resorting to a primitive tree-browser widget, this offers many possibilities for integrating the tree view with other views on the document.

Tree views are useful for giving an overview of large structures, such as a program source that consists of a number of different modules. By combining a tree presentation with a source editor, we can support the kind of project management that is found in integrated development environments, such as JBuilder or Eclipse [67].

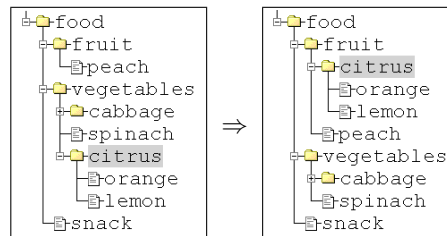


A source editor with a tree-browser pane.

The window in the screenshot consists of two panes, the right-hand pane contains a Haskell source editor and the left-hand pane contains a tree view of the module structure of the edited program. When the user clicks on a name in the left-hand pane, the corresponding module is shown in the right-hand pane.

Drag and drop

The tree browser supports drag and drop edit behavior that allows nodes in the tree to be dragged to new locations.



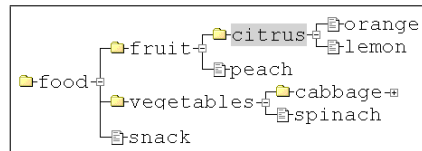
move "citrus" node

The screenshot shows the effect of dragging the node with label "citrus" to a new position immediately below "fruit". The operation results in a structural document change in which the element with presentation "citrus" becomes the first child of the element that has presentation "fruit".

In this example, the elements of the tree all have the same type and can therefore be moved anywhere in the structure. Using the tree view for outline editing in the word-processor example is slightly more complex, because a move operation may require a transformation of the element moved. For example, when a subsection is moved immediately under a chapter element, it must be changed to a section.

Customized tree views

Because the tree presentation is not primitive, the editor designer or user can customize it, or even define entirely different tree presentations.



An alternative tree-browser view.

The tree view in the screenshot is a more spacious presentation, in which the child nodes are presented to the right of the parent rather than below.

Requirements

Similar to the equation editor, the tree browser has a two-dimensional graphical presentation that requires fine control over the alignment of the presentation elements. Customizability of the tree view requires that the presentation specifications are transparent and reusable.

Edit operations on the tree structure are similar to edit operations on the table of contents in the word-processing example, because the tree is typically a derived structure that follows the structure of the document (or part of it). Updates on the tree need to be mapped on updates on the document itself. Navigation operations can be considered an update on the focus and hence the specification formalism for document-oriented edit operations must support focus updates.

An aspect that is specific to the tree browser is that it has a notion of state. Each node in the tree view is either collapsed or expanded, and this information must be stored somewhere. Such presentation state, or *presentation extra state*, as we call it, does not form part of the document, because if it is stored there, the document type will need to be changed if a tree view is added to the presentation. The fact that this state is not part of the document, but rather of the presentation of the document, makes it hard to model in a structure editor. A mapping between the document and the presentation state needs to be maintained to associate a document node with its expansion state, even when the document is edited and its nodes are reordered.

2.1.5 Simple tax form/spreadsheet

The last example is a simple tax-form application, which is basically a spreadsheet with a rather specialized presentation. It contains questions and explanatory text, mixed with input fields and fields that contain derived information.

Nr.	Description	Salary	Tax withheld
1	PhD Student	10.00	2.50
2	Programmer	20.00	6.00

Total income: 30.00
Tax paid: 8.50
Tax due: (35% of income - paid) 2.00

A much simplified tax form.

The tax-form editor has two different kinds of users: a user who designs the tax form, and a user who fills out the form. Both users use the same document type, albeit with different presentations. The form designer uses a presentation that shows the building blocks and structure of the form, as well as the formulas for the derived values. On the other hand, the user who fills out the form sees the input fields, the derived values, and the accompanying fragments of text. The structure of the form and the formulas for the derived values are not explicitly visible and cannot be modified in this presentation.

The distinction between the two kinds of users differs from the distinction in Section 1.1.3 between editor designer users and document editing users, because for the tax form, both users edit the document and therefore are document editing users rather than editor designers.

A difference between the tax form and the previous use cases is that, similar to a spreadsheet, it has computations that are specified in the document itself. Hence, these computations can be modified by an editing user, rather than the editor designer. Although this is probably not how an actual tax-form application would be designed, we use the presence of computations in the document as an example of spreadsheet behavior in a structure editor.

Presentation depending on document values

In most presentations, the structure of the presentation depends on the document structure. However, the presentation structure may also depend on a document value, rather than the structure. An example is the following section of the tax form:

The image shows two side-by-side screenshots of a 'Tax form' application window, separated by an arrow indicating a transition. Both windows have a menu bar with 'File', 'Edit', 'View', and 'Help'.
 The left window shows 'Number of jobs' set to 2. It contains a table with two rows:
Nr.	Description	Salary	Tax withheld
1	PhD Student	10.00	2.50
2	Programmer	20.00	6.00
Below the table, there are three text fields: 'Total income: 30.00', 'Tax paid: 8.50', and 'Tax due: (35% of income - paid) 2.00'.			
The right window shows 'Number of jobs' set to 3. The table now has three rows:			
Nr.	Description	Salary	Tax withheld
1	PhD Student	10.00	2.50
2	Programmer	20.00	6.00
3		0.00	0.00
 The summary fields below the table remain the same: 'Total income: 30.00', 'Tax paid: 8.50', and 'Tax due: (35% of income - paid) 2.00'.

increase number of jobs

The number of input fields for job information depends on the number of jobs. When the number is increased, the structure of the input form changes accordingly, showing an extra line of input fields. Decreasing the number hides the corresponding input fields, but after a subsequent increase, the fields reappear containing their previous values.

The tax-man view

A different presentation of the tax form allows a user to design the form by editing the structure of the form, rather than the values of its input fields.

The screenshot shows the 'Tax form' application window in a 'tax-man' view. The window has a menu bar with 'File', 'Edit', 'View', and 'Help'.
 At the top, there is a 'text: Number of jobs' field and a 'special: jobs.length' field with a value of 2.
 Below this is a list of jobs with columns for 'Nr.', 'Description', 'Salary', and 'Tax withheld'. The 'Nr.' column is derived from '.ix+1'. The 'Description' column is derived from 'input: ds'. The 'Salary' column is derived from 'input: ss' and the 'Tax withheld' column is derived from 'input: ts'.
 The table shows two rows:
 | derived: 1 | input: (ds [0]) | PhD Student | input: (ss [0]) | 10.00 | input: (ts [0]) | 2.50 |
 | derived: 2 | input: (ds [1]) | Programmer | input: (ss [1]) | 20.00 | input: (ts [1]) | 6.00 |
 Below the table, there are three text fields: 'text: Total income: derived: in sum jobs.ss 30.00', 'text: Tax paid: derived: tax sum jobs.ts 8.50', and 'text: Tax due: (35% of income - paid) derived: res 0.35*in-tax 2.00'.

The tax form for the tax man.

The screenshot gives an impression of a presentation in which the tax-form structure and layout are editable. The tax-form document is the same as for the first screenshot. Text blocks, as well as input fields and derived value fields can be inserted or deleted, and the computations for the derived values (“`.ix+1`”, “`sum jobs.ss`”, “`sum jobs.ts`”, and “`0.35*in-tax`”) can be modified. The input values of the input fields are still editable to allow for easy testing of the specified computations.

Requirements

In contrast to the other use cases, the tax-form presentation is rather similar to a user interface. Instead of just text and graphical elements, it contains widgets, such as check boxes, selection lists, and input fields, with the corresponding edit behavior.

The tax form also features computations with results that appear explicitly in the presentation itself. Unlike the type computations in the Haskell editor, the computations in the tax form are part of the document, and may be specified by an editing user (the tax man), rather than an editor designer. Therefore, similar to a spreadsheet application, the editor needs to dynamically interpret document structures that represent computations and display the results in the presentation.

2.2 Functional requirements

With the use cases of the previous section in mind, we now provide a number of functional requirements for a generic structure editor.

2.2.1 Genericity

The primary requirement for the editor is genericity: the editor must be generic in the sense that it is not built for a specific document type or class of document types. However, as mentioned in Section 1.1.1, we restrict ourselves to trees rather than graphs. Most documents can be represented by trees, including our five use cases. A formalism for specifying cross-links between tree nodes is desirable, but full graph editing is not a requirement.

2.2.2 Computation formalism

An interesting aspect of an editor that has knowledge of the structure of the document, is that it can show derived values over that structure to the user. Examples of derived values are chapter numbers and a derived table of contents, but also derived type information for a program source. An important point is that the computations we refer to are part of the presentation process. Based on values in the document, the derived value is computed, but the document itself is not affected. This is different from computations that map the document onto a new document containing derived values. The latter kind also allows computed values to be shown to the user, but in the process causes an update to the document. Hence, we view such computations as document-oriented edit operations, which are covered by the *editing power* requirement, discussed in Section 2.2.4.

Two aspects influence the usefulness of the computations: the expressivity of the formalism in which the computations are specified, and the integration of computed values and structures with the document presentation.

For program editing as well as the tax form, the expressivity of the computation formalism is important. Computations can provide static analysis, e.g. detecting name clashes and scoping problems, as well as a type derivation. In order to be able to specify these computations for arbitrary languages, a Turing-complete formalism, such as an attribute grammar [85], is desirable. Other options include constraint-based systems [7, 9, 30, 61] and tree transformation formalisms [21, 90]. Furthermore, the computation formalism should offer functionality for connecting to external tools, such as a compiler or a theorem prover.

For the word-processing example, as well as the tax form, the integration of computed values with the document presentation is important. Whereas type errors may be shown in separate windows or by underlining the location and showing the message in a tooltip, chapter numbers and a table of contents form an actual part of the presentation.

2.2.3 Presentation formalism

The presentation formalism has two different aspects, which we consider together here. One is the formalism in which the building blocks of the presentation are expressed (the *presentation target language*), whereas the other (the *presentation specification language*) is the formalism in which it is specified how a document is mapped onto an element of the presentation target language. For XML, a well-known presentation language is the Extensible Stylesheet Language (XSL) [1]. XSL is split into the mapping language XSLT [21] and the target language XSL Formatting Objects. Chapter 6 discusses presentation languages in more detail.

In many editors the *presentation target language* consists of just plain text, sometimes with color and font attributes. However, in order to support the graphical presentations of the equation editor and outline view use cases, a more advanced target formalism is required. It must be possible to specify graphical elements such as lines and boxes, as well as to show images. Furthermore, the presentation of a mathematical formula requires an advanced alignment model that offers full control over the positioning of presentation elements.

Another requirement for the presentation target language comes from the tax-form example. The tax form typically contains user-interface widgets, such as buttons, selection lists, and menus. Therefore, the target language must support user-interface widgets.

Finally, the word-processor use case requires that the presentation target language supports line and page breaking, preferably optimal [48].

The *presentation specification language* has to allow the specification of complex graphical presentations using compact readable style sheets. It must be possible to specify simple presentations in an easy way, while still allowing the specification of more complex presentations. For the exact choice of formalism we have similar options as for the computation formalism, including AGs, constraint-based systems, and tree transformation formalisms.

Although a presentation can be seen as a computed value, we make a separation between the presentation specification language and the computation formalism. One of the reasons is that the separation of computation and presentation makes it possible to specify multiple presentations of a document together with its computed values. Furthermore, the separation makes it easier to support edit operations on derived structures.

2.2.4 Editing power

The editing power of an editor is determined by the fact whether both document- and presentation-oriented editing is supported, together with the complexity of the edit operations and to what extent these operations are user-specifiable.

The equation editor as well as the outline editor rely heavily on document-oriented editing. Document-oriented edit operations typically include basic copy, paste, and delete operations, as well as selection and navigation operations.

Because a document is not always well typed while it is being constructed, the editor should support incomplete document structures, for example by allowing placeholders to appear in the document tree. Besides incomplete documents, it is desirable to have support for invalid documents in general. However, because it can be difficult to compute the presentation of an invalid document, we may wish to allow invalid documents only for certain presentations, such as a textual XML source presentation.

Presentation-oriented editing is required for the source editor, because it supports free textual editing of the program source. To a lesser extent, presentation-oriented editing is needed also for the equation editor (for textual structure entry) and the tax form (for editing the computations).

Finally, as the editable table of contents of the word processor use case shows, support for edit operations on derived structures is desirable. This is not to say that all derived structures and values should be editable, but in those cases in which it makes sense to a user, it should be possible to specify the edit behavior for derived structures.

For document-oriented edit operations, a transformation specification formalism is desirable. It allows an editor designer to define edit operations specific to a certain type of document. An example of such an edit operation is the rename operation in the Haskell editor. Furthermore, the formalism can be used to specify standard generic document-oriented edit operations such as split and join.

2.2.5 Modeless editing

Besides support for both document- and presentation-oriented editing, an important requirement is the integration of these two kinds of editing. A seamless integration provides a pleasant edit interface to the user, as the intended operation can be performed on the presentation the user is working on, without first having to explicitly switch modes. In case an edit operation is a meaningful operation on both the document and the presentation

(yielding different results), the editor can give preference to the document operation, with the possibility to override this preference. Sufrin and De Moor describe a basic modeless structure editor [82], but the idea of modeless editing is also found in earlier publications (e.g. Kaiser and Kant [43]).

The most extreme form of mode-switching is when edit operations on different levels have to take place in separate windows and also have a separate undo-history. This is the approach taken by many pure structure editors that offer some support for free text editing, as well as by all existing XML editors. Even worse, the separate free-editing text mode often has a special text-only format, in which derived values are not shown and interesting graphical presentations are not possible. In order to get back to document-oriented editing, the user needs to leave the text in a valid state, or abandon the text update.

If the editable textual presentation is displayed in-place in the document presentation, the mode-switching becomes less intrusive. However, the most user-friendly approach is to avoid mode switching altogether, thus allowing a user to freely edit the presentation, even if it contains computations and graphical presentations. Moreover, if a presentation-oriented edit operation makes the presentation invalid, the invalid area should be kept as small as possible, and document-oriented editing must still be available on the valid parts.

2.2.6 Extra state

If a document is edited, the presentation is updated accordingly by presenting the modified document. In some cases, however, a presentation may contain information that cannot be derived from the document. We refer to such information as *presentation extra state*. Analogously, the document may contain information that cannot be inferred from the presentation. This information is referred to as *interpretation extra state*.

A clear example of *presentation extra state* is found in the outline view example. The expansion state of the nodes of the tree view needs to be kept track of. However, this is not information that should be stored in the document tree structure, since the design of the document type should not have to consider what views may be defined for that document type. Moreover, several views may be opened simultaneously, each with their own expansion state. Hence, the expansion state is regarded as presentation extra state. Other examples of presentation extra state are focus information, local layout settings (e.g. whether or not auto-layout is turned on), and whitespace in the presentation.

Interpretation extra state, on the other hand, is any information in the document that cannot be inferred from its presentation. Hence, if a document is only partially presented, those parts that are not presented are considered interpretation extra state. An example is an editable table of contents of a word processor, in which the content of the chapters and sections is interpretation extra state.

In order to handle the use cases, a generic editor should support both presentation extra state, and interpretation extra state (in the form of editable partial presentations). We do not consider a generic editor to fully support presentation extra state if it only supports a

built-in form of it. Instead, it must be possible to explicitly declare parts of the presentation to be extra state.

In order to support extra state, an editor needs to maintain information about the mapping between the document and its presentation. If there is no extra state, no extra effort should be required from the editor designer. Extra state is discussed in more detail in sections 4.2 and 5.2

2.2.7 Summary

Summarizing, to support all five use cases, a generic structure editor must meet the following requirements.

- Genericity.
- Support for any computation over the document.
- A graphical presentation language with a powerful mapping formalism.
- Support for both presentation-oriented and document-oriented editing
- Modeless editing.
- Support for presentation extra state as well as interpretation extra state.

The requirements above all apply to the edit model, but of course many other requirements exist for a generic structure editor. Commonly recognized requirements for editors, which we will not discuss in detail, include: undo functionality, multiple window support, search/replace functionality, and a help facility.

2.3 Overview of structure editors

Because of the large number of existing systems, we can only mention a selection of editors in this overview. The editors mentioned are some of the early systems, together with a number of other editors that contain novel features.

It is important to note that the systems are discussed with the requirements from Section 2.2 in mind. If an editor does not meet our requirements or has been left out of the discussion, this does not necessarily say anything about the quality of that system. Many generic editors were designed to support a particular class of editor applications (e.g. program editors) rather than the entire range of use cases from Section 2.1. Moreover, many of these systems have interesting features that are orthogonal to our requirements, and the techniques for supporting these features may be applicable to Proxima editors as well.

2.3.1 Syntax-directed editors

Most of the editors in this section are specifically designed for program editing and hence have a rather text-oriented presentation formalism. Moreover, the computation formalism in such editors is aimed mainly at analyzing source code, and not at performing general-purpose computations.

Most syntax-directed editors allow partial presentations of the document, and hence offer support for interpretation extra state. On the other hand, presentation extra state is only found in built-in tree views on the document structure.

Synthesizer Generator

The Synthesizer Generator [77] is the successor of the Cornell Program Synthesizer [86], one of the early syntax-directed editors. Because the system is targeted at programming languages, the presentation is simple and text-only, although newer versions have some font and color control.

An interesting aspect of the Synthesizer Generator is its support for computations over the document structure. The presentation of the document can contain computed values, which are specified using an attribute grammar.

The edit model supports user-specified transformations on the structure, but plain text editing is poorly supported. The editor uses mode-switching, and after switching to the textual mode, the presentation must be left in a parsable state before structure editing is available again.

Over the years, the behavior and design have not undergone many drastic changes, but the system is still being used and commercially maintained.

LRC

The LRC attribute-grammar system [78] was a research project at Utrecht University. Higher-order attribute grammars are used to specify the derived values, as well as the presentation. The system is based on an efficient higher-order attribute-grammar evaluator. Higher-order attribute grammars allow some computations to be specified more elegantly than regular attribute grammars.

For the presentation of the document, the Tcl/Tk language is used. This allows for complex presentations with multiple windows, GUI widgets, colors, and basic graphical elements. However, the integration between the generated presentation and the editor is rather weak. No general focus model is present, and although edit events can be attached to the Tcl presentation, free editing is only possible in a separate window that contains a purely textual presentation of the document. The textual presentation cannot be used to edit the layout of the main presentation, and it does not contain derived values.

SbyS, Mjølner/Orm

SbyS is the structure editor of the Mjølner/Orm environment [54]. Mjølner/Orm is a generic language and software development environment. An interesting aspect of the environment is that it is truly a generic environment, since language descriptions can be changed without the need to recompile or regenerate the editor. In contrast, most of the other systems are editor generators.

SbyS supports textual editing only for entering expressions. In order to overcome the usability problems associated with pure syntax-directed editing, the editor employs the concept of direct manipulation. Program constructs are shown in a palette, from which they can be dragged to the program source or a clipboard.

No formalism for specifying transformations is present, and the only computations that can be specified are aimed at semantic analysis and code generation. Derived values cannot be part of the presentation.

PSG

PSG (Programming System Generator) [6] is a generator for language-based interactive environments, developed at the Technical University of Darmstadt. As the name suggests, the system is designed for programming languages. The presentations are text-only, and only LL(1) grammars are supported. The system generates an editor based on a number of formal descriptions for a language, including a syntax definition, a presentation sheet (called a *format syntax* in PSG), and a specification of the semantic analysis.

Special focus has been put on incremental analysis over incomplete program fragments. PSG uses a special form of the attribute-grammar formalism that supports sets of possible attribute values in order to handle attribution of incomplete document fragments.

However, the presentation may not contain derived values or structures. And although textual editing takes place in the same view as document-oriented editing, this does involve a mode switch. Furthermore, layout information cannot be edited freely, but is determined by the presentation sheet.

Other syntax-directed editors

Other textual syntax-directed editors for program editing are the Aloe editor in Gandalf environment [65], Mentor [24], its successor Centaur [10], Pragmatic [13], Poe [26], Dose [42], Gnome [31], Pecan [74], Muir [64], and Dice [29]. These systems have their own interesting aspects, but as far as the editors are concerned they do not deviate much from the systems already discussed, and hence are not discussed separately. Some more exotic editors that do not support editing on the presentation are Multiview [73] and VL-Eli [44].

2.3.2 Syntax-recognizing editors

Similar to the syntax-directed editors, most syntax-recognizing editors are designed for program editing. Regarding the computations, however, due to the difficulty of free editing in a presentation with derived values, none of the syntax-recognizing editors support arbitrary computations that may appear in the presentation.

Regarding extra state, syntax-recognizing editors are the opposite of syntax-recognizing editors: several built-in forms of presentation extra state (e.g. whitespace) are supported, but interpretation extra state is not.

Pan

Pan [7] is a text-only source editor environment. The presentations are text in multiple fonts, styles, and colors. The system has good support for handling partially incorrect or incomplete documents.

The computation formalisms in Pan are oriented towards semantic analysis. Logical constraint grammars are used for specifying, checking, and maintaining contextual constraints. Computed information is shown in the presentation by changing the font and color attributes of the text, but it is not possible to specify arbitrary computations that form part of the presentation. Furthermore, the editor does not support interpretation extra state. Hence, it is not possible to specify an editable presentation that shows only part of the document (e.g. a presentation in which function bodies may be hidden), as the editor is syntax-recognizing, and therefore the presentation must contain all information necessary to derive the document structure.

Pan offers some document-oriented editing, but edit operations on document structures are performed by editing the corresponding parts in the presentation and reparsing the presentation. Edit operations that modify the document structure directly are not supported, as these are believed to confuse the user. As a consequence, only basic document-oriented edit operations such as cut and paste are supported, and no document transformations can be specified. Free text editing, on the other hand, is fully supported, including layout editing.

GSE, ASF+SDF

The GSE [49] editor has been developed as part of the Esprit project “Generation of Interactive Programming Environment” (GIPE). It is part of the ASF+SDF meta environment [46] and under active development. The editor is primarily aimed at programming languages and the presentations are assumed to be lines of text. GSE supports free editing of the program text without an explicit mode switch. A powerful transformation formalism is available for specifying document edit operations that keep intact the layout [15]. On the other hand, such transformations cannot form part of the presentation process.

Ensemble

The Ensemble project is a successor to Pan, based on the recognition that structure editing cannot only be used for program editing, but also for editing documents of a more graphical nature, such as documentation. The system handles compound documents containing subdocuments of different types, and provides document management functionality, such as versioning.

Ensemble specifies formalisms for performing incremental semantic analysis, but arbitrary computations appearing in the presentation cannot be specified. However, some support for derived structures is present in the presentation formalism.

Ensemble has a powerful graphical presentation formalism, including a constraint-based box layout. The presentation specification language, however, does not elegantly allow presentations with a structure different from the document. The presentation formalism may be used to specify derived structures, but these are not editable.

The edit model supports modeless free text editing, including layout editing, as well as structural editing.

The Ensemble project has been terminated, but its successor, Harmonia [12], is still under development. Because the monolithic character and ambitious design requirements of Ensemble slowed down its development, Harmonia is a framework for incremental language analysis rather than a single editor generator. The services from Harmonia can be used to augment text editors, such as Emacs, with language-aware editing and navigation functionality.

Desert

Built with the experience of the FIELD [75] project, Desert [76] is a syntax-recognizing editor generator that uses the commercial editor system Framemaker for editing program sources. The system has many facilities for software development, including database facilities and an interface for easily defining (non-editable) software visualizations. The actual editor is a syntax-recognizing editor with attributed text and images in the presentation. However, no structural edit operations, or derived structures in the presentation are supported.

Other syntax-recognizing editors

Other syntax-recognizing editors similar to the ones that were discussed include Babel [35], Saga [19], and Pragmatic [13].

2.3.3 Editor toolkits

Besides generic editors and edit generators, an editor can also be built using an editor toolkit. The toolkit is a collection of libraries and tools that can be used when building an editor. The editor application itself, however, has to be written by hand. The distinction

between a toolkit and a generator is not always completely clear, since the specifications that an editor generator uses for specifying language, presentation, and semantics can be considered programs as well. The toolkits we consider here all require a substantial amount of programming in order to build an editor.

The advantage of a toolkit is that the final editor can be customized to a high degree, but this comes at the cost of the increased effort required for building an editor.

Amaya, Thot

Amaya [94] is the W3C web browser that is built on top of the editor toolkit Thot [71], which is a successor of Grif [72]. The Thot toolkit supports a number of specification languages for document structure, presentation, and transformation, but in order to build an actual editor C code is required to connect the various components.

The presentation formalism in Thot, called P, is a powerful graphical presentation formalism, somewhat similar to Proteus (Ensemble), but with more advanced alignment features. As a result, complex presentations are possible, such as the presentation for the equation editor use case.

Thot editors are of a syntax-directed nature. Multiple views on the document may be edited simultaneously, and user-specified transformations are supported. However, free text editing can only be done in a separate window in a different mode. Also, no computations are supported other than some basic counters in the presentation.

Visual Studio editor

The Microsoft Visual Studio environment includes an integrated source editor. Although the editor does not contain any novel features, and thousands of lines of code need to be written to tailor the editor for a specific language, we do include it in the discussion because it is a structure editor that is actually used by a rather large number of people.

The Visual Studio editor is of the syntax-recognizing kind with colored-text presentations. No document-oriented edit functionality is supported, other than performing semantic analysis and displaying the results. These results are displayed by marking a location in the source with a squiggly line, and displaying a corresponding message in a separate window pane as well as in a tooltip. Pop-up list boxes can be used to show auto-completion alternatives. Despite its simple model, in which semantic analysis is only possible when the entire presentation is syntactically correct, the editor provides a surprisingly usable environment.

2.3.4 XML editors

A large number of XML editors have been developed, but the differences between them are not fundamental. Almost all XML editors classify as pure structure editors with mode switching.

Because the differences are small, we discuss XML editors in general with respect to our functional requirements. Afterwards, two editors are discussed separately.

Genericity. The XML editors are generic. Most reviewed editors are actual generic editors, rather than editor generators, and support editing of documents with arbitrary DTDs. Although, compared to context-free grammars, DTDs have a few restrictions in order to make parsing easier [17], the type language is very similar to the EBNF grammar description formalism and powerful enough to describe the tree-based document structures we wish to edit.

Computation formalism. Support for computations is very weak for all reviewed editors. A few editors support basic numbering of elements in the document, but no arbitrary computations can be specified. Some editors support the transformation formalism XSLT, but none provide an editable view on the resulting transformed document.

Presentation formalism. Most XML editors only provide standard views on the document. Popular are the raw-text XML source view, a built-in tree view showing the document structure with the textual content in the leaves, and a slightly less raw view with tags, represented using a more graphical presentation.

Some editors support a user-defined presentation, or at least allow the user to specify some attributes for the presentation. However, the presentation formalisms are generally weak, and the presentations that can be used for editing have to follow the structure of the XML document. Moreover, there is hardly any support for textual presentations, making it impossible to present an XML tree that represents an abstract syntax tree as actual program source code.

It is remarkable that support for textual presentations of XML documents is this weak, since many languages for processing and describing XML documents are specified in XML itself (e.g. XML Schema [8, 87] and XSLT [21]) and editing these languages would be greatly simplified by providing the user with a concise concrete syntax, rather than the verbose XML syntax.

Editing power. Most XML editors offer simple document-oriented edit operations for structure entry and manipulation. However, none of the reviewed editors support user-specified transformations on the tree structure.

In each of the editors, free text editing is supported only in the raw XML source. Because most XML documents have text and whitespace in the leaves, it may appear that the document-oriented edit operations are free text editing, but this is not the case. Textual presentations other than the source presentation cannot be edited freely. On the other hand, as mentioned, most XML editors offer little support for textual presentations of the document.

Modeless editing. None of the editors support free editing on the presentation without a mode switch. Each type of view has a separate window, and though some editors have a shared undo history for some of the views, no editor has a shared undo

history for the XML source presentation and its other presentations. Hence, after switching to source mode, previous edit operations on other views cannot be undone, and vice versa.

Extra State. The XML editors support extra state similar to the syntax-recognizing editors. Interpretation extra state is supported in the form of partial presentations, but presentation extra state is only found in built-in tree views.

Two XML editors have a more sophisticated presentation engine and basic support for computations, and are therefore discussed separately.

X-Metal

The commercial system X-Metal from SoftQuad is a highly customizable XML editor, with support for many XML standards and database connectivity. Besides regular source and outline views, it offers built-in table editing and an editable CSS [11] presentation of the document. CSS provides a quick and easy way to specify a document presentation, but its expressive power is limited. Although general computations cannot be specified, CSS does allow the specification of basic counters in the presentation.

Document-oriented edit operations in X-Metal are rather weak, and transformations cannot be specified. Furthermore, the freely editable source presentation can only be edited in a separate mode.

XMLSPY

XMLSPY is a large system that has functionality similar to X-Metal. An important difference is the presentation system. XMLSPY supports a larger number of built-in presentations and also supports a user-defined presentation definition for the specification of simple derived structures. Values from the document that appear in the derived structure may be edited in place, but the structure itself is not editable.

2.4 Discussion

Figure 2.1 contains an evaluation of the strengths and weaknesses of each of the discussed editors according to the requirements from Section 2.2. None of the editors scores a positive for extra state, and besides that, each of the editors has at least one or more columns with a low score (\pm or less).

The reason why no editor scores positive on the extra state requirement is that, although it is rather straightforward to support either presentation or interpretation extra state, it is hard to support both forms. A syntax-directed editor without support for presentation-oriented editing may support interpretation extra state simply by ignoring it during presentation. Similarly, a syntax-recognizing editor without document-oriented editing may support presentation extra state by ignoring it during interpretation. The syntax-recognizing

	Genericity	Computation formalism	Presentation formalism	Editing power	Modeless editing	Extra state
Synthesizer Generator	++	++	±	+	--	±
LRC	++	++	+	+	--	±
PSG	++	+	--	±	+	±
SbyS	++	-	--	-	n/a	±
Pan	++	±	-	±	++	-
GSE	++	--	--	+	++	-
Desert	++	--	±	±	--	-
Ensemble	++	±	+	+	++	-
Amaya, Thot	+	±	+	+	--	-
Visual Studio	±	±	-	-	n/a	-
XMetal	++	-	±	±	--	±
XMLSPY	++	±	+	±	--	±
Other XML editors	++	max. -	max. ±	±	--	max. ±
Proxima	++	++	++	++	++	++

Figure 2.1: Editor comparison

editors score lower on extra state than the syntax-directed editors because only built-in forms of presentation extra state are supported, whereas the interpretation extra state for the syntax-directed editors may be specified by the editor designer.

The main reason why no editor has a line containing only positives is that the requirements for the computation and presentation formalisms interfere with the requirements for editing power and modelessness. The former two requirements determine the presentation complexity of the editor, whereas the latter determine the usability of the editor. A problem is that the more complex a presentation is, the harder it will be to still offer modeless free editing on the presentation level.

Syntax-directed editors. The syntax-directed editors tend to do well on the computation requirement, but at the same time, presentation-oriented editing is weakly supported, leading to a lower score on editing power. Furthermore, modelessness is not supported at all. However, if the presentation formalism is simple, and no computed values appear in the presentation, then modelessness can be supported (see PSG). Most syntax-directed editors support interpretation extra state, but none support presentation extra state adequately.

Syntax-recognizing editors. The syntax-recognizing editors on the other hand do well on the presentation-oriented editing and modelessness requirements, but the fact that a document is derived from its presentation has a number of consequences. Firstly, the presentation must at all times contain sufficient information to derive the document, which puts restrictions on the presentation formalism. Secondly, having derived values and structures in the presentation makes parsing a lot harder and is therefore not supported, hence the low scores on the computation formalism requirement. And finally, edit operations on the document are harder to implement. As a result, syntax-recognizing editors do not score maximally in the computation, presentation, and editing power columns. In contrast to syntax-directed editors, the syntax-recognizing editors support a form of presentation extra state, but lack interpretation extra state support.

XML editors. XML editors are similar to syntax-directed editors, but somehow the computation and presentation formalisms are not very well developed. Semantic analysis is, of course, not an essential requirement for an XML editor, but computations and derived structures have many applications also for XML editing. Furthermore, specification of a textual presentation with a parser is not supported, which is odd because the raw XML source has an extremely verbose syntax that is far from suitable for viewing or editing directly.

Although some XML editors have support for graphical presentations, the presentation specification formalisms are generally weak, disallowing the structure of the presentation to be different from the structure of the document. Hence, there exists a strong connection between an XML document and its presentation. A tree-structured document with text in the leaves lends itself well for editing with an XML editor, but other structures are harder or impossible to edit. An example is an XML representation of an abstract syntax tree, or a paragraph that is represented by a list of word elements. Current XML editors cannot handle such documents.

The close link between the XML document and its presentation sustains the view that an XML document is a piece of text with markup tags added to it. In this view, the current XML editors provide sufficient edit functionality. However, if a more powerful editor is available which releases the tight connection between a document and its presentation, the view might change, causing new applications for XML to arise.

Discussion

We already mentioned that a negative score in the evaluation table does not suggest that the editor in question is an inadequate system, but only that it is not suitable for our purposes. Many of these systems were simply designed with a different scope.

Another reason why some systems may look rather bad is that our requirements primarily concern the edit model. Several of the evaluated systems have been designed with a large number of other requirements in mind, which are not taken into account here because they are concerned more with the environment than with the editor. Structure editors often have many facilities for managing and versioning documents, as well as complex semantic analysis methods, whereas XML editors often offer built-in XSLT viewers, DTD viewers or editors, and database connectivity, as well as support for the many standards existing in the XML world. However, we do not view these requirements as being essential for the design of a generic structure editor.

Summarizing, the current and previous generations of structure editors are not powerful enough to edit the five use cases of Section 2.1. The editors either lack flexibility to express the required presentations, or have an edit model that is overly restrictive, or even suffer from both of these problems.

2.5 The Proxima editor

Proxima will be able to handle all five use cases from Section 2.1. It is designed according to the requirements from Section 2.2.

The editor uses an attribute-grammar formalism for performing semantic analysis, as well as for specifying derived structures and values, which may appear in the presentation. The presentation formalism supports graphical presentations and a box layout model with alignment, strong enough to specify presentations of mathematical equations. Furthermore, edit operations may be targeted at both presentation and document level, as well as at derived structures, without mode switching.

In order to support both presentation- and document-oriented editing, as well as presentation and interpretation extra state, the editor keeps track of bidirectional mappings between the document and its presentations. The layered architecture, which breaks up the presentation process, as well as the handling of edit operations in a number of steps, facilitates the process of keeping the mappings consistent.

An editor in Proxima is specified by a number of sheets that specify the computations, the presentation, the parser (inverse of the presentation), and the reducer (for handling edit operations on derived values and structures). The languages of the editor sheets are declarative and have a strong abstraction formalism, which helps to keep the specification of simple behavior short, while still allowing the specification of complex behavior as well.

2.6 Conclusions

Source editors, word processors, and equation editors can all be seen as possible applications, or instances, of a generic editor. The same thing holds for more exotic applications such as a tax form, or an outline editor or tree browser. However, no existing editor is able to handle this range of applications. We believe the reason for this is that existing editors lack complexity in presenting documents and/or have an edit model that is overly restrictive. Or, more precisely, because no editor meets all six of the following requirements.

- Genericity.
- Support for Turing-complete computations over the document.
- A graphical presentation language with a powerful mapping formalism.
- Support for both presentation-oriented and document-oriented editing.
- Modeless editing.
- Support for presentation extra state as well as interpretation extra state.

In contrast, the Proxima editor is designed meet all six requirements and thus will be able to handle all five use cases. In order to meet the requirements, Proxima makes use of the following concepts:

- A layered architecture.
- Bidirectional mappings between document and presentation.
- Concept of presentation/interpretation extra state on several levels of the presentation process.
- Declarative specification languages with strong abstraction mechanisms for specifying mappings between levels.

The use of many of the features of Proxima is optional rather than forced. Edit operations on derived structures may be specified or automatically derived in cases for which they make sense, but they may also be omitted. A similar thing holds for the extra state. Supporting extra state in a Proxima instantiation requires an effort of the editor designer, but if no extra state is present, no such effort is required.

Architecture of the Proxima editor

A generic editor is a large system consisting of many components. In this chapter, we focus on those components of the Proxima architecture that are involved in the process of presenting the internal document to the user, and interpreting the edit gestures given by the user. Of course, other functionality, such as IO handling, macro processing, or a search facility, is also important for the usability of the final editor, but implementation of these features is largely straightforward, despite the fact that it may require a substantial amount of engineering. The presentation of the document and the handling of edit gestures, on the other hand, is of paramount importance, because it determines for which applications the editor can be used, as well as how powerful the editing behavior will be.

The core architecture of Proxima consists of a number of layers, which only communicate with their direct neighbors. This layered structure is based on the staged nature of the presentation process. Instead of mapping a document directly onto its final rendering, it is first mapped onto an intermediate data structure. This intermediate data structure is mapped onto another intermediate data structure, until the last intermediate data structure is mapped onto the rendering.

As we mentioned in Chapter 1, the positions at which the document, the rendering, and the intermediate data structures reside are called *data levels*. Between each pair of levels is a *layer*, which is a component that maintains the mappings between the levels. Figure 3.1 schematically shows the levels and layers of Proxima. Only two data levels are visible to each layer: a higher and a lower level.

There are several reasons why the Proxima architecture is layered:

Staged presentation process. The presentation process is naturally staged. The process consists of repeatedly mapping structures that have a different meaning on a higher

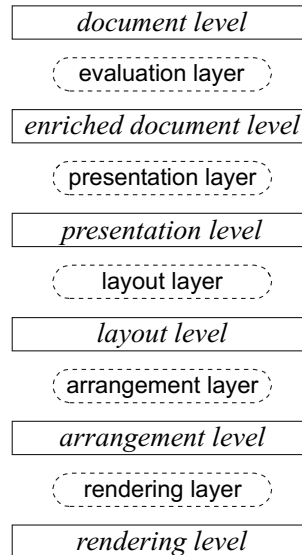


Figure 3.1: The levels and layers of Proxima.

level onto a common set of lower level structures. For example, a table of contents, once its structure has been computed, can be presented in the same way as a chapter structure. Similarly, a line that comes from a formatted paragraph is rendered in the same way as a line that was explicitly specified in the presentation. Mappings like these form stages in the presentation process that can be performed by separate layers.

Specification of presentation and edit behavior. A layered architecture provides natural hooks for the editor designer to specify specific parts of the presentation and edit behavior. A separate evaluation layer, makes it possible to separate computation and presentation, thus allowing different style sheets to be used for a document together with its derived structures. At the same time, layers offer more control over backward mappings, such as the specification of how edit operations on derived structures are to be interpreted as edit operations on the document.

Extra state. An important aspect of the Proxima editor is the concept of *extra state*, which is inherently connected to a layered architecture. If information on a level cannot be computed by presenting the level above, it is presentation extra state, and if it cannot be computed by interpreting the level below, it is interpretation extra state.

Maintaining bidirectional mappings. Because Proxima supports editing on all levels, a mapping between each pair of levels needs to be maintained. Maintaining such

mappings is easier in a layered architecture. Furthermore, the lower layers can maintain the mappings automatically.

Efficiency. Some steps in the presentation process, especially in the higher layers, may be time consuming because global computations need to be performed. In a layered architecture, it is possible to perform the higher layer computations not at every keystroke, but only once in a while. For example, in a program editor, parsing the program may be delayed until the user enters a whitespace character or performs a navigation operation. Type checking the program may be delayed until after a certain period of inactivity, or when requested by the user.

The remainder of this chapter contains an informal description of the levels and layers in the Proxima architecture. A more formal specification of the editor is provided in Chapter 5, preceded by an informal introduction in Chapter 4.

3.1 The levels of Proxima

A data level in Proxima is not just an intermediate value in the presentation computation, but an entity in its own right. Together, the data levels constitute the state of the editor. The six data levels of Proxima are:

Document: The edited document, the type of which is specified by a DTD or an EBNF grammar.

Enriched Document: The document enriched with computed information.

Presentation: A logical description of the presentation of the document, consisting of rows and columns of presentation elements with attributes. The presentation also supports formatting based on available space (e.g. line/page breaking).

Layout: Presentation with explicit whitespace.

Arrangement: Formatted presentation with absolute size and position information.

Rendering: A collection of user interface commands for drawing the absolutely positioned and sized arrangement.

In the discussion below, we illustrate the different data levels with examples. Note that when an element at one level is mapped onto elements at another level, an implementation will have to keep track of information about this mapping. For simplicity, the details regarding such mapping information have been left out of the examples.

3.1.1 Document

A document is the internal tree data structure that is edited by the user. The type of the document is specified by a context-free grammar, with special constructs for lists and optional values (similar to EBNF). Haskell data types, EBNF, DTDs and XML Schemas are all, possibly restricted, forms of context-free grammars suitable for describing the document type. In this thesis, we make use of simple monomorphic Haskell data types together with the list type.

The exact type formalism is important for document to document transformations (i.e. document-oriented edit operations), because it should be possible to guarantee type safety of such transformations. However, for the time being, the only supported document-oriented edit operations are simple tree-based operations, such as cut and paste, and basic operations on lists, such as selecting a segment of a list. Therefore, using a context-free grammar formalism for specifying the document structure is expressive enough.

Because different instances of a generic editor may have different document types, we give an example document type for a specific instance. The example document consists of a list of declarations, each of which is an identifier declaration or a comment. An identifier declaration contains a string that represents the declared identifier, as well as an expression that consists of conditional expressions, integers, and booleans. The third field of the declaration is a string that contains additional information about the declaration. It is used to illustrate the concept of interpretation extra state in the Proxima editor. A comment consists of a list of strings. The types `String`, `Int`, and `Bool` are primitive types. Although not very suitable for practical purposes, the chosen document type allows us to illustrate the different aspects of the Proxima data levels and layers.

```
data Document = RootDoc [DeclDoc]
data DeclDoc = DeclDoc String ExpDoc String
              | CommentDoc [String]
data ExpDoc = IfExpDoc ExpDoc ExpDoc ExpDoc
              | IntExpDoc Int
              | BoolExpDoc Bool
```

The explanation of the presentation process in Section 3.4 provides an example document of this type, as well as examples of the lower levels.

3.1.2 Enriched document

An enriched document is a copy of the document to which derived information has been added. In the word processor example, the enriched document contains a table of contents, and each section or subsection element has a field that contains its number. Such derived information is not present at the document level.

Besides containing extra information, the enriched document, or a subtree of it, may also be a reordered version of the document. For example, if the document contains a list of elements, the enriched document may contain a sorted list of these elements.

As an example of an enriched document type, we take the document type of the previous section and add a type declaration alternative `TypeDecl` to the `Decl` type. The type declaration is inferred for each declaration. The type of an expression may be integer, boolean, or erroneous (e.g. the type of `if True then 0 else False`). It is also possible to add the type as a field to the `Decl` alternative, but separate type declarations make it easier to show how edit operations targeted at the enriched document are handled in Section 3.5.5.

```
data EnrichedDocument = RootEnr [DeclEnr]
data DeclEnr = TypeDeclEnr String TypeEnr
             | DeclEnr String ExpEnr String
             | CommentEnr [String]
data ExpEnr = IfExpEnr ExpEnr ExpEnr ExpEnr
             | IntExpEnr Int
             | BoolExpEnr Bool

data TypeEnr = IntTypeEnr | BoolTypeEnr | ErrorTypeEnr
```

3.1.3 Presentation

A presentation is an abstract description of what the document will look like to the user. It consists of strings, images, and simple graphical elements (lines, boxes, etc.), which are grouped in rows, columns and matrices. A presentation element has attributes for colors, line styles, fonts, and alignment. Attribution may be influenced using a *with* element, which contains a rule that specifies how the attribution is affected.

There are three ways of positioning elements in the presentation. Firstly, the position can be specified relative to other elements in the presentation, by placing a list of elements next to each other in a *row*, or above each other in a *column*. Elements are aligned according to reference lines (e.g. the baseline for a string), and stretchable elements may be used to influence the positioning. Besides rows and columns, a *matrix* construct presents a list of lists of elements aligned both horizontally as well as vertically, and an *overlay* presents a list of elements in front of each other (e.g. for presenting a squiggly line).

The second way of positioning presentation elements is by using a *formatter* element, which positions a list of elements based on the available space. Currently, Proxima only supports horizontal formatting, suitable for line breaking in a paragraph. Vertical formatting (for page breaking) is not fundamentally different, but has not been implemented yet. Furthermore, support for a page model also requires extensions to the lower levels, which have not been realized yet.

Finally, a presentation (or part of it) may consist of a list of tokens, which may be identifiers, operators, integers, strings, etc. Token-list presentations support presentation ori-

ented editing; on interpretation, the (possibly edited) token list is parsed. Each token contains information about the whitespace (line breaks and spaces) that precedes the token in the presentation. This whitespace is an example of presentation extra state, since it is not stored in the enriched document.

A presentation consisting of tokens may contain parts that we do not want to be parsed. For example, a textual source editor may have a non-textual presentation for fractions ($\frac{1}{1+x}$). Such a presentation can be included in the token list presentation with a *structural token*. A structural token contains a presentation, and is treated specially by the parser. Although a structural presentation itself is not parsed, it may contain other token lists that will be parsed (e.g. the numerator and denominator of the fraction may be parsed again).

Unlike the document and the enriched document, the presentation has a fixed type, of which we present a slightly simplified subset here. The details of the attribution (e.g. color, font, and reference lines) of presentation elements have been left out by leaving the type `AttributionRule` abstract. Chapter 6 provides a discussion of these details.

```
data Presentation = EmptyPres
                  | StringPres String
                  | TokensPres [Token]
                  | RowPres [Presentation]
                  | ColumnPres [Presentation]
                  | OverlayPres [Presentation]
                  | MatrixPres [[Presentation]]
                  | FormatterPres [Presentation]
                  | WithPres AttributionRule Presentation

data Token = UCaseToken Whitespace String
           | LCaseToken Whitespace String
           | IdentToken Whitespace String
           | OpToken Whitespace String
           | IntToken Whitespace Int
           | StructuralToken Whitespace Presentation
           ...

type Whitespace = (LineBreaks, Spaces)
type LineBreaks = Int
type Spaces = Int

data AttributionRule = ...
```

3.1.4 Layout

The layout level is the same as the presentation level, except that there are no tokens anymore. Each token has been replaced by its string, and its whitespace has been replaced by strings of spaces and by starting a new row for each line break. Thus, at the layout level, all whitespace is explicit. Formatters are still present, because the exact size and position information required to remove them is not known at the layout level.

The similarity between the layout and the presentation level is clearly visible in the types: the `Layout` type is the `Presentation` type without the `Tokens` alternative.

```
data Layout = EmptyLay
           | StringLay String
           | RowLay [Layout]
           | ColumnLay [Layout]
           | OverlayLay [Layout]
           | MatrixLay [Layout]
           | FormatterLay [Layout]
           | WithLay AttributionRule Layout
```

```
data AttributionRule = ...
```

3.1.5 Arrangement

At the arrangement level, each element gets its position and size. The position is expressed in actual coordinates. These coordinates do not necessarily correspond to pixel coordinates because the rendering may be scaled.

Because the final positions have been determined, the distinction between rows, columns, etc. is no longer necessary at the arrangement level. Instead, all composite elements are represented by a `Node`, which contains a list of child arrangements.

Formatters have been resolved and are represented by a single node containing a list of nodes that represent the formatted lines (see Section 3.4.4 for an example). The neutral empty element of the layout is not visible and therefore not part of the arrangement type.

The arrangement has a tree rather than a list structure to enable the specification of properties, such as font and color information, for an entire subtree, instead of separately for each element of the arrangement. Moreover, a tree structure is helpful for incremental arranging, in which only part of the arrangement is recomputed after an edit operation, whereas the rest is simply copied from the previous arrangement.

Although the coordinates of each element in the arrangement are absolute, the coordinates are represented in the arrangement data structure relative to the coordinates of the parent in the tree. Relative positioning makes it easier to reposition a subtree in the arrangement, because it removes the need to update the position of each element in the moved subtree.

A `With` node does not have a geometry field, because it only influences the attribution of the arrangement tree, without being an actual part of the arrangement.

```
data Arrangement = StringArr Geometry String
                 | NodeArr Geometry [Arrangement]
                 | WithArr AttributionRule Arrangement

type Geometry = (Position, Size)
type Position = (Int, Int)      -- (x, y)
type Size     = (Int, Int)      -- (width, height)

data AttributionRule = ...
```

3.1.6 Rendering

A rendering is a set of user-interface drawing commands that actually draw on the screen. Positions are expressed in pixel coordinates. In contrast to the other levels, a rendering is a list rather than a tree. However, in the future it may change to a tree structure, to support incremental updates on the rendering.

Because the rendering is highly dependent on the GUI library that is used, we only give an abstract type.

```
type Rendering = [RenderingCommand]
data RenderingCommand = ...
```

3.2 Editing on different levels in Proxima

Before we proceed with a description of the layers, we briefly discuss which levels can be edited in Proxima. When a level is targeted by an edit operation, the edit operation is directly performed on that level. Afterwards, the other levels are updated indirectly, as a result of the interpretation and presentation processes. To offer document-oriented as well as presentation-oriented editing, several levels may be targeted in Proxima. However, on some levels it does not make much sense to support direct editing, whereas on others the semantics of direct editing is not clear. We briefly discuss how each level may be edited, starting with the rendering.

Rendering

A direct edit operation on the rendering would consist of moving around bitmaps, after which the updated rendering needs to be interpreted to yield a new arrangement. The semantics of such edit behavior are unclear and supporting it will probably be difficult,

and since direct rendering editing is not required by any of the use cases that drive the design of Proxima (Section 2.1), it is not supported.

Arrangement

Proxima does not support direct editing at the arrangement level. It is not possible to update the arrangement level and compute an edit operation on the presentation level. Edit operations targeted at the arrangement only involve the focus. A consequence is that although rectangular areas may be selected and deleted (the deletion takes place at a higher level), it is not possible to insert a rectangular area. Such edit behavior, often referred to as column editing in text editors, is not a strong requirement for Proxima. In text editors, column editing is used to edit a text that is organized in columns, but since Proxima is a structure editor, a document that has data that needs to be presented in columns can use a presentation-level matrix, the elements of which can be selected column-wise at presentation level.

Layout

The layout level is the lowest level that can be edited directly. All presentation-oriented editing takes place at the layout level. This includes entering program text in a source editor, but also text entry in the tax form editor or the word processor. Besides text insertion and deletion, also cut, copy and paste operations are supported.

Presentation

The presentation level does not need to be edited directly because of its direct correspondence to the layout level. Any edit operation that needs to be performed on the presentation level can be performed on the layout level.

Enriched document and Document

Because the enriched document and the document are both trees, the edit functionality on both levels is similar. The only difference between the two is that if the enriched document is edited, a subsequent interpretation takes place to compute the document update. Because of the similarities, both levels are discussed together.

The edit operations at the document level are basic tree operations, such as cut, copy and paste on subtrees. If a list element has a parent that is also a list (eg. a list of subsections in a list of sections), split and join operations can be applied. An editor designer may specify additional generic or domain-specific transformations.

For any element except a list, all child elements are required to be present and hence cannot be left out of the parent element. In order to still be able to manipulate an element without its children, Proxima employs the concept of a placeholder (also found in the Synthesizer Generator [77]). A placeholder of a type T is a dummy value that can be used in any place where an element of type T is required. A document containing a placeholder

is incomplete. The placeholders are typically only present during the construction of a document (or part of it), or as an intermediate situation during a document modification that consists of several steps. Lists and optional types do not require placeholders: for a list type, the empty list is the placeholder, and for an optional it is the empty alternative.

Summarizing, the following three levels may be directly edited in Proxima:

Layout. Mainly text-oriented editing, such as entering keywords (e.g. "if", "then", or "else") or moving the focus downward one line.

Enriched Document. Tree edit operations on derived structures, such as moving an entry in a generated table of contents.

Document. Tree edit operations, such as moving a section in word-processor document; deleting a declaration in Haskell source; or moving the focus to a certain subtree of the document.

3.3 The layers of Proxima

Between each pair of data levels is a *layer* that takes care of mapping the higher level onto the lower level (downward mapping, or *presentation*) and that maps edit operations on the lower level onto edit operations on the higher level (upward mapping, or *interpretation*). A layer consists of two components: one for presentation, and one for interpretation.

In the actual architecture, the downward mapping is not a mapping between the higher and lower levels, but between edit operations on those levels, similar to the upward mapping. Hence, the upward and the downward mappings are symmetrical, as both are concerned with edit operations.

Besides mapping edit operations onto edit operations, a layer also takes care of updating the data levels by performing the edit operations on them. Data levels are updated on presentation as well as on interpretation. Each layer only updates one of its neighboring levels, because otherwise levels would be updated twice (each level is surrounded by two layers). For implementation reasons that we will not discuss here, a layer component updates its argument level; the presentation component updates its higher level, whereas the interpretation component updates its lower level.

Figure 3.2 contains a picture of a single layer. In the figure, $level_H$ and $level_L$ denote the higher and lower levels, and δ_H and δ_L denote the edit operations on these levels. The present component computes δ_H from δ_L , $level_H$ and $level_L$. And, dually, the interpret component computes δ_L from δ_H , $level_L$ and $level_H$. The squiggly arrows (\rightsquigarrow) denote that present updates $level_H$ and interpret updates $level_L$.

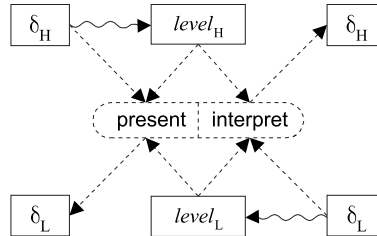


Figure 3.2: A single Proxima layer

Because each layer connects two levels and Proxima consists of six data levels, we have five layers, one for each of the following steps: evaluation, presentation, layout, arrangement and rendering.

Some of the components in the architecture are parameterized with so called *sheets*, which describe the specifics of a particular editor instance, similar to style sheets in a web browser. Together with the document type definition, the sheets determine the editor instance. Currently, the evaluation layer has an *evaluation sheet* and a *reduction sheet* and the presentation layer has a *presentation sheet* and a *parsing sheet*. The layout layer, on the other hand only has a *scanning sheet*, and the lower layers do not have any sheets at all.

The presentation sheet is specified by an attribute grammar, whereas the parsing sheet is specified using a Haskell parser combinator library. Section 7.2 provides example fragments of presentation and parsing sheets from the Proxima prototype. The formalisms for the evaluation, reduction, and scanning sheets have not been established exactly yet.

The reason why some components do not have sheets is that as of yet no sensible purpose has been found for them. The components without sheets can realize the required mappings without the need for parameters specific to a particular editor instance. However, a future version of Proxima may also support layout sheet and sheets for the lower levels. A layout sheet could be useful for specifying token based syntax coloring, whereas a rendering sheet could be used to specify rendering details, such as the rounding of corners in line drawings.

Figure 3.3 shows an overview of the layers of Proxima. For clarity, the arrows between the layers and the levels are omitted, and a single arrow is used to denote the arrows between the layers and the edit operations. Because the edit gesture is not an edit operation on the rendering, it has no update arrow (\rightsquigarrow). Furthermore, because the document only needs to be updated once, there is no update arrow on the top-right of the figure. Instead, δ_{Doc} is passed on to the evaluator, which performs the edit operation on the document.

The downward mappings from all layers together form a logical whole (the presentation process), as do the upward mappings (the interpretation process). Therefore, rather than

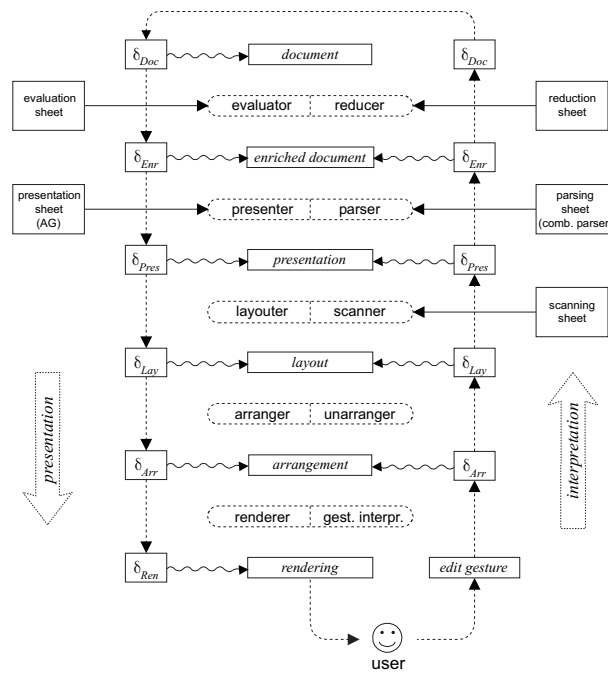


Figure 3.3: The layers of Proxima.

discussing the components pairwise from layer to layer, we give a description of all the components involved in the presentation process, followed by the components of the interpretation process.

The examples accompanying the component descriptions show a few instances of both presentation and interpretation extra state. Section 4.2 discusses extra state in more detail and Section 5.2 provides a formal specification.

3.4 Presentation process

The presentation process is the stepwise mapping of a document onto its final rendering. Although the mapping is actually a mapping between edit operations on each of the levels, we will present it here as a mapping between levels. The reason for this is that the mapping between edit operations is important mainly for incrementality, and viewing the presentation process as a mapping between levels makes it easier to explain.

In order to illustrate the different stages in the presentation process, we follow the presentation of a simple document. After the description of each layer component, the intermediate result of the presentation of the document is given.

The document type is the expression list document type from Section 3.1.1. The document itself consists of two items: a comment and an expression. For brevity, we denote strings on each level with `".."` instead of `StringLevel ".."`. Note that the comment is part of the document, unlike comments in regular programs.

Document:

```
RootDoc [ CommentDoc ["This", "is", "a", "simple", "expression"]
  , DeclDoc "simple1"
    (IfExpDoc (BoolExpDoc True) (IntExpDoc 1) (IntExpDoc 0))
    "info"
  ]
```

To a user, this document appears as:

<pre>This is a simple expression simple1 :: Int simple1 = if True then 1 else 0</pre>
--

3.4.1 Evaluation layer: Evaluator

The first step in the presentation process is the computation of the derived information in the document. The component that takes care of this is the evaluator. The evaluator is parameterized with an *evaluation sheet*, which is a declarative specification of the derived values. The evaluation sheet may be specified with an attribute grammar, but no final choice for the formalism has been made yet.

Besides basic values, such as section numbers or the outcome of a computation in a spreadsheet, the evaluator may also derive tree structures, such as a table of contents. The derived structures may be partial, duplicated, or reordered versions of the document.

Example: For each declaration, the evaluator computes a type declaration in the enriched document. In the example this means that a type declaration for "simple1" with type `IntType` is included in the item list.

Enriched document:

```
RootEnr [ CommentEnr [ "This", "is", "a", "simple", "expression" ]
  , TypeDeclEnr "simple1" IntTypeEnr
  , DeclEnr "simple1"
    (IfExpEnr (BoolExpEnr True) (IntExpEnr 1) (IntExpEnr 0))
    "info"
  ]
```

3.4.2 Presentation layer: Presenter

The enriched document is mapped onto the presentation by the presenter. Similar to the evaluator, the presenter is parameterized with a *presentation sheet* that specifies the presentation. The presentation sheet is an attribute grammar that defines the presentation as a synthesized attribute for each element in the enriched document.

In order to support extra state, the presentation layer keeps track of the mappings between the elements of the enriched document and the elements of the presentation. The presenter stores the enriched document origin in each presentation element, whereas the parser stores the presentation elements that were used for parsing in the enriched document.

Tokens are an example of presentation extra state in the presenter, because a token contains its own whitespace, which is not represented in the document or enriched document levels. If an enriched document structure is re-presented, the original tokens (and corresponding whitespace) are reused.

If an enriched document element that is presented with tokens, is newly created or has not been presented before, a default value for the whitespace of its tokens is chosen. This default may come from a pretty printing algorithm. A default value may also be used in case a structure has been edited in such a way that reusing the old tokens does not make sense.

The mechanism of reusing presentation tokens is also used to handle ambiguities in token representations. For example, when a user has entered the text “001” which is stored in the document as the integer 1, the mapping between the enriched document and the presentation ensures that on re-presentation, the integer is presented as “001” instead of “1”.

The separation between evaluating and presenting is more pragmatic than theoretical. On the one hand, the entire presentation can be regarded as a derived structure, and on the other hand the presentation sheet can reorder elements in the presentation and introduce structures, which is more appropriately done in the evaluation sheet.

The editor designer must make a careful decision on where to specify document evaluation and presentation. Whenever editing on a derived structure is supported, the computation of the structure must be part of the evaluation. But even if a derived structure is not intended to be editable, it may be wise to compute it during evaluation. Take for example a table of contents or a bibliography in a word processor. If the computation of such a structure is part of the evaluation, then its presentation can be changed without having to know the details of the computation.

Example: The example presentation of the enriched document is basic: a comment is put in a formatted paragraph and a (type) declaration is presented in a textual infix representation using tokens. The third string field of the declaration "info" is not included in the presentation in order to have an example of interpretation extra state. The pair of numbers in each token represents the whitespace (*line breaks, spaces*) preceding it. This is a rather basic representation, in which spaces at the end of a line cannot be encoded, but for demonstration purposes it is sufficient.

The `With` nodes specify the font for the presentation of the declarations. To keep things simple, the exact details of the attribution rule are not shown: the bracketed declaration `{fontFamily = "name", fontSize = size}` specifies the font family and size for the child of the `with` node.

Presentation:

```
ColPres [ WithPres { fontFamily = "Times New Roman", fontSize = 12 }
  (FormatterPres [ "This", "is", "a", "simple", "expression" ])
  , WithPres { fontFamily = "Courier New", fontSize = 12 }
    (TokensPres [ LCaseTokenPres (1,0) "simple1", OpTokenPres (0,1) ":@"
      , UCaseTokenPres (0,1) "Int"
    ])
  , WithPres { fontFamily = "Courier New", fontSize = 12 }
    (TokensPres [ LCaseTokenPres (1,0) "simple1", OpTokenPres (0,1) "="
      , LCaseTokenPres (1,2) "if", UCaseTokenPres (0,1) "True"
      , LCaseTokenPres (0,1) "then", IntTokenPres (0,1) "1"
      , LCaseTokenPres (1,10) "else", IntTokenPres (0,1) "0"
    ])
]
```

3.4.3 Layout layer: Layouter

The layouter processes the tokens in the presentation level, yielding the layout level. Each list of tokens is mapped onto a column that contains rows of strings. The spaces in the whitespace of a token are represented as strings of spaces, whereas line breaks cause new rows in the layout.

Structural tokens are treated specially, because these are not strings. A structural token is represented in the layout level by its `Presentation` field, whereas its whitespace is handled similarly to the other tokens. The presentation of a structural token in the layout level is tagged in order to be able to retrieve the structural token during scanning.

Example: The tokens in the presentation level are replaced by strings in the layout level (the `␣` character denotes a space). The formatter is not affected. The line break before the type declaration is represented by an empty string.

Layout:

```
ColLay [ WithLay { fontFamily = "Times New Roman", fontSize = 12 }
  (FormatterLay [ "This", "is", "a", "simple", "expression" ])
  , WithLay { fontFamily = "Courier New", fontSize = 12 }
    (ColLay [ "
      , RowLay [ "simple1", "␣", ":", "␣", "Int" ]
    ])
  , WithLay { fontFamily = "Courier New", fontSize = 12 }
    (ColLay [ RowLay [ "simple1", "␣", "=" ]
      , RowLay [ "␣", "if", "␣", "True", "␣", "then", "␣", "1" ]
      , RowLay [ "␣␣␣␣␣␣", "else", "␣", "0" ]
    ])
  ]
```

3.4.4 Arrangement layer: Arranger

The arranger computes the exact sizes and positions for all elements in the layout level, yielding the arrangement. Fonts are queried to determine the size of strings, and child elements of composite elements such as rows and columns are aligned and positioned.

The arrangement layer also resolves formatters. Based on the amount of available space for a formatter, its child elements are distributed along rows using a (possibly optimal) line-breaking algorithm. These rows are put in a column, and the resulting column of rows is then mapped onto arrangement nodes in the same way as regular rows and columns.

Example: The formatter, which is the first element in the top-level column of the example layout, is replaced by a node that contains a list of nodes in the arrangement. Furthermore, each element in the arrangement tree has an exact size and a position relative to its parent, which are denoted with superscripts: $element^{(x,y)(width \times height)}$.

Note that the `With` elements are not removed because the font information is required to render the arrangement.

Arrangement:

```

NodeArr(0,0)(80×84)
  [ WithArr { fontFamily = "Times New Roman", fontSize = 12 }
    (NodeArr(0,0)(80×24)
      [ NodeArr(0,0)(80×12) [ "This"(0,0)(17×12), "is"(25,0)(6×12), "a"(41,0)(4×12)
        , "simple"(53,0)(27×12) ]
      , NodeArr(0,12)(80×12) [ "expression"(0,0)(42×12) ]
      ]
    , WithArr { fontFamily = "Courier New", fontSize = 12 }
      (NodeArr(0,24)(75×24)
        [ ""(0,0)(0×12)
        , NodeArr(0,12)(75×12) [ "simple1"(0,0)(35×12), "_"(35,0)(5×12), "::(40,0)(10×12)
          , "_"(50,0)(5×12), "Int"(55,0)(15×12) ]
        ]
      , WithArr { fontFamily = "Courier New", fontSize = 12 }
        (NodeArr(0,48)(80×36)
          [ NodeArr(0,24)(50×12) [ "simple1"(0,0)(35×12), "_"(35,0)(5×12), "="(40,0)(5×12) ]
          , NodeArr(0,36)(80×12) [ ... ]
          , NodeArr(0,48)(80×12) [ ... ]
          ]
        ]
      ]
  ]

```

3.4.5 Rendering layer: Renderer

The renderer maps each element of the arrangement onto a set of drawing commands for the user interface. While traversing the arrangement, information from the `With` nodes is used to generate commands that set the font, style, and color of the rendering. All positions and sizes have already been computed by the arranger, and the renderer only scales these positions and sizes according to the current scaling factor of the view.

Example: The result of applying the renderer to the example arrangement is a set of rendering commands that display the comment and the declaration when executed.

Rendering:

```
[ setFont "Times New Roman" 12, drawText (0,0) "This", drawText (25,0) "is"
, ... ]
```

The result of executing these commands was already shown at the start of this section, but for completeness we repeat it here. Note that the comment is rendered in a different font than the declaration.

```
This is a simple
expression

simple1 :: Int
simple1 =
  if True then 1
      else 0
```

3.5 Interpretation process

The interpretation of edit operations is layered in the same way as the presentation process. However, there are important differences between the two. For the interpretation process, edit operations are the main focus. Therefore we will not view interpretation mappings as mappings between levels (as we did for the presentation process) but as mappings between edit operations.

An edit operation may be interpreted either *indirectly*, or *directly* (not to be confused with directly/indirectly editing a level). If an edit operation is interpreted *indirectly*, the operation is performed on the lower level, yielding an updated lower level. The updated lower level is then mapped onto a new higher level, from which a higher-level edit operation is computed by taking the difference between the new and the previous higher level.

For an example of indirect interpretation, consider the insertion of a token in the presentation level. Rather than mapping this edit operation directly onto an edit operation on the enriched document, the token is inserted in the presentation, the presentation is parsed, and the edit operation is distilled from the new enriched document and its previous value.

An edit operation that is *directly* interpreted, on the other hand, is immediately mapped onto an edit operation on the higher level, without first performing the operation on the lower level. An example is the interpretation of a mouse click on a position in the arrangement, which is mapped onto a mouse click on a tree path in the layout level.

Note the difference between direct or indirect interpretation versus direct or indirect editing. A level is edited directly if an edit gesture is targeted at that level, whereas the level is edited indirectly if it changes due to an edit gesture targeted at another level. When a certain level is directly edited, this results in direct interpretation by all layers below that level, and indirect interpretation by the layers above. Hence, because the rendering and arrangement levels may not be edited directly, the lowest two layers (rendering and arrangement) only support direct interpretation.

Because the interpretation of edit operations does not always go through all stages like the presentation does, and because of the variation in edit operations, it is not helpful to give a running example of the interpretation process. Instead, a number of separate examples are provided together with the descriptions of the interpretation components.

3.5.1 Rendering layer: Gesture interpreter

The gesture interpreter has two tasks. It maps edit gestures onto edit operations for the designated levels, and it interprets direct edit operations on the rendering as edit operations on the arrangement, which means that absolute positions in pixel coordinates are descaled to arrangement level coordinates.

Example: We give two interpretation examples for the gesture interpreter: a mouse click and a key press.

The mouse edit operation is a single left-click at pixel coordinates (84,57) in a rendering that has been scaled to 150%. Because of the scaling factor, the coordinates are divided by 1.5 to get the arrangement coordinates.

$\text{MouseClicked}_{Ren} \text{ Left } (84,57) \mapsto \text{MouseClicked}_{Arr} \text{ Left } (56, 38)$

The second example is a key press of the letter ‘a’, which is mapped onto an insert event. However, a textual insert event is targeted at the layout level instead of the arrangement level, since the arrangement level cannot be edited textually. Because the insert operation is not of the arrangement edit type, it is wrapped with a Wrap_{Arr} constructor. The arrangement layer will remove the Wrap_{Arr} constructor and pass the insert operation on to the layout layer.

$\text{KeyPress}_{Ren} \text{ 'a'} \mapsto \text{Wrap}_{Arr} (\text{Insert}_{Lay} \text{ 'a'})$

3.5.2 Arrangement layer: Unarranger

The main task of the unarranger is to map locations in arrangement-level edit operations onto locations in layout-level edit operations. A location that is specified in absolute coordinates is first converted to a location in the arrangement tree, which is specified as a tree path. Subsequently, the arrangement tree path is mapped onto a layout tree path. The arrangement tree is largely isomorphic to the layout tree, except for the formatter subtrees, as these are represented by nodes that contain lists of nodes (representing the formatted lines) in the arrangement. Therefore, the mapping is almost the identity function, except for paths to children of nodes originating from a formatter, which are mapped onto paths to the corresponding child elements of the formatter.

Example: A left-click mouse event at position (56,38) in the example arrangement from Section 3.4.4 represents a click on the string “Int” in the layout from Section 3.4.3. To be precise, it is a click on the left side of the letter ‘I’. If we represent a path in the arrangement tree by a list of integers and a 0 denotes the first child, then this position is represented by [1,0,1,4,0]. Thus:

$\text{unarrange} (\text{MouseClicked}_{Arr} (56,38) \text{ Left}) = \text{MouseClicked}_{Lay} [1,0,1,4,0] \text{ Left}$

3.5.3 Layout layer: Scanner

The scanner is the first layer that supports indirect interpretation, since the layout level may be edited by the user. Edit operations targeted at the layout level are performed on the layout level, after which the level is scanned, yielding the new presentation. An edit operation on the presentation is obtained by taking the difference between the new presentation and its previous value.

The scanner operates only on the subtrees in the layout layer that originate from a token list on the presentation level, while leaving other parts of the tree unaffected. Every such subtree, which is always a column of rows, is scanned by inspecting it row by row, and recognizing the tokens that are represented by the strings in each row. For each token, the whitespace (spaces and row transitions) preceding it is recorded and stored in the token. The parts of the layout tree that originate from structural tokens, have been tagged by the layouter and are scanned as structural tokens.

If the scanner encounters an error during the scanning process, (e.g. in case of an unterminated string), it generates an error token, which will subsequently cause a parse error in the parsing layer. The details of this process are not discussed here.

The scanner is parameterized with a *scanner sheet*, which contains a set of regular expressions describing the allowed set of tokens. Since different parts of a layout may require different scanners (e.g. in a presentation that shows both Java and Haskell code), a scanner sheet can contain several sets of token descriptions. The layout level contains information on what kind of presentation gave rise to the parts that need to be scanned, which allows the scanner to use the appropriate set of token descriptions.

Scanning is usually a localized process, so rather than re-scanning an entire token list, only the edited part of the layout level is scanned and used to compute the appropriate presentation-oriented edit commands. An exception to this local behavior is formed by tokens with an explicit start and end tag, such as strings or characters. If a string or character quote is entered, this may affect more than just the edited part of the layout.

The scanner layer does not yet support the handling of comments that may appear anywhere in the source. Because such comments are not part of the document, they need to be handled as presentation extra state, similar to whitespace. Comments may affect the locality of the scanning process in the same way as strings.

Example: Consider the example layout level from Section 3.4.3 and assume that the edit operation on the layout is the insertion of a space between the characters ‘e’ and ‘1’ in the identifier “simple1” of the declaration. Because “simple 1 = if ...” is not a valid declaration, the edit operation will cause a parse error in the presentation layer, but this has no consequences for the example at the layout layer.

In order to compute the edit operation on the presentation, we first apply the layout edit operation to the layout level, yielding:

```

...
ColLay [ RowLay [ "simple_1", "_", "=" ]
          , RowLay [ "_", "if", "_", "True", "_", ... ]
          ... ]
...

```

Now, the scanner is invoked on the updated parts of the layout, which gives rise to the following list of tokens.

```

TokensPres [ ...
              , LCaseTokenPres (1,0) "simple", IntToken (1,0) 1,
              , OpTokenPres (0,1) "="
              , LCaseTokenPres (1,2) "if", UCaseTokenPres (0,1) "True"
              ... ]

```

From the new token list and the old presentation, an edit operation on the presentation level can be derived:

```

insertLay ' '
↳
replacePres [ LCaseTokenPres (1,0) "simple1" ]
by [ LCaseTokenPres (1,0) "simple", IntToken (1,0) 1 ]

```

We use an informal notation for both the *insert* and the *replace* edit operations to improve readability. The actual insert operation also contains a reference to the target location of the inserted character, and the replace operation contains the location of the target token list, rather than the list itself.

3.5.4 Presentation layer: Parser

It is not possible to map edit operations on the presentation directly onto edit operations on the enriched document. Consider, for example, the insertion of the string “if” in a source editor. From this insertion command only, we cannot compute an enriched document update. Instead, we need to take the indirect approach: the edit operation is applied to the presentation, which is then parsed to yield a new enriched document. Similar to the scanner, the parser layer computes an edit operation on the enriched document by taking the difference between the new enriched document and its previous value.

The parser component is parameterized with a *parsing sheet*, which contains the actual specification of the parser. The parser is specified in Haskell, using a specialized parser combinator library. The parser combinators from this library parse a presentation tree instead of a string. Furthermore, special support is available for handling interpretation extra state.

The parser makes a distinction between token lists and the rest of the presentation. Only the token lists are actually parsed. The other parts of the presentation may not be edited at

the presentation level and are therefore mapped onto their originating enriched document structures. Because parsed presentations may contain parts that are not parsed, and vice versa, the two processes alternate.

Each part of the presentation that is not parsed is mapped directly onto the enriched document element of which it is the presentation. In order to do so, the parser layer uses the information on the enriched document origin that is stored in each presentation element by the presenter.

If the originating enriched document element has children that are also presented, these children are determined via the same process. A child that does not appear in the presentation is reused from the previous enriched document level, or, if this is not possible, it is initialized to a default value. If a child has a token list presentation, the parser process takes over.

The parser that is applied to the list of tokens is specified in the parsing sheet. Parse errors are represented in the document by error nodes that may appear anywhere in the document tree. If a `StructuralToken` is encountered, the previously described mapping process is invoked again. Because the enriched document may contain information that is not presented (i.e. interpretation extra state), the parser tries to reuse the enriched document nodes from the previous version of the enriched document. If a node cannot be reused, the extra information is initialized to a default value.

In order to support presentation extra state for tokens (e.g. whitespace), the parser stores a reference to the parsed tokens in the resulting enriched document node. This information is used when the enriched document is presented. Because currently the only form of presentation extra state is whitespace in tokens, the mapping information only needs to be kept for enriched document elements that are parsed.

Example: We consider a delete operation on a number of successive tokens in the presentation. For simplicity, each declaration or type declaration in our example is presented as a separate token list and will therefore be parsed separately. Thus, presentation-oriented edit operations have to be local to a single declaration. In an actual source editor, the token lists are concatenated, and editing may span several declarations.

The presentation level for the example is the same as in Section 3.4.2. The part that is affected by the edit operation is:

```
...
TokensPres [ LCaseTokenPres (1,0) "simple1", OpTokenPres (0,1) "="
             , LCaseTokenPres (1,2) "if", UCaseTokenPres (0,1) "True"
             , LCaseTokenPres (0,1) "then", IntTokenPres (0,1) "1"
             , LowercaseTokenPres (1,10) "else", IntTokenPres (0,1) "0"
             ]
...
```

The delete operation removes the "if", "True", "then", "1", and "else" tokens, giving rise to a new presentation level.

```

...
TokensPres [ LCaseTokenPres (1,0) "simple1", OpTokenPres (0,1) "=",
              , IntTokenPres (0,1) "0"
              ]
...

```

A parser is invoked on the new list of tokens. The result is not an entire enriched document, but only an updated declaration for `simple1`. Interpretation extra state children, such as the third child of the declaration (see the example in Section 3.4.2), are not in the presentation and are therefore reused from the previous value of the declaration. The previous value of the declaration is obtained from a reference that was stored in its tokens on presentation.

```
DeclEnr "simple1" (IntExpEnr 0) "info"
```

The edit operation is computed from the new enriched document part and the previous enriched document, yielding:

```

deletePres "if" ... "else"
↳
replaceEnr (IfExpEnr (BoolExpEnr True) (IntExpEnr 1) (IntExpEnr 0))
by (IntExpEnr 0)

```

It should be noted that the second `(IntExpEnr 0)` in the replacement operation is exactly the same as the one in the else part of the if expression. In this case, the expression is uniquely determined by its presentation, and hence parsing it gives an exact copy, but even if this were not the case, both expressions would be the same, due to the reuse strategy of the parser. For example, if the integer expression has an extra field that is not presented, the replacement integer expression gets the value for that field from the integer expression in the else part. This is analogous to the `"info"` string for the declaration.

3.5.5 Evaluation layer: Reducer

The reducer takes care of mapping edit operations on derived structures onto edit operations on the document. The specification of the mapping is in the *reduction sheet*, which in many cases can be automatically derived from the evaluation sheet. Automatic reduction behavior is typically possible for parts of the enriched document that are duplicated, reordered, or partial versions of parts of the document.

If a document structure is duplicated in the enriched document, and one of the duplicates has been edited, the reducer resolves the situation by taking the duplicate that was edited. If both duplicates have been edited, either the edit operation is blocked, or a choice between the two is made. Reordered and partially presented document structures are handled by maintaining a mapping between each enriched document element and the document element from which it originated.

Although it is possible for a derived structure to be editable, this behavior is not enforced. For many derived structures, a reverse mapping does not make much sense. For example, it is hard to give a clear semantics to the direct editing of a chapter number. However, when two chapters in a table of contents are swapped, swapping the two corresponding chapters in the document could well be the desired effect. When the reverse mapping makes sense, it can be specified in the reduction sheet. When not, editing derived structures can be forbidden.

Besides regarding the reduction as the reverse of the evaluation, it is also possible to use reduction as an extension of the parser. As an example, consider a program source that contains definitions of infix operators with user-specified associativity and precedence. Parsing such operators in one pass requires a sophisticated parser, whereas the two-pass solution is straightforward.

Another application of reduction is the handling of redundancy in a document presentation. For example, when a document type for expressions does not have an explicit representation for parentheses, redundant parentheses that are entered by a user can be removed by the reducer, to be added again by the evaluator.

Example: The example editor supports editing on the function name in a type declaration, which causes an update on the name in the corresponding declaration. The example edit operation is an update on the function name in the type declaration of the enriched document from Section 3.4.1. The name is changed from "simple1" to "simple". Thus, we get the following updated enriched document.

```
RootEnr [ CommentEnr [ "This", "is", "a", "simple", "expression" ]
  , TypeDeclEnr "simple" IntTypeEnr
  , DeclEnr "simple1"
    (IfExpEnr (BoolExpEnr True) (IntExpEnr 1) (IntExpEnr 0))
    "info"
  ]
```

Both the type declaration and the declaration are presentations of the declaration in the document. Hence, we have two duplicate presentations, the first of which (the type declaration) is a partial presentation, since the type declaration does not contain enough information to compute the declaration in the document. Thus, the missing right-hand side is interpretation extra state.

The reducer processes the enriched document and maps both the type declaration and the declaration onto a document level declaration. Because the type declaration has no right-hand side expression, the value for the expression is reused from the previous value of the declaration in the document. The result of interpreting the type declaration is a declaration with string "simple" and expression (if True then 1 else 0). On the other hand, the interpretation of the declaration itself is the identity and yields a declaration with string "simple1" and expression (if True then 1 else 0).

We now have two conflicting interpretations for the same document declaration. The conflicts are resolved in favor of the updated fields. In this case, it means that the string "simple" is chosen. The result is a new document.

```
RootDoc [ CommentDoc ["This", "is", "a", "simple", "expression"]
          , DeclDoc "simple"
            (IfExpDoc (BoolExpDoc True) (IntExpDoc 1) (IntExpDoc 0))
            "info"
          ]
```

And the document-level edit operation thus becomes:

```
replaceEnr "simple1" by "simple"  $\mapsto$  replaceDoc "simple1" by "simple"
```

3.6 The choice of layers in Proxima

The choice of layers for the Proxima editor is based on pragmatic considerations. On the one extreme, the entire editing process could be put in one large layer, resulting in an unwieldy presentation relation. On the other extreme a separate layer could be defined for every small step in the process, giving an awkward and inefficient editing process. The choice of layers in Proxima is a balance between these two extremes. This section explains why Proxima consists of the five layers that were presented in this chapter.

The separation of document evaluation and presentation serves two purposes. Firstly, the separation makes it possible to have different presentations for a document together with its derived structures. Secondly, it facilitates the specification of edit behavior on derived structures. When parsing and reduction are mixed, such behavior is harder to specify.

The reasons for a separate layout layer are the automatic whitespace handling for token presentations, as well as efficiency, since first scanning and then parsing is more efficient than parsing on a character basis. A downside is that different document types require slightly different scanning methods, and a generic scanner that is able to handle all exotic cases is hard to construct. Moreover, in some cases, an editor designer may be interested in dealing with whitespace at enriched document or document level. However, in these cases, the scanner layer may be bypassed altogether, allowing the editor designer to parse on a character basis and explicitly deal with whitespace.

Below the layout level are the arrangement level and the rendering level. The separation between these two levels and the higher levels is obvious, since keeping the position and size computations in a separate layer prevents cluttering the higher layers with needless detail. The reason why the arrangement and rendering have been split is to keep the part of the architecture that deals with the GUI-specific issues as small as possible. Furthermore, the arrangement contains exact information on the location and size of each item that is

to be rendered, which is useful for resolving pointing issues and performing incremental updates.

The arrangement process itself also consists of steps, since a formatter is mapped onto an intermediate column of rows, which is then arranged in the same way as the other rows and columns. However, these steps are very closely intertwined, and separating the arrangement layer into different layers does not seem worthwhile.

Besides the current layers, several other layers are imaginable. For example, a post-arrangement layer could process the arrangement in order to handle footnotes or the formatting of paragraphs that contain text in languages with different reading directions. Similarly, an extra evaluation layer is conceivable when computations over computed structures need to be specified. When yet other computations are desired on the resulting computed structures, even multiple evaluation layers may be required. This brings up the issue of using higher-order attribute grammars [91] for the evaluation layer. However, when using higher-order attribute grammars, it is not straightforward anymore to handle extra state at intermediate levels, nor is it clear how to interpret edit operations on lower levels. In Proxima, these problems are dealt with by the layered architecture. Before it is possible to do presentation, or even just evaluation, with higher-order attribute grammars in Proxima, more research is necessary.

3.7 Conclusions

The architecture of the Proxima editor consists of five layers connecting six data levels. Several layers are parameterized with sheets that specify the behavior of the layer. The most important sheets are the presentation and parsing sheets, which are specified, respectively, with an attribute grammar and a combinator parser. The other sheets are the evaluation and reduction sheets of the evaluation layer, and the scanning sheet of the scanning layer, but no final choice has been made about the formalisms for these sheets.

Proxima has an open architecture, to which an extra layer can be added with relatively little effort. Moreover, each layer may easily be extended or modified. Instead of specifying the computations in an evaluation sheet, the evaluation layer may, for example, invoke an external type checker.

The value of each level in the Proxima editor contributes to the total editor state, rather than just being an intermediate value in a computation. One reason for this is to support incrementality, but a second reason is that a level may contain extra state. More specifically, from the perspective of a single layer, a higher level does not always contain enough information to compute the lower level and vice versa. By storing both levels, together with information on how elements in one level depend on elements in the other, it is possible to compute the lower level from an updated higher level and vice versa.

Each layer in the Proxima system maintains information on the bidirectional mapping between the neighboring higher and lower levels. In the lower layers, the mappings can be maintained by the editor, because the lower layers operate between levels that have

fixed types, and furthermore, have presentation and interpretation mappings that are less customizable by the editor designer. For the higher levels, however, the editor designer in some cases needs to specify how the mappings are to be maintained.

The formalisms for the sheets at the presentation layer offer special support for maintaining the mapping information, but more research is required to improve this support. Furthermore, the formalisms for the evaluation and reduction sheets need to be determined. Ideally, only the evaluation sheet needs to be provided, from which the reduction sheet is derived automatically. However, for the time being, we consider it an acceptable solution that both sheets are provided by the editor designer. For frequently appearing patterns in the evaluation process, automatic reduction could be supported. Similar functionality is desirable for the presentation layer: for frequently appearing patterns in the presentation, a parser may automatically be generated.

Prelude to the specification

In this chapter, we informally introduce several concepts that play a role in the specification of a layered editor, to be given in Chapter 5. The discussion in this chapter also recapitulates some issues already mentioned in Chapter 3. Although many examples come from the Proxima editor, the concepts are general and apply to any layered structure editor.

Section 4.1 introduces the presentation invariant that is to be maintained by an editor and sketches the different steps of the editing process. In Section 4.2 we explore the concept of extra state for presentation mappings between tree structures, and sketch a method for handling such extra state. Finally, Section 4.3 discusses how to handle edit operations on presentations that contain duplicate information.

4.1 The editing process

An editor maintains a presentation relation between a document and its presentation. We denote the document by $level_H :: Level_H$, and the presentation by $level_L :: Level_L$. The relation between the two is $Present :: Level_L \sim Level_H$. The level order in the relation type is such that the notation $l Present h$ looks similar to the notation for a presentation mapping represented by a function: $l = present h$.

The invariant, maintained by the editor is that the lower level is a valid presentation of the higher level:

Presentation invariant: $level_L Present level_H$

The precondition of an edit step is that the presentation invariant holds. The user then modifies the presentation (giving $level'_L$), and most likely breaks the invariant. In order to

restore the invariant, the editor updates the document level by interpreting the modified presentation, which yields $level''_H$. However, because the user-updated lower level is often not a completely valid presentation, the document update alone may not yet restore the presentation invariant. For example, a program fragment may have been entered without proper syntax coloring, or a chapter title may have been modified in the chapter itself but not in the table of contents. In order to guarantee the restoration of the presentation invariant, the updated document is presented again ($level''_L$).

Figure 4.1 schematically shows the editing process for one edit step. The final higher-level value is called $level''_H$ rather than $level'_H$ to enable a consistent notation for final values. Furthermore, in the multi-layered editor, introduced in the next section, $level'_H$ will denote an intermediate value for the higher level.

The final values $level''_H$ and $level''_L$ satisfy the presentation invariant and serve as initial values for the next edit step. In the rest of this chapter, as well as in the next, we use this notation for the values of the data levels in the different stages of an edit step.

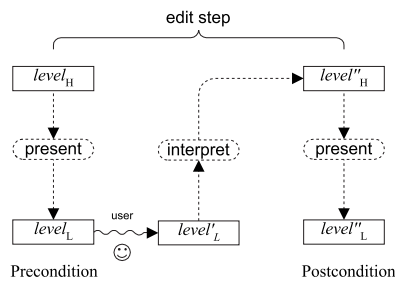


Figure 4.1: A single edit step.

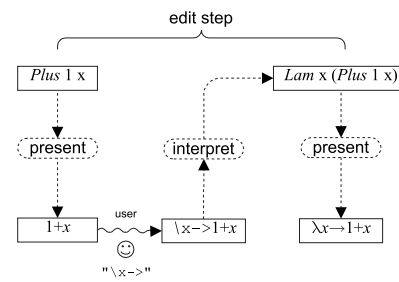


Figure 4.2: A concrete example.

Figure 4.2 shows the values of the higher and lower levels for an actual example. The document is an expression that is presented using an italic style for identifiers and mathematical symbols for operators. The presentation of a sum is modified by entering the text `"\x->"` at the beginning. After the document update, the presentation invariant clearly does not hold. It is restored by re-presenting the document, and thus changing `"\x->1+x"` to `"\lambda x \to 1 + x"`.

Instead of updating the lower level (presentation-oriented editing), a user may also perform a direct update on the higher level (document-oriented editing). However, we do not consider document-oriented editing in much detail, because after a document update the presentation invariant can be restored by simply re-presenting the updated document.

A layered editor

In order to describe the layered architecture of Proxima (see Chapter 3), we refine the simple editor by splitting the presentation relation. A *Present* relation that is split into n components ($Present_i :: Level_i \sim Level_{i-1}$) gives rise to $n + 1$ data levels ($Level_{0..n}$).

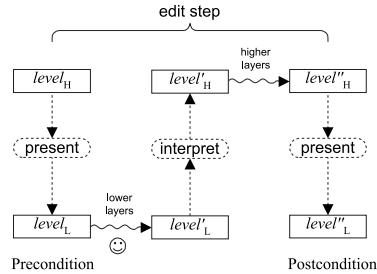


Figure 4.3: An edit step at one layer.

Lower levels get higher indices, with $Level_0$ being the document level and $Level_n$ the presentation level.

From the perspective of a single layer, we only need to consider a single $Present_i$ relation and two data levels $Level_{i-1}$ and $Level_i$. We drop the subscript of the relation and refer to the data levels as $Level_H$ and $Level_L$. Figure 4.3 shows the updates during an edit step from the perspective of a single layer. Instead of being updated immediately by the user, the lower level gets updated by the layer below (which is the effect of the user updating the lowest level). At the top of the figure, the layer computes an intermediate $level'_H$, instead of directly computing $level''_H$. This $level'_H$ is processed by the layers above, yielding $level''_H$. At the document level, $level'_0$ is copied to $level''_0$. Note that there is no incrementality in the model: level values are passed up and down, but not to the right.

A simple example shows the need for the intermediate $level'_H$. Consider a document that consists of a list of numbers, which is mapped onto an enriched document that contains the list together with its sum. The presentation is a textual presentation of both the list and the sum. Suppose that the list is edited at the presentation level. At the presentation layer, the updated presentation is parsed, yielding an intermediate enriched document value $level'_H$, which is the new list with the old sum. This intermediate value is then interpreted and presented by the evaluation layer above, yielding the final $level''_H$, which is the updated list with a correct sum.

4.2 Extra state

As we saw in Section 2.2.6, a lower level may contain information that cannot be computed by presenting its adjacent higher level. Similarly, a higher level may contain information that cannot be computed by interpreting the lower level. The information in a level that cannot be computed by presenting, or interpreting an adjacent level, is referred to as *extra state*. In this section, we give an informal description of extra state, as well as a number of examples. Section 5.2 presents a formal specification.

Generally speaking, *presentation extra state* is information that influences the way in which the higher level is viewed without being part of the higher level. An example of this is found in a tree-browser presentation (see Section 2.1.4). The information whether a node in the tree is expanded or collapsed is not part of the document. Therefore, this expansion state is part of the presentation extra state.

Interpretation extra state, on the other hand, consists of the parts of the higher level that are not shown in the presentation. Again, the tree browser provides an example, since it shows the structure of the document while leaving out the content. The content that is left out is part of the interpretation extra state.

Because the presentation or interpretation does not specify the value of the extra state in the resulting level, extra state is reused, if possible, from the previous value of the level. A general method for reusing extra state is difficult, if not impossible, to give. Therefore, in this section, we consider a special case of extra state, for which we sketch a method for reuse. We only consider extra state that is associated with a specific parent node in the tree. Whether or not the extra state can be reused depends on whether or not its parent node can be tracked down in the updated level. This form of extra state is sufficient for specifying the use cases presented in Section 2.1. Chapter 5 contains a more formal specification of extra state.

4.2.1 The presentation mapping

In order to specify when a node is extra state, we first take a closer look at the presentation relation. In this section, we assume that both $Level_H$ and $Level_L$ are tree structures. Furthermore, we assume that *Present* not only relates a higher level to a lower level, but that it also establishes a relation between the nodes on these levels. If, for example, a document containing an if expression is mapped onto a presentation that contains the three tokens “if”, “then”, and “else”, then this establishes a relation between the If node in the document, and the three token nodes in the presentation. Thus, for any two levels $Level_H$ and $Level_L$ between which the presentation invariant holds, we also have a relation between the nodes of these two levels.

Figure 4.4 shows two levels for which the presentation invariant holds. In addition, the figure shows the relation between the nodes of both levels, using dotted arrows. To reduce the number of arrows in the figure, the lower-level nodes are grouped. An arrow between two nodes only relates the nodes and not the subtrees rooted at these nodes.

We assume that the relation between nodes is a $1 : n$ relation, which implies that the presentation of a higher-level node may consist of several lower-level nodes, but each lower-level node is in the presentation of at most one higher-level node. This restriction concerns only the node mapping; the *Present* relation itself between two levels is $n : m$.

The choice for $1 : n$ relations follows naturally from the compositional way in which a presentation relation is usually specified: by specifying a presentation rule for each higher-level node. No practical examples have been found that suggest $n : m$ relations are needed.

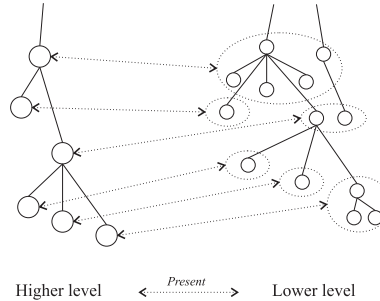
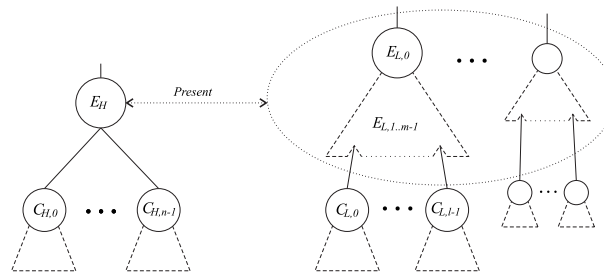


Figure 4.4: A mapping between the nodes of two levels.

Figure 4.5: The presentation of node E_H .

The $1 : n$ restriction is only for dealing with extra state. It is still possible to have a lower-level node depend on several higher-level nodes, but when dealing with extra state, each lower-level node is assumed to be the presentation of at most one higher-level node.

Thus, besides relating a set of higher-level values to a set of lower-level values, *Present* also relates each node of the higher level to zero or more lower-level nodes. Figure 4.5 schematically shows the presentation of a node (E_H) in the higher level. E_H has n children ($C_{H,0..n-1}$). The presentation of E_H is a number of trees (although usually just one) that may consist of several nodes. In the figure, the first tree is shown in more detail. It consists of m nodes ($E_{L,0..m-1}$) and is rooted at $E_{L,0}$.

The $C_{L,0}, \dots, C_{L,l-1}$ subtrees in the lower-level tree are not part of the presentation of E_H . Typically, these are the presentations of the children of E_H , but in general they can be (part of) the presentation of any node in the higher level.

Figure 4.6 provides a more concrete example of a presentation, taken from a source editor (see Section 2.1.1). A parenthesized sum in the document is presented as a list of tokens,

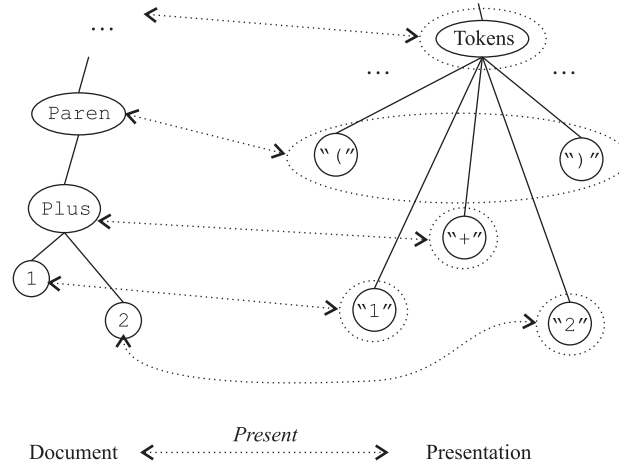


Figure 4.6: The presentation of a parenthesized sum.

which are represented by strings. The figure shows only part of the document and its presentation. To the editing user, the expression will appear as "(1+2)". The example is somewhat simplified, because it does not contain whitespace. Whitespace is part of the presentation extra state, which is discussed in following subsection.

4.2.2 Extra-state nodes

In Figures 4.4 and 4.6, each node is connected to a node in the adjacent level, either by a direct arrow, or by being inside an ellipse that is connected by an arrow. However, it is possible that a node does not have an arrow connecting it to a node in the adjacent level. Such nodes are not determined by the presentation relation, and hence are part of the extra state of the level.

Extra state may occur on the higher level as well as on the lower level. On presentation, a lower level is computed from a higher level, which means that the extra state in the lower level needs to be dealt with. Hence, lower-level extra state is referred to as *presentation extra state*. Similarly, on interpretation of a level, we only need to deal with the higher-level extra state, which is referred to as *interpretation extra state*. In fact, if we consider not just one layer, both kinds of extra state may exist on a single level. This is explained in Section 4.2.3.

Figure 4.7 shows two examples of extra state, one in the lower level and one in the higher level. On the left-hand side, a document node is presented as a token. The shaded whitespace node (0,1) (denoting the line-breaks and spaces before the token) is not specified by

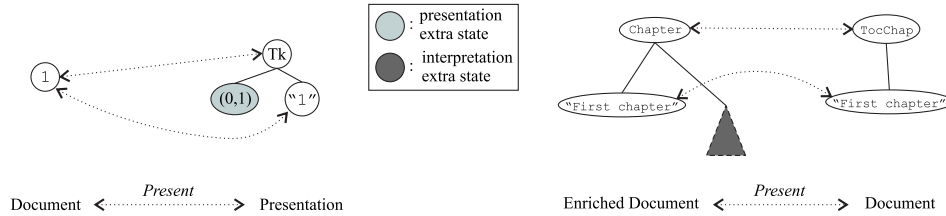


Figure 4.7: Two examples of extra state.

the presentation mapping, and hence part of the lower level presentation extra state. On presentation, tokens are reused, causing the whitespace information to stay the same. In order to do this, we need to know exactly on which presentation nodes a document node was mapped when it was previously presented. In case a token has no previous whitespace (e.g. because it is the presentation of a newly inserted document part) a default value is used.

The right-hand side of Figure 4.7 shows an example of interpretation extra state: a word processor with an editable table of contents. A document chapter is presented only partially, since the content of the chapter is left out.

For simplicity, we assume that the enriched document only contains the table of contents and not the chapters themselves. Thus, we only have to consider the extra state here and not the duplication (titles that appear in the table of contents as well as in the chapters). Section 4.3 discusses how to handle duplications in general, and the same method can be used for the table of contents.

On interpretation, the table of contents is mapped back onto a complete document, which includes the shaded content parts that are not in the enriched document. The title of a chapter comes from the enriched document, whereas its content is reused from the previous document.

4.2.3 Each intermediate level has two kinds of extra state

Figure 4.7 only shows one layer, but since each data level except the document and the rendering is in between two layers, each level between the document and the rendering may have both presentation and interpretation extra state.

Figure 4.8 shows a data level between two layers. At the middle level, a shaded left or right half denotes that a node is extra state. Nodes with a shaded left half are presentation extra state for the higher layer, and nodes with a shaded right half are interpretation extra state for the lower layer. Extra state in one direction is independent of extra state in the other direction, hence a node in the figure can have zero, one, or two shaded halves.

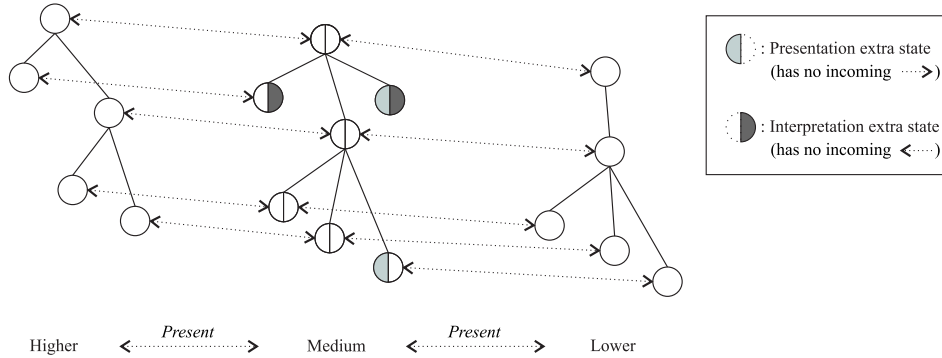


Figure 4.8: Presentation and interpretation extra state in one level.

The whitespace extra state in tokens shows that extra state in one direction is independent of extra state in the other direction. Whitespace is presentation extra state of the presentation level, since it cannot be computed when presenting the enriched document. On the other hand, when scanning, whitespace in tokens is computed from strings and line breaks in the layout level. Hence, whitespace is not interpretation extra state of the presentation level.

4.2.4 Reusing extra state

When a new value for a level is computed on presentation, or interpretation, the values for its Extra-state nodes are taken from the previous value of that level.

In order to reuse nodes, a layer needs to keep track of additional information regarding the presentation mapping. For each higher-level node, the layer must keep references to the lower-level nodes of its presentation, and for each lower-level node there must be a reference to the higher-level node whose presentation it is part of. These references corresponds to the dotted arrows between nodes in the figures of this section.

We sketch the process of reusing for the presentation direction. For interpretation, the process is analogous.

We assume that the update on the higher level is incremental; except for the parts that have been edited, the updated level is equal to its previous value. On presentation of a higher-level node, the lower-level nodes of its previous presentation are used to construct its new presentation, if possible. If the higher-level node is new, the presentation will consist of new lower-level nodes. Furthermore, also in case the node was changed in such a way that its previous lower-level nodes cannot be reused, new lower-level nodes are used.

The resulting lower level is not a completely new value, but an incrementally updated version of the previous value. Only those parts of the presentation that correspond to a changed part of the higher level are changed.

When a lower-level node is reused, also its Extra-state children are reused. For nodes that are no longer present in the presentation, Extra-state children are lost. Extra-state children for new lower-level nodes are set to a default value. Thus, if an entry is added to a table of contents in the (presentation of the) enriched document of a word processor, an empty chapter (or section) is added to the document. Similarly, if a document-oriented edit operation in a source editor adds a new structure to the document, its presentation gets a default layout.

4.2.5 Safety of extra state

The mechanism of reusing extra state based on its parent nodes is not infallible. Several situations can cause the loss of extra state, both in the presentation as in the interpretation direction. We present two situations here.

Firstly, if the presentation of a node depends not only on the node itself, but also on nodes elsewhere in the tree, then the presentation may change, even if the node itself remains unchanged. In that case, it is possible that the lower-level nodes of the previous presentation cannot be used for the new presentation.

For interpretation extra state, updates elsewhere in the tree can be even more dangerous. When a presentation is computed by a parser, updates on the presentation before a certain token may greatly influence how the token is parsed. Hence, extra state in a presentation that allows full text editing is vulnerable.

Secondly, if a node is transformed, it may make sense to reuse its extra state in the result of the transformation. For example, when in an expression editor, a sum node at document level is transformed to a product node, it makes sense to reuse its whitespace extra state. However, if the transformation is performed by simply removing the old sum and inserting a fresh product node, then the extra state is lost. In such a case, an editor designer may specify for a transformations that source nodes are reused in the result, thereby also reusing their extra state.

More research is necessary to establish clearly in what situations extra state is vulnerable, and also how reusing it can be maximized. Although in general it cannot be guaranteed that extra state is always recoverable, this does not necessarily pose a problem. Because presentation extra state generally consists of non-essential information, it is not a big problem that in some rare cases, it is reset to a default value.

Interpretation extra state, on the other hand, may represent essential information, but by restricting the edit behavior on presentations that have interpretation extra state, an editor designer can protect it from accidentally getting lost. For example, the edit behavior on a table of contents can be restricted to updates on titles, and insertion and deletion of entire entries. For these edit operations, the reuse of extra state does not fail. Furthermore, a

warning may be issued when document extra state is about to get lost, for example when a user deletes a table of contents entry.

4.2.6 Conclusions

The method of handling extra state, presented in this section, applies only to extra state that is clearly associated with a certain parent node. The method works for specifying the whitespace extra state for token presentations as well as interpretation extra state for editing partial presentations. However, the method of reusing is only sketched, and the exact way in which to reuse extra state is left to the editor designer.

Extra state is vulnerable, and it is easy to create situations in which reusing it is impossible, for example by allowing full text editing in a presentation that hides a large part of the document. However, the point is that the model allows those instances of extra state that are useful and for which a clear method of reuse can be established.

Extra state in one layer also has consequences for the other layers in the editor. In order to reuse presentation extra state, the higher-level nodes must have a reference to their previous presentation. Hence, the layer above must reuse the higher-level nodes when computing the higher level (by presenting the level above it). The references to the lower level can be considered presentation extra state of the higher level. The same thing holds for interpretation extra state: references from the lower level to the higher level can be considered interpretation extra state of the lower level.

Besides extra state that is attached to a certain parent node, other forms of extra state exist. For example, when a list in the document has a fixed order, but is allowed to be reordered by the user in the presentation, the order can be seen as either presentation or interpretation extra state.

Since extra state arises whenever a presentation or interpretation mapping has no unambiguous inverse, many other kinds of extra state exist. Further research is required not only to establish more precisely how to deal with extra state attached to a certain parent, but also to establish other kinds of extra state, as well as methods of handling it. Finally, it would be desirable to have automatic handling of extra state for certain kinds of presentations (or interpretations).

4.3 Duplicates in the presentation

Duplication occurs when a higher-level structure is presented on more than one lower-level structure. An example of duplication is a chapter title that appears both in the derived table of contents as well as in the document. But also multiple windows with (possibly different) presentations of the same part of the document give rise to duplications.

The problem with duplicate values is that if only one of the duplicates is edited, a conflict may arise during interpretation, and the editor needs to choose which value to use for

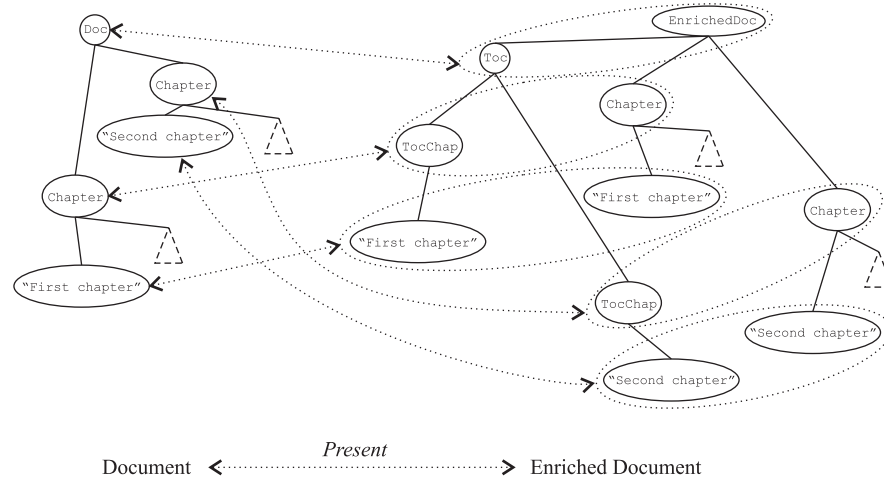


Figure 4.9: The presentation of a word processor document with a table of contents.

computing the higher level. As mentioned already in Section 3.5.5, we tackle the problem by giving priority to the changed duplicate. In case both duplicates have been edited and yield different higher-level results, the editor can either prohibit the edit operation, or make a default choice for one of the duplicates.

It is hard to give a precise definition of duplication of information without making additional assumptions on the presentation mappings and level types. Moreover, even if part of a presentation is a duplicate, the editor designer may prefer not to regard it as such. The "(" and ")" tokens in Figure 4.6 are both in the presentation of the Paren node, but it is not immediately obvious to consider these tokens duplicates. Consider for example a presentation $(1+2)*(3+4)$ in which the middle two parentheses are deleted. If parentheses are regarded as duplicates, the result will be the deletion of both pairs of parentheses, yielding $1+2*3+4$. However, it may be considered more natural to get the result $(1+2*3+4)$.

In Proxima, the only layers potentially giving rise to duplicates are the evaluation and the presentation layers. The presentation mappings for the other layers are largely predefined and do not duplicate any structures.

Figure 4.9 shows an example of a duplicated structure in the form of a table of contents for a word processor. On the left-hand side of the figure is a document that contains two chapters, of which only the titles are shown. The enriched document contains both a table of contents subtree (Toc), as well as a copy of the chapters. The table of contents subtree follows the structure of the original document, but only contains the title of each chapter rather than the chapter itself. To keep the figure simple, the table of contents only contains

chapters and not sections and subsections, but in a real editor the table of contents may reflect the entire document structure.

The evaluator maps a chapter in the document on a chapter in the enriched document, as well as on a chapter entry in the table of contents tree, in the latter case leaving out everything but the title. The presenter presents both the table of contents tree as well as the chapter tree, and does not duplicate any structures. The example also involves extra state, since a chapter is shown without its content in the table of contents. However, the extra state is orthogonal to the presence of duplicates and can be handled as sketched in Section 4.2.

When a user performs an update on the presentation, this results in an updated enriched document. Updates can be either in the table of contents, or in the chapters themselves. If the update is not in the table of contents, the document is computed from the chapters in the enriched document, while ignoring the table of contents altogether. On the other hand, if the table of contents tree is modified, the chapters in the enriched document tree are ignored on interpretation, and the document is obtained from the table of contents tree only. In this case, the chapter content is treated as interpretation extra state.

By treating the table of contents as a partial duplicate of the chapters, it is easy to support structural updates on the table of contents, such as swapping two titles. The reuse process for extra state takes care of swapping the chapter content correspondingly. The editor designer only needs to specify what happens when a new title is inserted, or when an entry is moved to a different level (e.g. a section to a subsection).

4.4 Conclusions

In this section we described the editing process of a layered editor, as well as two concepts that play a role in the design of such an editor. The first concept is extra state, which may have two forms. Presentation extra state is information that cannot be computed by presenting the higher level, whereas interpretation extra state cannot be computed by interpreting the lower level. Both forms may exist independently at one level.

When the higher and lower levels are tree structures, we can identify a form of extra state consisting of nodes (or subtrees) that are clearly attached to a certain parent node in the tree. Such extra state may be reused after an edit operation by looking up the parent node in the previous value of the level (before the edit operation was applied). If this fails, a default value is used.

The second concept that was discussed is the duplication of information in a presentation. When a presentation contains duplicates and a user edits one (or both) of these duplicates, a conflict may arise on interpretation. In Proxima, we resolve such conflicts by giving priority to the edited duplicate. In case both duplicates are edited, a default choice is made, or the edit operation is prohibited, depending on which behavior is specified by the editor designer.

Specifying a layered editor

In this chapter we develop a formal specification of a layered presentation-oriented editor, such as Proxima. We represent the presentation mapping *Present* by a function `present`. Given `present`, we state a number of requirements for a corresponding function `interpret`, which is an inverse of `present` that maps an updated presentation back onto the document. Furthermore, we specify how the data levels are updated after the presentation has been edited.

Because we make only few assumptions on the presentation mapping and the level types, the specification gives requirements for `interpret`, rather than an exact specification of how the inverse can automatically be computed for a specific `present`. The burden of defining an `interpret` function that meets the requirements lies with the editor designer.

An automatically derived `interpret` is desirable, but this is not yet feasible for the complex presentation functions of the use cases from Chapter 2. Examples of presentation formalisms supporting the automatic construction of an inverse function for basic presentation functions can be found in [57], [60], and [32].

We develop the specification of the layered editor by starting with a simplified editor without layers or extra state, and then step by step adding extra functionality. Section 5.1 specifies a simple editor that consists of a single layer and does not support extra state. In Section 5.2, the specification is extended with a general model of extra state. Section 5.3 provides a more concrete specification of extra state for tree data structures. The last step is the addition of layers to the specification in Section 5.4. Finally, Section 5.5 informally discusses how the specification may be adapted to handle duplications in the presentation.

5.1 A single layer

First we consider a simple abstract editor consisting of a single layer between two data levels. The higher level is the document level and the lower level is the presentation level. In this section, we assume that the presentation mapping *Present* maps each document onto exactly one presentation, and thus can be represented by a total function *present*:

$$\text{present} :: \text{Level}_H \rightarrow \text{Level}_L$$

Because not every value of type Level_L is the presentation of a document is, *present* need not be surjective.

Given *present*, we will specify a total function *interpret*, which maps a lower level back onto a higher level:

$$\text{interpret} :: \text{Level}_L \rightarrow \text{Level}_H$$

Note that in contrast to the architecture description in Chapter 3, *present* and *interpret* in the specification are functions between values rather than edit operations.

The INTERPRESENT requirement

A minimal condition for a *present* and *interpret* pair to model an editor is that presentation of a higher level followed by interpretation should yield the original higher level. In other words, *interpret* is a left inverse of *present*. We express this with the INTERPRESENT requirement:

$$l = \text{present } h \Rightarrow h = \text{interpret } l \qquad \text{INTERPRESENT}$$

Or, equivalently, using function composition:

$$\text{interpret} \circ \text{present} = \text{id}_{\text{Level}_H}$$

INTERPRESENT implies injectivity of *present* and (together with totality of *present*) surjectivity of *interpret*.

Note that we do not require *interpret* to be a right inverse ($\text{present} \circ \text{interpret} = \text{id}_{\text{Level}_L}$). The reason for this is to make it possible to interpret edit operations that are not entirely exact but still clearly show the intended update.

For example, take a presentation function that presents a return statement by using a bold style for the keyword: $\text{present}(\text{Return True}) = \text{“return True”}$. If we want to allow a user to enter a return statement without having to add the exact styles to the text, we must have $\text{interpret} \text{“Return True”} = \text{Return True}$. Hence, $\text{present} \circ \text{interpret}$ is not the identity, since $\text{present} \circ \text{interpret} \text{“Return True”} = \text{“return True”}$.

Since we assume *present* is given, INTERPRESENT is a requirement for *interpret*. Yet, it is only a minimal requirement. In order to model an editor, we also want to say something about the result of *interpret* on lower-level values outside the range of *present*. The next section provides such a requirement.

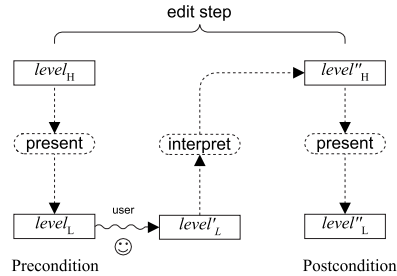


Figure 5.1: A single edit step.

5.1.1 Editing

A user may perform an edit operation on either the higher level (document) or the lower level (presentation). Because after a document update, the presentation invariant can be restored by simply presenting the updated document, we only consider presentation-oriented editing here. When the lower level is edited, we need to interpret the updated lower level in order to find a new higher level, which subsequently may need to be presented again (as explained in Section 4.1).

Let $level_L$ and $level_H$ denote the state of the layer at the beginning of an edit step.

$$level_L = \text{present } level_H$$

The user then updates the lower level:

$$level_L \rightsquigarrow level'_L$$

after which the editor computes the final values for the data levels: $level''_L$ and $level''_H$, in order to restore the presentation invariant. Figure 5.1 (which was also shown in the previous chapter) illustrates the edit process.

Besides the presentation invariant, we impose several other requirements on the final levels.

Firstly, if $level'_L$ is a valid presentation of $level_H$, we do not want $level_H$ to change because of the edit operation, and thus $level''_H = level_H$. Because, in this section, the presentation mapping is a function, $level'_L$ can only be a valid presentation of $level_H$ if it is equal to $level_L$, meaning that the user did not change the lower level. However, in the next section, when extra state is added to the model, the reason for this property becomes clear. Because the property states that the document stays the same under an edit operation, we refer to it as DOC-INERT.

Secondly, if a user edits the lower level in such a way that $level'_L$ is a valid presentation of some higher-level value, then the editor should not perform any further updates on the

lower level, and the final value $level''_L$ should be equal $level'_L$. Analogous to the previous property, this property is referred to as PRES-INERT.

Our final requirement specifies what happens when $level'_L$ is not a valid presentation of any higher level. In this case, the editor could simply forbid the edit operation, but a more user-friendly solution is to regard the operation as an inexact edit operation and perform the intended edit operation. This means that the editor chooses a $level''_H$ and $level''_L$ such that $level''_L$ is as close as possible to the $level'_L$ that came from the user (and, of course, $level''_L = \text{present } level''_H$).

Summarizing, if we let *Comp* be a program fragment that assigns new values to the data levels, then the requirements on *Comp*, specified as Hoare triples, are:

$\{\text{true}\} \text{Comp } \{level''_L = \text{present } level''_H\}$	POSTCONDITION
$\{level'_L = \text{present } level_H\} \text{Comp } \{level''_H = level_H\}$	DOC-INERT
$\{level'_L = \text{present } h\} \text{Comp } \{level'_L = level''_L\}$	PRES-INERT
$\{\text{true}\} \text{Comp } \{level'_L \text{ “close to” } level''_L\}$	INTENDED

Note that PRES-INERT is a special case of INTENDED, in which “close to” is equality. In order to prevent a conflict with POSTCONDITION, INTENDED is slightly weaker than POSTCONDITION.

Definition of *Comp*

A straightforward definition of *Comp* that meets all requirements is:

$$\begin{aligned} \text{Comp} &\hat{=} \begin{array}{l} level''_H := \text{interpret } level'_L; \\ level''_L := \text{present } level''_H \end{array} \\ &\hspace{15em} \text{Comp-DEF} \end{aligned}$$

Together with *Comp*, the requirements above provide a specification of *interpret*. We can prove that POSTCONDITION holds, and furthermore, if INTERPRESENT holds, then DOC-INERT and PRES-INERT can be proved as well. INTENDED cannot be proved from INTERPRESENT, and hence is a requirement on *interpret* (together with *Comp*).

Basic Hoare logic

Before we prove the requirements, we introduce a few laws from the Hoare calculus. We use the following law to relate Hoare triples to weakest precondition propositions:

$$\{P\} S \{Q\} \equiv P \Rightarrow wp(S, Q) \hspace{10em} \text{wp-CHAR}$$

Because the program only consists of assignments and a composition, we do not need to consider termination, and moreover, we only need two laws from the weakest precondition calculus. For assignments we have:

$$\text{wp}(x := e, P) \equiv P[x/e] \quad \text{wp-:=}$$

And for composition we have:

$$\text{wp}(S; T, P) \equiv \text{wp}(S, \text{wp}(T, P)) \quad \text{wp-;}$$

In Section 5.4.4 we need two more laws, which we introduce here as well, for completeness. We need a law for weakening the postcondition.

$$(Q \Rightarrow Q') \Rightarrow (\text{wp}(S, Q) \Rightarrow \text{wp}(S, Q')) \quad \text{wp-MONO}$$

And a law for combining two postconditions.

$$(P \Rightarrow \text{wp}(S, Q)) \wedge (P \Rightarrow \text{wp}(S, Q')) \Rightarrow (P \Rightarrow \text{wp}(S, Q \wedge Q')) \quad \text{wp-AND}$$

POSTCONDITION requirement

According to wp-CHAR, $\{\text{true}\} \text{Comp} \{level''_L = \text{present } level''_H\}$ is equivalent to:

$$\text{true} \Rightarrow \text{wp}(\text{Comp}, level''_L = \text{present } level''_H)$$

Proof:

$$\begin{aligned} & \text{wp}(\text{Comp}, level''_L = \text{present } level''_H) \\ \equiv & \quad \{ \text{Comp-DEF} \} \\ & \text{wp}(level''_H := \text{interpret } level'_L; level''_L := \text{present } level''_H \\ & \quad , level''_L = \text{present } level''_H) \\ \equiv & \quad \{ \text{wp-;} \} \\ & \text{wp}(level''_H := \text{interpret } level'_L \\ & \quad , \text{wp}(level''_L := \text{present } level''_H, level''_L = \text{present } level''_H)) \\ \equiv & \quad \{ \text{wp-:=} \} \\ & \text{wp}(level''_H := \text{interpret } level'_L, \text{present } level''_H = \text{present } level''_H) \\ \equiv & \quad \{ \text{reflexivity of } = \} \\ & \text{wp}(level''_H := \text{interpret } level'_L, \text{true}) \\ \equiv & \quad \{ \text{wp-:=} \} \\ & \text{true} \end{aligned}$$

□

DOC-INERT requirement

In the proofs below, we will implicitly use the equivalence between the Hoare notation and the weakest precondition notation suggested by wp-CHAR. Hence, DOC-INERT is:

$$level'_L = \text{present } level_H \Rightarrow \text{wp}(Comp, level''_H = level_H)$$

Proof:

$$\begin{aligned}
& \text{wp}(Comp, level_H = level''_H) \\
\equiv & \quad \{ Comp-DEF \} \\
& \text{wp}(level''_H := \text{interpret } level'_L; level'_L := \text{present } level''_H, level_H = level''_H) \\
\equiv & \quad \{ \text{wp-}; \} \\
& \text{wp}(level''_H := \text{interpret } level'_L, \text{wp}(level'_L := \text{present } level''_H, level_H = level''_H)) \\
\equiv & \quad \{ \text{wp-} := \} \\
& \text{wp}(level''_H := \text{interpret } level'_L, level_H = level''_H) \\
\equiv & \quad \{ \text{wp-} := \} \\
& level_H = \text{interpret } level'_L \\
\Leftarrow & \quad \{ INTERPRESENT \} \\
& level'_L = \text{present } level_H
\end{aligned}$$

□

PRES-INERT requirement

The proof of PRES-INERT is similar to the proof of DOC-INERT

$$level'_L = \text{present } h \Rightarrow \text{wp}(Comp, level'_L = level''_L)$$

Proof:

In the proof, we assume $level'_L = \text{present } h$:

$$\begin{aligned}
& \text{wp}(Comp, level'_L = level''_L) \\
\equiv & \quad \{ Comp-DEF \} \\
& \text{wp}(level''_H := \text{interpret } level'_L; level'_L := \text{present } level''_H, level'_L = level''_L) \\
\equiv & \quad \{ \text{wp-}; \} \\
& \text{wp}(level''_H := \text{interpret } level'_L, \text{wp}(level'_L := \text{present } level''_H, level'_L = level''_L)) \\
\equiv & \quad \{ \text{wp-} := \} \\
& \text{wp}(level''_H := \text{interpret } level'_L, level'_L = \text{present } level''_H) \\
\equiv & \quad \{ \text{wp-} := \} \\
& level'_L = \text{present } (\text{interpret } level'_L) \\
\Leftarrow & \quad \{ \text{assumption} \}
\end{aligned}$$

$$\begin{aligned}
 level'_L &= \text{present} (\text{interpret} (\text{present } h)) \\
 \equiv & \quad \{ \text{INTERPRESENT} \} \\
 level'_L &= \text{present } h
 \end{aligned}$$

□

INTENDED requirement

If $level'_L$ is not a valid presentation, only `POSTCONDITION` and `INTENDED` are of importance:

$$\{ \text{true} \} \text{Comp} \{ level'_L \text{ "close to" } level''_L \}$$

This requirement states that `interpret` returns a higher level such that its presentation resembles what the user intended with the edit operation. Because the intention of a user is a rather vague concept, we cannot give a formal specification of the “close to” relation here. It is left to the editor designer to assign a meaning to it and provide an `interpret` function that meets the requirement given.

5.2 Extra state

In this section, we extend the specification of the single-layered editor with extra state (see Section 4.2). The document level may contain interpretation extra state, whereas the presentation may contain presentation extra state.

In the presence of extra state, a single document may have several presentations, and a single presentation may be the presentation of several documents. This means that we can no longer represent the relation *Present* by a function `present` between values, since `present h` would have to be a set of lower-level levels. Similarly, `interpret l` would have to return a set of higher-level values.

To be able to use functional representations for the presentation and interpretation mappings after all, we introduce an equivalence class representation for extra state. Because two levels that only differ in extra state are in a sense equivalent, we model extra state using an equivalence relation on the data level. The elements of an equivalence class are equal except for their extra state. Using this equivalence class representation for extra state, we can express `present` and `interpret` as set-valued functions between equivalence classes.

We provide two examples to illustrate the equivalence class representation for extra state. Both examples are set in an editor for a simple programming language, for which layout is not part of the syntax.

As an example of interpretation extra state, we take a partially presented declaration. If we hide the right-hand side of a declaration, `Decl "f" (Sum 1 2)` is presented as `f = ...`, which is also a presentation of, for example, `Decl "f" (Product 3 4)`.

Thus, the right-hand sides (Sum 1 2) and (Product 3 4) are interpretation extra state. If we denote the extra state equivalence relation by $H :: Level_H \sim Level_H$, we have $(Decl\ "f"\ (Sum\ 1\ 2))\ H\ (Decl\ "f"\ (Product\ 3\ 4))$, which means that the two declarations are equal up to extra state. More generally, according to H a declaration is equivalent to every declaration that has the same left-hand side identifier.

Whitespace is an example of presentation extra state. Consider an expression Sum 1 2, which is presented as “1_+_2”. The presentation is equivalent to a string containing ‘1’, ‘+’, and ‘2’ with any configuration of whitespace. Let $L :: Level_L \sim Level_L$ be the equivalence relation for presentation extra state, then, for example, “1_+_2” L “1+2”.

Because no assumptions are made about the types of the two levels, the model for extra state is rather abstract. Section 5.3 provides a concrete representation of extra state for tree-structured data types.

5.2.1 Equivalence classes

We start by introducing some notation for equivalence classes. A typed binary relation $R :: T \sim T$ is an equivalence relation if it is reflexive, symmetric, and transitive. The equivalence class for a value $x :: T$ is denoted by $[x]_R$. Its definition is:

$$[x]_R = \{y \mid x R y\}$$

The factor set T/R is the set of all equivalence classes of R . It is a set of mutually exclusive and jointly exhaustive subsets of T (if we regard T as a set).

$$T/R = \{[x]_R \mid x :: T\}$$

We have the following property for an equivalence relation R :

$$x \in [y]_R \equiv [x]_R = [y]_R \quad \text{[-]-MEMBER}$$

5.2.2 An equivalence class for extra state

If several lower-level values are related to the same higher-level value by the presentation relation, this means that, when disregarding extra state, the lower-level values are equal. Similarly, if two higher-level values are related to the same lower value, the higher-level values are equal up to extra state. We use an equivalence relation to express that two values are equal up to extra state.

The two data levels give rise to two equivalence relations: H and L .

$$H :: Level_H \sim Level_H \quad \text{and} \quad L :: Level_L \sim Level_L$$

In order to specify that extra state should be reused after an update, without making any assumptions on the level type, we use the “close to” notion from the previous section. We

specify that an updated value x' contains reused extra state from x , by requiring that x' is the element of its equivalence class that is as close as possible to x .

As an example, consider an update on the presentation of Decl "f" (Sum 1 2) from the example above: "f = ..." \rightsquigarrow "g = ...". The new higher level should have left-hand side "g" and reuse the sum. This can be specified by requiring that from the equivalence class of declarations with left-hand side "g", we select the element that is as close as possible to Decl "f" (Sum 1 2). The obvious solution is Decl "g" (Sum 1 2).

5.2.3 Presenting and interpreting

In the presence of extra state, both the presentation and interpretation mappings may have several results for a single argument, and hence cannot directly be represented by functions anymore. Regarding the mappings as relations (with the levels ordered as in Section 4.1), we have:

$$\begin{aligned} \textit{Present} &:: \textit{Level}_L \sim \textit{Level}_H \\ \textit{Interpret} &:: \textit{Level}_H \sim \textit{Level}_L \end{aligned}$$

If we model the extra state on the higher and lower levels with two equivalence relations H and L , we can express *Present* and *Interpret* as functions between equivalence classes:

$$\begin{aligned} \textit{present} &:: \textit{Level}_{H/H} \rightarrow \textit{Level}_{L/L} \\ \textit{interpret} &:: \textit{Level}_{L/L} \rightarrow \textit{Level}_{H/H} \end{aligned}$$

The correspondence between functions *present* and *interpret*, and relations *Present* and *Interpret* is made explicit by the two characterizations:

$$\begin{aligned} l \textit{Present} h &\equiv [l]_L = \textit{present} [h]_H && \textit{present-CHAR} \\ h \textit{Interpret} l &\equiv [h]_H = \textit{interpret} [l]_L && \textit{interpret-CHAR} \end{aligned}$$

In the remainder of this section, we will use the functional representation.

INTERPRESENT requirement

The INTERPRESENT requirement changes slightly, since we use equivalence class notation to make explicit that the arguments and results are equivalence classes:

$$[l]_L = \textit{present} [h]_H \Rightarrow [h]_H = \textit{interpret} [l]_L \quad \textit{INTERPRESENT}$$

Or, equivalently:

$$\textit{interpret} \circ \textit{present} = \textit{id}_{\textit{Level}_{H/H}}$$

Similar to the previous section, INTERPRESENT states that *interpret* is a left inverse of *present* and implies injectivity of *present* as well as surjectivity of *interpret*.

5.2.4 Editing

Similar to Section 5.1.1, we assign new values to the higher and the lower data levels when a user edits the lower level:

$$\begin{aligned} [level_L]_L &= \text{present } [level_H]_H \\ level_L &\rightsquigarrow level'_L \end{aligned}$$

The old requirements, rewritten with equivalence classes, are:

$$\begin{aligned} \{\text{true}\} \text{Comp } \{[level''_L]_L &= \text{present } [level''_H]_H\} && \text{POSTCONDITION} \\ \{[level'_L]_L &= \text{present } [level_H]_H\} \text{Comp } \{level_H &= level''_H\} && \text{DOC-INERT} \\ \{[level'_L]_L &= \text{present } [h]_H\} \text{Comp } \{level'_L &= level''_L\} && \text{PRES-INERT} \\ \{\text{true}\} \text{Comp } \{level'_L &\text{“close to” } level''_L\} && \text{INTENDED} \end{aligned}$$

With extra state, the DOC-INERT requirement becomes interesting, since it is now possible to update $level_L$ to a (different) $level'_L$, which is still a valid presentation of $level_H$. For example, if whitespace is not stored in the document, and a user edits only whitespace, then the document should remain unchanged.

In addition to the four requirements of the previous section, we need to require that extra state on both levels is reused after an update. For interpretation extra state on the higher level, we require that from the equivalence class specified by POSTCONDITION, the element that is as close as possible to the original value $level_H$ is selected:

$$\{\text{true}\} \text{Comp } \{level_H \text{ “close to” } level''_H\} \quad \text{DOC-PRESERVE}$$

DOC-PRESERVE is weaker than the other requirements. It specifies which element of the equivalence class determined by POSTCONDITION is the final value of the lower level.

On the lower level, we need a similar requirement for presentation extra state, but here it coincides with INTENDED. Thus, INTENDED has a double function: it states that the resulting equivalence class of `present` must be close to the equivalence class of $level'_L$, as well as that the presentation extra state in $level''_L$ must resemble the extra state in $level'_L$ as much as possible.

Reusing extra state

`Comp` realizes a mapping between data levels, but both `present` and `interpret` are mappings between equivalence classes on these data levels. Hence, we need a way to get from values to equivalence classes and back. We use the $[-]$ function to denote the equivalence class

of a value. For going from an equivalence class to a value, we need a selection function that takes as an extra argument the old value of the level.

The resulting class of `present` or `interpret` contains all possible extra state configurations. From this class, we need to select an element for which the extra state resembles the extra state of the previous value as much as possible. Thus, we introduce the function $\triangleright :: X'_R \rightarrow X \rightarrow X$, which takes an equivalence class and a previous value, and selects the element from the class that is closest to the previous value. Intuitively, \triangleright “reuses” the extra state from the previous value. In this section, we will only give a number of requirements for \triangleright . Section 5.3 provides a more concrete instance.

Because we need to reuse extra state on the results of `present` as well as `interpret`, we need two \triangleright functions: one for each level. In order to disambiguate between the two functions, we add the corresponding relation as a subscript (e.g. \triangleright_H and \triangleright_L). For the higher level we have $\triangleright_H :: Level_{H/H} \rightarrow Level_H \rightarrow Level_H$, and for the lower level $\triangleright_L :: Level_{L/L} \rightarrow Level_L \rightarrow Level_L$.

We require several properties of \triangleright . First of all, the equivalence class that is the result of \triangleright must be equal to the argument equivalence class:

$$[[x]_R \triangleright y]_R = [x]_R \quad \triangleright\text{-VALID}$$

Furthermore, if the value argument is in the equivalence class argument, then \triangleright returns the value argument.

$$[y]_R = [x]_R \Rightarrow [x]_R \triangleright y = y \quad \triangleright\text{-IDEM}$$

Note that, by $\triangleright\text{-VALID}$, we also have the reverse: $[x]_R \triangleright y = y \Rightarrow [y]_R = [x]_R$.

Finally, if the value argument is not in the equivalence class argument, then \triangleright returns an element of that class that is as close as possible to the value argument.

$$[x]_R \triangleright y \text{ “close to” } y \quad \triangleright\text{-CLOSE}$$

Similar to the `DOC-INERT` and `INTENDED` requirements, $\triangleright\text{-IDEM}$ is a special case of $\triangleright\text{-CLOSE}$, for which “close to” is equality.

Definition of *Comp*

Using $[-]$ and \triangleright we can define *Comp*. The definition is split in *Up* and *Dwn* to make the proofs of the requirements more readable.

$$Comp \hat{=} Up; Dwn \quad Comp\text{-DEF}$$

$$Up \hat{=} level''_H := \text{interpret } [level'_L]_L \triangleright_H level_H \quad Up\text{-DEF}$$

$$Dwn \hat{=} level''_L := \text{present } [level''_H]_H \triangleright_L level'_L \quad Dwn\text{-DEF}$$

Similar to the previous section, we can prove that *Comp* meets the POSTCONDITION, and, if we assume INTERPRESENT, also DOC-INERT and PRES-INERT.

POSTCONDITION requirement

The weakest precondition notation of POSTCONDITION is:

$$\text{true} \Rightarrow \text{wp}(\text{Comp}, [level''_{L}] = \text{present } [level''_{H}])$$

Proof:

$$\begin{aligned} & \text{wp}(\text{Comp}, [level''_{L}] = \text{present } [level''_{H}]) \\ \equiv & \quad \{ \text{Comp-DEF} \} \\ & \text{wp}(\text{Up}; \text{Dwn}, [level''_{L}] = \text{present } [level''_{H}]) \\ \equiv & \quad \{ \text{wp-}; \} \\ & \text{wp}(\text{Up}, \text{wp}(\text{Dwn}, [level''_{L}] = \text{present } [level''_{H}])) \\ \equiv & \quad \{ \text{Dwn-DEF} \} \\ & \text{wp}(\text{Up} \\ & \quad , \text{wp}(level''_{L} := \text{present } [level''_{H}] \triangleright_L level'_{L}, [level''_{L}] = \text{present } [level''_{H}])) \\ \equiv & \quad \{ \text{wp-} := \} \\ & \text{wp}(\text{Up}, [\text{present } [level''_{H}] \triangleright_L level'_{L}] = \text{present } [level''_{H}]) \\ \equiv & \quad \{ \triangleright\text{-VALID} \} \\ & \text{wp}(\text{Up}, \text{present } [level''_{H}] = \text{present } [level''_{H}]) \\ \equiv & \quad \{ \text{reflexivity of } = \} \\ & \text{wp}(\text{Up}, \text{true}) \\ \equiv & \quad \{ \text{Up-DEF} \} \\ & \text{wp}(level''_{H} := \text{interpret } [level'_{L}] \triangleright_H level_H, \text{true}) \\ \equiv & \quad \{ \text{wp-} := \} \\ & \text{true} \end{aligned}$$

□

DOC-INERT requirement

$$[level'_{L}] = \text{present } [level_H] \Rightarrow \text{wp}(\text{Comp}, level_H = level''_H)$$

Proof:

$$\begin{aligned} & \text{wp}(\text{Comp}, level_H = level''_H) \\ \equiv & \quad \{ \text{Comp-DEF} \} \\ & \text{wp}(\text{Up}; \text{Dwn}, level_H = level''_H) \end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{wp-; } \} \\
&\quad \text{wp}(Up, \text{wp}(Dwn, level_H = level''_H)) \\
&\equiv \{ Dwn\text{-DEF } \} \\
&\quad \text{wp}(Up, \text{wp}(level'_L := \text{present } [level''_H]_H \triangleright_L level'_L, level_H = level''_H)) \\
&\equiv \{ \text{wp-:= } \} \\
&\quad \text{wp}(Up, level_H = level''_H) \\
&\equiv \{ Up\text{-DEF } \} \\
&\quad \text{wp}(level''_H := \text{interpret } [level'_L]_L \triangleright_H level_H, level_H = level''_H) \\
&\equiv \{ \text{wp-:= } \} \\
&\quad level_H = \text{interpret } [level'_L]_L \triangleright_H level_H \\
\leftarrow &\quad \{ \triangleright\text{-IDEM } \} \\
&\quad [level_H]_H = \text{interpret } [level'_L]_L \\
&\equiv \{ \text{INTERPRESENT } \} \\
&\quad [level'_L]_L = \text{present } [level_H]_H \\
&\square
\end{aligned}$$

PRES-INERT requirement

$$[level'_L]_L = \text{present } [h]_H \Rightarrow \text{wp}(Comp, level'_L = level''_L)$$

Proof:

We assume $[level'_L]_L = \text{present } [h]_H$.

$$\begin{aligned}
&\text{wp}(Comp, level'_L = level''_L) \\
&\equiv \{ Comp\text{-DEF } \} \\
&\quad \text{wp}(Up; Dwn, level'_L = level''_L) \\
&\equiv \{ \text{wp-; } \} \\
&\quad \text{wp}(Up, \text{wp}(Dwn, level'_L = level''_L)) \\
&\equiv \{ Dwn\text{-DEF } \} \\
&\quad \text{wp}(Up, \text{wp}(level''_L := \text{present } [level''_H]_H \triangleright_L level'_L, level'_L = level''_L)) \\
&\equiv \{ \text{wp-:= } \} \\
&\quad \text{wp}(Up, level'_L = \text{present } [level''_H]_H \triangleright_L level'_L) \\
&\equiv \{ Up\text{-DEF } \} \\
&\quad \text{wp}(level''_H := \text{interpret } [level'_L]_L \triangleright_H level_H \\
&\quad \quad , level'_L = \text{present } [level''_H]_H \triangleright_L level'_L)
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{wp-:=} \} \\
&\quad \text{level}'_L = \text{present} [\text{interpret} [level''_L]_L \triangleright_H \text{level}_H]_H \triangleright_L \text{level}'_L \\
&\equiv \{ \text{assumption} \} \\
&\quad \text{level}'_L = \text{present} [(\text{interpret} (\text{present} [h]_H) \triangleright_H \text{level}_H)]_H \triangleright_L \text{level}'_L \\
&\equiv \{ \text{INTERPRESENT} \} \\
&\quad \text{level}'_L = \text{present} [([h]_H \triangleright_H \text{level}_H)]_H \triangleright_L \text{level}'_L \\
&\equiv \{ \triangleright\text{-VALID} \} \\
&\quad \text{level}'_L = \text{present} [h]_H \triangleright_L \text{level}'_L \\
&\equiv \{ \text{assumption} \} \\
&\quad \text{level}'_L = [level''_L]_L \triangleright \text{level}'_L \\
&\equiv \{ \triangleright\text{-IDEM} \} \\
&\quad [level''_L]_L = [level''_L]_L \\
&\equiv \{ \text{reflexivity of } = \} \\
&\quad \text{true} \\
&\square
\end{aligned}$$

DOC-PRESERVE and INTENDED requirements

For the DOC-PRESERVE and INTENDED requirements, we give an informal argumentation because of the informal nature of the “close to” requirement.

From *Up-DEF*, we know that $level''_H$ is the result of $\text{interpret} [level''_L]_L \triangleright_H \text{level}_H$, which implies $level_H$ “close to” $level''_H$ by $\triangleright\text{-CLOSE}$. Hence, DOC-PRESERVE holds.

The INTENDED requirement is somewhat more subtle, since it has a double function. On the one hand, it states that the equivalence class $[level''_L]_L$ must be close to $[level'_L]_L$. This corresponds to the INTENDED requirement in Section 5.1.1 without extra state. In this sense, closeness is used to express that the final lower level resembles what the user intended. Thus, if $level'_L$ is not a valid presentation, then $[level''_H]_H$, must be chosen such that its presentation equivalence class $[level''_L]_L$ is close to $[level'_L]_L$. This part of the requirement is therefore a specification of *interpret*.

On the other hand, INTENDED also states that from the equivalence class $[level''_L]_L$, we want to choose an element that is as close as possible to $level'_L$ regarding its extra state. This closeness is guaranteed by $\triangleright\text{-CLOSE}$ in the same way as on the higher level for DOC-PRESERVE.

5.3 A wildcard representation for equivalence classes

In this section, we give a representation for equivalence classes for a simple kind of tree data structures. We introduce a wildcard notation for representing extra state. Furthermore, we provide an instance of the reuse function (\triangleright), specific to wildcard values.

The specification for reusing extra state is a first step, and needs to be extended further to allow reusing extra state for more edit operations. The final subsection sketches such an extension.

5.3.1 Trees with wildcards

In a tree-structured data type, we can model extra state by leaving certain parts of the tree undetermined. We do this by representing each extra state-part by a wildcard, which stands for any possible value of the correct type. The functions `present` and `interpret` return a value that contains wildcards, thus leaving the extra state in the result undetermined.

For a type T , the corresponding wildcard is denoted by $*_T$ and has type $*_T$ (note the different font):

```
data  $*_T = *_T$ 
```

The wildcard for type T represents any possible value of type T . Thus, the value $*_{Bool}$ represents any value from the set `{True, False}`, and the value $*_{Int}$ represents any integer. An example shows how wildcards can be used to model extra state equivalence classes in a data type.

Consider a simple binary tree data type:

```
data Tree = Bin Bool Tree Tree | Leaf Bool Int
```

We can recursively define a type $Tree^*$ in which we specify for each child type whether or not it is extra state. If, for example, we wish to declare the boolean in the `Bin` node and the integer in the `Leaf` as extra state, we represent these children by wildcards:

```
data  $Tree^* = \text{Bin } *_{Bool} \text{ } Tree^* \text{ } Tree^* \text{ } | \text{Leaf } Bool^* \text{ } *_{Int}$ 
```

Note that for a type without children, such as `Bool`, the wildcard type is the same as the original type ($Bool^* = Bool$). Nevertheless, for uniformity, we replace every child type T by a type T^* , even if T has no children.

For simplicity, the constructors for type T^* are the same as for T . A sample value of type $Tree^*$ is: `Bin $*_{Bool}$ (Leaf True $*_{Int}$) (Leaf False $*_{Int}$)`.

For a type T , a type T^* specifies which parts of it are extra state. Note that for a single type T several types T^* are possible: whether or not a value is extra state is not determined

by its type, but rather by the T^* declarations in which the value appears as a child. In the example, a boolean in a `Bin` node is extra state, whereas a boolean in a `Leaf` is not. The editor designer specifies which parts of a *Level* type are extra state by declaring an appropriate *Level**.

Because a wildcard represents the set of all possible values of a certain type, a value containing a wildcard also represents a set of values. A value that contains several wildcards also represents the set that is formed by taking all possible combinations of values for the wildcards.

We use the notation $\llbracket x \rrbracket$ for the set of values represented by a value $x :: T^*$. Thus, $\llbracket C *_{Bool} *_{Bool} \rrbracket = \{C \text{ True True}, C \text{ True False}, C \text{ False True}, C \text{ False False}\}$.

In order to obtain a T^* value for a value of type T , we define a function $\text{core} :: T \rightarrow T^*$, which drops all information that is masked by a wildcard in the T^* definition. For example, for the type definition $\text{data } T^* = T \text{ Int } *_{Bool}$, we have $\text{core } (T \ 1 \ \text{True}) = T \ 1 \ *_{Bool}$.

The root of the document cannot be extra state, because it is not a child of any constructor. Even though it is unlikely that we want the root to be extra state, it may still be declared as such by using a dummy root type $\text{data } R = R \ \text{Root}$, and declare the *Root* child as extra state: $\text{data } R^* = R \ *_{Root}$.

In general, we regard a recursive first-order Haskell data type T as:

$$\text{data } T = C_0 \ T_{0,0} \dots T_{0,m_0} \mid \dots \mid C_n \ T_{n,0} \dots T_{n,m_n}$$

We can specify which parts of T are extra state by defining a wildcard type T^* . In the definition of T^* , we specify for each child of each constructor of T whether or not is extra state. Thus, for constructor C_i , child number j is either $T_{i,j}^*$ or $*_{T_{i,j}}$.

$$\begin{aligned} \text{data } T^* = & C_0 \ (T_{0,0}^* \text{ or } *_{T_{0,0}}) \dots (T_{0,m_0}^* \text{ or } *_{T_{0,m_0}}) \\ & \mid \dots \\ & \mid C_n \ (T_{n,0}^* \text{ or } *_{T_{n,0}}) \dots (T_{n,m_n}^* \text{ or } *_{T_{n,m_n}}) \end{aligned}$$

If we use the notation $\wp T$ for the power set of T , the definition of $\llbracket _ \rrbracket$ is:

$$\begin{aligned} \llbracket _ \rrbracket &:: T^* \rightarrow \wp T \\ \llbracket *T \rrbracket &= \{x \mid x :: T\} \\ \llbracket C \ x_0^* \dots x_n^* \rrbracket &= \{C \ x_0 \dots x_n \mid x_0 \in \llbracket x_0^* \rrbracket \wedge \dots \wedge x_n \in \llbracket x_n^* \rrbracket\} \end{aligned} \quad \llbracket _ \rrbracket\text{-DEF}$$

And the definition of core is:

$$\begin{aligned} \text{data } T^* = & \dots \mid C_i \ U_{i,0} \dots U_{i,m_i} \mid \dots \quad (\text{with } U_{i,j} = T_{i,j}^* \text{ or } *_{T_{i,j}}) \\ \text{core} &:: T \rightarrow T^* \\ \text{core } (C_i \ x_0 \dots x_{m_i}) &= C_i \ x'_0 \dots x'_{m_i} \\ \text{where } x'_j &= \begin{cases} *_{T_{i,j}} & , \text{ if } U_{i,j} = *_{T_{i,j}} \\ \text{core } x_j & , \text{ otherwise} \end{cases} \end{aligned} \quad \text{core-DEF}$$

5.3.2 T^* induces an equivalence relation on T

We can define two values of type T to be equivalent when they are equal, or only differ in extra state. The resulting binary relation on T is denoted by $\simeq :: T \sim T$. For the definition of \simeq we use the function `core`, which is uniquely determined by T^* :

$$x \simeq y \hat{=} \text{core } x = \text{core } y \quad \simeq\text{-DEF}$$

To see that \simeq is an equivalence relation, we need the theorem below, the simple proof of which has been omitted:

$$\text{"}R \text{ is an equivalence relation"} \equiv \exists f : \forall x, y : x R y \equiv (f x = f y)$$

By substituting `core` for f and \simeq for R in this theorem, we can immediately conclude that \simeq is an equivalence relation on T .

Without giving a proof, we mention that the equivalence classes of \simeq can be expressed using $\llbracket _ \rrbracket$ and `core`.

$$\llbracket x \rrbracket_{\simeq} = \llbracket \text{core } x \rrbracket \quad \simeq\text{-CLASSES}$$

5.3.3 Reuse on wildcard types

We use wildcard types to represent the results of `present` and `interpret`. Because each wildcard represents a set of values, we need to select an element from each of these sets to obtain a final value for the result. This selection corresponds to filling in a value of type T for each wildcard $*_T$ in the result. The values that are filled in for the wildcards are taken from the previous value of the level, if possible.

In this section, we define a function \triangleright^* that fills in the wildcards in its first argument by reusing values from its second argument. The type of \triangleright^* is:

$$\triangleright^* :: T^* \rightarrow T \rightarrow T$$

We give two examples to show how \triangleright^* is used.

Consider a `Token` data type with a presentation extra state tuple to denote the whitespace. If a token for "False" with a whitespace of one line break and three spaces is updated to a token "True", then the whitespace of the old token is reused by \triangleright^* :

$$(\text{Token } *_{(Int,Int)} \text{ "True"}) \triangleright^* (\text{Token } (1,3) \text{ "False"}) = \text{Token } (1,3) \text{ "True"}$$

For the second example, we take the declaration with the hidden right-hand side from Section 5.2.2: `Decl "f" (Sum 1 2)`. If a user renames the left-hand side in the presentation to `g`, the result of `interpret` contains the new identifier "g", but the right-hand side is interpretation extra state and will be a $*_{Exp}$. Again, \triangleright^* recovers the extra state from the previous value:

$(\text{Decl } "g" \ *_{Exp}) \triangleright^* (\text{Decl } "f" \ (\text{Sum } 1 \ 2)) = \text{Decl } "g" \ (\text{Sum } 1 \ 2)$

In some cases, reusing extra state from a previous value is not possible. For example, when a new subtree is inserted in a level, its extra state will not be present in the previous value of the level. Furthermore, if a level is structurally changed, even if all extra state is present in the previous value, it may not always be possible to recover it. In case extra state cannot be reused, we use a function to obtain a default value: $\text{default} :: a^* \rightarrow a$.

We give a default function for the declaration example. If a new declaration is inserted, its hidden right-hand side is initialized to the special expression `Undefined`.

$$\text{default } (\text{Decl } i \ e) = \begin{cases} \text{Decl } i \ \text{Undefined} & , \text{if } e = *_{Exp} \\ \text{Decl } i \ e & , \text{otherwise} \end{cases}$$

We require that the result of `default` is equal to its argument, except for the wildcards in the argument. Wildcards are replaced by default values.

$\text{default } x^* \in \llbracket x^* \rrbracket$ default-VALID

Because $\text{default} :: T^* \rightarrow T$ strongly depends on type T , we cannot give a general definition of the function. However, it will generally have this pattern:

$$\begin{aligned} \text{default } *_{T} &= \text{default value for type } T \\ \text{default } (C \ x_0^* \dots x_n^*) &= C \ (\text{default } x_0^*) \dots (\text{default } x_n^*) \end{aligned}$$

The definition is just a sketch, because it suggests that the default for a value only depends on its type, whereas values of the same type that appear in different places may have different defaults. In the `default` for declarations, as defined above, a $*_{Exp}$ wildcard appearing in a `Decl` gets the default value `Undefined`, but in other places the default may be different.

Another difficulty in giving a general definition of `default` is that a default value may also depend on information elsewhere in the tree, for example when `default` whitespace is computed by a pretty-printing algorithm. In that case, the `default` function requires an extra argument containing the context of the subtree that is to be pretty-printed.

Using `default` we can give a first definition of \triangleright^* . It takes a new value containing wildcards, together with an old value, and returns the new value with extra state from the old value.

$$\begin{aligned} *_{T} \triangleright^* y &= y \\ C \ x_0^* \dots x_n^* \triangleright^* C \ y_0 \dots y_n &= C \ (x_0^* \triangleright^* y_0) \dots (x_n^* \triangleright^* y_n) \\ C \ x_0^* \dots x_n^* \triangleright^* C' \ y_0 \dots y_m &= \text{default } (C \ x_0^* \dots x_n^*) \end{aligned} \quad \triangleright^* \text{-DEF}$$

This is only a basic definition of \triangleright^* that recovers extra state only if the old and the new value are structurally similar. If the constructors for the two values are different, a default value is chosen for extra state. Although sufficient for some cases, this method of reusing, based on structure only, is too coarse in general.

Consider an element that is deleted from a list (e.g. $[e_0, e_1] \rightsquigarrow [e_1]$). A structure-based reuse will recover extra state from e_0 for the new value e_1 , which is not the desired behavior. Similarly, when swapping child nodes in a tree, the extra state will not swap. And finally, when a subtree is updated to a structurally similar subtree, such as $\text{Sum } e_1 e_2 \rightsquigarrow \text{Product } e_1 e_2$, we might want the extra state of the children to be reused, which is not possible with this basic \triangleright^* .

We can handle the problems mentioned above by extending the model with a notion of subtree identities, and defining a more powerful reuse function. Section 5.3.5 sketches how such a function may be defined.

5.3.4 Reuse on equivalence classes

The reuse function \triangleright^* defined in the previous section takes a wildcard argument of type T^* , whereas the reuse function required for *Comp* in Section 5.2.4 takes an equivalence class argument. Because a wildcard type definition induces an equivalence relation, we can use \triangleright^* for \triangleright . However, since not every equivalence relation has a wildcard representation, we thus put a restriction on the equivalence relations that we can define on the results of *present* and *interpret*, and hence on the kind of extra state we can model.

Recall the types of *present* and *interpret*:

$$\begin{aligned} \text{present} &:: \text{Level}_{H/H} \rightarrow \text{Level}_{L/L} \\ \text{interpret} &:: \text{Level}_{L/L} \rightarrow \text{Level}_{H/H} \end{aligned}$$

A wildcard type Level_H^* induces an equivalence relation \simeq and the corresponding *core* and \triangleright^* functions. We restrict ourselves by requiring that the extra state on Level_H has a wildcard representation. Thus, we require the existence of a Level_H^* , such that $H \equiv \simeq$. A similar restriction applies to the lower level.

To disambiguate instances of *core* for the different levels, we use subscripts H and L (rather than the verbose Level_H^* and Level_L^*). The equivalence classes on the higher and lower levels can be expressed as:

$$\begin{aligned} h \ H \ h' &\hat{=} \text{core}_H h = \text{core}_H h' \\ l \ L \ l' &\hat{=} \text{core}_L l = \text{core}_L l' \end{aligned} \qquad \text{RESTRICT}$$

Because the results in the remainder of this section apply to either level, we drop the subscripts to *core* and other level-specific functions. We use R to denote either relation H or L . Thus, from \simeq -CLASSES, we have:

$$[x]_R = \llbracket \text{core } x \rrbracket \qquad \text{RESTRICT-CLASSES}$$

We define \triangleright in terms of \triangleright^* :

$$\begin{aligned} \triangleright &:: T/R \rightarrow T \rightarrow T \\ [x]_R \triangleright y &\hat{=} \text{core } x \triangleright^* y \end{aligned} \qquad \triangleright\text{-DEF}$$

In order for this definition to be valid, we have to verify that it returns the same result for all elements in an equivalence class.

$$x R x' \Rightarrow [x]_R \triangleright y = [x']_R \triangleright y$$

Proof:

$$\begin{aligned} & [x]_R \triangleright y = [x']_R \triangleright y \\ \equiv & \quad \{ \triangleright\text{-Def} \} \\ & \text{core } x \triangleright^* y = \text{core } x' \triangleright^* y \\ \Leftarrow & \quad \{ \text{Leibniz} \} \\ & \text{core } x = \text{core } x' \\ \equiv & \quad \{ \text{RESTRICT} \} \\ & x R x' \end{aligned}$$

□

Before we prove that \triangleright meets the \triangleright -VALID, \triangleright -IDEM, and \triangleright -CLOSE requirements from Section 5.2.4, we introduce a more uniform wildcard type that will simplify the proofs.

The T^* type

The structure of a wildcard type definition T^* leads to awkward proofs, because each child of a constructor can be either a wildcard or a regular child. When pattern matching, this means that children cannot be handled uniformly. Therefore, we introduce a uniform construction assigning a data type T^* , which has an extra wildcard constructor, to every type T , including the primitive types. (Note the different star symbols: ‘ \star ’ for the uniform type versus ‘ $*$ ’ for the regular wildcard type.)

For a type T :

$$\mathbf{data} \ T = C_0 \ T_{0,0} \dots T_{0,m_0} \mid \dots \mid C_n \ T_{n,0} \dots T_{n,m_n}$$

we can construct a data type T^* by:

$$\mathbf{data} \ T^* = C_0 \ T_{0,0}^* \dots T_{0,m_0}^* \mid \dots \mid C_n \ T_{n,0}^* \dots T_{n,m_n}^* \mid *T$$

For the binary tree from Section 5.3.1, we get:

$$\mathbf{data} \ Tree^* = \text{Bin } Bool^* \ Tree^* \ Tree^* \mid \text{Leaf } Bool^* \ Int^* \mid *Tree$$

Note that the construction also applies to the primitive types $Bool$ and Int , and extends these types with a wildcard constructor. Hence, the possible values for the type $Bool^*$ are: $True$, $False$, and $*Bool$.

Unlike T^* types, a data type T has only one T^* type. Because T^* and T^* share the $*_T$ constructor, any T^* value is also a T^* value. For example, for the type $Tree^*$ from Section 5.3.1, we have $\text{Leaf True } *_Int :: Tree^*$ as well as $\text{Leaf True } *_Int :: Tree^*$. Because the constructors are shared, the functions \triangleright^* and $\llbracket _ \rrbracket$, which are defined by pattern matching on T^* values, are also functions on T^* .

Validity

The validity requirement follows from a more general statement on T^* :

$$\begin{aligned}
& \llbracket [x]_R \triangleright y \rrbracket_R = [x]_R \\
\equiv & \quad \{ [-]\text{-MEMBER} \} \\
& [x]_R \triangleright y \in [x]_R \\
\equiv & \quad \{ \triangleright\text{-DEF and RESTRICT-CLASSES} \} \\
& \text{core } x \triangleright^* y \in \llbracket \text{core } x \rrbracket \\
\Leftarrow & \quad \{ \text{taking core } x \text{ for } x^* \} \\
& x^* \triangleright^* y \in \llbracket x^* \rrbracket
\end{aligned}$$

We prove $x^* \triangleright^* y \in \llbracket x^* \rrbracket$ by structural induction on x^*

Proof: Case $x^* = *_T$:

$$\begin{aligned}
& *_T \triangleright^* y \in \llbracket *_T \rrbracket \\
\equiv & \quad \{ \triangleright^*\text{-DEF} \} \\
& y \in \llbracket *_T \rrbracket \\
\equiv & \quad \{ [-]\text{-DEF} \} \\
& y \in \{ t \mid t :: T \} \\
\Leftarrow & \quad \{ y :: T \} \\
& \text{true}
\end{aligned}$$

Case $x^* = \mathbf{C} x_0^* \dots x_n^*$ and $y = \mathbf{C} y_0 \dots y_n$:

The induction hypothesis is $x_i^* \triangleright^* y_i \in \llbracket x_i^* \rrbracket$

$$\begin{aligned}
& \mathbf{C} x_0^* \dots x_n^* \triangleright^* \mathbf{C} y_0 \dots y_n \in \llbracket \mathbf{C} x_0^* \dots x_n^* \rrbracket \\
\equiv & \quad \{ \triangleright^*\text{-DEF} \} \\
& \mathbf{C} (x_0^* \triangleright^* y_0) \dots (x_n^* \triangleright^* y_n) \in \llbracket \mathbf{C} x_0^* \dots x_n^* \rrbracket \\
\equiv & \quad \{ [-]\text{-DEF} \} \\
& \mathbf{C} (x_0^* \triangleright^* y_0) \dots (x_n^* \triangleright^* y_n) \in \{ \mathbf{C} x_0 \dots x_n \mid x_0 \in \llbracket x_0^* \rrbracket \wedge \dots \wedge x_n \in \llbracket x_n^* \rrbracket \} \\
\equiv & \quad \{ \text{property of set comprehension} \}
\end{aligned}$$

$$\begin{aligned}
& x_0^* \triangleright^* y_0 \in \llbracket x_0^* \rrbracket \wedge \cdots \wedge x_n^* \triangleright^* y_n \in \llbracket x_n^* \rrbracket \\
\equiv & \quad \{ \text{Induction Hypothesis} \} \\
& \text{true}
\end{aligned}$$

Case $x^* = C x_0^* \dots x_n^*$ and $y = C' y_0 \dots y_m$, with $C \neq C'$:

$$\begin{aligned}
& C x_0^* \dots x_n^* \triangleright^* C' y_0 \dots y_m \in \llbracket C x_0^* \dots x_n^* \rrbracket \\
\equiv & \quad \{ \triangleright^* \text{-DEF} \} \\
& \text{default}(C x_0^* \dots x_n^*) \in \llbracket C x_0^* \dots x_n^* \rrbracket \\
\equiv & \quad \{ \text{default-VALID} \} \\
& \text{true}
\end{aligned}$$

□

Idempotency

Similar to validity, we prove idempotency by proving a more general statement on T^* .

$$\begin{aligned}
& [y]_R = [x]_R \Rightarrow [x]_R \triangleright y = y \\
\equiv & \quad \{ [-]\text{-MEMBER} \} \\
& y \in [x]_R \Rightarrow [x]_R \triangleright y = y \\
\equiv & \quad \{ \text{RESTRICT-CLASSES and } \triangleright\text{-DEF} \} \\
& y \in \llbracket \text{core } x \rrbracket \Rightarrow \text{core } x \triangleright^* y = y \\
\Leftarrow & \quad \{ \text{taking core } x \text{ for } x^* \} \\
& y \in \llbracket x^* \rrbracket \Rightarrow x^* \triangleright^* y = y
\end{aligned}$$

The more general statement $y \in \llbracket x^* \rrbracket \Rightarrow x^* \triangleright^* y = y$ is proven by structural induction on x^* :

Proof: Case $x^* = *T$:

$$\begin{aligned}
& y \in \llbracket *T \rrbracket \Rightarrow *T \triangleright^* y = y \\
\Leftarrow & \quad \{ \text{propositional calculus} \} \\
& *T \triangleright^* y = y \\
\equiv & \quad \{ \triangleright^* \text{-DEF} \} \\
& \text{true}
\end{aligned}$$

Case $x^* = C x_0^* \dots x_n^*$ and $y = C y_0 \dots y_n$:

The induction hypothesis is $y^i \in \llbracket x_i^* \rrbracket \Rightarrow x_i^* \triangleright^* y_i = y_i$

$$\begin{aligned}
& x^* \triangleright^* y = y \\
\equiv & \quad \{ \text{definitions of } x^* \text{ and } y \} \\
& \mathbf{C} x_0^* \dots x_n^* \triangleright^* \mathbf{C} y_0 \dots y_n = \mathbf{C} y_0 \dots y_n \\
\equiv & \quad \{ \triangleright^* \text{-DEF} \} \\
& \mathbf{C} (x_0^* \triangleright^* y_0) \dots (x_n^* \triangleright^* y_n) = \mathbf{C} y_0 \dots y_n \\
\Leftarrow & \quad \{ n \text{ times Leibniz} \} \\
& x_0^* \triangleright^* y_0 = y_0 \wedge \dots \wedge x_n^* \triangleright^* y_n = y_n \\
\Leftarrow & \quad \{ \text{Induction Hypothesis} \} \\
& y_0 \in \llbracket x_0^* \rrbracket \wedge \dots \wedge y_n \in \llbracket x_n^* \rrbracket \\
\Leftarrow & \quad \{ \text{property of set comprehension} \} \\
& \mathbf{C} y_0 \dots y_n \in \{ \mathbf{C} x_0 \dots x_n \mid x_0 \in \llbracket x_0^* \rrbracket \wedge \dots \wedge x_n \in \llbracket x_n^* \rrbracket \} \\
\equiv & \quad \{ \llbracket - \rrbracket \text{-DEF} \} \\
& \mathbf{C} y_0 \dots y_n \in \llbracket \mathbf{C} x_0^* \dots x_n^* \rrbracket \\
\equiv & \quad \{ \text{definitions of } x^* \text{ and } y \} \\
& y \in \llbracket x^* \rrbracket
\end{aligned}$$

Case $x^* = \mathbf{C} x_0^* \dots x_n^*$ and $y = \mathbf{C}' y_0 \dots y_m$, with $\mathbf{C} \neq \mathbf{C}'$:

This case does not occur, because we have $\mathbf{C} = \mathbf{C}'$ from the assumption $y \in \llbracket x^* \rrbracket$:

$$\begin{aligned}
& \mathbf{C}' = \mathbf{C} \\
\equiv & \quad \{ \text{property of set comprehension} \} \\
& \mathbf{C}' y_0 \dots y_m \in \{ \mathbf{C} x_0 \dots x_n \mid x_0 \in \llbracket x_0^* \rrbracket \wedge \dots \wedge x_n \in \llbracket x_n^* \rrbracket \} \\
\equiv & \quad \{ \llbracket - \rrbracket \text{-DEF} \} \\
& \mathbf{C}' y_0 \dots y_m \in \llbracket \mathbf{C} x_0^* \dots x_n^* \rrbracket \\
\equiv & \quad \{ \text{definitions of } x^* \text{ and } y \} \\
& y \in \llbracket x^* \rrbracket
\end{aligned}$$

□

Closeness

For the requirement \triangleright -CLOSE, we cannot give a formal proof because we have no formal description of closeness. In fact, as we have seen in Section 5.3.3, our first definition of \triangleright^* does not guarantee much closeness after a structural update. The next section sketches a \triangleright^* function that can handle structural updates.

5.3.5 Improving reuse

A reuse strategy based only on the structure of the old and new level values cannot recover extra state when the level is structurally changed. In this section we sketch a more advanced reuse strategy that is based on identities of parent nodes of extra state.

If tree nodes have identities that are preserved by edit operations, an extra state child of a node can be recovered by looking up its value in the old tree. A definition of the new reuse function is:

$$\begin{aligned} \triangleright^* &:: T^* \rightarrow Level \rightarrow T \\ C_{id} x_0^* \dots x_n^* \triangleright^* level &= C_{id} x'_0 \dots x'_n \\ \text{where } x'_i &= \begin{cases} x_i^* \triangleright^* level & \text{if } x_i^* \neq * \\ y_i & \text{if lookup } id \text{ level} = C y_0 \dots y_n \\ y_j & \text{if lookup } id \text{ level} = C' y_0 \dots y_n \wedge C \neq C' \\ & \exists j : y_j \text{ "is similar to" } x_i \\ \text{default } (C x_0^* \dots x_n^*) & \text{otherwise} \end{cases} \end{aligned}$$

In contrast to \triangleright^* from Section 5.3.3, this version of \triangleright^* only recurses on its first argument. The second argument (*level*) is the previous value of the root of the level, from which a previous value of a node can be looked up based on its identity. In the definition, there are four cases for x'_i . The first case is when x_i is not an extra state value, and its result is the recursive application of \triangleright^* .

If x_i is extra state, then `lookup id y` is used to obtain the previous value of its parent. If the lookup succeeds and the previous value has the same constructor (*C*), then all extra state children can be copied from the previous value.

If the lookup returns a parent with a different constructor *C'*, it may still be possible to recover its previous value, provided that $C' y_0 \dots y_n$ has a child that represents the same information as x_i . This is expressed by the phrase “is similar to” in the third case of the definition. An example illustrates this case.

Consider an expression editor with an edit operation that transforms a sum into a product. Both sums and products have an extra state child that represents the whitespace in the presentation. For $(\text{Product}_{id_0} * e_1 e_2) \triangleright^* y$, we have `lookup id_0 y = Sum_{id_0} extra_0 e_1 e_2`, in which *extra*₀ represents the whitespace. It makes sense to reuse the whitespace from the old sum, because of the similarity between the presentation of the sum and the product. Hence, the third case applies, and we have $j = 0$, which yields the final result $\text{Product}_{id_0} \text{extra}_0 e_1 e_2$.

The last case of the definition applies if the extra state from the old value cannot be reused or if lookup fails. In that case, extra state is set to a default value.

Note that since the recursion is only on the first argument, the type is slightly different from the reuse function from Section 5.3.3, which is $\triangleright^* :: T^* \rightarrow T \rightarrow T$. However, for the application to the root of the level, *T* is equal to *Level*, yielding the type $\triangleright^* :: Level^* \rightarrow Level \rightarrow Level$.

The \triangleright^* from this section is able to reuse extra state after a structural update, such as moving a subtree or deleting from a list. Moreover, it also allows reuse when a node is updated to a similar node that has the same kind of extra state. However, it does require that editing preserves node identities.

5.4 A composite layer

In order to describe the architecture of Proxima, we refine the single-layer model from the previous sections to accommodate multiple layers. To make it explicit that present and interpret are composite functions in this section, we denote the functions by present_C and interpret_C .

We assume that present_C is given and that it can be split into components present_H and present_L . For these components, we recursively obtain specifications for interpret_H and interpret_L . The specification for interpret_C is a composition of interpret_H and interpret_L .

Instead of splitting a layer into n components, we split it in two: an upper and a lower layer with a middle level in between. The upper layer may itself be a composite layer, whereas the lower layer is an atomic layer, such as specified in the previous section. Section 5.4.3 explains the reason for this distinction.

It turns out that if we split the presentation mapping and construct interpret_H and interpret_L according to the specification, this does not always result in a valid specification for interpret_C . Hence, we provide extra requirements on the components to ensure that the specification is valid.

5.4.1 Composite present_C and interpret_C

Before we show the compositions for present_C and interpret_C , we first take a look at what the compositions look like if we view the mappings as relations Present_C and Interpret_C rather than as functions between equivalence classes. The relation view is somewhat simpler because it does not explicitly refer to the various extra state equivalence classes in the composition.

Composite Present_C and Interpret_C

For the specification of a composite interpretation mapping, we split (or decompose) the presentation mapping, whereas we compose the interpretation mapping. However, from a formal point of view there is no distinction between splitting and composing, since in both cases the composite relation is the relational composition of two other relations. Hence, we ignore this distinction for Present_C and Interpret_C .

The components of $Present_C$ and $Interpret_C$ are

$$\begin{aligned} Present_L &:: Level_L \sim Level_M & Present_H &:: Level_M \sim Level_H \\ Interpret_H &:: Level_H \sim Level_M & Interpret_L &:: Level_M \sim Level_L \end{aligned}$$

Instead of using the notation $Present_L \circ Present_H$ and $Interpret_L \circ Interpret_H$, we explicitly write out the relational compositions, yielding:

$$\begin{aligned} Present_C &:: Level_L \sim Level_H \\ l Present_C h &\equiv \exists m : l Present_L m \wedge m Present_H h \end{aligned}$$

and

$$\begin{aligned} Interpret_C &:: Level_H \sim Level_L \\ h Interpret_C l &\equiv \exists m : h Interpret_H m \wedge m Interpret_L l \end{aligned}$$

Composite $present_C$ and $interpret_C$

We can split $present_C$ and compose $interpret_C$, similar to $Present_C$ and $Interpret_C$ with the difference that all relations are now represented by functions between equivalence classes. However, even if two relations are both functions between equivalence classes, this does not guarantee that their composition is also function between equivalence classes. Hence, the composite definitions will need additional restrictions to be valid. We will provide these restrictions in the form of two extra requirements for the components of $present_C$ and $interpret_C$.

We assume the existence of $present_C :: Level_H/_{CH} \rightarrow Level_L/_{CL}$, which, for a certain type $Level_M$ (and certain equivalence classes HH , HL , LH , and LL), can be decomposed into $present_H :: Level_H/_{HH} \rightarrow Level_M/_{HL}$ and $present_L :: Level_M/_{LH} \rightarrow Level_L/_{LL}$.

In the types of the $present_C$, $present_H$ and $present_L$, there are six different equivalence relations: both the upper and lower layer have equivalence relations on their respective upper and lower levels (HH , HL , LH , and LL), and from $present_C$ we have CH and CL . Although the type of the middle level ($Level_M$) is equal for both layers, the equivalence relations are likely to be different ($HL \neq LH$) because the presentation extra state of the upper layer is most likely different from the interpretation extra state of the lower layer. Similarly, on $Level_H$ and $Level_L$, the equivalence relations for $present_C$ (CH and CL) do not have to be equal to the relations of the two component layers (HH and LL). Thus, in total, we have the six equivalence relations that are shown in Figure 5.2.

We cannot use function composition to compose $present_H$ and $present_L$, because the equivalence classes on the middle do not match ($Level_M/_{HL}$ versus $Level_M/_{LH}$). Hence, we write the composition as a relational composition, similar to $Present_C$. (Note the reversed order of the types, as was explained in Section 4.1.)

$$\begin{aligned} present_C &:: Level_H/_{CH} \rightarrow Level_L/_{CL} \\ [l]_{CL} = present_C [h]_{CH} &\equiv \exists m : [l]_{LL} = present_L [m]_{LH} \\ &\quad \wedge [m]_{HL} = present_H [h]_{HH} \end{aligned} \quad \text{present-COMPOSE}$$

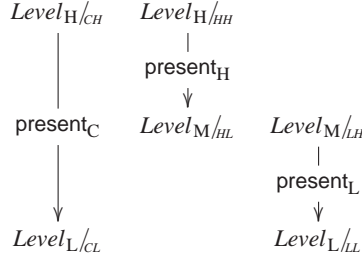


Figure 5.2: Equivalence relations in a composite layer.

For the components present_L and present_H , we have a specification of $\text{interpret}_L :: Level_{L/LL} \rightarrow Level_{M/LH}$ and $\text{interpret}_H :: Level_{M/HL} \rightarrow Level_{H/HH}$, which we compose to define interpret_C :

$$\begin{array}{l}
 \text{interpret}_C :: Level_{L/CL} \rightarrow Level_{H/CH} \\
 [h]_{CH} = \text{interpret}_C [l]_{CL} \equiv \exists m : [h]_{HH} = \text{interpret}_H [m]_{HL} \\
 \quad \quad \quad \wedge [m]_{LH} = \text{interpret}_L [l]_{LL} \quad \quad \quad \text{interpret-COMPOSE}
 \end{array}$$

Although we cannot use function composition, we can use Kleisli composition on set-valued functions to give an equivalent point-free definition of the compositions:

$$\begin{array}{l}
 \text{present}_C \circ [-]_{CH} = (\text{present}_L \circ [-]_{LH}) \diamond (\text{present}_H \circ [-]_{HH}) \\
 \text{interpret}_C \circ [-]_{CL} = (\text{interpret}_H \circ [-]_{HL}) \diamond (\text{interpret}_L \circ [-]_{LL})
 \end{array}$$

Before we discuss the restrictions we need for both compositions to be valid, we give a simple example, which shows that equivalence relations on the same level can be different. Consider a function present_C with its type written in the wildcard style from Section 5.3.

$$\begin{array}{l}
 \text{present}_C :: (Int, *Int) \rightarrow Int \\
 \text{present}_C(x, y) = x
 \end{array}$$

Because $\text{present}_C = \text{present}_C \circ \text{id}$, we can split present_C as follows:

$$\begin{array}{ll}
 \text{present}_L :: (Int, *Int) \rightarrow Int & \text{present}_H :: (Int, Int) \rightarrow (Int, Int) \\
 \text{present}_L(x, y) = x & \text{present}_H(x, y) = (x, y)
 \end{array}$$

The equivalence relation LH for the argument of present_L is (the relation induced by) $(Int, *Int)$, whereas HL is (Int, Int) . Furthermore, CH is $(Int, *Int)$, and HH is (Int, Int) . Hence, $LH \neq HL$ and $CH \neq HH$. A similar example can be constructed to show that CL may be different from LL .

Restrictions on the components

The assumed condition `present-COMPOSE` puts a restriction on the components `presentH` and `presentL`. We can only split the presentation mapping into components for which `present-COMPOSE` hold. However, it turns out that even if `present-COMPOSE` holds for `presentH` and `presentL`, and `interpretH` and `interpretL` are constructed according to the specification, this does not guarantee that `interpret-COMPOSE` is valid for the composition. As a result, without any additional restrictions, the specification of `interpretC` may not always be valid.

Moreover, even if the definition `interpret-COMPOSE` is valid, which implies that the composition of `interpretH` and `interpretL` can be viewed as a function between equivalence classes (i.e. `interpretC :: LevelL/CL → LevelH/CH`), this does not necessarily imply that these classes are equal to the `CH` and `CL` classes in `presentC :: LevelH/CH → LevelL/CL`. In case the classes are not equal, this also results in an invalid specification.

We provide an example of the latter case (when the equivalence classes for `interpretC` are different from `presentC`). Consider `presentC = idInt`, which is split into the `presentH` and `presentL` below. The `interpretH` and `interpretL` provided meet the requirements of the previous section.

$$\begin{array}{ll}
 \text{present}_L :: (*Int, Int) \rightarrow Int & \text{present}_H :: Int \rightarrow (Int, Int) \\
 \text{present}_L (*Int, y) = y & \text{present}_H x = (x, x) \\
 \text{interpret}_L :: Int \rightarrow (*Int, Int) & \text{interpret}_H :: (Int, Int) \rightarrow Int \\
 \text{interpret}_L y = (*Int, y) & \text{interpret}_H (x, y) = x
 \end{array}$$

By `interpret-COMPOSE`, we have `interpretC [0]CL = [0]CH`, if we take $(0, 0)$ for the middle level m . Similarly, for $m = (1, 0)$, we get `interpretC [0]CL = [1]CH`, and hence, `[0]CH = [1]CH`. However, because `presentC` is the identity function, `CH` is the equality relation, which implies `[0]CH ≠ [1]CH`. Thus, even though `interpretC` is a mapping between equivalence classes, it is not a mapping between the equivalence classes of `presentC`.

Without showing the details of their derivation, we introduce two explicit conditions, which guarantee that `present-COMPOSE` and `interpret-COMPOSE` hold, as well as that the `CL` and `CH` classes of both definitions are equal. First, we define an auxiliary function \mathcal{I} . The application $\mathcal{I} m$ denotes the set of all higher-level elements that are in the interpretation of some member of the `LH` equivalence class of m .

$$\begin{array}{l}
 \mathcal{I} :: Level_M \rightarrow \wp Level_H \\
 \mathcal{I} m = \{h \mid \exists m' \in [m]_{LH} : [h]_{HH} = \text{interpret}_H [m']_{HL}\} \quad \mathcal{I}\text{-DEF}
 \end{array}$$

The two conditions are:

$$\begin{array}{l}
 h, h' \in \mathcal{I} m \wedge [m']_{HL} = \text{present}_H [h]_{HH} \Rightarrow \\
 \exists m'' \in [m']_{LH} : [m'']_{HL} = \text{present}_H [h']_{HH} \quad \text{present-MATCH}
 \end{array}$$

and

$$\begin{aligned} h, h' \in \mathcal{I} m \wedge [h]_{HH} = \text{interpret}_H [m']_{HL} &\Rightarrow \\ \exists m'' \in [m']_{LH} : [h']_{HH} = \text{interpret}_H [m'']_{HL} &\quad \text{interpret-MATCH} \end{aligned}$$

The `interpret-MATCH` condition guarantees that we can represent the composition of `interpretH` and `interpretL` by a mapping between equivalence classes, which implies validity of `interpret-COMPOSE`. On the other hand, `present-MATCH` implies validity of `present-COMPOSE` and guarantees that the *CH* and *CL* classes of `present-COMPOSE` are equal to *CH* and *CL* of `interpret-COMPOSE`. In the remainder of this section we assume `present-MATCH` and `interpret-MATCH`.

5.4.2 The INTERPRESENT requirement

The `INTERPRESENT` requirement holds for a composition if it holds for the upper and lower layer.

$$[l]_{CL} = \text{present}_C [h]_{CH} \Rightarrow [h]_{CH} = \text{interpret}_C [l]_{CL} \quad \text{INTERPRESENT}$$

The proof is simple:

$$\begin{aligned} [h]_{CH} &= \text{interpret}_C [l]_{CL} \\ \equiv &\quad \{ \text{interpret-COMPOSE} \} \\ \exists m : [m]_{LH} = \text{interpret}_L [l]_{LL} \wedge [h]_{HH} = \text{interpret}_H [m]_{HL} \\ \Leftarrow &\quad \{ \text{INTERPRESENT for higher and lower layers} \} \\ \exists m : [l]_{LL} = \text{present}_L [m]_{LH} \wedge [m]_{HL} = \text{present}_H [h]_{HH} \\ \equiv &\quad \{ \text{present-COMPOSE} \} \\ [l]_{CL} &= \text{present}_C [h]_{CH} \end{aligned}$$

5.4.3 An inductive definition of present_C and interpret_C

In this section, we give inductive versions of `present-COMPOSE` and `interpret-COMPOSE` for an editor consisting of n layers. The building blocks for the composite layer are $n + 1$ data levels and n pairs of presentation and interpretation functions with their corresponding equivalence relations:

$$\begin{aligned} \text{present}_i &:: \text{Level}_{i-1}/_{H_i} \rightarrow \text{Level}_i/_i \\ \text{interpret}_i &:: \text{Level}_i/_i \rightarrow \text{Level}_{i-1}/_{H_i} \end{aligned}$$

Because Level_0 is the document and Level_n the presentation, the top layer consists of `present1` and `interpret1`, whereas the lowest layer consists of `presentn` and `interpretn`. A composite layer consists of two composite functions: `presentC,i` and `interpretC,i`.

Layers are composed in a “top-associative” way: the higher component layer may itself be a composite layer, whereas the lower component is an atomic layer. The reason for this

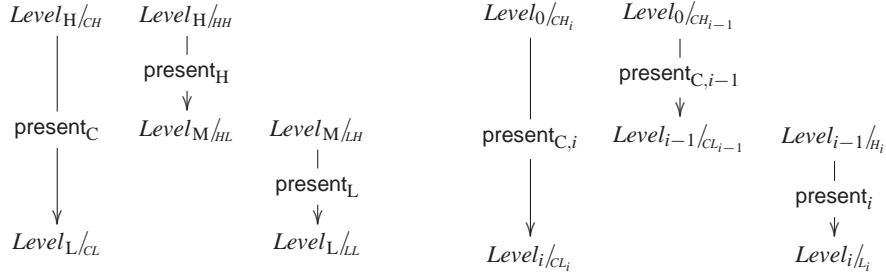


Figure 5.3: Indices in a composite layer.

choice stems from the order in which the updated lower level is computed. If we abuse the notation for function composition to denote the composition of layer functions, we can sketch the computation:

$$(\text{present}_n \circ \text{present}_{n-1} \circ \dots \circ \text{present}_1 \circ \text{interpret}_1 \circ \dots \circ \text{interpret}_{n-1} \circ \text{interpret}_n)$$

The most logical way to decompose this computation is by splitting it into present_n , interpret_n , and $(\text{present}_{n-1} \circ \dots \circ \text{interpret}_{n-1})$. This corresponds to splitting off an atomic layer at the bottom of a composite layer.

Figure 5.3 shows an overview of the levels, present functions, and equivalence relations that appear in a composite layer. The left-hand side shows the view from Figure 5.2 with relative subscripts H, M, and L, whereas the right-hand side shows the inductively defined versions with numbered subscripts. The figure does not show interpret because it is similar to present .

In a composite layer, there are six equivalence relations. The two equivalence relations that are induced by the composite layer are CH_i and CL_i . The upper layer is the composition of the first $i - 1$ layers, and hence its equivalence relations are CH_{i-1} and CL_{i-1} . For the lower layer, the equivalence relations are the H_i and L_i relations that are associated with present_i and interpret_i .

The inductive definition of $\text{present}_{C,i}$ is right-associative, whereas $\text{interpret}_{C,i}$ is left-associative. The reason for this difference is that in the computation, the order of the present_i functions is the reverse of the order of the interpret_i functions. The basis for both definitions is id . In the definition, we use the point-free style for the composite functions:

$$\begin{aligned} \text{present}_{C,i} &:: \text{Level}_0/CH_i \rightarrow \text{Level}_i/CL_i \\ \text{present}_{C,0} &= \text{id} \\ \text{present}_{C,i} \circ [-]_{CH_i} &= (\text{present}_i \circ [-]_{H_i}) \diamond (\text{present}_{C,i-1} \circ [-]_{CH_{i-1}}) \end{aligned}$$

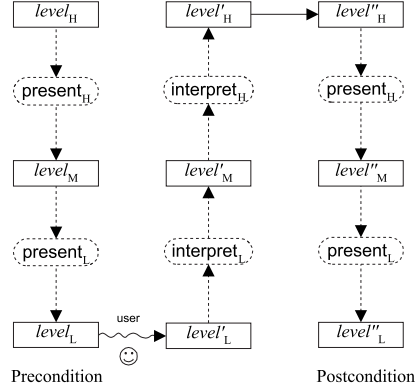


Figure 5.4: Single edit step in a composite layer.

$$\begin{aligned}
 \text{interpret}_{C,i} &:: Level_0/_{CH_i} \rightarrow Level_i/_{CL_i} \\
 \text{interpret}_{C,0} &= \text{id} \\
 \text{interpret}_{C,i} \circ [-]_{CL_i} &= (\text{interpret}_{C,i-1} \circ [-]_{CL_{i-1}}) \diamond (\text{interpret}_i \circ [-]_{L_i})
 \end{aligned}$$

The equivalence relations for the basis (CH_0 and CL_0) both are the equality relation ($=$). The reason for this is that the presentation function at the basis (present_0) is the identity, which leaves no room for extra state since each element in $Level_0$ is mapped onto itself. Hence, all equivalence classes are singleton sets, which correspond to the equivalence classes of the equality relation.

Because indices make proofs harder to read, and because at any time we only regard two layers and three levels, we use the notation from the left-hand side of Figure 5.3 in the rest of this section.

5.4.4 Editing

Besides $level_H$ and $level_L$, a composite layer also keeps track of the middle level, $level_M$. Before the edit operation we know that $[Level_L]_{CL} = \text{present}_C [Level_H]_{CH}$. By present-COMPOSE , this implies there exists a middle level m for which we have $[level_L]_{LL} = \text{present}_L [m]_{LH}$ and $[m]_{HL} = \text{present}_H [level_H]_{HH}$. Although not explicitly denoted, we assume that $level_M$ is this m .

Before the lower level is updated, we have:

$$\begin{aligned} [level_L]_{LL} &= \text{present}_L [level_M]_{LH} \wedge [level_M]_{HL} = \text{present}_H [level_H]_{HH} \\ level_L &\rightsquigarrow level'_L \end{aligned}$$

After the lower-level update, the lower layer computes an intermediate value $level'_M$, from which the higher layer (which may be a composite layer itself) computes $level''_H$. At the top, $level'_H$ is assigned to $level''_H$, which is subsequently presented onto $level''_M$. Finally, $level''_M$ presented onto $level''_L$.

Figure 5.4 sketches the updates to the various data levels. Note that the figure is just a sketch, since *present* and *interpret* are functions between equivalence classes instead of values (as the figure might suggest).

Requirements

Apart from a few changed subscripts, the requirements for $level''_H$ and $level''_L$ are equal to the requirements for the single-layered editor in Section 5.2.4. The only difference is that we explicitly added the precondition of the edit step ($[level_L]_{CL} = \text{present}_C [level_H]_{CH}$) to the precondition of DOC-INERT. We need it here, because DOC-INERT can only hold if the middle level is a valid presentation of $level_H$.

$$\begin{aligned} \{\text{true}\} \text{Comp} \{ [level''_L]_{CL} = \text{present}_C [level''_H]_{CH} \} & \quad \text{POSTCONDITION} \\ \left\{ \begin{array}{l} [level_L]_{CL} = \text{present}_C [level_H]_{CH} \wedge \\ [level'_L]_{CL} = \text{present}_C [level_H]_{CH} \end{array} \right\} \text{Comp} \{ level_H = level''_H \} & \quad \text{DOC-INERT} \\ \{ [level'_L]_{CL} = \text{present}_C [h]_{CH} \} \text{Comp} \{ level'_L = level''_L \} & \quad \text{PRES-INERT} \\ \{\text{true}\} \text{Comp} \{ level_H \text{ "close to" } level''_H \} & \quad \text{DOC-PRESERVE} \\ \{\text{true}\} \text{Comp} \{ level'_L \text{ "close to" } level''_L \} & \quad \text{INTENDED} \end{aligned}$$

Definition of *Comp*

The inductive definition of *Comp* reads:

$$\begin{aligned} \text{Comp}_0 &\hat{=} level''_H := level'_H \\ \text{Comp}_n &\hat{=} \text{Up}; \text{Comp}_{n-1}; \text{Dwn} \end{aligned} \quad \text{Comp-DEF}$$

$$\text{Up} \hat{=} level'_M := \text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M \quad \text{Up-DEF}$$

$$\text{Dwn} \hat{=} level''_L := \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L \quad \text{Dwn-DEF}$$

We prove that *Comp* meets the first three requirements.

5.4.5 POSTCONDITION requirement

In wp notation, the POSTCONDITION is:

$$\text{true} \Rightarrow \text{wp}(\text{Comp}_n, [\text{level}''_{\text{L}}]_{\text{CL}} = \text{present}_{\text{C}} [\text{level}''_{\text{H}}]_{\text{CH}}) \quad \text{POSTCONDITION}$$

Proof: We prove POSTCONDITION by induction over the number of layers n .

Case $n = 0$:

Filling in the base cases for $\text{present}_{\text{C}}$, the data levels, and relations CH and CL yields:

$$\text{true} \Rightarrow \text{wp}(\text{Comp}, [\text{level}''_0]_{=} = \text{id} [\text{level}''_0]_{=})$$

which holds because it is equivalent to $\text{true} \Rightarrow \text{wp}(\text{Comp}, \text{true})$.

Case $n > 0$:

The induction hypothesis is:

$$\text{true} \Rightarrow \text{wp}(\text{Comp}_{n-1}, [\text{level}''_{\text{M}}]_{\text{HL}} = \text{present}_{\text{H}} [\text{level}''_{\text{H}}]_{\text{HH}}) \quad \text{I.H.}$$

In the proof, we need the property that the lower presentation of a valid middle level is in the result of the composite presentation:

$$[m]_{\text{HL}} = \text{present}_{\text{H}} [h]_{\text{HH}} \Rightarrow [\text{present}_{\text{L}} [m]_{\text{LH}} \triangleright_{\text{LL}} l]_{\text{CL}} = \text{present}_{\text{C}} [h]_{\text{CH}}$$

It has a simple proof:

$$\begin{aligned} & [\text{present}_{\text{L}} [m]_{\text{LH}} \triangleright_{\text{LL}} l]_{\text{CL}} = \text{present}_{\text{C}} [h]_{\text{CH}} \\ \equiv & \quad \{ \text{present-COMPOSE} \} \\ & \exists m' : [\text{present}_{\text{L}} [m]_{\text{LH}} \triangleright_{\text{LL}} l]_{\text{LL}} = \text{present}_{\text{L}} [m']_{\text{LH}} \wedge [m']_{\text{HL}} = \text{present}_{\text{H}} [h]_{\text{HH}} \\ \Leftarrow & \quad \{ \text{let } m' = m \} \\ & [\text{present}_{\text{L}} [m]_{\text{LH}} \triangleright_{\text{LL}} l]_{\text{LL}} = \text{present}_{\text{L}} [m]_{\text{LH}} \wedge [m]_{\text{HL}} = \text{present}_{\text{H}} [h]_{\text{HH}} \\ \equiv & \quad \{ \triangleright\text{-VALID} \} \\ & \text{present}_{\text{L}} [m]_{\text{LH}} = \text{present}_{\text{L}} [m]_{\text{LH}} \wedge [m]_{\text{HL}} = \text{present}_{\text{H}} [h]_{\text{HH}} \\ \equiv & \quad \{ \text{reflexivity of } = \} \\ & [m]_{\text{HL}} = \text{present}_{\text{H}} [h]_{\text{HH}} \end{aligned}$$

Using this result, we prove POSTCONDITION.

$$\begin{aligned} & \text{wp}(\text{Comp}, [\text{level}''_{\text{L}}]_{\text{CL}} = \text{present}_{\text{C}} [\text{level}''_{\text{H}}]_{\text{CH}}) \\ \equiv & \quad \{ \text{Comp-DEF} \} \\ & \text{wp}(\text{Up}; \text{Comp}_{n-1}; \text{Dwn}, [\text{level}''_{\text{L}}]_{\text{CL}} = \text{present}_{\text{C}} [\text{level}''_{\text{H}}]_{\text{CH}}) \end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{wp-}; \} \\
&\text{wp}(Up; \text{Comp}_{n-1}, \text{wp}(\text{Down}, [level''_L]_{CL} = \text{present}_C [level''_H]_{CH})) \\
&\equiv \{ \text{Down-DEF} \} \\
&\text{wp}(Up; \text{Comp}_{n-1}, \text{wp}(level'_L := \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L \\
&\quad, [level''_L]_{CL} = \text{present}_C [level''_H]_{CH})) \\
&\equiv \{ \text{wp-} := \} \\
&\text{wp}(Up; \text{Comp}_{n-1}, [\text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L]_{CL} = \text{present}_C [level''_H]_{CH}) \\
&\Leftarrow \{ \text{wp-MONO and previous result} \} \\
&\text{wp}(Up; \text{Comp}_{n-1}, [level''_M]_{HL} = \text{present}_H [level''_H]_{HH}) \\
&\equiv \{ \text{wp-}; \} \\
&\text{wp}(Up, \text{wp}(\text{Comp}_{n-1}, [level''_M]_{HL} = \text{present}_H [level''_H]_{HH})) \\
&\Leftarrow \{ \text{wp-MONO and I.H.} \} \\
&\text{wp}(Up, \text{true}) \\
&\equiv \{ \text{Up-DEF} \} \\
&\text{wp}(level'_M := \text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M, \text{true}) \\
&\equiv \{ \text{wp-} := \} \\
&\text{true} \\
&\square
\end{aligned}$$

5.4.6 DOC-INERT requirement

$$\begin{aligned}
&[level_L]_{CL} = \text{present}_C [level_H]_{CH} \wedge [level'_L]_{CL} = \text{present}_C [level_H]_{CH} \Rightarrow \\
&\text{wp}(\text{Comp}_n, level_H = level'_H) \quad \text{DOC-INERT}
\end{aligned}$$

Besides the present-Match and interpret-Match conditions from Section 5.4.1, we need an additional condition for DOC-INERT to hold. From INTERPRESENT, we know that if $level'_L$ is a presentation of $level_H$, then $level_H$ is in the interpretation of $level'_L$. Thus, there exists a middle level m in the interpretation of $level'_L$, which has $level_H$ in its interpretation. However, the mere existence of such a middle level does not guarantee that this is the middle level resulting from *Comp*.

The problem lies in the fact that $level'_M$ is selected by \triangleright_{LH} from an *LH* class of elements, instead of an *HL* class. Thus, reusing extra state from $level_M$ could cause $level'_M$ to end up in an *HL* class that does not contain $level_M$, which would break DOC-INERT. We avoid the problem by requiring that the result of \triangleright_{LH} is in the correct *HL* class, if possible:

$$[m']_{HL} = [m]_{HL} \Rightarrow [[m']_{LH} \triangleright_{LH} m]_{HL} = [m]_{HL} \quad \text{ORTHOGONAL}$$

If both HL and LH are described by the wildcard types from Section 5.3, ORTHOGONAL holds.

Proof:

The proof of DOC-INERT is by induction over n .

Case $n = 0$:

$$\begin{aligned}
& [level_0]_{=} = \text{id } [level_0]_{=} \wedge [level'_0]_{=} = \text{id } [level_0]_{=} \Rightarrow \text{wp}(Comp_0, level_0 = level'_0) \\
& \text{wp}(Comp_0, level_0 = level'_0) \\
\equiv & \quad \{ \text{Comp-DEF and Level}_H \equiv Level_0 \} \\
& \text{wp}(level'_0 := level'_0, level_0 = level'_0) \\
\equiv & \quad \{ \text{wp-:=} \} \\
& level_0 = level'_0 \\
\Leftarrow & \quad \{ \text{property of equivalence class and symmetry of } = \} \\
& level'_0 \in [level_0]_{=} \\
\Leftarrow & \quad \{ [-]\text{-MEMBER} \} \\
& [level_0]_{=} = [level_0]_{=} \wedge [level'_0]_{=} = [level_0]_{=} \\
\equiv & \quad \{ \text{definition of id} \} \\
& [level_0]_{=} = \text{id } [level_0]_{=} \wedge [level'_0]_{=} = \text{id } [level_0]_{=}
\end{aligned}$$

Case $n > 0$:

For the inductive case, we assume the antecedent of the implication. The first conjunct ($[level_L]_{CL} = \text{present}_C [level_H]_{CH}$) has already been rewritten according to present-COMPOSE and the implicit assumption that the middle level between $level_H$ and $level_L$ is $level_M$:

$$\begin{aligned}
& [level_L]_{LL} = \text{present}_L [level_M]_{LH} \wedge [level_M]_{HL} = \text{present}_H [level_H]_{HH} \wedge \\
& [level'_L]_{L} = \text{present}_C [level_H]_{H}
\end{aligned}$$

In the proof, we need the existence of an m_0 that is in the equivalence class of the interpretation of $level'_L$ and at the same time in the equivalence class of the presentation of $level_H$:

$$\begin{aligned}
& [m_0]_{LH} = \text{interpret}_L [level'_L]_{LL} \wedge [m_0]_{HL} = \text{present}_H [level_H]_{HH} \\
\Leftarrow & \quad \{ \text{let } m_0 \text{ be the } m \text{ from the quantification} \} \\
& \exists m : [m]_{LH} = \text{interpret}_L [level'_L]_{LL} \wedge [m]_{HL} = \text{present}_H [level_H]_{HH} \\
\equiv & \quad \{ \text{INTERPRESENT} \} \\
& \exists m : [level'_L]_{LL} = \text{present}_L [m]_{LH} \wedge [m]_{HL} = \text{present}_H [level_H]_{HH}
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{present-COMPOSE} \} \\
&\quad [level'_L]_{CL} = \text{present}_C [level_H]_{CH} \\
&\equiv \{ \text{assumption} \} \\
&\quad \text{true}
\end{aligned}$$

From the precondition, we know $\text{present}_H [level_H]_{HH} = [level_M]_{HL}$, which, together with $[m_0]_{HL} = \text{present}_H [level_H]_{HH}$, implies that m_0 and $level_M$ are in the same *HL* class: $[m_0]_{HL} = [level_M]_{HL}$. This result is needed to apply *ORTHOGONAL*.

For the inductive step, we have the following induction hypothesis:

$$\begin{aligned}
&[level_M]_{HL} = \text{present}_H [level_H]_{HH} \wedge [level'_M]_{HL} = \text{present}_H [level_H]_{HH} \Rightarrow \\
&\text{wp}(\text{Comp}_{n-1}, level_H = level''_H) \quad \text{I.H.}
\end{aligned}$$

Note that the first conjunct in the antecedent of the induction hypothesis is already satisfied due to the assumption above.

Because we assumed the antecedent of *DOC-INERT* (in weakest precondition notation), we complete the proof of the implication by proving its conclusion.

$$\begin{aligned}
&\text{wp}(\text{Comp}_n, level_H = level''_H) \\
&\equiv \{ \text{Comp-DEF} \} \\
&\quad \text{wp}(\text{Up}; \text{Comp}_{n-1}; \text{Dwn}, level_H = level''_H) \\
&\equiv \{ \text{wp-}; \} \\
&\quad \text{wp}(\text{Up}; \text{Comp}_{n-1}, \text{wp}(\text{Dwn}, level_H = level''_H)) \\
&\equiv \{ \text{Dwn-DEF} \} \\
&\quad \text{wp}(\text{Up}; \text{Comp}_{n-1} \\
&\quad \quad , \text{wp}([level'_L] := \text{present}_L [level'_M]_{LH} \triangleright_{LL} level'_L, level_H = level''_H)) \\
&\equiv \{ \text{wp-} := \} \\
&\quad \text{wp}(\text{Up}; \text{Comp}_{n-1}, level_H = level''_H) \\
&\equiv \{ \text{wp-}; \} \\
&\quad \text{wp}(\text{Up}, \text{wp}(\text{Comp}_{n-1}, level_H = level''_H)) \\
&\Leftarrow \{ \text{wp-MONO, I.H., and } [level_M]_{HL} = \text{present}_H [level_H]_{HH} \} \\
&\quad \text{wp}(\text{Up}, [level'_M]_{HL} = \text{present}_H [level_H]_{HH}) \\
&\equiv \{ \text{Up-DEF} \} \\
&\quad \text{wp}(level'_M := \text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M \\
&\quad \quad , [level'_M]_{HL} = \text{present}_H [level_H]_{HH}) \\
&\equiv \{ \text{wp-} := \} \\
&\quad [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{HL} = \text{present}_H [level_H]_{HH}
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ [m_0]_{LH} = \text{interpret}_L [level'_L]_{LL} \} \\
&[[m_0]_{LH} \triangleright_{LH} level_M]_{HL} = \text{present}_H [level_H]_{HH} \\
&\equiv \{ \text{ORTHOGONAL and } [m_0]_{HL} = [level_M]_{HL} \} \\
&[m_0]_{HL} = \text{present}_H [level_H]_{HH} \\
&\equiv \{ [m_0]_{HL} = \text{present}_H [level_H]_{HH} \} \\
&\text{present}_H [level_H]_{HH} = \text{present}_H [level_H]_{HH} \\
&\equiv \{ \text{symmetry of } = \} \\
&\text{true}
\end{aligned}$$

□

5.4.7 PRES-INERT requirement

$$[level'_L]_{CL} = \text{present}_C [h]_{CH} \Rightarrow \text{wp}(Comp_n, level'_L = level''_L) \quad \text{PRES-INERT}$$

Similar to DOC-INERT, we cannot guarantee PRES-INERT without assuming extra conditions. It turns out that we can prove PRES-INERT if we assume present-MATCH together with a reversed ORTHOGONAL, in which the *HL* and *LH* relations are swapped. However, because ORTHOGONAL refers to \triangleright_{LH} , its reverse would have to refer to \triangleright_{HL} , which does not exist. The reason for this is the association order in *Comp*: reuse in the presentation direction of the higher layer is the result of the combined effect of the \triangleright_{LL} functions of the components of the higher layer. Thus, the only way to specify a reversed ORTHOGONAL is as a condition on the composite layer:

$$[level'_L]_{LH} = [m]_{LH} \Rightarrow \text{wp}(Comp_{n-1}, [level''_L]_{LH} = [m]_{LH}) \quad \text{ORTHOGONAL-R}$$

Because this condition is rather complex, we introduce a stronger condition ABSORPTION that is easier to verify. ABSORPTION implies present-MATCH and interpret-MATCH, as well as ORTHOGONAL and ORTHOGONAL-R.

$$[m]_{LH} = [m']_{LH} \Rightarrow [m]_{HL} = [m']_{HL} \quad \text{ABSORPTION}$$

The condition states that two $Level_M$ values that are in the same *LH* equivalence class of the lower layer, also share the *HL* equivalence class of the higher layer. Thus, interpretation extra state on the middle level is absorbed by presentation extra state on that level, hence the name.

As an example of ABSORPTION for the wildcard types of Section 5.3, consider the *LH* relation represented by $(Int, (Int, *Int))$. This relation is absorbed by an *HL* relation $(Int, (*Int, *Int))$, and also by $(Int, *Int)$.

For wildcard types, ABSORPTION holds if and only if for all possible values of type $Level_M$, any node that is a *** according to the wildcard definition of *LH*, is either also a *** according to *HL* or has an ancestor that is a ***.

More generally, ABSORPTION holds when each HL equivalence class is a union of LH classes. A result of ABSORPTION is that CH is equal to HH . For the proof of PRES-INERT, ABSORPTION implies that the intermediate level that results from interpreting a valid lower level, is itself also valid, which allows us to apply PRES-INERT inductively to the higher layer.

Because of ABSORPTION we have $CH = HH$. If we denote this with the explicit indices from Figure 5.3, we get $CH_i = CH_{i-1}$. Furthermore, since $CH_0 = (=)$, this means that $CH_i = (=)$, and, thus, no composite layer has interpretation extra state on $Level_0$. This is clearly not a desirable situation, and hence we do not assume ABSORPTION for the top-most combination. Nevertheless, the result is still quite restrictive, since now the interpretation extra state relation CH_i for each composite layer will be equal to the interpretation extra state relation H_1 for $present_1$. Future research should determine whether this is a problem. If it is, a less-strict condition needs to be established.

Because we do not assume ABSORPTION at the top-most composite layer, we have to give a separate proof for case $n = 1$ of PRES-INERT, in which we do not use the condition. Furthermore, in the proof of DOC-INERT, we cannot use ABSORPTION to imply present-MATCH, interpret-MATCH, and ORTHOGONAL for case $n = 1$. However, since for $n = 1$, these three conditions trivially hold, we do not need to assume ABSORPTION for this case.

Proof: In the inductive step of the proof, we need the fact that the lower level is not affected by the assignments of the higher layer. Therefore, we strengthen the pre- and postcondition of PRES-INERT:

$$\begin{aligned} [level'_L]_{CL} = present_C [h]_{CH} \wedge level'_{n+1} = x \Rightarrow \\ wp(Comp_n, level'_L = level''_L \wedge level'_{n+1} = x) \end{aligned}$$

Case $n = 0$: We omit the trivial proof of this case.

Case $n = 1$:

For $n = 1$, we have $level_H = level_M = level_0$ and $level_L = level_1$. Furthermore, since $present_H = present_{C,0} = id$ and HH and HL both are $(=)$, we have $present_C [h]_{CH} = present_L [h]_{LH}$. Thus, we rewrite the requirement to:

$$\begin{aligned} [level'_L]_{LH} = present_L [h]_{LH} \wedge level'_2 = x \Rightarrow \\ wp(Comp_1, level'_L = level''_L \wedge level'_2 = x) \end{aligned}$$

First, we prove the first conjunct of the postcondition, for which we only need to assume half of the antecedent $[level'_L]_{LL} = present_L [h]_{LH}$. The proof is similar to the proof of PRES-INERT for a single layer (see Section 5.2.4).

$$\begin{aligned} & wp(Comp_1, level'_L = level''_L) \\ \equiv & \quad \{ Comp-DEF \} \\ & wp(Up; Comp_0; Dwn, level'_L = level''_L) \\ \equiv & \quad \{ wp-; \} \end{aligned}$$

$$\begin{aligned}
& \text{wp}(Up; Comp_0, \text{wp}(Dwn, level'_L = level''_L)) \\
\equiv & \quad \{ Dwn-DEF \} \\
& \text{wp}(Up; Comp_0, \text{wp}(level'_L := \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L, level'_L = level''_L)) \\
\equiv & \quad \{ wp-:= \} \\
& \text{wp}(Up; Comp_0, level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L) \\
\equiv & \quad \{ wp-; \} \\
& \text{wp}(Up, \text{wp}(Comp_0, level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L)) \\
\equiv & \quad \{ Comp-DEF \text{ and } Level_M \equiv Level_0 \} \\
& \text{wp}(Up, \text{wp}(level''_M := level'_M, level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L)) \\
\equiv & \quad \{ wp-:= \} \\
& \text{wp}(Up, level'_L = \text{present}_L [level'_M]_{LH} \triangleright_{LL} level'_L) \\
\equiv & \quad \{ Up-DEF \} \\
& \text{wp}(level'_M := \text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M \\
& \quad , level'_L = \text{present}_L [level'_M]_{LH} \triangleright_{LL} level'_L) \\
\equiv & \quad \{ wp-:= \} \\
& level'_L = \text{present}_L [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{LH} \triangleright_{LL} level'_L \\
\equiv & \quad \{ \text{assumption} \} \\
& level'_L = \text{present}_L [\text{interpret}_L (\text{present}_L [h]_{LH}) \triangleright_{LH} level_M]_{LH} \triangleright_{LL} level'_L \\
\equiv & \quad \{ INTERPRESENT \} \\
& level'_L = \text{present}_L [[h]_{LH} \triangleright_{LH} level_M]_{LH} \triangleright_{LL} level'_L \\
\equiv & \quad \{ \triangleright-VALID \} \\
& level'_L = \text{present}_L [h]_{LH} \triangleright_{LL} level'_L \\
\equiv & \quad \{ \text{assumption} \} \\
& level'_L = [level'_L]_{LL} \triangleright_{LL} level'_L \\
\equiv & \quad \{ \triangleright-IDEM \} \\
& [level'_L]_{LL} = [level'_L]_{LL} \\
\equiv & \quad \{ \text{reflexivity of } = \} \\
& \text{true}
\end{aligned}$$

This completes the proof of the first conjunct of the postcondition. For the second conjunct, we need to prove:

$$\begin{aligned}
& [level'_L]_{LH} = \text{present}_L [h]_{LH} \wedge level'_2 = x \Rightarrow \\
& \text{wp}(Comp_1, level'_2 = x)
\end{aligned}$$

which trivially holds, because $Comp_1$ only performs assignments on levels 0 and 1. Hence, by wp-AND, we conclude:

$$\begin{aligned} [level'_L]_{LH} = \text{present}_L [h]_{LH} \wedge level'_2 = x &\Rightarrow \\ \text{wp}(Comp_1, level'_L = level''_L \wedge level'_2 = x) & \end{aligned}$$

Case $n > 1$:

Similar to the $n = 1$ case, prove the first part of postcondition conjunct:

$$[level'_L]_{CL} = \text{present}_C [h]_{CH} \wedge level'_{n+1} = x \Rightarrow \text{wp}(Comp_n, level'_L = level''_L)$$

We assume half of the antecedent: $[level'_L]_{CL} = \text{present}_C [h]_{CH}$.

From the assumption, we know there exists an m_0 that is in the presentation of h , and that has $level'_L$ in its presentation:

$$\begin{aligned} [level'_L]_{LL} = \text{present}_L [m_0]_{LH} \wedge [m_0]_{HL} = \text{present}_H [h]_{HH} \\ \equiv \quad \{ \text{let } m_0 \text{ be the } m \text{ from the quantification} \} \\ \exists m : [level'_L]_{LL} = \text{present}_L [m]_{LH} \wedge [m]_{HL} = \text{present}_H [h]_{HH} \\ \equiv \quad \{ \text{present-COMPOSE} \} \\ [level'_L]_{CL} = \text{present}_C [h]_{CH} \\ \equiv \quad \{ \text{assumption} \} \\ \text{true} \end{aligned}$$

Furthermore, by the first part of this conjunct and INTERPRESENT on the lower layer, we also have $[m_0]_{LH} = \text{interpret}_L [level'_L]_{LL}$

We need two intermediate results for the inductive step of the proof. The first one is:

$$\begin{aligned} [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{HL} = \text{present}_H [h]_{HH} \\ = \quad \{ [m_0]_{HL} = \text{present}_H [h]_{HH} \} \\ [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{HL} = [m_0]_{HL} \\ \Leftarrow \quad \{ \text{ABSORPTION} \} \\ [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{LH} = [m_0]_{LH} \\ \equiv \quad \{ [m_0]_{LH} = \text{interpret}_L [level'_L]_{LL} \} \\ [[m_0]_{LH} \triangleright_{LH} level_M]_{LH} = [m_0]_{LH} \\ \equiv \quad \{ \triangleright\text{-VALID} \} \\ [m_0]_{LH} = [m_0]_{LH} \\ \equiv \quad \{ \text{reflexivity of } = \} \\ \text{true} \end{aligned}$$

A corollary of this proof is that $[\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{LH} = [m0]_{LH}$, which is used in the proof of the second intermediate result:

$$\begin{aligned}
& level'_L = \text{present}_L [\text{interpret}_L [level'_L]_{LH} \triangleright_{LH} level_M]_{LH} \triangleright_{LL} level'_L \\
& \\
& level'_L \\
= & \quad \{ \triangleright\text{-IDEM and } [level'_L]_{LL} = [level'_L]_{LL} \} \\
& [level'_L]_{LL} \triangleright_{LL} level'_L \\
= & \quad \{ [level'_L]_{LL} = \text{present}_L [m0]_{LH} \} \\
& \text{present}_L [m0]_{LH} \triangleright_{LL} level'_L \\
= & \quad \{ \text{corollary: } [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{LH} = [m0]_{LH} \} \\
& \text{present}_L [\text{interpret}_L [level'_L]_{LL} \triangleright_{LH} level_M]_{LH} \triangleright_{LL} level'_L
\end{aligned}$$

Now we can prove the inductive step using the induction hypothesis:

$$\begin{aligned}
& [level'_M]_{HL} = \text{present}_H [h]_{HH} \wedge level'_L = x \Rightarrow \\
& \text{wp}(Comp_{n-1}, level'_M = level''_M \wedge level'_L = x) \quad \text{I.H.} \\
& \\
& \text{wp}(Comp_n, level'_L = level''_L) \\
\equiv & \quad \{ \text{Comp-DEF} \} \\
& \text{wp}(Up; Comp_{n-1}; Dwn, level'_L = level''_L) \\
\equiv & \quad \{ \text{wp-;} \} \\
& \text{wp}(Up; Comp_{n-1}, \text{wp}(Dwn, level'_L = level''_L)) \\
\equiv & \quad \{ \text{Dwn-DEF} \} \\
& \text{wp}(Up; Comp_{n-1} \\
& \quad , \text{wp}(level'_L := \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L, level'_L = level''_L)) \\
\equiv & \quad \{ \text{wp-:=} \} \\
& \text{wp}(Up; Comp_{n-1}, level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L) \\
\Leftarrow & \quad \{ \text{wp-MONO and Leibniz} \} \\
& \text{wp}(Up; Comp_{n-1}, level'_M = level''_M \wedge level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L) \\
\equiv & \quad \{ \text{wp-;} \} \\
& \text{wp}(Up, \text{wp}(Comp_{n-1} \\
& \quad , level'_M = level''_M \wedge level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L)) \\
\Leftarrow & \quad \{ \text{wp-MONO and I.H.} \} \\
& \text{wp}(Up, [level'_M]_{HL} = \text{present}_H [h]_{HH} \wedge level'_L = \text{present}_L [level''_M]_{LH} \triangleright_{LL} level'_L) \\
\equiv & \quad \{ \text{Up-DEF} \}
\end{aligned}$$

$$\begin{aligned}
& \text{wp}(\text{level}'_M := \text{interpret}_L [\text{level}'_L]_{LL} \triangleright_{LH} \text{level}_M \\
& \quad , [\text{level}'_M]_{HL} = \text{present}_H [h]_{HH} \wedge \text{level}'_L = \text{present}_L [\text{level}'_M]_{LH} \triangleright_{LL} \text{level}'_L) \\
\equiv & \quad \{ \text{wp-} := \} \\
& [\text{interpret}_L [\text{level}'_L]_{LL} \triangleright_{LH} \text{level}_M]_{HL} = \text{present}_H [h]_{HH} \wedge \\
& \text{level}'_L = \text{present}_L [\text{interpret}_L [\text{level}'_L]_{LL} \triangleright_{LH} \text{level}_M]_{LH} \triangleright_{LL} \text{level}'_L \\
\equiv & \quad \{ \text{the two intermediate results we just proved} \} \\
& \text{true}
\end{aligned}$$

This completes the proof of the first postcondition conjunct:

$$[\text{level}'_L]_{CL} = \text{present}_C [h]_{CH} \wedge \text{level}'_{n+1} = x \Rightarrow \text{wp}(\text{Comp}_n, \text{level}'_L = \text{level}''_L)$$

For the second part of the postcondition, we need:

$$[\text{level}'_L]_{CL} = \text{present}_C [h]_{CH} \wedge \text{level}'_{n+1} = x \Rightarrow \text{wp}(\text{Comp}_n, \text{level}'_{n+1} = x)$$

which holds, because Comp_n only does assignments on levels $0 \dots n$. Thus, by wp-AND, we have:

$$\begin{aligned}
& [\text{level}'_L]_{CL} = \text{present}_C [h]_{CH} \wedge \text{level}'_{n+1} = x \Rightarrow \\
& \text{wp}(\text{Comp}_n, \text{level}'_L = \text{level}''_L \wedge \text{level}'_{n+1} = x)
\end{aligned}$$

□

5.4.8 DOC-PRESERVE and INTENDED requirements

Similar to the single-layered editor, DOC-PRESERVE states that interpretation extra state is reused, whereas INTENDED has a double function. The requirement states both that presentation extra state is reused, as well as that the final lower level resembles the updated lower level.

Consider the interpretation extra state of the composition, which is represented by the equivalence classes of CH . Each of these classes consists of a number of HH classes. If we do not assume ABSORPTION, then closeness in a class of CH is established in two steps. The lower layer selects the closest HH class by \triangleright -CLOSE, after which DOC-PRESERVE of the higher layer selects the closest element from this class.

On the other hand, if ABSORPTION holds, then each CH class corresponds to exactly one HH class, and closeness is guaranteed by DOC-PRESERVE on the higher layer.

The part of INTENDED for reusing presentation extra state is guaranteed by the component layers in a similar way as for interpretation extra state (without assuming ABSORPTION). The other part of INTENDED must be guaranteed by interpret_L and interpret_H , and thus serves as the specification of these functions.

5.4.9 Conclusions

A layered editor consisting of two component layers can be specified in the same manner as a single-layered editor, but imposes additional requirements on the components. In order to guarantee that the composition is valid, we need the present-MATCH and interpret-MATCH conditions on the components. Furthermore, for DOC-INERT and PRES-INERT requirements, we need to assume the conditions ORTHOGONAL and the rather complex ORTHOGONAL-R.

The stronger condition ABSORPTION implies all four additional conditions:

$$[m]_{LH} = [m']_{LH} \Rightarrow [m]_{HL} = [m']_{HL} \quad \text{ABSORPTION}$$

For the top-most composition, we cannot assume ABSORPTION because this would rule out interpretation extra state altogether. However, present-Match, interpret-Match, ORTHOGONAL trivially hold for the top-most composition, and PRES-INERT is proven without assuming ABSORPTION for the top-most composition.

Because an editor consisting of n layers is specified by repeatedly splitting of layers from underneath, we need to prove ABSORPTION for each of the $n - 1$ composite layers, except the top-most one.

ABSORPTION is a rather strong condition. It only allows interpretation extra state at the document level, since all other interpretation extra state is absorbed by presentation extra state. Further research is necessary to determine whether this restriction disallows the specification of useful editors, in which case a workable but less strict condition needs to be established.

5.5 Duplicate presentations

The specification developed in the previous sections does not yet offer support for presentations that duplicate information (see also Section 4.3). A simple example of a duplicate presentation is the function $\text{present } x = (x, x)$. An intuitive way to handle edit operations on duplicates is to use the duplicate that was edited for computing the document update. This behavior, however, cannot be expressed by the specification from the previous section, since the specification does not take the original $level_L$ into account.

Duplicates become particularly problematic in combination with extra state and multiple layers, but although the edit behavior for duplicates is difficult to specify formally, the implementation of this behavior is often rather clear. Hence, in this section, we only sketch how the editor specification may be adapted to support the handling of duplicate presentations.

5.5.1 Dealing with duplicates

A precise definition of when a presentation contains duplicates is hard to give without making additional assumptions on the presentation formalism and the level types. Therefore, we adopt an informal notion of duplication, illustrated by a number of examples.

Besides the obvious present $x = (x, x)$, we speak of duplication whenever a presentation contains two or more values that depend on the same document value (e.g. present $x = (2x, -x)$). This holds even if the computation for the derived value does not have an inverse, as in present $x = (x, x \bmod 256)$. Although the term may seem somewhat odd in this case, we do refer to such presentations as duplicates. A related example present $x = (x \operatorname{div} 256, x \bmod 256)$ shows the subtlety of duplication, since in this case there is no duplication of information, and both elements of the presentation tuple can be edited without causing a conflict.

Further, besides depending on a single document value, a duplicate may also depend on several values. Hence, an average value of a list of integers (when presented together with the list) can be regarded as a duplication of the values in the list. Other examples of duplication are derived type signatures for functions, or even the color of a keyword in a syntax-coloring editor.

The easiest way to deal with duplicate information is to simply ignore all but one of the duplicates on interpretation and thus making the ignored duplicates non-editable. This is in fact the only way that is allowed by the specification of the previous sections. For present $x = (x, x)$, this leaves a choice of interpret functions: with interpret $(x, y) = x$, only the first value is editable, and with interpret $(x, y) = y$ only the second value.

In some cases non-editable duplicates are perfectly acceptable. In case of the list of integers with its average, few people will expect to be able to edit the average value. Even fewer people will expect to be able to edit colors in a syntax-coloring editor and thereby modify the edited program source. In many cases, however, we do want to be able to edit duplicate values.

We discuss how each of the requirements from the previous sections is affected by the presence of duplicates.

5.5.2 Adapting the INTENDED requirement

When a duplicate value in a presentation is edited, the edited duplicate should determine the document update. This is in line with the requirement that if an update not exact, the editor will perform the operation that the user intended.

Edit operations on duplicates most probably result in an invalid presentation, unless a user has managed to edit all occurrences of the duplicated element in a consistent way. Hence, to a large extent, the result of an edit operation on a duplicate value is specified by the INTENDED requirement:

$\{\text{true}\} \text{Comp } \{level'_L \text{ “close to” } level''_L\}$

INTENDED

However, in its current form, INTENDED cannot specify the desired behavior for edited duplicates, since it relates $level''_L$ to $level'_L$ without taking $level_L$ into account. An example shows the problem that can occur.

Consider an editor with $\text{present } x = (x, x)$. Because the edit operation $(0, 0) \rightsquigarrow (1, 0)$ should result in presentation $(1, 1)$, we must have $(1, 0)$ “close to” $(1, 1)$. On the other hand, the desired result of $(1, 1) \rightsquigarrow (1, 0)$ is $(0, 0)$, implying $(1, 0)$ “close to” $(0, 0)$. Thus, based on $level'_L$ alone, INTENDED cannot specify the correct behavior for an edit operation that updates the lower level to $(1, 0)$.

A second example shows the problem more precisely. Consider the function $\text{present } x = (x, x, x)$. If a user edits the first element of the presentation $((0, 0, 0) \rightsquigarrow (1, 0, 0))$ the intuitive result would be to change the document to 1. However, since $(1, 0, 0)$ is arguably closer to $(0, 0, 0)$ than $(1, 1, 1)$, $level''_H$ is specified to be 0 instead of 1. Therefore, an editor conforming to the specification does not allow any editing in the presentation, unless two or more values are edited simultaneously and consistently.

The problem with the INTENDED requirement is that the duplicates of the edited part of the presentation influence the final result of the edit operation. We tackle the problem by introducing an operator $\Delta :: T \rightarrow T \rightarrow T^*$, to ignore duplicates of edited values.

The application $level \Delta level'$ returns a value that is structurally the same as $level'$, but in which for each edited value, all duplicates have been replaced by wildcards. (Note that Δ thus depends on the presentation mapping.) For a presentation without duplicates, we have $level \Delta level' = level'$. If two duplicates are updated simultaneously, the result of Δ is not defined.

Similar to Section 5.3, a wildcard stands for any value. A wildcard is ignored when testing for equality, or evaluating closeness. Hence, both $(1, 1, 1)$ and $(1, 3, 5)$ are equally (and maximally) close to $(1, *_{Int}, *_{Int})$. If we use $level \Delta level'$ instead of $level'$ in the INTENDED requirement, the duplicates of the edited part of the presentation will have been replaced by wildcards and thus no longer influence the final result.

We provide two examples to illustrate the behavior of Δ . For the first example, consider the presentation function $\text{present } (x, y) = (x, x, x, y, y)$. If a user updates the second element of the presentation tuple (e.g. $(1, 1, 1, 5, 5) \rightsquigarrow (1, 2, 1, 5, 5)$), we compute $(1, 1, 1, 5, 5) \Delta (1, 2, 1, 5, 5) = (*_{Int}, 2, *_{Int}, 5, 5)$, which can be interpreted unambiguously as $(2, 5)$.

Another example is $\text{present } (x, y) = (x, y, x + y)$. For an update on the first element of the tuple $((1, 2, 3) \rightsquigarrow (10, 2, 3))$, we get $(1, 2, 3) \Delta (10, 2, 3) = (10, 2, *_{Int})$, which is interpreted as the document $(10, 2)$. If the sum is edited, for example by $(1, 2, 3) \rightsquigarrow (1, 2, 13)$, we get $(1, 2, 3) \Delta (1, 2, 13) = (*_{Int}, *_{Int}, 13)$. In this case, the specification leaves a choice for the document value, since any value of the form $(z, 13 - z)$ is a correct interpretation. If we take into account PRES-PRESERVE (or DOC-PRESERVE), the choices are reduced to $(11, 2)$ and $(1, 12)$.

It is difficult to give a formal specification of Δ , because we do not have a formal definition of the notion of duplicate information. Moreover, when a value influences the structure of a presentation, rather than a specific value in the structure, this cannot be modeled with wildcards. For these reasons, we keep the description of Δ informal, and leave the behavior of the editor unspecified in conflict situations.

Using Δ , we define an INTENDED requirement that does not require closeness on duplicates of the edited part of the presentation:

$$\{\text{true}\} \text{Comp } \{level_L \Delta level'_L \text{ "close to" } level''_L\} \quad \text{INTENDED}$$

However, this requirement alone is not sufficient. Recall that the old INTENDED also served to preserve the presentation extra state. Because duplicates are ignored by the new INTENDED, the requirement says nothing about their extra state. Therefore, we need to add a weaker requirement for preserving extra state of the duplicates. The new requirement corresponds to the old INTENDED requirement:

$$\{\text{true}\} \text{Comp } \{level'_L \text{ "close to" } level''_L\} \quad \text{PRES-PRESERVE}$$

With the new requirements, the two examples at the start of this section are no longer problematic. For the present $x = (x, x)$ example, after $(0, 0) \rightsquigarrow (1, 0)$ the requirement states $(1, *_{Int})$ "close to" $level''_L$, with $(1, 1)$ as a solution, whereas for $(1, 1) \rightsquigarrow (1, 0)$ it states $(*_{Int}, 0)$, with $(0, 0)$, as a solution. For the triple presentation, the requirement after updating the first element states $(1, *_{Int}, *_{Int})$ "close to" $level''_L$, which has $(1, 1, 1)$ as a solution.

5.5.3 Adapting the PRES-INERT requirement

Besides the INTENDED requirement, we also need to adapt the PRES-INERT requirement to support duplicates. An example shows the problem that can occur with the old PRES-INERT requirement.

Consider an editor for a very simple functional language. The document is a list of declarations, which is presented as a list of strings together with a message about type correctness. Furthermore, the body of a function may be hidden (see Section 2.1.1), in which case it is represented by the string "...".

A type-correct document $[f = a + 2, a = 1]$ can be presented while hiding the body of f , yielding: "f = ...; a = 1; ok". The string ok signals that the program is type correct.

If a user performs the update $a = 1 \rightsquigarrow a = True$, the most natural result would be an updated document $[f = a + 2, a = True]$, with a presentation that shows the type error: "f = ...; a = True; error". Of course, in a real-world editor we would like to have a somewhat more informative error message, but this basic message is sufficient for the example.

The problem that occurs is that because $level'_L$ ("f = ... ; a = True; ok") is a valid presentation (take the document [$f = 2, a = True$]), PRES-INERT states that $level'_L = level''_L$. Therefore, according to the specification, the editor has to update the document such that its presentation is "f = ... ; a = True; ok". The only possible way to do this is by changing the hidden body of f , which is clearly not the desired behavior.

To prevent PRES-INERT from suggesting updates on hidden parts of the document, we adapt the requirement in a similar way as INTENDED; both the pre- and the postcondition are changed to ignore duplicates of the updated parts of the lower level. We use $=^*$ to denote equality on values that may contain wildcards.

$$\{[level_L \Delta level'_L]_L =^* \text{present } [h]_H\} \text{ Comp } \{level_L \Delta level'_L =^* level''_L\}$$

PRES-INERT

The precondition has been weakened because otherwise the presence of duplicates in the presentation may disable the requirement. The postcondition, on the other hand, is weakened because we only require the updated parts to stay the same. Duplicates of the updated part, such as the type error in the example above, may change.

Because the precondition of PRES-INERT has been weakened, it is satisfied by the two examples $\text{present } x = (x, x)$ and $\text{present } x = (x, x, x)$. Hence, the editing behavior for these examples is no longer specified by the INTENDED requirement. The INTENDED requirement now, more appropriately, only applies to presentation updates that are not valid even when duplicates are ignored. An example of such an update occurs in an expression editor with syntax coloring and two views. If an expression is entered in one of the views, it is not a valid presentation even if the second view is ignored, because the new expression does not have the correct syntax coloring. Hence, the INTENDED requirement applies.

5.5.4 The remaining requirements

The precondition of DOC-INERT could be changed in a similar way as the precondition of PRES-INERT, but this is not necessary. Because the precondition of the edit step ensures that the unchanged parts of the presentation are valid with respect to $level_H$, only the changed parts may break the precondition. Hence, it makes no difference to also include the unchanged parts in the precondition of DOC-INERT, as the changed parts determine whether the requirement applies or not.

Further, POSTCONDITION should still hold between the entire higher and lower level, and DOC-PRESERVE only refers to higher-level values and is therefore not affected by duplicates in the presentation.

Finally, we have INTERPRESENT: $[l]_L = \text{present } [h]_H \Rightarrow [h]_H = \text{interpret } [l]_L$, which is still a valid requirement in the presence of duplicates. However, the requirement does not say anything about presentations in which only one duplicate is edited, because in that case the presentation is invalid. Hence, INTERPRESENT is no longer sufficient to

prove PRES-INERT (although it probably still implies DOC-INERT). Moreover, because it does not refer to the data levels and *Comp*, INTERPRESENT cannot be modified in a straightforward way, similar to DOC-INERT and INTENDED.

5.5.5 Conclusions

Summarizing, we have the following set of requirements:

$[l]_L = \text{present } [h]_H \Rightarrow [h]_H = \text{interpret } [l]_L$	INTERPRESENT
$\{\text{true}\} \text{Comp } \{[level''_L]_L = \text{present } [level''_H]_H\}$	POSTCONDITION
$\{[level'_L]_L = \text{present } [level]_H\} \text{Comp } \{level_H = level''_H\}$	DOC-INERT
$\{[level_L \Delta level'_L]_L =^* \text{present } [h]_H\} \text{Comp } \{level_L \Delta level'_L = level''_L\}$	PRES-INERT
$\{\text{true}\} \text{Comp } \{level_L \Delta level'_L \text{ "close to" } level''_L\}$	INTENDED
$\{\text{true}\} \text{Comp } \{level'_L \text{ "close to" } level''_L\}$	PRES-PRESERVE
$\{\text{true}\} \text{Comp } \{level_H \text{ "close to" } level''_H\}$	DOC-PRESERVE

If the presentation does not contain duplicates, then the requirements correspond exactly to the requirements of the previous section.

Further research is needed to establish a formal notion of duplicates in the presentation, as well as a specification of the Δ operator. Furthermore, it would be desirable to have a form of INTERPRESENT that implies PRES-INERT even in the presence of duplicates. Using these concepts, we could provide a verification that *Comp* meets the requirements, similar to the verification in Section 5.4.

With duplicates it is easy to specify presentation mappings for which it is difficult, if not impossible, to specify an interpretation mapping. However, it must be noted that it is not the aim of the specification to be able to handle every possible presentation mapping. Rather, we wish to be able to specify editors for the common cases of duplication, which we know how to handle.

5.6 Conclusions and related work

In this chapter, we have provided a specification of the `interpret` mapping, given a presentation mapping `present`. Together with the provided computation `Comp` this constitutes a specification of a presentation-oriented editor. The specification is layered and supports extra state in both presentation and interpretation direction.

The combination of extra state and duplication makes it easy to construct a presentation mapping for which we cannot specify an `interpret` function. However, the aim of the specification is not to specify an editor for every imaginable presentation mapping, but rather to be able to formally specify the editors for real-world examples in which basic cases of duplication and extra state occur.

The specification establishes requirements on the pair of `present` and `interpret` functions that constitute an editor, and helps to clarify the notion of a layered presentation-oriented editor. Furthermore, the specification gives a more precise definition of the concept of extra state, and what it entails to reuse extra state.

Related to the specification in this chapter is the work by Meertens [57] on maintainers for constraints between a document and its presentation. Given a constraint between two values, Meertens formulates a number of requirements for a pair of functions to be a maintainer for that constraint. In our case, the constraint would be the presentation relation. Meertens also provides a more formal approach to the concept of closeness and gives a number of maintainers for specific constraints.

Rather than specifying requirements for an editor, Greenwald, et al. [32], and Mu, Hu, and Takeichi [60], describe injective languages that allow the computation of a document from its (possibly edited) presentation. The language by Mu, Hu, and Takeichi also takes into account duplication of information.

The three formalisms have in common that their focus is not immediately on real-world presentation functions. In their current state they only handle rather basic presentations. It is not yet clear whether these basic presentations are scalable to the presentations of the use cases from Chapter 2. Furthermore, all of the formalisms only take into account interpretation extra state, while disregarding presentation extra state. Finally, only [57] deals explicitly with a layered presentation relation.

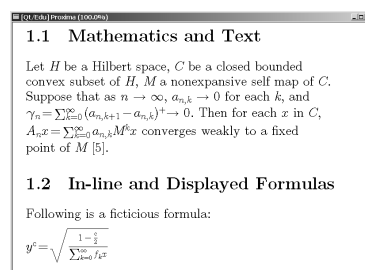
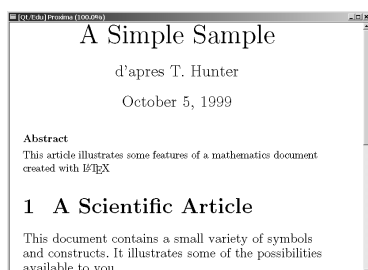
An important area of future research lies in the development of a presentation language that is powerful enough to specify all use cases, and which automatically constructs `interpret`. Because an efficient inverse cannot always be computed automatically, complex parts of the presentation can be annotated, whereas for simpler presentations `interpret` is constructed automatically. Until such a language has been developed, it is up to the implementor to guarantee correctness of the editor.

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 XPRES 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 XPRES 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 XPRES 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 easy.

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

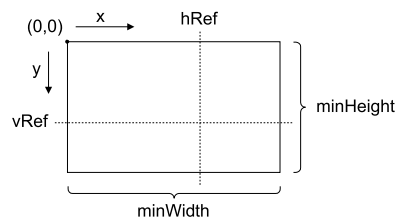
6.2 The XPREZ target language

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. `onMouseClicked :: 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 `overly` 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 `overlyR` 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.

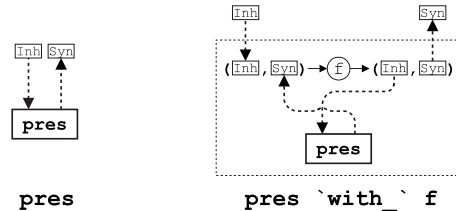
Here is an example XPREZ presentation that illustrates alignment and stretching in a row:

```
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 XPRES 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 XPRES 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:

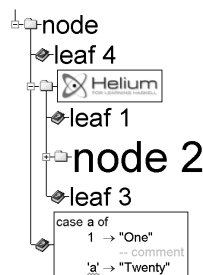
$$\frac{x}{2}$$

$$1 + y$$

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: XPRES 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 XPRES 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 XPRES, 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 XPRES is part of the Proxima prototype.

XPRES 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 XPRESZ 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 XPRESZ.

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, XPRESZ 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 XPRESZ 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 XPRESZ expressiveness and efficient evaluation are considered to be more important than safety. An XPRESZ 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 XPRESZ 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 XPRESZ, 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 XPRESZ 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 XPRESZ presentations. With such a model, we can describe the exact behavior of the combinators with laws, rather than textual descriptions.

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 XPRES 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.

```

Proxima v0.2
File
Focused expression :: a -> b
Top level identifiers: 'list'; 'large'; 's'; 'f';

module Main where

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

large :: Int -> Int -> Int -> Int -- Value: <function>
large = [..]

s :: (a -> b -> c) -> (a -> b) -> a -> c -- Value: <function>
s = \f -> \g -> \x -> f x (g x);

f :: Int -> Int -- Value: <function>
f = \x -> x2+2*x+ $\frac{(3*x)*(2*x)*1}{(x+1)^2}$ ;

Variables in scope:
f :: a -> b -> c
g :: a -> b
large :: Int -> Int -> Int -> Int
list :: [Int]

```

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.

$$\begin{array}{ccc}
 \boxed{\text{list} :: [\text{Int}] \text{ -- Value: [15,} \\ \text{list} = [3*5, 1+2, 27] ;} & \Rightarrow & \boxed{\text{list} :: [\text{Int}] \text{ -- Value: [3,} \\ \text{list} = [1+2, 27] ;} & \Rightarrow & \boxed{\text{list} :: [\text{Int}] \text{ -- Value: [3,} \\ \text{list} = [1+2, 27, 3*5] ;}
 \end{array}$$

*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:

$$\begin{array}{ccc}
 \boxed{\text{list} = [1+2, 27, 3*5] ;} & \Rightarrow & \boxed{\text{list} = [1+2, 27, 3*5, \frac{\{\text{Exp}\}}{\{\text{Exp}\}}] ;} & \Rightarrow & \boxed{\text{list} = [1+2, 27, 3*5, \frac{\{\text{Exp}\}}{\{\text{Exp}\}}] ;}
 \end{array}$$

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.

$$\boxed{\text{list} = [1+2, 27, 3*5, \frac{\{\text{Exp}\}}{\{\text{Exp}\}}] ;} \Rightarrow \boxed{\text{list} = [15, \frac{\{\text{Exp}\}}{\{\text{Exp}\}}] ;}$$

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+  $\frac{(3+True)*(2+x)*1}{(x+1)^2}$ ;
```

```
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`.

<pre>x :: Int x = f 3;</pre>	⇒	<pre>x :: Int -- Value: 16 x = (λx → x²+2*x+ $\frac{(3+x)*(2+x)*1}{(x+1)^2}$) 3;</pre>	⇒	<pre>x :: Int -- Value: 16 x = (3²+2* 3+ $\frac{(3+3)*(2+3)*1}{(3+1)^2}$);</pre>
“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

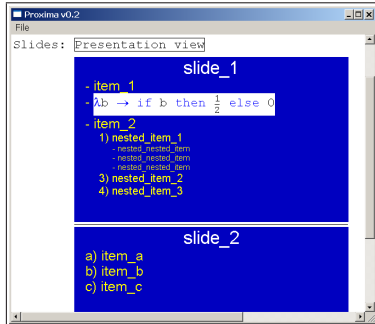


Figure 7.2: A slide editor.

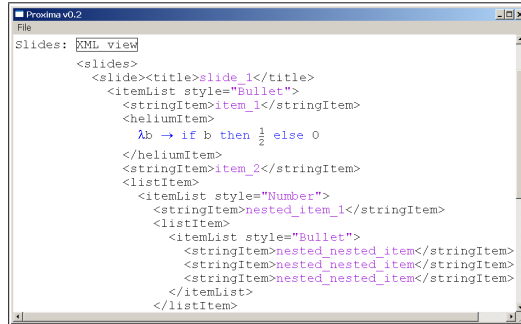


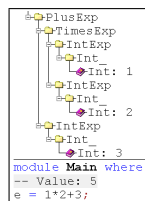
Figure 7.3: Slides as viewed as XML.

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.

Top level identifiers: 'list'; 'large'; 'x'; 's'; 'f'; <pre>module Main where list :: [Int] -- Value: [3, 27, 15] list = [1+2, 27, 3*5];</pre>	⇒	Top level identifiers: 'list1'; 'large'; 'x'; 's'; 'f'; <pre>module Main where list1 :: [Int] -- Value: [3, 27, 15] list1 = [1+2, 27, 3*5];</pre>
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

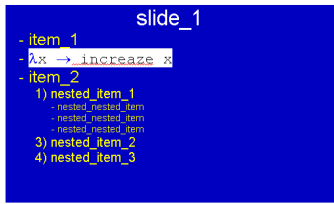
```

module Main where

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

Slides: Presentation view

```



```

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

```

Figure 7.4: Helium slides.

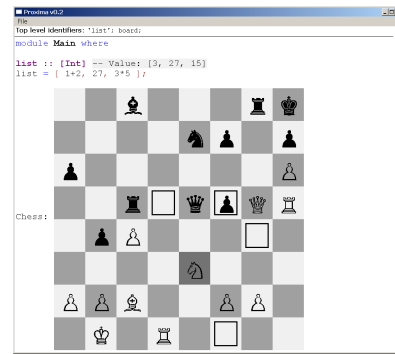


Figure 7.5: A game of chess: Ne3-g4?

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 `increase`.

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: $\{\text{ident}_1 \dots \text{ident}_n\}$. 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 “ \rightarrow ”), 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 $\lambda x \rightarrow x+1$. 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 “ \rightarrow ”). 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 child1 ... childi ... childn)
```

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 c1...cn :: 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 XPRES 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 XPRES are implemented for a large part at the arrangement layer.

Rendering

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

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 XPRESZ 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.

Conclusions and further research

In this thesis, we have introduced the presentation-oriented structure editor Proxima. Proxima provides an intuitive and user-friendly way of editing for complex document presentations which may contain derived structures.

In Chapter 2 we investigated a number of use cases for a generic editor, and formulated a set of functional requirements based on these use cases. We have provided an evaluation of existing generic editors with regard to the requirements. It turns out that none of the existing systems is able to handle all of the use cases. In our opinion, the reason for this is that these editors either lack the power to support the complex presentations of the use cases, or have an edit model that is overly restrictive. The chapter ended with a brief description of the Proxima editor, which has been designed to meet the requirements and will be able to handle all of the use cases.

Proxima has a layered architecture that makes it possible to support both presentation-oriented and document-oriented editing. An overview of the layers and data levels of the architecture has been provided in Chapter 3, followed by a specification in chapters 4 and 5.

In Chapter 6 we introduced the declarative presentation formalism XPRES. The language is a Haskell combinator library for creating graphical presentations with an advanced alignment model. Using Haskell's abstraction facilities, complex presentations may be defined in a concise way.

A prototype for Proxima has been implemented and was discussed Chapter 7. To instantiate an editor, a basic Haskell document type definition must be provided together with a presentation sheet and a parsing sheet. The presentation sheet is an attribute grammar,

and the parsing sheet is a combinator parser. A number of example editors have been instantiated with the prototype.

8.1 Further research

The formal specification has provided a wildcard representation for extra-state equivalence classes. However, the specified reuse function is rather basic and only allows reusing extra state after simple edit operations. The specification should be improved with a more advanced reuse function, a sketch of which was provided. Furthermore, the specification for an editor supporting duplicates in the presentation was only informal. A more formal definition of the notion of duplication as well as of the mechanism of ignoring duplicates is needed to establish a formal specification for the editor that supports duplicates.

In Proxima, functions for both the presentation and the interpretation direction need to be specified. The inverse of the presentation sheet is the parsing sheet, and the inverse of the evaluation sheet is the reduction sheet. Because specifying these inverses by hand is a possible source of errors, a bidirectional presentation formalism, which automatically generates an inverse function, is desirable.

The evaluation layer of Proxima is an ideal place for such a bidirectional presentation (or rather evaluation) formalism. Thus, the reduction sheet will be automatically generated from the evaluation sheet. For the presentation layer the situation is somewhat more complicated because it does not seem realistic to assume that an efficient parser can always be generated automatically from the presentation sheet. On the other hand, many parts of the presentation are straightforward and could be inverted automatically. In the prototype, for example, it is already clear how the structure recognizers in the parsing sheet can be derived from the presentation sheet. For the more difficult parts of the presentation, the presentation sheet could contain special directives for parsing, or even an explicit specification of the appropriate part of the parser.

Chapter 7 already provided an overview of the future research concerning the Proxima prototype. We recapitulate a few main issues here. The most important area of research is support for incrementality, consisting both of built-in incremental behavior for the lower layers, as well as the application of techniques for incremental attribute evaluation to the attribute-grammar compiler. A second important area concerns the evaluation layer for which we need to establish the formalisms for the sheets, as well as provide an implementation. Furthermore, we need extensions to the attribute grammar formalism and the XPRES presentation language. And finally, libraries of useful functions must be compiled, to facilitate the specification of common editor behavior.

8.2 Final remarks

The approach taken for Proxima is different from most other generic editing projects. Most of these projects take as the starting point a specification formalism that guarantees a correct and efficient editor for a limited range of applications. Proxima, on the other hand, provides a general architecture with presentation and computation formalisms that are powerful enough to build serious editor applications. In the initial stages of the Proxima project, it is left to the editor designer to guarantee safety and efficiency of the implemented editor. Support for automatic interpretation and incrementality will be added at a later stage.

Because both the presentation and the interpretation need to be specified, it is possible to specify an editor for which the parser does not match the presentation. Furthermore, because the presentation formalism allows arbitrary computations, it is possible to specify a presentation that is too slow for editing, or even crashes. Nevertheless, the safety of an editor built with Proxima is already much easier to guarantee than if a similar editor had been built by hand. Further, in practice, it turns out to be rather straightforward to avoid inconsistencies between the presentation and interpretation functions.

An advantage of our approach is that even in an early stage, complex editors can be specified (albeit with a little more effort). And, moreover, it is possible to specify editors for which automatic interpretation is not yet an option. A related advantage is that by building and experimenting with editors, it becomes clear which parts of the generic system should get the highest development or research priorities.

Even without the language support and libraries for common presentation patterns, it is already straightforward to specify a complex editor in Proxima. Thus, the Proxima prototype shows that it is possible to combine a powerful presentation formalism with a modeless integration of document-oriented and presentation-oriented editing. The resulting editors are powerful, yet easy to use.

Bibliography

- [1] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible Stylesheet Language (XSL) Version 1.0. *W3C Recommendation 15 October 2001*, <http://www.w3.org/TR/2001/REC-xsl-20011015>, 2001.
- [2] M. Altheim and S. McCarron. XHTML 1.1 – Module-based XHTML. *W3C Recommendation 31 May 2001*, <http://www.w3.org/TR/2001/REC-xhtml11-20010531/>, 2001.
- [3] G. J. Badros, A. Borning, K. Marriott, and P. J. Stuckey. Constraint cascading style sheets for the web. In *ACM Symposium on User Interface Software and Technology*, pages 73–82, 1999.
- [4] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.
- [5] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of the 10th international conference on Software engineering*, pages 356–366. IEEE Computer Society Press, 1988.
- [6] R. Bahlke and G. Snelling. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.
- [7] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):95–127, 1992.
- [8] P. Biron and A. Malhotra. XML Schema Part 2: Datatypes. *W3C Recommendation 2 May 2001*, <http://www.w3.org/TR/2001/REC-xmlschema-2-20010502>, 2001.
- [9] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst.*, 3(4):353–387, 1981.

- [10] P. Borrás, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 14–24. ACM Press, 1988.
- [11] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, level 2 (CSS2) Specification. *W3C Recommendation 12 May 1998*, <http://www.w3.org/TR/1998/REC-CSS2-19980512>, 1998.
- [12] M. Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report CSD-01-1149, University of California, Berkeley, 2001.
- [13] M. G. J. van den Brand. *Pregmatic, a generator for incremental programming environments*. PhD thesis, Katholieke Universiteit Nijmegen, Netherlands, 1992.
- [14] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):1–41, 1996.
- [15] M. G. J. van den Brand and Vinju, J.J. Rewriting with layout. In C. Kirchner and N. Dershowitz, editors, *Proceedings of RULE2000*, 2000.
- [16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. *W3C Recommendation 4 February 2004*, <http://www.w3.org/TR/2004/REC-xml11-20040204>, 2004.
- [17] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Inf. Comput.*, 140(2):229–253, 1998.
- [18] F. J. Budinsky, R. C. Holt, and S. G. Zaky. SRE – a syntax-recognizing editor. *Software – Practice and Experience*, 15(5):489–497, May 1985.
- [19] R. H. Campbell and P. A. Kirslis. The SAGA project: A system for software development. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 73–80. ACM Press, 1984.
- [20] D. Carlisle, P. Ion, R. Miner, and N. Poppelier. Mathematical Markup Language (MathML) Version 2.0 (Second Edition). *W3C Recommendation 21 October 2003*, <http://www.w3.org/TR/2003/REC-MathML2-20031021>, 2003.
- [21] J. Clark. XSL Transformations (XSLT) Version 1.0. *W3C Recommendation 16 November 1999*, <http://www.w3.org/TR/1999/REC-xslt-19991116>, 1999.
- [22] J. Clark. TREX – Tree Regular XML. *Thai Open Source Software Center*, <http://www.thaiopensource.com/trex>, 2001.
- [23] J. Clark and M. Makoto. RELAX NG specification. *OASIS*, <http://www.oasis-open.org/committees/relax-ng/spec.html>, 2001.

- [24] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, New York, 1984.
- [25] M. Ettrich et al. Lyx – The Document Processor. <http://www.lyx.org>, 2004.
- [26] C. N. Fischer, G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock. The Poe language-based editor project. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 21–29, 1984.
- [27] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, pages 1–23. Springer, Berlin, Heidelberg, 1995.
- [28] C. W. Fraser. A generalized text editor. *Communications of the ACM*, 23(3):154–158, 1980.
- [29] P. Fritzson. Preliminary experience from the DICE system – a Distributed Incremental Compiling Environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 113–123. ACM Press, 1984.
- [30] J. Ganzevoort. Maintaining presentation invariants in the Views system. Technical Report CS-R9262, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, December 1992.
- [31] D. B. Garlan and P. L. Miller. GNOME: an introductory programming environment based on a family of structure editors. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 65–72, 1984.
- [32] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan 2004.
- [33] R. Heckmann and R. Wilhelm. A functional description of \TeX 's formula layout. *Journal of Functional Programming*, 7(5):451–485, Sept. 1997.
- [34] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.
- [35] M. Horton. *Design of a multi-language editor with static error detection capabilities*. PhD thesis, University of California, Berkeley, 1981.
- [36] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of LNCS. Springer Verlag, 1995.

- [37] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.
- [38] International Organization for Standardization. *ISO 8879:1986: Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, Aug. 1986.
- [39] International Organization for Standardization. *ISO 10179:1996(E): Information technology – Text and office systems – Document Style Semantics and Specification Languages (DSSSL)*. International Organization for Standardization, Geneva, Switzerland, Apr. 1996.
- [40] J. Jeuring. Incremental algorithms on lists. In J. v. Leeuwen, editor, *Proceedings SION Computing Science in the Netherlands*, pages 315–335, 1991. Also appeared in EURICS Workshop on Calculational Theories of Program Structure, Hollum-Ameland, 1991.
- [41] W. Kahl. Beyond pretty-printing: Galley concepts in document formatting combinators. In G. Gupta, editor, *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 1999, Proceedings*, volume 1551 of LNCS, pages 76–90. Springer-Verlag, 1999.
- [42] G. E. Kaiser, P. H. Feiler, F. Jalili, and J. H. Schlichter. A retrospective on DOSE: an interpretive approach to structure editor generation. *Software – Practice and Experience*, 18(8):733–748, 1988.
- [43] G. E. Kaiser and E. Kant. Incremental Parsing without a Parser. *Journal of Systems and Software*, 5:121–144, 1985.
- [44] U. Kastens and C. Schmidt. VL-Eli: a generator for visual languages. In M. G. J. van den Brand and R. Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.
- [45] J. H. Kingston. Design and implementation of the Lout document formatting language. *Software – Practice and Experience*, 23(9):1001–1041, 1993.
- [46] P. Klint. A meta-environment for generating programming environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, Mar. 1993.
- [47] D. E. Knuth. *The TeXbook*. Addison Wesley, 1984.
- [48] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software – Practice and Experience*, 11(11):1119–1184, Nov. 1982.
- [49] J. W. C. Koorn. GSE: a generic text and structure editor. Technical Report P9202, University of Amsterdam, 1992.
- [50] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, 1966.

- [51] D. Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.
- [52] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM Press, 2003.
- [53] A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, The Netherlands, Sept. 2004.
- [54] B. Magnusson, M. Bengtsson, L. Dahlin, G. Fries, A. Gustavsson, G. Hedin, S. Minor, D. Oscarsson, and M. Taube. An overview of the Mjølner/ORM environment: Incremental language and software development. In *TOOLS'90, Paris, France*, 1990.
- [55] M. Makoto. RELAX (regular language description for XML). *INSTAC (Information Technology Research and Standardization Center)*, <http://www.xml.gr.jp/relax>, 2001.
- [56] J. P. M. Marden and E. V. Munson. PSL: An alternate approach to style sheet languages for the World Wide Web. *J.UCS: Journal of Universal Computer Science*, 4(10):792–806, 1998.
- [57] L. G. L. T. Meertens. Designing constraint maintainers for user interaction. <ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps>, 1998.
- [58] S. Minör. *On Structure-Oriented Editing*. PhD thesis, Department of Computer Science, Lund University, Sweden, 1990.
- [59] O. de Moor and J. Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1):3–27, 1999.
- [60] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Mathematics of Program Construction (MPC 2004)*, LNCS. Springer-Verlag, 2004.
- [61] B. A. Myers, D. Giuse, R. B. Dannenberg, B. Vander Zanden, D. Kosbie, E. Previn, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical highly-interactive user interfaces. *IEEE Computer*, 23(11), November 1990.
- [62] L. R. Neal. Cognition-sensitive design and user modeling for syntax-directed editors. In *Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, pages 99–102. ACM Press, 1987.
- [63] J. van der Hoeven. GNU TexMacs. <http://www.texmacs.org>, 2004.
- [64] K. Normark. Muir - a language development environment. In *Proceedings of the 1988 ACM SIGSMALL/PC symposium on ACTES*, pages 22–27. ACM Press, 1988.

- [65] D. Notkin. The GANDALF Project. *Journal of Systems and Software*, 5:91–105, 1985.
- [66] D. Notkin, N. Habermann, R. Ellison, G. Kaiser, and D. Garlan. Response to Waters' article on structure oriented editors. *SIGPLAN Notices*, 18(4), 1983.
- [67] Object Technology International, Inc. Eclipse platform – a universal tool platform. available at: <http://www.eclipse.org>, 2001.
- [68] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, 1990.
- [69] D. C. Oppen. Prettyprinting. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.
- [70] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.
- [71] V. Quint. The languages of Thot. Technical report, INRIA, <http://www.inrialpes.fr/opera/Thot/Doc/languages.html>, 1997. Translated by E. V. Munson.
- [72] V. Quint and I. Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 200–213. Cambridge University Press, 1986.
- [73] M. Read and C. Marlin. Generating direct manipulation program editors within the multiview programming environment. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 232–236. ACM Press, 1996.
- [74] S. P. Reiss. Pecan: Program development systems that support multiple views. In *Proceedings of the 7th international conference on Software engineering*, pages 324–333, 1984.
- [75] S. P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, 1995.
- [76] S. P. Reiss. The Desert environment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(4):297–342, 1999.
- [77] T. Reps and T. Teitelbaum. The Synthesizer Generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 42–48. ACM Press, 1984.
- [78] J. Saraiva, S. D. Swierstra, and M. Kuiper. Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International Conference on Compiler Construction, CC/ETAPS2000*, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, Mar. 2000.

- [79] A. Sellen, G. Kurtenbach, and W. Buxton. The role of visual and kinesthetic feedback in the prevention of mode errors. In *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*, pages 667–673. North-Holland, 1990.
- [80] U. Shani. Should program editors not abandon text oriented commands? *SIGPLAN Notices*, 18(1):35–41, 1983.
- [81] R. M. Stallman. EMACS the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 147–156, 1981.
- [82] B. Sufrin and O. de Moor. Modeless structure editing. In J. Davies, A. W. Roscoe, and J. Woodcock, editors, *Proceedings of the Oxford–Microsoft symposium in Celebration of the work of Tony Hoare*, Sept. 1999.
- [83] S. D. Swierstra. Combinator parsers: From toys to tools. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.
- [84] S. D. Swierstra, P. R. Azero Alocer, and J. Saraiava. Designing and implementing combinator languages. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.
- [85] S. D. Swierstra and A. Baars. *Attribute Grammar System*. Utrecht University, The Netherlands, <http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem>, 2004.
- [86] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.
- [87] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. *W3C Recommendation 2 May 2001*, <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502>, 2001.
- [88] M. L. Van De Vanter. Practical language-based editing for software engineers. In *ICSE Workshop on SE-HCI*, pages 251–267, 1994.
- [89] R. Verhoeven. *The Design of the MathSpad Editor*. PhD thesis, Eindhoven University, The Netherlands, 2000.
- [90] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
- [91] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 131–145. ACM Press, 1989.

- [92] N. Walsh, L. Muellner, and B. Stayton. *DocBook: The Definitive Guide*. O'Reilly, 2.0.9 edition, 2003.
- [93] R. C. Waters. Program editors should not abandon text oriented commands. *SIG-PLAN Notices*, 17(7):39–46, 1982.
- [94] World Wide Web Consortium. Amaya web editor/browser. <http://www.w3.org/Amaya>, 2004.

Samenvatting

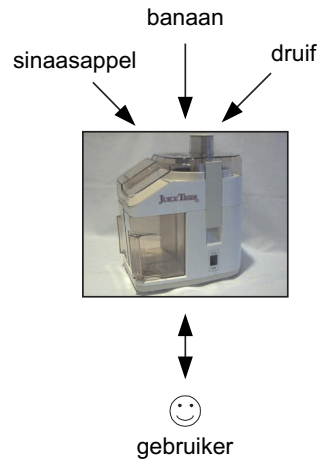
Een computergebruiker heeft over het algemeen te maken met een grote verscheidenheid aan documenten, zoals tekstbestanden, spreadsheets en webpagina's. De applicaties waarmee deze documenten bewerkt kunnen worden zijn zogenoemde *editors*, met als voorbeelden tekstverwerkers, spreadsheet-applicaties en HTML-editors. Ondanks de uiterlijke verschillen tussen editors vertonen de edit-operaties (tekst invoeren, selecteren, knippen/plakken, etc.) sterke overeenkomsten.

Proxima is een generieke editor waarmee een groot aantal verschillende documenttypes bewerkt kan worden. Om in te zien wat een generieke editor precies is, kunnen we editen vergelijken met het maken van vruchtensap. Van sinaasappels, bananen en druiven kan sap gemaakt worden met een citruspers, een blender en een druivenpers. Er zijn dus drie verschillende apparaten nodig, die elk een eigen gebruiksaanwijzing hebben. Bovendien moet voor een ander soort vrucht misschien weer een nieuw apparaat aangeschaft worden. Handiger is het daarom gebruik te maken van een algemene (of generieke) sapcentrifuge, zoals de Juice Tiger™ in Figuur 1 op de volgende bladzijde. Door middel van verschillende hulpstukken kan dit apparaat sap maken van diverse soorten vruchten en zo een aantal losse apparaten vervangen.

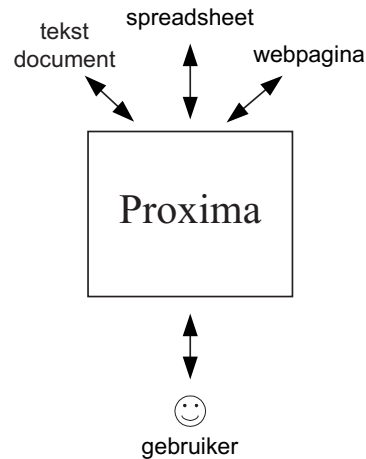
Zoals de sapcentrifuge een aantal losse fruitpersen vervangt, zo kan een generieke editor een aantal losse editors vervangen (zie Figuur 2). De rol van hulpstukken wordt nu vervuld door *style sheets*, waarin het specifieke gedrag van de editor voor een bepaald documenttype beschreven wordt. De voordelen van een generieke editor zijn vergelijkbaar met die van de sapcentrifuge. In plaats van verschillende applicaties is er bijvoorbeeld nog maar één applicatie, met een uniforme user interface. Het belangrijkste voordeel is echter dat het bouwen van een editor voor een nieuw type document met een generieke editor veel minder inspanning vergt dan op de conventionele wijze. Dit is vooral gunstig voor het bouwen van editors voor XML documenten.

Ondanks de voordelen zijn generieke editors echter weinig populair. Om een beeld te krijgen van de reden hiervoor, werpen we eerst een wat nauwkeuriger blik op de interne structuur van een generieke editor.

In de meeste editors kunnen we twee niveaus onderscheiden: het *document* en de *presentatie*. Het document is een interne representatie van de data die door de gebruiker geëdit wordt. Het document is niet rechtstreeks zichtbaar, maar wordt afgebeeld op een presentatie.



Figuur 1: Een generieke sapcentrifuge.



Figuur 2: Een generieke editor.

tatie die aan de gebruiker getoond wordt. De presentatie kan variëren van platte tekst tot aan opgemaakte tekst met grafische elementen zoals lijnen en plaatjes. Het afbeelden van het document op de presentatie wordt het presentatie proces genoemd, en het weer terug afbeelden van een gewijzigde presentatie op een nieuw document het interpretatie proces.

Op het niveau van zowel het document als de presentatie zijn edit-operaties denkbaar. Een documentgerichte edit-operatie is een verandering gericht op de structuur van het document, zoals het verwisselen van twee hoofdstukken. Presentatiegerichte operaties daarentegen zijn gericht op wat er op het scherm zichtbaar is. Voor een tekstuele presentatie betekent dit dat de tekst vrij geëdit kan worden, zelfs als dit niet correspondeert met een edit-operatie op de structuur.

Gerelateerd aan dit onderscheid tussen edit-operaties kunnen bestaande generieke editors in twee categorieën ingedeeld worden. Aan de ene kant zijn er de *syntax-directed* editors, die een krachtig presentatiemechanisme kunnen bieden, maar vrijwel alleen documentgerichte edit-operaties bieden. Dit wordt door gebruikers vaak als beperkend ervaren. De andere categorie wordt gevormd door de *syntax-recognizing* editors. Deze editors staan het vrij editen van de presentatie toe, maar beschikken weer over een beperkt presentatiemechanisme. Er bestaan ook tussenvormen (*hybrid* editors), maar die zijn vrijwel altijd te beschouwen als hoofdzakelijk *syntax-directed* danwel *syntax-recognizing* editors.

In dit proefschrift onderzoeken we hoe de voordelen van een generieke editor met een krachtig presentatiemechanisme te combineren zijn met een presentatiegericht edit model.

Na een algemene inleiding en een introductie van relevante begrippen in Hoofdstuk 1, wordt in Hoofdstuk 2 een aantal uiteenlopende toepassingen van een generieke editor beschreven. Onder deze voorbeelden bevinden zich bekende applicaties als een editor

voor programmacode, een tekstverwerker en een formule editor, maar bijvoorbeeld ook een elektronisch belastingformulier. Tezamen helpen de voorbeelden het toepassingsgebied van de Proxima editor vast te leggen. Aan de hand van de voorbeelden, die elk hun specifieke eisen stellen aan de editor, formuleren we een zestal functionele eisen, of requirements, voor een generieke editor:

Genericiteit. Uitgangspunt van het onderzoek is dat de editor generiek is, en niet ontworpen voor een specifiek documenttype.

Berekeningen in de presentatie. De presentatie moet berekende waarden en structuren kunnen bevatten, zoals hoofdstuknummers en een automatische inhoudsopgave.

Krachtig grafisch presentatieformalisme. Het presentatieformalisme moet krachtig genoeg zijn om tekstverwerkingsdocumenten met wiskundige formules te tonen, maar ook een elektronisch formulier met invoervelden en knoppen.

Presentatiegericht en documentgericht editen. Edit-operaties op zowel het document als op de presentatie moeten ondersteund worden. Tevens moeten edit-operaties voor specifieke documentsoorten gespecificeerd kunnen worden.

Modeless editen. Het moet het mogelijk zijn om eenvoudig te wisselen tussen presentatiegericht en documentgericht editen, zonder de editor expliciet in een andere toestand (mode) te moeten brengen.

Extra state. In sommige gevallen bevat een document informatie die niet in de presentatie zichtbaar is, en soms bevat de presentatie weer informatie die niet in het document opgeslagen is. Deze informatie noemen we *extra state*. De editor moet beide vormen van extra state ondersteunen.

Aan de hand van bovenstaande requirements wordt een aantal bestaande systemen onder de loep genomen en met elkaar vergeleken. Het blijkt dat geen van de bestaande systemen aan alle requirements voldoet, wat betekent dat er dus geen systeem is dat alle voorbeelden kan ondersteunen. Eén van de problemen is dat een editor aan de ene kant moet beschikken over een krachtig presentatiemechanisme met ondersteuning voor berekende waarden, grafische elementen en eventuele duplicatie van informatie. Aan de andere kant moet voor een gebruiksvriendelijk edit model het editen van de presentatie mogelijk zijn. Dit laatste houdt echter in dat een gewijzigde presentatie terugvertaald (geïnterpreteerd) moet worden naar een nieuw document, wat moeilijker is naarmate het presentatiemechanisme complexer is.

In hoofdstuk 3 stellen we een gelaagde architectuur vast, voor een editor die voldoet aan de zes requirements. Het probleem om presentatiegerichte editfunctionaliteit te bieden wordt door de gelaagde architectuur opgesplitst in een aantal eenvoudigere deelproblemen. De lagen vinden hun oorsprong in het presentatie proces dat in een aantal logische stappen verdeeld kan worden. Voorbeelden van deze stappen zijn het berekenen van afgeleide waarden, het bepalen van de logische opmaak van de presentatie, tot aan het uitrekenen van alle posities en het tonen van de presentatie aan de gebruiker. Het interpretatie

proces doorloopt dezelfde stappen in omgekeerde volgorde. Elk van de tussenresultaten wordt een data-niveau of *level* genoemd. Tussen twee niveaus bevindt zich een laag (of *layer*) die waarden van het ene niveau op het andere afbeeldt in zowel de presentatie als interpretatie richting. Het hoofdstuk geeft een overzicht van de verschillende data-niveaus en de lagen daartussen. De presentatie en interpretatie processen worden beschreven met behulp van voorbeelden.

Hoofdstukken 4 en 5 geven een specificatie van de Proxima editor. Hoofdstuk 4 is een inleiding op de specificatie, en geeft een model van het edit proces. Ook worden de begrippen extra state en duplicaten in de presentatie beschreven. De specificatie zelf is het onderwerp van Hoofdstuk 5. Het uitgangspunt is dat, gegeven een presentatiefunctie, een corresponderende interpretatiefunctie gespecificeerd wordt. De specificatie wordt in een aantal stappen ontwikkeld. De eerste stap is de specificatie van een editor die uit slechts één laag bestaat en geen rekening houdt met extra state. Deze specificatie wordt vervolgens stap voor stap uitgebreid tot de specificatie van een gelaagde editor met ondersteuning voor extra state. Het hoofdstuk eindigt met een schets van de behandeling van presentaties die duplicaten kunnen bevatten.

Om een document af te beelden op het scherm maakt Proxima gebruik van de presentatietaal XPRES. XPRES is een declaratieve presentatietaal die geschikt is voor een verscheidenheid aan toepassingen. Hoofdstuk 6 inventariseert een aantal requirements voor een presentatietaal en bevat een vergelijking van een aantal van zulke talen. Vervolgens wordt aan de hand van voorbeelden een informele beschrijving van de taal XPRES gegeven.

Hoofdstuk 7 beschrijft het prototype van Proxima, dat geïmplementeerd is in de functionele programmeertaal Haskell. Het prototype is platform-onafhankelijk en voldoet al aan vier van de zes genoemde requirements, terwijl aan de overige twee (presentatieformalisme en modelessness) grotendeels is voldaan. De berekende waarden en de presentatie van het document worden gespecificeerd met behulp van een attributengrammatica. Voor de interpretatie wordt gebruik gemaakt van een combinator parser. Het hoofdstuk toont screenshots van het prototype en bevat ook een aantal voorbeelden van de style sheets waarmee specifieke editors beschreven worden.

Ondanks de nog prille staat van het prototype is het al mogelijk om met relatief weinig moeite complexe editors te bouwen die toch prettig in het gebruik zijn. Op basis van de ervaringen met het prototype kunnen we concluderen dat de combinatie van een krachtig presentatiemechanisme en een presentatiegericht edit model zeker mogelijk is.

Dankwoord

Al is het schrijven van een proefschrift een grotendeels solitaire bezigheid, je doet het toch zeker niet zonder hulp van een heleboel anderen. En dit is de uitgelezen plek om iedereen daar eens uitgebreid voor te bedanken.

Allereerst natuurlijk dank aan mijn promotoren, Doaitse Swierstra, Johan Jeurig en Lambert Meertens. Johan, als dagelijks begeleider heb je heel wat stukjes van mij langs zien komen, die je elke keer weer van gedetailleerd commentaar voorzag. Ook kon ik, ondanks jouw steeds drukker wordende schema, altijd bij je terecht als ik iets wilde bespreken. Dat heeft me vaak weer een stapje in de juiste richting geholpen. Doaitse, ook met jou kon ik regelmatig over Proxima van gedachten wisselen. Bedankt voor het koffiezetapparaat, het theeglas en je heldere en kritische blik op mijn ideeën, zelfs al waren dat – zoals je nog wel eens verzuchtte – niet de ideeën die jij oorspronkelijk voor mij in gedachten had.

En ten slotte Lambert: onze langdurige brainstormsessies bij het koffiezetapparaat behoren tot de leukste herinneringen aan mijn promotieonderzoek. Een rode draad was er vrijwel nooit, en eventuele toehoorders zouden conclusies over het al dan niet in chocolatjes kunnen lopen wellicht wat vreemd hebben gevonden, maar deze gesprekken hebben niettemin een zeer grote invloed gehad op de inhoud van dit proefschrift. Bedankt!

I would also like to thank the members of the examination committee (Roland Backhouse, Paul Klint, Arno Siebes, and Masato Takeichi) for reading (and approving) the manuscript, and providing me with several useful comments.

And, continuing in English, a big thank you to Zhenjiang Hu, Shin-Cheng Mu, Masato Takeichi, and the other members of the Programmable Structured Documents group at the University of Tokyo. Your enthusiasm for Proxima, and the prospect of a visit to your research group in Japan after submitting the manuscript have been a wonderful motivation in the last stages of writing this thesis.

In Centrumgebouw Noord werden de dagen altijd op een leuke manier onderbroken door de middag- en andere pauzes met mijn lunchmaatjes Arjan en Daan. Bedankt voor de gezelligheid en de vele interessante discussies en gesprekken, die soms zowaar over de informatica gingen. Ook bedankt alle kamergenoten die ik de afgelopen jaren in CGN voorbij heb zien komen, en in het bijzonder Silja. Leuk dat ik je na drieënhalf jaar nu eindelijk terug kan bedanken! Verder ben ik René van Oostrum zeer erkentelijk voor

het beschikbaar stellen van zijn \LaTeX -stijl, die de basis was voor de vormgeving van dit proefschrift.

Naast al deze informatici ben ik natuurlijk ook mijn familie en vrienden dankbaar. Het zijn er te veel om allemaal te vermelden, maar toch wil ik een aantal mensen even apart noemen. Ten eerste mijn paranimfen Pascal en Jan (Michiel). Jullie hebben het proces van het schrijven van dit proefschrift van dichtbij meegemaakt, en ook al weten jullie volgens mij (nog) niet veel over de inhoud, toch heb ik bij de totstandkoming ervan veel gehad aan jullie vriendschap. Ook Eelke wil ik bedanken, met name voor een bijzonder zinvol gesprekje in het Lepelenpark, op een moment dat ik mijn promotietoekomst even wat minder helder voor ogen had. Bedankt ook Marcel en Mildred voor de gezellige etentjes, en Iris en Tessa, al heb ik jullie met alle drukte natuurlijk veel te weinig gezien de laatste tijd.

Verder zijn er nog veel mensen die me afleiding en steun bezorgd hebben. Bedankt Ingrid, Miriam, Jan-Willem, Evelyne, Carlein, Joep, Cindy, Dirk, Hester, en alle andere vrienden en familieleden die hier niet genoemd staan.

Tot slot is de laatste plek in dit dankwoord voor Joke. Je bent er weliswaar niet bij geweest, maar het doorzettingsvermogen dat ik soms moest aanspreken om dit boekje te schrijven is zonder twijfel van jou afkomstig. En wat zou het mooi geweest zijn als je het resultaat had kunnen zien.

Martijn

Curriculum Vitae

Martijn Michiel Schrage

4 juni 1973

Geboren te Veendam.

augustus 1986 – juni 1991

VWO-Atheneum aan:

Dr. Aletta Jacobs College, Hoogezand. (1985–1988)

Scholengemeenschap Winkler Prins, Veendam. (1988–1989)

H. N. Werkman College, Groningen. (1989–1989)

Niels Stensen College, Utrecht. (1990–1991)

Diploma behaald op 7 juni 1991.

september 1991 – februari 1997

Studie Informatica aan de Universiteit Utrecht.

Doctoraal diploma (cum laude) behaald op 28 februari 1997.

april 1997 – februari 1999

Wetenschappelijk medewerker bij het Skit-project aan het Informatica Instituut van de Universiteit Utrecht.

september 1999 – augustus 2003

Assistent in Opleiding aan het Informatica Instituut van de Universiteit Utrecht.

september 2004 – heden

Docent/wetenschappelijk programmeur aan het Instituut voor Informatica en Informatiekunde van de Universiteit Utrecht.

Titles in the IPA dissertation series

- J.O. Blanco.** *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-01
- A.M. Geerling.** *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-02
- P.M. Achten.** *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-03
- M.G.A. Verhoeven.** *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-04
- M.H.G.K. Kessler.** *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-05
- D. Alstein.** *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-06
- J.H. Hoepman.** *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-07
- H. Doornbos.** *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-08
- D. Turi.** *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-09
- A.M.G. Peeters.** *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10
- N.W.A. Arends.** *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11
- P. Severi de Santiago.** *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12
- D.R. Dams.** *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13
- M.M. Bonsangue.** *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14
- B.L.E. de Fluiter.** *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01
- W.T.M. Kars.** *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02
- P.F. Hoogendijk.** *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03
- T.D.L. Laan.** *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

- C.J. Bloo.** *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05
- J.J. Vereijken.** *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06
- F.A.M. van den Beuken.** *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07
- A.W. Heerink.** *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01
- G. Naumoski and W. Alberts.** *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02
- J. Verriet.** *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03
- J.S.H. van Gageldonk.** *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04
- A.A. Basten.** *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05
- E. Voermans.** *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01
- H. ter Doest.** *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02
- J.P.L. Segers.** *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03
- C.H.M. van Kemenade.** *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, UL. 1999-04
- E.I. Barakova.** *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05
- M.P. Bodlaender.** *Scheduler Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06
- M.A. Reniers.** *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07
- J.P. Warners.** *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08
- J.M.T. Romijn.** *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09
- P.R. D'Argenio.** *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10
- G. Fábíán.** *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11
- J. Zwaneburg.** *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12
- R.S. Venema.** *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13
- J. Saraiva.** *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14
- R. Schiefer.** *Viper, A Visualisation Tool for Parallel Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15
- K.M.M. de Leeuw.** *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01
- T.E.J. Vos.** *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02
- W. Mallon.** *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03
- W.O.D. Griffioen.** *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

- P.H.F.M. Verhoeven.** *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05
- J. Fey.** *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06
- M. Franssen.** *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07
- P.A. Olivier.** *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08
- E. Saaman.** *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10
- M. Jelasity.** *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01
- R. Ahn.** *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02
- M. Huisman.** *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03
- I.M.M.J. Reymen.** *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04
- S.C.C. Blom.** *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05
- R. van Liere.** *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06
- A.G. Engels.** *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07
- J. Hage.** *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08
- M.H. Lamers.** *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09
- T.C. Ruys.** *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10
- D. Chkhaev.** *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11
- M.D. Oostdijk.** *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12
- A.T. Hofkamp.** *Reactive machine control: A simulation approach using χ .* Faculty of Mechanical Engineering, TU/e. 2001-13
- D. Bošnački.** *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14
- M.C. van Wezel.** *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01
- V. Bos and J.J.T. Kleijn.** *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02
- T. Kuipers.** *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03
- S.P. Luttkik.** *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04
- R.J. Willemen.** *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05
- M.I.A. Stoelinga.** *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06
- N. van Vugt.** *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

- A. Fehnker.** *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08
- R. van Stee.** *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09
- D. Tauritz.** *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10
- M.B. van der Zwaag.** *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11
- J.I. den Hartog.** *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12
- L. Moonen.** *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13
- J.I. van Hemert.** *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14
- S. Andova.** *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15
- Y.S. Usenko.** *Linearization in μ CRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16
- J.J.D. Aerts.** *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01
- M. de Jonge.** *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02
- J.M.W. Visser.** *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03
- S.M. Bohte.** *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04
- T.A.C. Willemse.** *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05
- S.V. Nedeia.** *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06
- M.E.M. Lijding.** *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07
- H.P. Benz.** *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08
- D. Distefano.** *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09
- M.H. ter Beek.** *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10
- D.J.P. Leijen.** *The λ Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11
- W.P.A.J. Michiels.** *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01
- G.I. Jojgov.** *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02
- P. Frisco.** *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03
- S. Maneth.** *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04
- Y. Qian.** *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05
- F. Bartels.** *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

- L. Cruz-Filipe.** *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07
- E.H. Gerding.** *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08
- N. Goga.** *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09
- M. Niqui.** *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10
- A. Löh.** *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11
- I.C.M. Flinsenberg.** *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12
- R.J. Bril.** *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13
- J. Pang.** *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14
- F. Alkemade.** *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15
- E.O. Dijk.** *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16
- S.M. Orzan.** *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17
- M.M. Schrage.** *Proxima – A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

