Institut für Informatik
der Technischen Universität München

Specification of Optimizing Document Formatters

Aurel Huber

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Florian Matthes
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Dr.h.c. Jürgen Eickel
	2.	Univ.-Prof. Tobias Nipkow, Ph.D.

Die Dissertation wurde am 20.06.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.10.2007 angenommen.

Acknowledgements

First of all, I thank Prof. Jürgen Eickel for giving me the opportunity to work on this dissertation at his chair. I received valuable comments on draft versions from him which led to significant improvement.

Thanks also go to Prof. Tobias Nipkow who agreed to take part in the dissertation committee. The discussion with him and his review of my work allowed me to improve the dissertation in several places.

I would also like to thank my former colleagues at Prof. Eickel's chair for providing a good working atmosphere, fruitful discussions, and technical help.

Finally, I thank my parents for giving me the opportunity to study Computer Science and for additional support and encouragement.

Abstract

Document processing is a very important application of computers. Most documents for personal and business use are today created and processed using computer systems.

Processing of documents is eased by creating and storing documents in a structured form. Besides that, it is beneficial to separate logical document structures from layout. This allows, for instance, to easily produce multiple layouts of the same document, which has been created in its logical structure, on different output media and in different layout styles. The generic structure and semantic properties of documents can be conveniently specified using attribute grammars.

A document formatter (formatter for short) is the part of a document processing system whose task is to provide a mapping from logical document structures to layout structures. Complex requirements are usually imposed on this mapping: firstly, consistency constraints have to be satisfied concerning, for instance, the placement of floating objects or footnotes; secondly, a formatter is often supposed to optimize certain aspects of layout in order to present a document in the most readable and visually most appealing form to readers.

A formatter that meets such demands to a great extent is hard to implement and maintain using a general-purpose programming language. Languages tailored to the specification of formatters exist, but are limited in their support for formatting constraints and/or optimization.

In this work a new model for optimizing transformations between attributed trees is introduced. The model is based on rules involving constraints on the syntactical structure as well as on semantic properties of source and target trees. The transformation model and its semantics is defined in a completely formal manner. It is described how transformations based on the model can be efficiently executed and the correctness of the resulting execution model w.r.t. the transformation model's semantics is formally proved.

A practical specification language based on the new transformation model is introduced which allows to specify constraint-based optimizing formatters on a high level. It has been implemented in Java and provides, together with a set of tools specific to document processing, a complete system for realizing practical formatters.

The applicability of the new approach is demonstrated by several example formatters that have been successfully realized using the new system. Besides formatters, the new transformation model can be applied in other fields such as user interfaces and code generation.

Contents

1 Introduction **1**
1.1 Documents . . . 1
1.1.1 The structured document model . . . 2
1.1.2 Context-free grammars . . . 3
1.1.3 Attribute grammars . . . 4
1.1.4 Layout structure of documents . . . 4
1.1.5 Separation of logical document structure and layout . . . 5
1.2 Document Formatters . . . 5
1.2.1 Typical tasks of a document formatter . . . 5
1.2.2 Declarative specification of document formatters . . . 6
1.2.3 Generative approach to document formatters . . . 6
1.3 Approach to document formatting taken in this work . . . 6
1.4 Previous approaches to document formatting . . . 7
1.4.1 Solutions to specific formatting problems . . . 7
1.4.2 Transformation languages . . . 8
1.4.3 Specification languages for document formatters . . . 8
1.5 Mathematical notation . . . 8
1.6 Overview over the following chapters . . . 9

2 Examples of formatting problems **11**
2.1 Logical document structure . . . 11
2.2 Layout document structure . . . 12
2.3 Line breaking . . . 13
2.3.1 Problem description . . . 13
2.3.2 An approach to a solution . . . 14
2.4 Page breaking . . . 21
2.4.1 Requirements on page breaking . . . 22
2.4.2 Declarative rules for specifying page layout . . . 23
2.5 Conclusion . . . 24

3 A formal model for optimizing transformations of attributed trees **25**
3.1 Modeling attributed trees and constraints . . . 25
3.1.1 Attributed trees . . . 25
3.1.2 Semantic constraints . . . 30
3.2 Specification of optimizing transformations of attributed trees . . . 31
3.2.1 Specification of relations between attributed trees . . . 31

3.2.2 Semantics of a specification . . . 35
3.2.3 Optimizing transformation specifications . . . 39
3.3 Efficient execution of transformations . . . 42
3.3.1 Construction of target trees . . . 42
3.3.2 Tree fragments and their attribution . . . 44
3.3.3 Attribute dependencies in templates . . . 49
3.3.4 LR specifications . . . 50
3.3.5 An execution scheme for LR specifications . . . 53
3.3.6 Soundness of the execution scheme . . . 58
3.3.7 Completeness of the execution scheme . . . 60
3.3.8 Optimization . . . 64
3.3.9 Application of dynamic programming . . . 66
3.3.10 Automatic construction of LR specifications . . . 69
3.4 Comparison with existing approaches . . . 70
3.5 Conclusion . . . 74

4 The Coala system 77
4.1 Coala's specification language . . . 77
4.1.1 Syntax notation . . . 78
4.1.2 Lexical structure . . . 78
4.1.3 Basic language elements . . . 80
4.1.4 Expressions . . . 81
4.1.5 User-defined semantic functions . . . 84
4.1.6 Structure of specifications . . . 85
4.1.7 Inclusion of specifications . . . 85
4.1.8 Specifying attribute grammars . . . 85
4.1.9 Semantics . . . 87
4.1.10 Attribute evaluation strategy . . . 88
4.1.11 Transformation rules . . . 88
4.1.12 Example specification . . . 90
4.2 The Coala runtime system . . . 91
4.2.1 Loading and interpretation of Coala specifications . . . 92
4.2.2 Function libraries . . . 93
4.2.3 A simple page layout format with PostScript language output . . . 96
4.3 Conclusion . . . 97

5 Example applications 99
5.1 Line breaking . . . 99
5.1.1 Global optimization . . . 99
5.1.2 Variants . . . 102
5.1.3 Example runs . . . 103
5.2 A page formatter . . . 103
5.2.1 Area model . . . 106
5.2.2 Managing cross references . . . 112
5.2.3 Transformation rules . . . 115
5.2.4 Example run . . . 118

5.3 Ideas for further applications . . . 118
5.3.1 Code generation optimizing order of execution . . . 118
5.3.2 User interface layout . . . 122

6 Related Work 123
6.1 ODA . . . 123
6.1.1 ODA's document model . . . 123
6.1.2 Formatting of documents . . . 124
6.1.3 Comparison with Coala . . . 124
6.2 SGML/DSSSL, XML/XSL . . . 124
6.2.1 SGML's and XML's document model . . . 125
6.2.2 Flow object trees . . . 125
6.2.3 Transformation to target structures . . . 126
6.2.4 Comparison with Coala . . . 126
6.3 Agenda . . . 127
6.4 (Constraint) Logic Programming . . . 128
6.4.1 Logic programs . . . 128
6.4.2 Constraint logic programs . . . 130
6.5 Conclusion . . . 130

7 Summary and outlook 133
7.1 Summary . . . 133
7.2 Contributions . . . 134
7.3 Possible extensions and improvements . . . 135

References 137

A Glossary 141

B Missing proofs from Chapter 3 145
B.1 Auxiliary lemmata . . . 145
B.2 Proofs from Section 3.3.2 . . . 152
B.3 Proofs from Section 3.3.5 . . . 153
B.4 Proofs from Section 3.3.6 . . . 153
B.5 Proofs from Section 3.3.7 . . . 155

C Extensions to attribute grammars and their semantics 159
C.1 Context-free grammars . . . 159
C.2 Attributes . . . 160
C.3 Attribution rules . . . 160
C.4 Initialization of synthesized attributes at terminal nodes . . . 161
C.5 Attribute grammars . . . 161
C.6 Semantics . . . 161
C.6.1 Syntax trees . . . 161
C.6.2 Tree attribute equations . . . 161
C.6.3 Consistent attributions . . . 162
C.6.4 Attribute dependencies . . . 162

C.6.5 Monotonicity property 162
C.7 Construction of an attribute grammar from a Coala specification 165
C.7.1 Context-free grammar 165
C.7.2 Attributes 166
C.7.3 Attribution rules 166

D Summary of Coala syntax **167**

E Haskell implementation **171**

CHAPTER 1

Introduction

Document processing has become an increasingly important use of computers over the last years, especially of personal computers. Most documents (for both business and private use) are today created, stored, and processed electronically.

As shown in the literature [8, 41, 42, 43, 55], processing of documents greatly benefits from organizing documents in a structured way and keeping logical document structure and content separated from its layout.

A part of a document processing system which is important for realizing this separation is the document formatter whose task is to create layout representations from logical document structures. Complex requirements are usually imposed on formatters as their result must firstly satisfy often difficult to handle consistency constraints and secondly be as visually pleasing as possible. This makes implementing formatters by hand using a general-purpose programming language very difficult and error-prone.

In this work the novel approach of applying *constraint optimization* to the specification of document formatters is taken, i.e., formatters are described using highly descriptive executable specifications which are concise and easy to create and maintain.

1.1 Documents

In the scope of electronic document processing the term **document** usually denotes data representing some information, stored persistently in a computer system. A document's data can consist for instance in textual data, graphical data, or multimedial data such as sound and movie clips.

Documents are usually—besides being stored in computer systems— processed in various ways, including editing, printing, online viewing/browsing, archiving and transforming. A collection of tools supporting such tasks is called a **document processing**

system.

Very often, many documents of the same or of a similar kind exist, for example conference papers or technical reports. In order to be able to process such documents using a single system, they are most often grouped into **document class**es which formalize important common characteristics, including the generic structure, for instance.

In the following, the generally accepted [8, 41, 42, 43, 55] structured document model and well-known techniques of specifying document classes based on that model are introduced at an informal level.

1.1.1 The structured document model

Documents were originally created and processed in an unstructured or weakly structured way. For example, a document created with a usual word-processing application was (and often still is) just structured into a sequence of paragraphs, which themselves consist in a sequence of characters.

In contrast, with the **structured document model**, a document is seen as a hierarchical tree structure of elements. Figure 1.1 shows, as a typical example, a report document which is structured into chapters, sections, and paragraphs.

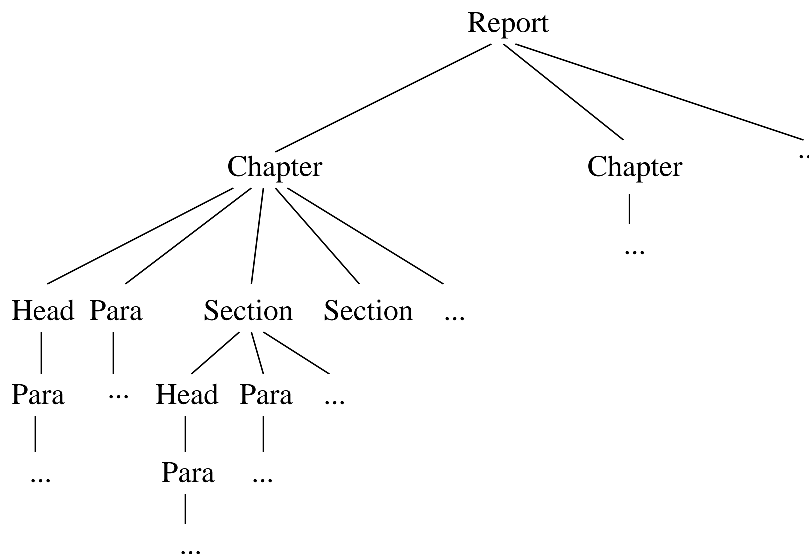


Figure 1.1: Example of a structured document

The structured document model became popular in the 1980s with `SGML` (Standardized Markup Language) [12, 18] and ODA [19]. The most popular system in use today is `XML` [45].

Some of the benefits of using the structured document model are:

- editing of large documents is eased as quick navigation within the document is possible

- reuse of parts of a document, e.g., for assembling and reconstructing documents is made easier (see for example [15]); a general approach for creating documents using highly reusable units is realized with `DITA` [6]
- some processing tasks like automatic creation of tables of contents or numbering of sections are greatly simplified

1.1.2 Context-free grammars

Individual structured documents can be represented using structure trees. When dealing with document classes, however, the *generic* structure of documents of a given class needs to be specified. A **context-free grammar** can be used for this purpose.

A context-free grammar formally describes the syntactic structure of a language using grammar symbols and productions describing their syntactic structure. A context-free grammar can be used to specify the *concrete* syntax of a language, and a parser mapping a sequence of input tokens to a structure tree can be derived automatically. However, it is also possible to specify the *abstract* syntax of a language using a context-free grammar by dropping all delimiters needed for disambiguating the grammar. The latter approach will result in a grammar which cannot be used for parsing, but specifies the structure of a language in a way that is perfectly suitable for processing. Abstract syntax is usually used in compiler construction, but Figure 1.2, which gives a grammar for the report document type described above, shows that it can be used for describing the generic structure of documents as well. Note that context-free grammars require sequences to be specified using recursive productions while sequences are shown in Figure 1.1 on the facing page in a flat fashion.

$$
\begin{array}{lcl}
\mathit{Report} & ::= & \mathit{ParaSeq}\ \mathit{ChapterSeq} \\
\mathit{ChapterSeq} & ::= & \varepsilon \\
 & | & \mathit{Chapter}\ \mathit{ChapterSeq} \\
\mathit{Chapter} & ::= & \mathit{Head}\ \mathit{ParaSeq}\ \mathit{SectionSeq} \\
\mathit{SectionSeq} & ::= & \varepsilon \\
 & | & \mathit{Section}\ \mathit{SectionSeq} \\
\mathit{Section} & ::= & \mathit{Head}\ \mathit{ParaSeq} \\
\mathit{Head} & ::= & \mathit{Para} \\
\mathit{ParaSeq} & ::= & \varepsilon \\
 & | & \mathit{Para}\ \mathit{ParaSeq} \\
\mathit{Para} & ::= & \ldots
\end{array}
$$

Figure 1.2: Partial context-free grammar for document class *Report*

Context-free grammars are also used in `SGML` and `XML` for the specification of the generic structure of documents. Document Type Definitions (DTDs) and XML schemata [47] there serve for this purpose. With `SGML` and `XML`, concrete syntax is simply derived by enclosing each nonterminal element in start and end tags. The resulting concrete language can be parsed very easily.

1.1.3 Attribute grammars

While context-free grammars can be used to specify the generic *syntactic* structure of documents, they cannot be used to describe *semantic* properties of documents.

Attribute grammars, which were introduced in [26] and are often used in the field of compiler construction, provide a means to fill this gap, as they allow to enrich syntax trees with semantic information.

An attribute grammar associates attributes with symbols of a context-free grammar and defines equations for each production, describing how attributes are computed.

When dealing with documents, one can take advantage of attribute grammars as well. Many problems of computing semantic information about documents can be specified using attribute grammars. Well known examples include the construction of tables of contents and section numbering [8, 9].

1.1.4 Layout structure of documents

The layout of a document is a visual representation of the document which can be output on a printer or computer screen.

We have already seen that the structured document model can be applied to *logical* document structures. However, it is also an appropriate means to naturally describe the *layout* of documents [8, 9]. Figure 1.3 indicates how a possible concrete layout might be structured.

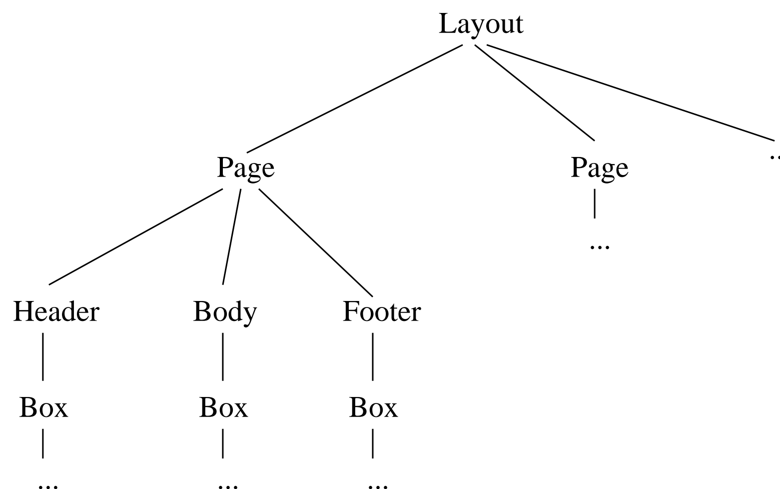


Figure 1.3: Example of a structured document layout

The structured document model for layout can be found in practice in widely used standards like `Portable Document Format` (`PDF`) [1] or `Scalable Vector Graphics` (`SVG`) [44].

Attribute grammars can again be used to associate semantic information with layout structure [8, 9]. Page numbering is a simple example; another example is the computation of geometric properties of boxes, like position in a page's coordinate system, or box dimensions.

1.1.5 Separation of logical document structure and layout

We have seen that both logical content of documents and document layout can be suitably represented using tree structures. It is, however, also very important that logical structure and layout always be kept in *separate* structures. Creating a document in its purely logical structure and keeping it independent from layout allows for instance to easily recreate a document in different layout styles and formats and for different output media (e.g., on paper as well as online).

1.2 Document Formatters

Separating logical structure and layout of documents requires the existence of components in a document processing system which map a given logical document structure to layouts. Such a component is usually called a **document formatter** (or formatter for short) [33, 41, 42, 43]. There are two basic requirements on the mapping provided by a formatter [33]:

- the layouts which are produced are *faithful to the author's intentions*
- layouts are *aesthetically pleasing.*

As a formatter does the job which formerly a human typesetter did, it should be driven by a set of (formalized) rules which a human typesetter would use to craft the same layout. Thus, a formatter's functionality can be imagined as is depicted in Figure 1.4.

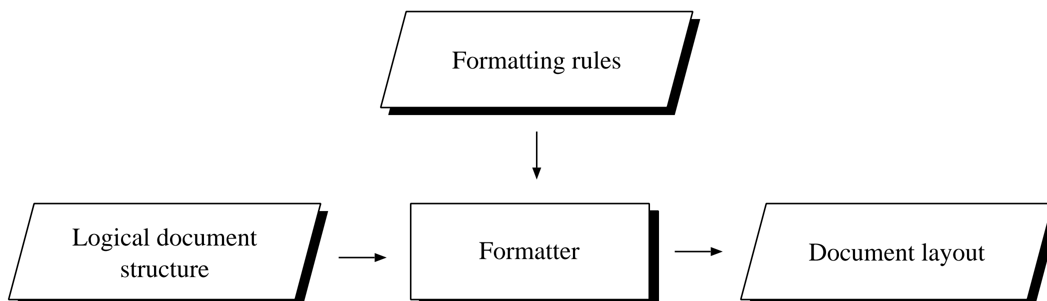


Figure 1.4: Functionality of a formatter

1.2.1 Typical tasks of a document formatter

A formatter constructs layouts from basic objects with certain graphic characteristics and arranges those objects on the output medium. The constructed layout must satisfy certain consistency constraints in order to fulfill the two main requirements of a formatter listed above.

On the one hand, for a layout to be *faithful to the author's intentions*, constraints like preservation of the order of certain elements or the usage of fonts must be fulfilled in order to communicate the structure of the document correctly. An example of more complex constraints is the placement of floating objects, which is described, e.g., in [33]:

a floating object may typically not appear on a page (or page spread) before its first or main reference in the text.

On the other hand, further constraints have to be satisfied for a layout in order to be *aesthetically pleasing*. For example, pages should neither be underfull nor overfull.

The constraints a formatter must fulfill usually still allow some freedom. So, another desired feature of a formatter is to choose the *optimal* consistent document layout w.r.t. some measure. A well-known example for this is optimal line breaking [9, 27, 55] where all words contained in a paragraph are distributed as evenly as possible in the resulting sequence of lines, leaving as little gaps as possible between words. Another example is optimal page breaking; [5, 36] describe for example the optimization problem that floating objects should be placed as near as possible to their main reference in the text.

1.2.2 Declarative specification of document formatters

A formatter should ideally be driven by rules which are specified in a highly declarative manner. An appropriate way to specify rules in such a declarative manner is by using **constraints**, i.e., logical formulas describing desired characteristics of a solution instead of an algorithm to compute it. Constraints have been used in various fields described, e.g., in [10, 30, 31].

1.2.3 Generative approach to document formatters

Using a declarative specification technique with formal semantics allows a special approach to be taken: the program implementing a given specification can be automatically *generated* from a specification. This approach is well-known in the area of compiler construction where, e.g., lexical, syntactic, and semantic analyzers are usually generated from formal specifications [2, 56].

The main benefit of this approach is that the implementation phase of the software engineering process can be completely automated, which causes an important shortening of software development time and significantly eases maintenance of the software.

This generative approach can be applied to the development of document formatters which has previously been shown [43]. The resulting architecture is depicted in Figure 1.5 on the next page.

1.3 Approach to document formatting taken in this work

As has been pointed out, logical document content as well as layout can be modeled using attribute grammars, and we wish to be able to declaratively specify and generate optimizing formatters, i.e., mappings from logical document structures to layout structures.

That means that our actual task is to describe *optimizing transformations from attributed trees to other attributed trees*. The new approach taken in this work is using constraints, so our model for specifying document formatters can be concisely characterized as

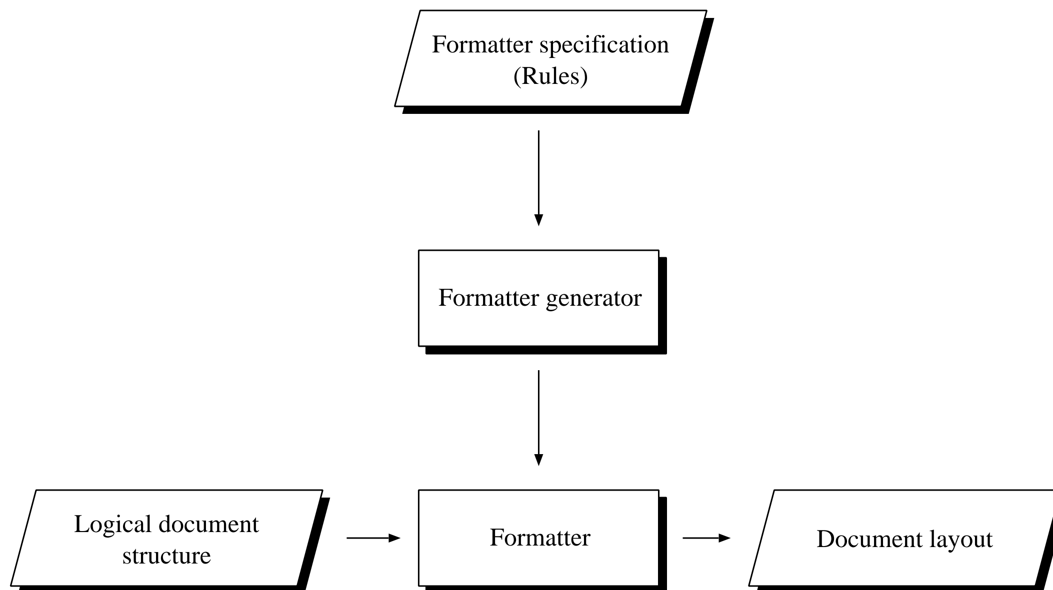


Figure 1.5: Generating formatters from formal specifications: architecture overview

Constraint-based specification of optimizing transformations of attributed trees.

The general idea of the constraint-based approach is the following: in order to specify a formatting (sub-) problem, a *relation* that relates source document structures to a set of consistent target structures is defined. This is accomplished by a set of rules, where constraints are used to specify consistency (pertaining to syntactic as well as semantic properties). An *objective function* defines a measure for which is to be optimized, i.e., the relation describing possible consistent target structures is further constrained to contain only *optimal* results according to the desired measure.

As will be shown in this work, the new approach allows to specify formatting problems in a very natural way. Problems which were hard to tackle with existing methods and systems (see below) can be declaratively described.

1.4 Previous approaches to document formatting

The approach to document formatting described above in Section 1.3 has not been taken before. In the following, characteristics and shortcomings of the most prominent representatives of existing approaches to document formatting are discussed which are meant to be leviated by the new approach. Note that a more elaborate discussion will be given in Chapter 6.

1.4.1 Solutions to specific formatting problems

Several methods for solving specific formatting problems have been developed and have been implemented in general-purpose programming languages. Popular examples are

optimal line breaking [27, 35] and optimal page breaking [5, 54].

Using general-purpose languages ist is very difficult to create complex formatters. The resulting implementations are hard to read and maintain. It would also be difficult to extend or modify the behavior of a formatter created in this manner to suit the needs that may arise with new document processing systems.

1.4.2 Transformation languages

As it would be difficult and tedious to implement special-purpose formatters using the approach described above in Section 1.4.1 for each new document processing system that has to be developed, the most popular method used today is to first transform logical documents to a widely usable intermediate language; this first transformation is relatively simple to implement. Fixed programs are then used to transform from this intermediate format to the actual layout format solving the complex formatting tasks like line and page breaking.

An example of such an intermediate language widely used today is `XSL-FO` (*XSL formatting objects*) [48], for which transformers are available which perform the actual formatting tasks and produce various output formats like `HTML` and `PDF`.

For the transformation from logical structures to the intermediate structure a declarative transformation language is often used, for example `DSSSL` (see for example [37]) or `XSLT` [49].

This approach works well in practice. However, it has the limitation that formatting tasks beyond those expressible in the fixed intermediate language cannot be solved.

1.4.3 Specification languages for document formatters

Some special-purpose languages for implementing document formatters have been developed. An example is the `Lout` document preparation system [25], which allows a user to specify formatters for structured documents on a relatively high level. It is based on a lazily evaluated functional language extended for special concepts for cases that exceed simple bottom-up construction of layout (e.g., placement of floating objects).

The `Agenda` system [43] allows to specify source and target structure of a formatter using attribute grammars and the formatting process itself using a stream-based model [34].

Both `Lout` and `Agenda`, and other systems of this kind, provide a means to declaratively specify document formatters. However, declarativity is provided only at the price of some limitations:

- both systems do not allow to specify general formatting constraints, and thus make some formatting tasks quite hard or impossible to specify
- while `Lout` offers a limited form of optimization, an optimizing formatter cannot be specified using `Agenda` at all

1.5 Mathematical notation

The following introduces the mathematical notation used in this work.

Sets. The set of natural numbers is denoted by $\mathbf{Nat} =_{def} \{1, 2, \ldots\}$. The set of natural numbers including zero is denoted by $\mathbf{Nat_0} =_{def} \{0\} \cup \mathbf{Nat}$. $\mathbf{Int}$ denotes the set of all integer numbers $\mathbf{Nat_0} \cup \{-n \mid n \in \mathbf{Nat}\}$. We will write $min(M)$ for the minimum of a non-empty, finite set $M \subseteq \mathbf{Int}$. The empty set is given by $\emptyset$. The power set of a set M is denoted by $\mathcal{P}ow(M)$. The union of two disjoint sets M_1 and M_2 is denoted by $M_1 \dot\cup M_2$.

Sequences. Let M be a set. A sequence over M is denoted by $\langle m_1 m_2 \ldots m_n \rangle$ where $n \in \mathbf{Nat_0}$ and $m_i \in M$ for each $1 \leq i \leq n$. If no doubt exists that we refer to a list we will also shortly write $m_1 m_2 \ldots m_n$. We will further write ε for the empty sequence $\langle\rangle$. The set of sequences of fixed length n over M is denoted by M^n. The set of all sequences of arbitrary finite length over M is defined as $M^* =_{def} \bigcup_{n \in \mathbf{Nat_0}} M^n$. Given two sequences $\bar{m} = m_1 m_2 \ldots m_n$ and $\bar{m}' = m'_1 m'_2 \ldots m'_{n'}$, $\bar{m} \circ \bar{m}' =_{def} m_1 m_2 \ldots m_n m'_1 m'_2 \ldots m'_{n'}$ denotes the concatenation of both sequences. A sequence $\bar{m}_1$ is called a **prefix** of a sequence $\bar{m}_2$ iff a sequence $\bar{m}_3$ exists such that $\bar{m}_1 \circ \bar{m}_3 = \bar{m}_2$.

Functions. Given two sets X and Y, $X \to Y$ denotes the set of functions from X to Y. For functions f from the set $X \to Y$ we write $f\colon X \to Y$. If $f\colon X \to Y$ and $M \subseteq X$, then we write $f|_M$ for the restriction of f to M, for which the following holds: $f|_M\colon M \to Y$ and $f|_M(m) = f(m)$ for all $m \in M$.

1.6 Overview over the following chapters

The following chapter will present realistic examples of document formatting problems, and a new specification technique based on constraint optimization is motivated.

A formal model for optimizing transformations of attributed trees is developed in Chapter 3, including a formal semantics. Methods for efficient execution of transformations based on the model are developed and proved to be correct.

Chapter 4 describes a new system called Coala providing a specification language based on the theoretical transformation model introduced in Chapter 3, and its runtime system, suitable for realizing practical document formatters.

In Chapter 5 several example applications are shown that have been realized using the new system.

Chapter 6 describes previous work in this area in comparison with the new specification technique.

Chapter 7 gives a summary of the achievements of this dissertation and a brief outlook to future work.

The appendix contains a glossary of terms used in the field of document processing, technical proofs that have been omitted in Chapter 3, a precise description of the slightly extended attribute grammar model used in this work, a summary of the context-free syntax of Coala's specification language, and a functional implementation of the theoretical transformation model.

CHAPTER 2

Examples of formatting problems

In this chapter some basic formatting problems are addressed, showing how logical and layout document structures can be modeled using attribute grammars and how optimizing transformations between logical and layout document structures can be specified in a natural and declarative way. The examples deal with the formatting of documents containing text structured into paragraphs and sections, and are as such classic formatting problems. The target of transformations is a hierarchical box model as the one known from the TEX system [27].

The ideas gained from these examples, which are treated in this chapter at an informal level, will build the basis for the formal document transformation model developed in Chapter 3.

2.1 Logical document structure

The source structure (logical document structure) of our example formatting application is defined by the following context-free grammar giving the abstract syntax of documents structured into sections and paragraphs.

Document	::=	*SectionSeq*
SectionSeq	::=	*Section*
	|	*Section SectionSeq*
Section	::=	*Header ParagraphSeq*
Heading	::=	*Paragraph*
ParagraphSeq	::=	*Paragraph*
	|	*Paragraph ParagraphSeq*
Paragraph	::=	*Word*
	|	*Word Paragraph*
Word	::=	*text*

2.2 Layout document structure

The following context-free grammar defines the abstract syntax of a simple page description language used as the target structure of transformations.

Layout	::=	*PageSeq*	
PageSeq	::=	ε	
	|	*Page PageSeq*	
Page	::=	*Box*	
Box	::=	*HBox* | *VBox*	
	|	*TextBox* | *EmptyBox*	
HBox	::=	*Box Box*	
VBox	::=	*Box Box*	
TextBox	::=	*Str*	
EmptyBox	::=	*Int Int Int*	(* *width height depth* *)

Layout is defined as a sequence of pages, where page content is described using a hierarchy of boxes, terminal elements being *TextBox* and *EmptyBox*; boxes can be composed horizontally (*HBox*) or vertically (*VBox*).

The box geometry shown in Figure 2.1 on the facing page is used, where the dimensions of each box are described by three attributes *width*, *height* and *depth*. Each box has a reference point used for the alignment of *HBox*es and *VBox*es. The geometry of those compound boxes is shown in Figure 2.2 on the next page. The box geometry used here is very close to the one used in TEX [27].

An attribute grammar can describe box geometry in a natural way. Figure 2.3 on page 14 shows the attribution of layout structure necessary to compute the dimensions of compound boxes. In Figure 2.4 on page 15 the computation of the coordinates of the reference point of each box is given. The content of a page is placed in a way such that the top left corner of the box is at the origin of the coordinate system, which lies at the upper left corner of the page.

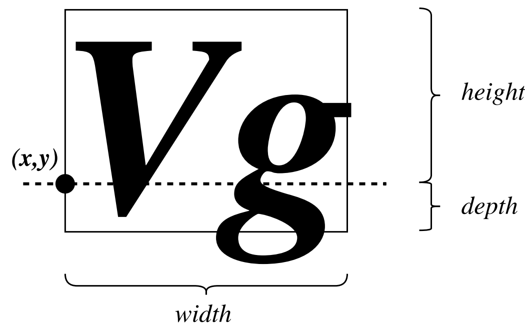


Figure 2.1: Geometry of boxes (in this case *TextBox*es)

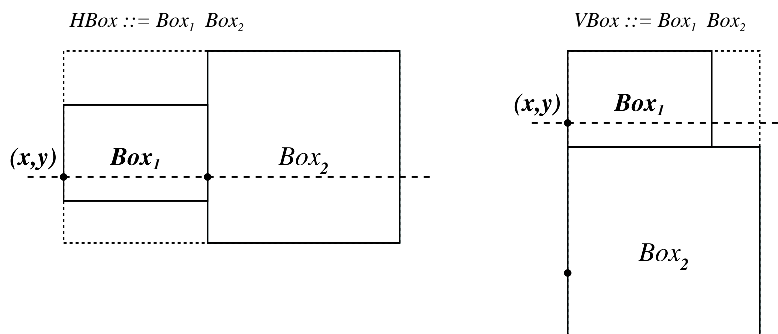


Figure 2.2: Geometry of *HBox*es and *VBox*es

2.3 Line breaking

2.3.1 Problem description

We will first discuss the classic problem of (optimal) line breaking. Solutions to optimal line breaking have already been built into prominent document preparation systems such as TEX or Lout and have been described in [9, 35]. However, these solutions have up to now been described only in an operational manner and deal only with flat document structures. The highly declarative specification of line breaking is still an open question and interesting to analyze when searching for a more general solution to the specification of optimizing document formatting problems.

The (slightly simplified[1]) task of line breaking is to arrange the words contained in a paragraph among justified lines of a given width. Points where lines are allowed to be broken lie between any two adjacent words. In order to allow lines to extend to the desired width of the paragraph, white space of variable width is inserted between words. However, to avoid underfull and overfull lines, whitespace can be shrunken or stretched only within given limits.

[1]We omit the handling of hyphenation

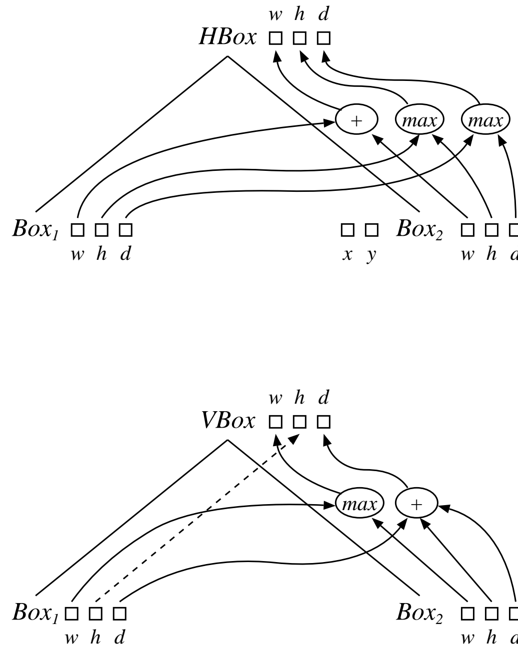


Figure 2.3: Computation of the dimensions of *HBox*es and *VBox*es

The description of line breaking so far will in general allow more than one possible solution; so the actual result of line breaking should be the one that is optimal with respect to some measure. The following approaches are described in [35]:

- break each line as early as possible (stretching white space as much as possible)
- break each line as late as possible (shrinking white space as much as possible)
- always choose the next line such that its white space is stretched or shrunken as little as possible (locally optimal, also called 'best fit')
- choose all lines such that the white space of all lines is stretched or shrunken as little as possible (globally optimal)

The last approach—which we will take in the following—is the most interesting. Because it might occur that choosing a locally optimal line leaves a rest paragraph that can be only broken into very bad subsequent lines it is not possible to produce the resulting lines one after another. Instead, all consistent complete sequences of lines have to be constructed and the best one has to be selected among them. This makes this approach very hard to implement in a general-purpose programming language.

2.3.2 An approach to a solution

Attribute grammar describing sequences of lines

White space of variable width can be modeled using the concept of *glue*. The valid range of width of a glue is defined by three parameters: the natural width (*nat*), the amount by

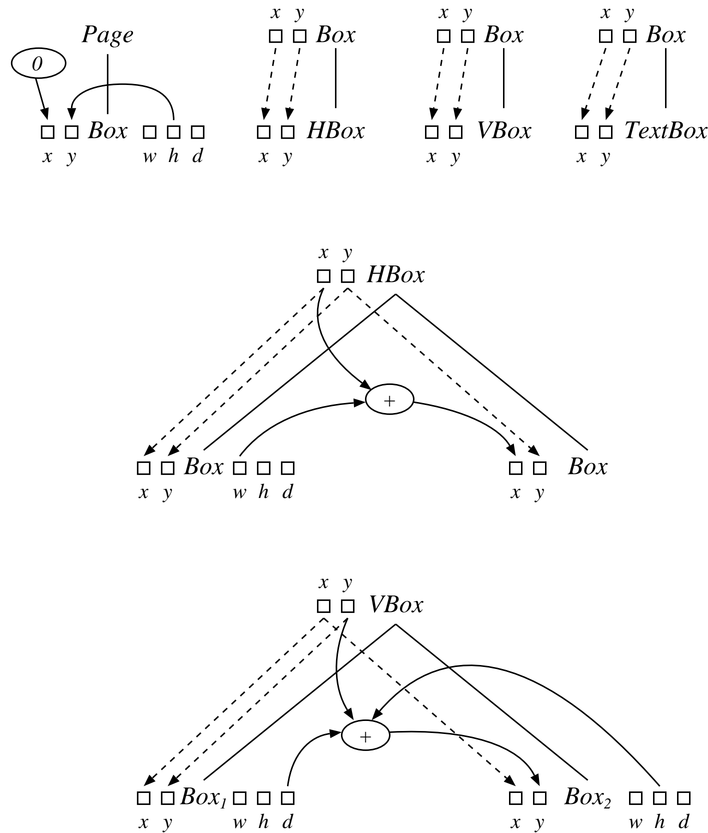


Figure 2.4: Computation of reference point coordinates

which it can be shrunken (*minus*) and the amount by which it can be stretched (*plus*). The range of valid widths is then $[nat - minus, nat + plus]$.

As our target document structure provides no means to represent lines containing glue, we will first introduce an intermediate structure *LineSeq* for this purpose:

LineSeq	::=	ε	
	|	*Line LineSeq*	
Line	::=	ε	
	|	*Box Line*	
	|	*Glue Line*	
Glue	::=	*Int Int Int*	(* *nat plus minus* *)

A *LineSeq* can later be easily transformed to our target structure where each line becomes an *HBox* containing *EmptyBox*es for glues, when the actual width of glues is known.

The computation of the actual width of each glue element can be described using the attribute grammar depicted in Figure 2.5 on page 17. In this attribute grammar, each glue element of a line is stretched or shrunken according to an *adjustment factor*

α which is computed from the sums of natural widths, stretchability and shrinkability of all the elements of the line and the constant *lineWidth* by a function f, defined as follows:

$$f(Nat, Plus, Minus) \quad =_{def}$$

$$\begin{cases} 0 & \text{if} Nat = lineWidth \\ \frac{lineWidth - Nat}{Plus} & \text{if} Nat < lineWidth \ \wedge \ Plus \neq 0 \\ \frac{lineWidth - Nat}{Minus} & \text{if} Nat > lineWidth \ \wedge \ Minus \neq 0 \\ \infty & \text{if} Nat < lineWidth \ \wedge \ Plus = 0 \\ -\infty & \text{if} Nat > lineWidth \ \wedge \ Minus = 0 \end{cases}$$

Using the adjustment factor, the actual width of each glue is computed using the function g, defined so:

$$g(\alpha, nat, plus, minus) \quad =_{def} \quad \begin{cases} nat + \alpha \times plus & \text{if} \quad \alpha \geq 0 \\ nat + \alpha \times minus & \text{otherwise.} \end{cases}$$

If the width of all glue elements of a line is computed like this a justfied line of width *lineWidth* results.

Box kerf sequences

The core of the line breaking problem is to transform a sequence of word boxes into a sequence of lines where breaks occur at allowable points (in our case between adjacent words). Allowable points can be represented by special elements called *kerfs*, so the input to the line breaking process is actually a sequence of boxes and kerfs. We will use *BoxKerfSeq* structures, defined as follows, to represent such sequences.

BoxKerfSeq	::=	ε
	$\vert$	*Box BoxKerfSeq*
	$\vert$	*Kerf BoxKerfSeq*
Kerf	::=	*Glue*

*Kerf*s contain a glue element which is used as an inter-word space if a line is not broken at that kerf[2]. If lines are actually broken at some kerf element, it will simply vanish.

[2]Note that in a setting with hyphenation, a more complicated notion of *Kerf* supporting the insertion of hyphen characters can be used.

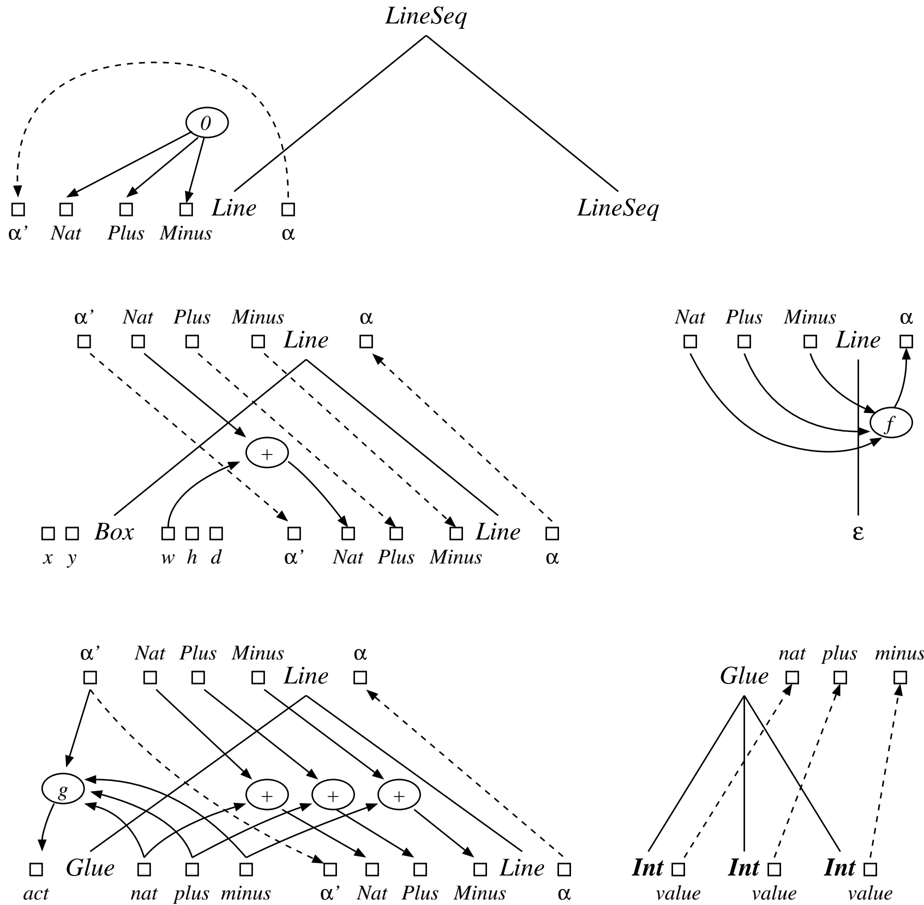


Figure 2.5: Computation of actual glue widths

Specification of a transformation to consistent sequences of lines

Having introduced all the necessary structures, we can now address the actual transformation task of line breaking. The transformation of our source document structure (*Paragraph*) to *BoxKerfSeq*, simply inserting kerf elements between adjacent words, is straightforward and rather uninteresting, so we will assume that we already have a transformation function *paragraphToBoxKerfSeq*: *Paragraph* → *BoxKerfSeq* available which performs this task. The more interesting part is the transformation *BoxKerfSeq* → *LineSeq* which can be specified as shown in the following. We omit optimization at first and only describe the construction of *consistent* sequences of lines.

An overview of the transformation is shown in Figure 2.6 on the next page. A single sequence of lines is constructed by choosing a consistent first line leaving a rest box kerf sequence, and applying the same process recursively until the rest box kerf sequence is empty. The specification of how a single line is to be constructed will describe the structure of *all* consistent lines, so the whole line breaking process will result in all consistent sequences of lines.

An idea to describe the construction of a single line is using rules consisting of tree

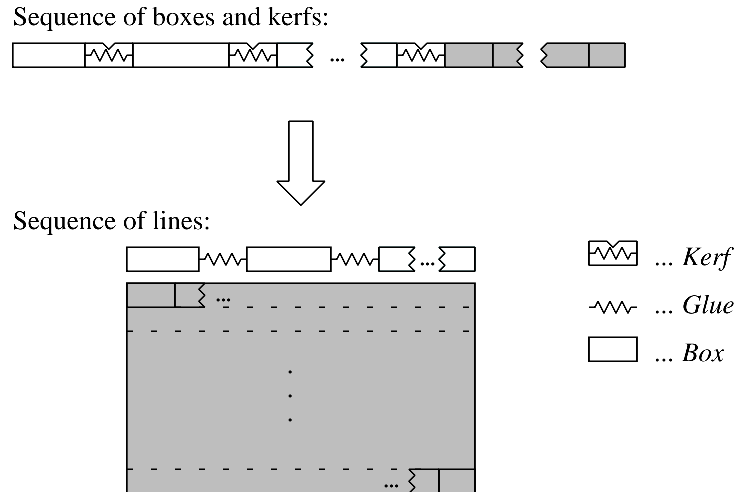
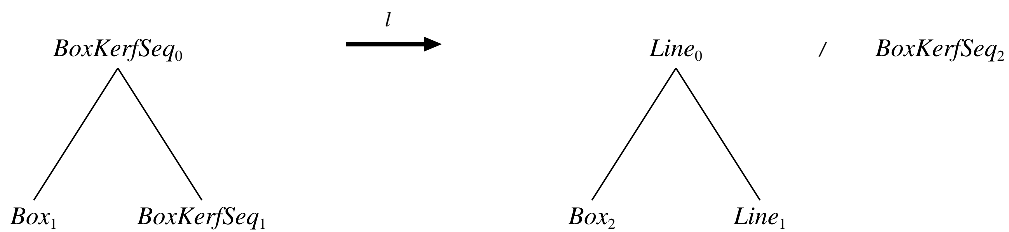


Figure 2.6: Constructing a sequence of lines from a sequence of boxes and kerfs

templates and constraints. Three rules are necessary to describe the transformation of a box kerf sequence depending on the structure of the source box kerf sequence.

The case where the source box kerf sequence starts with a box can be handled as is shown in Figure 2.7. Here, the box is appended to the current line and the transformation proceeds with the sub box kerf sequence.



where $\quad$ *copy:* $Box_1 \rightarrow Box_2$
l*:* $BoxKerfSeq_1 \rightarrow Line_1 \,/\, BoxKerfSeq_2$

Figure 2.7: Processing of a box

If the source box kerf sequence starts with a kerf the current line can either be broken or not broken, in the latter case appending the inter-word glue to the current line. These rules are shown in Figure 2.8 on the next page and Figure 2.9 on the facing page.

The rules consist of:

- a tree template matching the source box kerf sequence structure

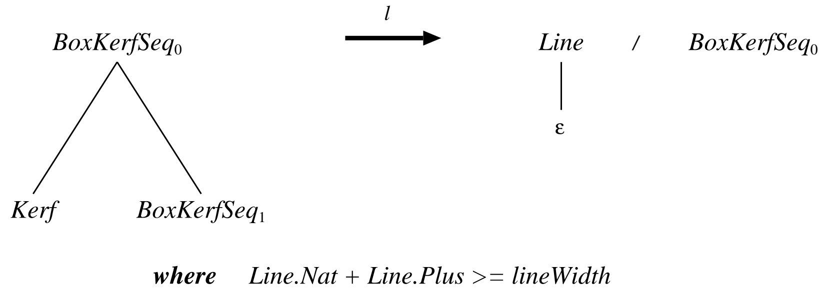


Figure 2.8: Processing of a kerf breaking the current line

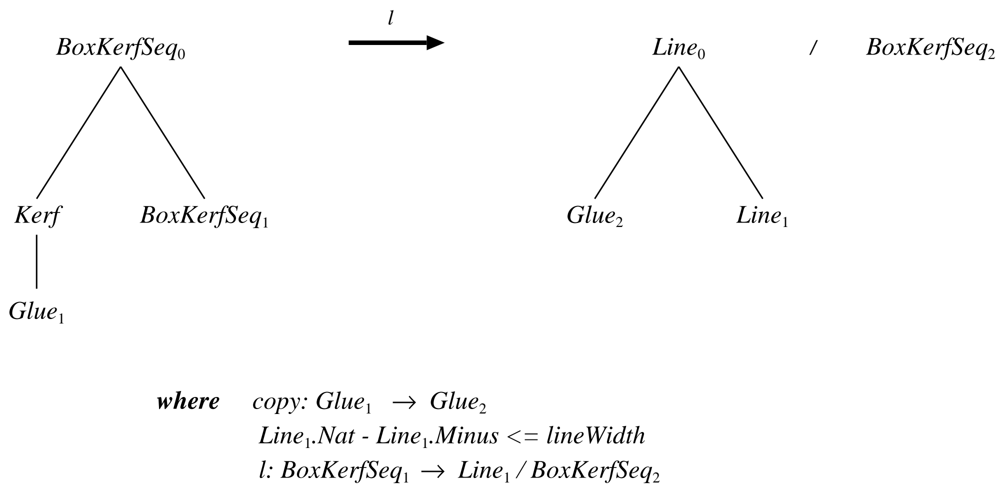


Figure 2.9: Processing of a kerf not breaking the current line

- a tree template describing the structure of the target line structure
- a variable giving the rest box kerf sequence remaining after applying the rule
- constraints describing necessary sub transformations and conditions on attribute occurrences that must be satisfied for the resulting line to be neither underfull nor overfull

The rules describe a transformation to *all* possible consistent lines that can be constructed from the beginning of a box kerf sequence. Note here that two rules can be applied to a rest box kerf sequence beginning with a kerf as long as the resulting lines are consistent. Note also that the existence of built-in transformations for structurally copying boxes and glue from the source structure to the target structure is assumed.

We are now ready to describe the complete line breaking process as a transformation ls. For producing a consistent sequence of resulting lines, this transformation uses l as a sub transformation to create line after line until the remaining source box kerf sequence is empty. Notice again that *all* consistent sequences of lines result, as l describes a

relation to all consistent single lines. The transformation *ls* can be specified using rules such as shown in Figure 2.10.

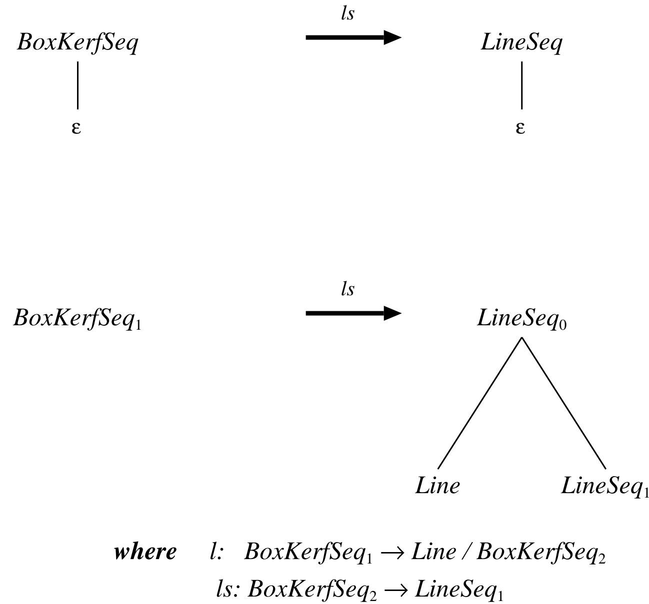


Figure 2.10: Construction of the target line sequence

Specification of optimization

The specification as discussed so far defines a transformation relation *ls* which relates a given source box kerf sequence to all consistent sequences of lines. In order to find the *optimal* possible line break, we can now constrain this relation by introducing a measure of *cost* (or *badness*) associated with sequences of lines and then choosing the optimal result w.r.t. this measure. The appearance of a line is best when the contained glue elements' widths are as close as possible to their natural widths, i.e., the least amount of stretching or shrinking is required. Thus, the amount of necessary stretching or shrinking can serve as a measure of cost for a line. The average cost of all lines can then serve as a measure of cost for a sequence of lines. The computation of cost can again be specified using an attribute grammar whose rules are shown in Figure 2.11 on the facing page. The function $cost : \mathbf{Int} \times \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int} \cup \{\infty\}$ used there is defined as follows:

$$cost(Nat, Plus, Minus) = \begin{cases} \dfrac{lineWidth - Nat}{Plus} & \text{if } Nat \leq lineWidth \leq Nat + Plus \\ \dfrac{Nat - lineWidth}{Minus} & \text{if } Nat - Minus \leq lineWidth < Nat \\ \infty & \text{otherwise} \end{cases}$$

This is a simplified variant of the measure of cost used in the TEX system [28]. Note

that, as desired, cost is 0 (i.e., minimal) when no stretching or shrinking is necessary and cost rises with the required amount of stretching and shrinking. The cost is infinite if glues are stretched or shrunken beyond their limits.

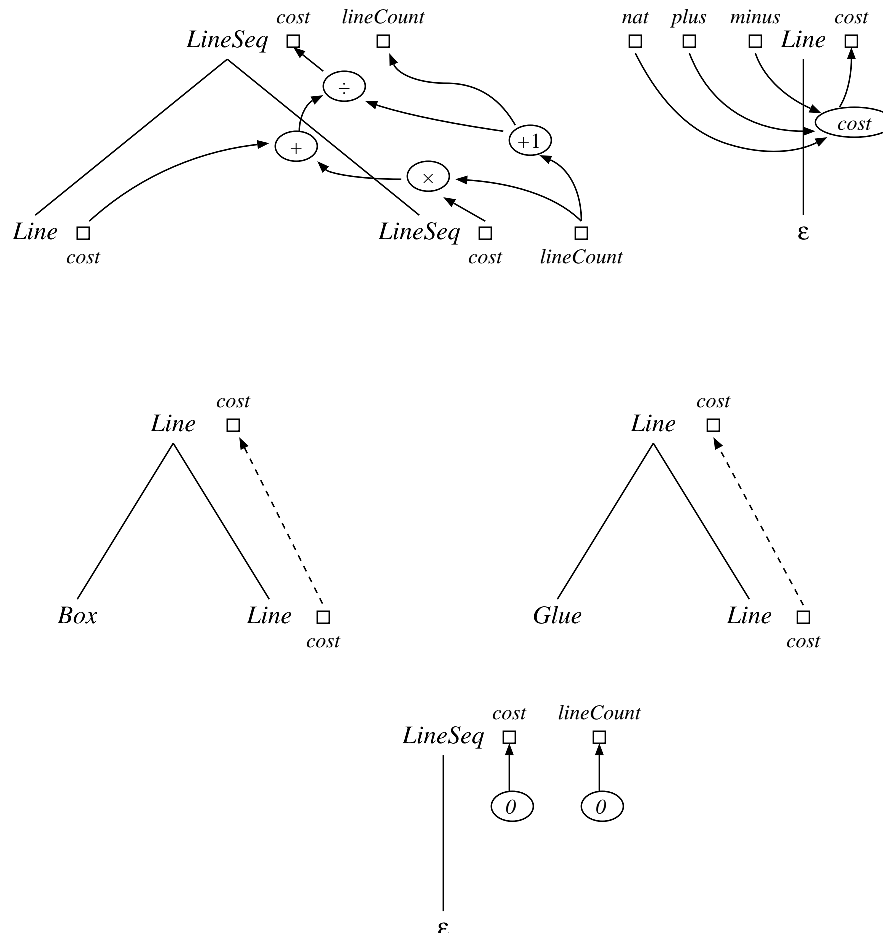


Figure 2.11: Computation of cost of line breaking results

Our specification of line breaking described so far can be extended to allow minimization of cost simply by attaching a corresponding instruction to the rule construction line sequences as shown in Figure 2.12 on the next page. The intended meaning is to choose, after considering all consistent variants of sequences of lines, the result producing the least cost.

As this example has shown, the problem of breaking paragraphs into lines can be described declaratively using relatively simple transformation rules.

2.4 Page breaking

A more difficult formatting problem as compared to line breaking is page breaking. The increased difficulty is caused by the requirement that in addition to normal text

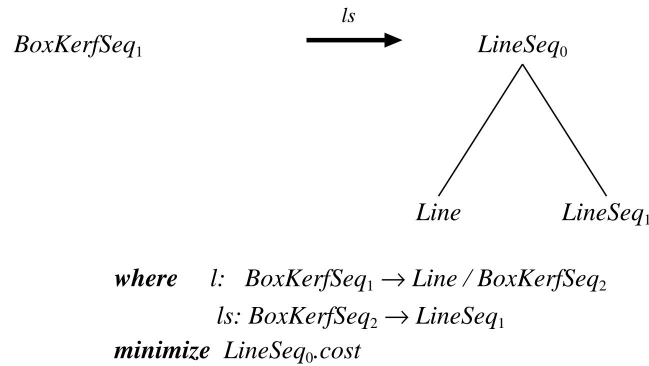


Figure 2.12: Rule for construction of optimal sequence of lines

other kinds of document elements, e.g., figures or footnotes, are placed on pages rather independently, apart from the flow of the body text. The placement of such elements is constrained by some typographical rules. E.g., a figure must not appear on a page before its first reference in the text.

Besides such formatting constraints, requirements of optimality w.r.t. some measures also exist for the problem of page breaking [5, 36, 54]. For instance, besides requiring that a figure is placed after its first reference, it is desirable for the figure to appear *as near as possible* to the reference, in order to minimize the reader's time looking for referenced figures. Apart from that it is desirable to optimize the visual appearance by minimizing the stretching and shrinking of white space just like with line breaking as described above.

Existing solutions to isolated page breaking problems have so far only been implemented in an operational fashion, e.g., in [28, 36]. These operational implementations are hard to create, read, and maintain. Thus, it is certainly desirable to be able to specify page breaking declaratively at a high level.

In the following a closer look at typical requirements arising with page breaking is taken. After that an outline to the declarative specification with techniques similar to the ones described for line breaking above is given.

2.4.1 Requirements on page breaking

Some selected common requirements on page breaking are described in more detail below. A more complete description can be found in [33].

Avoidance of widows and orphans

An important requirement is the avoidance of ☞*widow*s and ☞*orphan*s. The first line of text of a paragraph appearing solitarily at the bottom of a page is called an orphan. A widow is the last line of a paragraph located at the top of a page. Both widows and orphans impair the visual appearance of a document and should therefore be avoided.

Similarly, it is not desirable that section headings appear at the bottom of a page. Instead, the beginning of the first paragraph of a section should always go to the same

page as the heading.

Footnotes

A new kind of requirement occurs when placing footnotes. A footnote consists of an anchor in the text and a body which is usually placed at the bottom of pages. The general rule is that a footnote body can spread over multiple pages, but must start on the page containing the footnote anchor.

Floating objects

Similar to footnotes, floating objects such as figures or tables appear separately from the flow of the main document text. The rules for placing floats are usually less restrictive than those for footnotes: a float can be placed on any page not preceding the page containing the float anchor in the text.

2.4.2 Declarative rules for specifying page layout

We now outline an approach to the specification of page breaking. As we will see, similar techniques as the one used before for line breaking can be applied. This example will be picked up again in Chapter 5, when our transformation model and a practical specification language have been developed.

We restrict ourselves to the problems of the avoidance of ☞*widow*s and ☞*orphan*s, and the placement of floating objects. The handling of footnotes is omitted, but can be realized similar to floating objects.

Problems like the avoidance of ☞*widow*s and ☞*orphan*s can be modeled with box kerf sequences used before for the specification of line breaking. Box kerf sequences are now vertical and boxes represent lines of text (as for instance produced like described above for line breaking) and kerfs indicate where page breaks may occur. So, for a sequence of lines representing a text paragraph kerfs are inserted between any two adjacent lines, but not between the first and the second and not between the second last and the last. The glue contained in kerfs now realizes inter-line space (☞*leading*).

In order to specify the transformation of breaking text lines and floating objects into pages we decide to divide a page into two areas, one containing a sequence of floating objects and the other containing text lines. For this purpose it is a good idea to introduce an intermediate structure again. It is rather straightforward to define such a structure of *PageArea*s and *PageAreaSeq*s, similar to *Line* and *LineSeq* which were used above when specifying line breaking.

A requirement described above is that floating objects must not be placed on a page before its anchor in the text. To recognize if a sequence of pages is consistent w.r.t. this condition, it is possible to attribute page sequence structures with cross reference tables similar to the way symbol tables are used in the field of compiler construction. For each page both anchors of floating objects as well as floating objects themselves are added to this table, so inconsistent sequences of pages can be recognized easily.

It is also quite straightforward to realize the computation of the cost associated with a sequence of pages. Cost related to the visual appearance can be computed in a similar way as done in the line breaking example above. In addition, the computation of cost

can use the cross reference table described above to incorporate, e.g., the distance of floating objects from their anchor references.

Using those intermediate structures it is possible to specify the page breaking process using transformation rules like outlined in Figure 2.13.

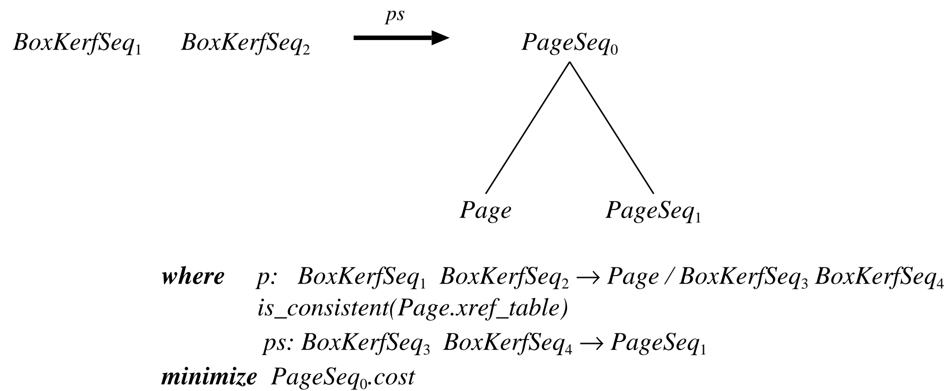


Figure 2.13: Rules specifying the breaking of text lines and floating objects into pages

We do not further describe how to specify the transformation relation p at this point. It should be clear by now that the techniques described before can be applied again. A more complete description will be given in Chapter 5 using the specification language then having been developed.

2.5 Conclusion

We have developed ideas for techniques allowing to specify important problems of optimizing document formatters. In the following chapters we will use these ideas to develop a formal model for the transformation of attributed trees and a practical specification language based upon it.

CHAPTER 3

A formal model for optimizing transformations of attributed trees

In this chapter a formal model for optimizing transformations between attributed trees is developed, incorporating the ideas for specifying formatting problems described in the previous chapter.

First, a representation of attributed trees and semantic constraints is introduced.

Then, a constraint-based language for specifying optimizing transformations between attributed trees is introduced and its formal semantics is defined.

After that, it is described how transformations based on that model can be efficiently executed. This execution scheme is first defined for non-optimizing specifications and proved to be sound and complete w.r.t. the transformation model's semantics. After that handling of optimization is added, resulting in a system of recursive functions implementing an optimizing transformation specification. Finally, it is shown how dynamic programming can be applied in order to increase efficiency by avoiding unnecessary re-computations of common sub transformation results.

3.1 Modeling attributed trees and constraints

3.1.1 Attributed trees

Before developing a model for the specification of transformations, we first define the representation of attributed trees that will be used. Note that, as discussed before, syntactic correctness of trees can and should be enforced using a context-free grammar; however, this is not relevant for the theoretical transformation model described here and is therefore omitted. It is also abstracted from how trees are actually attributed, i.e., attribute values are obtained by some, not further specified, attribution function.

Definition 3.1.1 (Σ-Trees)
Let Σ be a set of syntactic symbols. The set $\mathcal{T}_\Sigma$ of **Σ-trees** is inductively defined as follows:

- if $F \in \Sigma$ and $t_1, t_2, \ldots, t_k$ are Σ-trees for some $k \geq 0$, then $F(t_1, t_2, \ldots, t_k)$ is a Σ-tree.

$\Diamond$

In order to address nodes of a tree *positions* are introduced. A position specifies the path from the root of a tree to a node by a sequence of natural numbers each giving the index of the child node to visit next. The following defines the set of positions of a tree (which is a subset of $\mathbf{Nat}^*$).

Definition 3.1.2 (Positions)
The set $Pos(t) \subseteq \mathbf{Nat}^*$ of **positions** of a Σ-tree t is defined by induction on the structure of t:

$$Pos\big(F(t_1, t_2, \ldots, t_k)\big) \quad =_{def} \quad \{\varepsilon\} \cup \Bigg(\bigcup_{1 \leq i \leq k} \{\langle i \rangle \circ pos \mid pos \in Pos(t_i)\} \Bigg)$$

$\Diamond$

The following definition introduces some useful operations on trees.

Definition 3.1.3 (Operations on trees)
Let Σ be a set of symbols. Given a Σ-tree t, $symbol(t)$ denotes the symbol at the root of the tree:

$$symbol\big(F(t_1, \ldots, t_k)\big) \quad =_{def} \quad F$$

For a tree t and a position $pos \in Pos(t)$, $subtree(pos, t)$ denotes the subtree at position pos:

$$subtree(pos, t) \quad =_{def} \quad \begin{cases} t & \text{if} \quad pos = \varepsilon \\ subtree(pos', t_i) & \text{if} \quad pos = \langle i \rangle \circ pos', \\ & \qquad t = F(t_1, \ldots, t_k) \text{ and } 1 \leq i \leq k \end{cases}$$

$t[t'/pos]$ denotes the tree obtained by replacing the subtree from position $pos \in Pos(t)$ by the new subtree t' and is defined by induction on pos:

$$\begin{aligned} t[t'/\varepsilon] \quad &=_{def} \quad t' \\ F(t_1, \ldots, t_k)[t'/\langle i \rangle \circ pos'] \quad &=_{def} \quad F\big(t_1, \ldots, t_{i-1}, t_i[t'/pos'], t_{i+1}, \ldots, t_k\big) \end{aligned}$$

The number of direct subtrees of a tree t is given by:

$$subtree_count\big(F(t_1, \ldots, t_k)\big) \quad =_{def} \quad k$$

$\Diamond$

A node of a tree can be specified by the tree itself together with a position. We will now define, given a set Σ of symbols, the set of all nodes of a single Σ-tree and the set of nodes of all Σ-trees.

Definition 3.1.4 (Nodes)
Let Σ be a set of symbols. The set of **nodes** of a Σ-tree t is given by $Node(t) =_{def} \{(t, pos) \mid pos \in Pos(t)\}$. The set of all nodes of Σ-trees is defined as $\mathcal{N}_\Sigma =_{def} \bigcup_{t \in \mathcal{T}_\Sigma} Node(t)$. We will write $t_{\downarrow pos}$ for elements of $\mathcal{N}_\Sigma$. $\Diamond$

Example 3.1.5
Let $\Sigma_T = \{A, T, E, H, V\}$. An example tree according to this signature is

$$t = T(E(H(E(V(E(A()), E(A))), E(A()))))$$

which is depicted in Figure 3.1. The set of nodes of this tree is

$$\begin{aligned} Node(t) = \{ \; & t_{\downarrow\varepsilon},\ t_{\downarrow 1},\ t_{\downarrow 11},\ t_{\downarrow 111}, \\ & t_{\downarrow 1111},\ t_{\downarrow 11111},\ t_{\downarrow 111111},\ t_{\downarrow 11112}, \\ & t_{\downarrow 111121},\ t_{\downarrow 112},\ t_{\downarrow 1121} \quad \} \end{aligned}$$

■

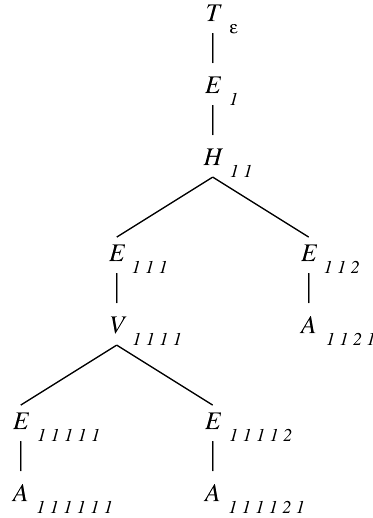


Figure 3.1: Example tree

Some useful operations on tree nodes needed later are introduced in the following definition.

Definition 3.1.6 (Operations on tree nodes)
The subtree at a node $t_{\downarrow pos}$ is given by:

$$subtree(t_{\downarrow pos}) \quad =_{def} \quad subtree(pos, t)$$

The symbol used at a node σ is denoted by

$$symbol(\sigma) \quad =_{def} \quad symbol\big(subtree(\sigma)\big)$$

The number of children of a node σ is given by:

$$child_count(\sigma) \quad =_{def} \quad subtree_count\big(subtree(\sigma)\big)$$

A descendant from position pos' of a node $\sigma = t_{\downarrow pos}$ is given by:

$$\sigma_{\downarrow pos'} \quad =_{def} \quad t_{\downarrow pos \circ pos'}$$

$\sigma_{\downarrow i}$ is an abbreviation for $\sigma_{\downarrow \langle i \rangle}$ addressing the i-th child of σ.

Replacing the subtree at a node with a new subtree yields the node at the same position in a new tree:

$$replace(t_{\downarrow pos}, t') \quad =_{def} \quad t[t'/pos]_{\downarrow pos}$$

The sequence of children of a node σ is given by

$$children(\sigma) \quad =_{def} \quad \sigma_{\downarrow 1}\, \sigma_{\downarrow 2} \ldots \sigma_{\downarrow k}$$

where $k = child_count(\sigma)$. $\Diamond$

We have now defined a representation of trees and tree nodes. In order to enforce syntactic validity of trees with respect to a context-free grammar the necessary additions can be easily made (see Appendix C).

As we will work with *attributed* trees, the following introduces the notion of *attributions* which allow attributes to be associated with tree nodes. An attribution specifies a set of attribute symbols, a domain of attribute values, an attribution function, and a relation describing the functional dependencies between attribute occurrences in a tree. The attribution function yields, given a tree node and an attribute symbol, an attribute value or the special value $\perp$ for undefined attributes, e.g., if some attribute is not associated with a node at all.

Definition 3.1.7 (Attribution)
An **attribution** is a tuple $A = (\Sigma, Inh, Synth, \mathcal{D}, att, Dep)$ where

(i) Σ is a set of syntactic symbols.

(ii) Inh and $Synth$ are two disjoint sets of **inherited** and **synthesized attributes**, respectively. Let $\mathcal{A} = Inh \cup Synth$. A **tree attribute occurrence** is given by an element of $Attr =_{def} \mathcal{N}_\Sigma \times \mathcal{A}$. The tree attribute occurrences of a tree t are denoted by $Attr(t) =_{def} Node(t) \times \mathcal{A}$.

(iii) $\mathcal{D}$ is a **domain** of attribute values.

(iv) $att\colon \mathcal{N}_\Sigma \times \mathcal{A} \to \mathcal{D} \cup \{\perp\}$ is a function associating attribute values with tree attribute occurrences; the special symbol $\perp \notin \mathcal{D}$ represents the undefined attribute value. As usual for attribute grammars, root nodes have no inherited attributes, so $att(t_{\downarrow \varepsilon}, a) = \perp$ for any $t \in \mathcal{T}_\Sigma$ and $a \in Inh$.

(v) $Dep \subseteq Attr \times Attr$ is a transitive and irreflexive relation reflecting the functional dependencies between attributes. We define the restriction to the dependencies within a single tree t as $Dep(t) =_{def} Dep \cap \big(Attr(t) \times Attr(t)\big)$.

◊

Note that *att* and *Dep* are in practice defined by the semantics of an attribute grammar. An attribution representing an example attribute grammar is shown below in Example 3.1.8. A general mapping from attribute grammars to attributions is described in detail in Appendix C.

Also note that attribute grammar semantics traditionally allow more than one consistent attribution of a tree, e.g., if underspecification occurs. The present model, in contrast, enforces unambiguous attribution of trees which suffices for most applications of attribute grammars and is more appropriate for our purposes.

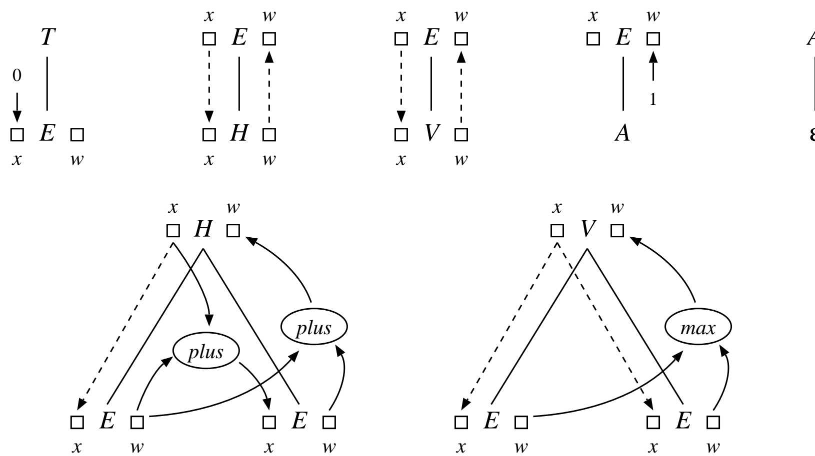


Figure 3.2: Example attribute grammar

Example 3.1.8
Let Σ_T be defined as in Example 3.1.5 on page 27.

The attribute grammar depicted in Figure 3.2 defines how the inherited attribute x and the synthesized attribute w (both integers) can be associated with tree nodes of symbols E, H and V. The functions *plus* and *max* are intended to have their usual meaning here. In this case we have an attribution $\big(\Sigma_T, Inh = \{x\}, Synth = \{w\}, \mathcal{D} = \mathbf{Int}, att_T, Dep_T\big)$. att_T attributes the example tree t from Example 3.1.5 on page 27 as follows (compare

with Figure 3.3):

$$att_T(t_{\downarrow pos}, x) = \begin{cases} 0 & \text{if} \quad pos \in \{1, 11, 111, 1111, 11111, 11112\} \\ 1 & \text{if} \quad pos = 112 \\ \perp & \text{otherwise.} \end{cases}$$

$$att_T(t_{\downarrow pos}, w) = \begin{cases} 1 & \text{if} \quad pos \in \{11111, 11112, 1111, 111, 112\} \\ 2 & \text{if} \quad pos \in \{11, 1\} \\ \perp & \text{otherwise.} \end{cases}$$

The value $\perp$ is used for attributes which are not associated with a node's symbol.

$Dep_T(t)$ reflects the attribute dependency information for t and can be seen as the transitive closure of the attribute dependency graph shown in Figure 3.3. ■

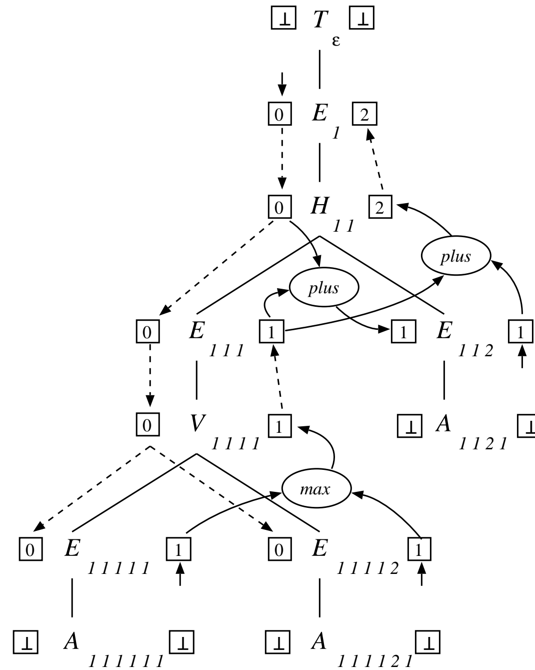


Figure 3.3: Attribution of the example tree

3.1.2 Semantic constraints

In order to be able to express semantic constraints (i.e., constraints on attributes of tree nodes), a *constraint system* defines a family of *predicate symbols* for each arity and an interpretation of all predicate symbols in some domain.

Definition 3.1.9 (Constraint system)
A **constraint system** over a domain $\mathcal{D}$ is a tuple $CS = \big((Pred_k)_{k \in \mathbf{Nat_0}}, I\big)$ consisting of a family of predicate symbols for each arity $k \in \mathbf{Nat_0}$ and an interpretation I associating with each $P \in Pred_k$ a relation $I(P) \subseteq \mathcal{D}^k$. ◊

3.2 Specification of optimizing transformations of attributed trees

After the preliminaries, we now come to the definition of our model for the specification of tree transformations. As motivated by the previous chapter, we first deal with specifying relations between source and target trees satisfying syntactic and semantic consistency constraints and later make the additions needed for optimization.

Throughout the rest of the chapter, let Σ be a fixed, but arbitrary, set of symbols and $\mathcal{D}$ a fixed, but arbitrary semantic domain. Further, let $A = (\Sigma, Inh, Synth, \mathcal{D}, att, Dep)$ and $CS = ((Pred_k)_{k \in \mathbf{Nat_0}}, I)$ be fixed, but arbitrary, Σ-attribution and constraint system over $\mathcal{D}$, respectively. Further, let $\mathcal{A} = Inh \cup Synth$. As we will only deal with a single signature Σ, we will write $\mathcal{T}$ for $\mathcal{T}_\Sigma$ and $\mathcal{N}$ for $\mathcal{N}_\Sigma$ in order to save indices.

3.2.1 Specification of relations between attributed trees

A central concept used for the specification of transformations is that of *templates*. A template describes the structure of a contiguous set of nodes of a tree and will be used for matching both source and target subtrees. Templates can be imagined as tree patterns where a *node variable* is located at each pattern node. Using those node variables attributes of all nodes may be addressed when specifying semantic aspects of transformations. A syntactic symbol is associated with each node variable and a node variable will only be allowed to hold a node of its symbol.

Definition 3.2.1 (Node variables)
A set of **node variables** is given by a countable set $\mathcal{V}ar$ of variables together with a mapping $symbol\colon \mathcal{V}ar \to \Sigma$. ◊

In examples we will use variables consisting of a symbol from Σ and disambiguating numbers or primes identifying the variable; i.e., given a symbol $F \in \Sigma$, we will use F, F', F'', F_0, F_1, F_2, etc. as variables of symbol F.

Definition 3.2.2 (Templates)
Given a set $\mathcal{V}ar$ of node variables, the syntactic set *Template* of **templates** is defined inductively:

(i) $X \in Template$ for each $X \in \mathcal{V}ar$

(ii) $X \langle\, T_1 T_2 \ldots T_k \,\rangle \in Template$ if $X \in \mathcal{V}ar$ and $T_i \in Template$ for each $1 \leq i \leq k$.

The set of variables of a template T is denoted by $Var(T)$, inductively defined as follows:

$$\begin{aligned} Var(X) &=_{def} \{X\} \\ Var\big(X \langle T_1 \ldots T_k \rangle\big) &=_{def} \{X\} \cup \Big(\bigcup_{1 \leq i \leq k} Var(T_i) \Big) \end{aligned}$$

For a sequence of templates $\bar{T} = T_1 \dots T_k$ we define $Var(\bar{T}) =_{def} \bigcup_{1 \leq i \leq k} Var(T_i)$.

The set of leaf variables of a template T is denoted by $Leaf_Var(T)$, inductively defined so:

$$\begin{array}{lll} Leaf_Var(X) & =_{def} & \{X\} \\ Leaf_Var\big(X \langle T_1 \dots T_k \rangle\big) & =_{def} & \bigcup_{1 \leq i \leq k} Leaf_Var(T_i) \end{array}$$

The set of **positions** of a template is inductively defined:

(i) $Pos(X) \quad =_{def} \quad \{\varepsilon\}$

(ii) $Pos(X \langle T_1 \dots T_k \rangle) \quad =_{def} \quad \{\varepsilon\} \cup \left(\bigcup_{1 \leq i \leq k} \{ \langle i \rangle \circ pos \mid pos \in Pos(T_i) \} \right)$

Let T be a template and $pos \in Pos(T)$. The variable at a position pos in T is denoted by $T_{\downarrow pos}$, defined by induction on pos:

(i) $X_{\downarrow \varepsilon} \quad =_{def} \quad X$

(ii) $X \langle T_1 \dots T_k \rangle_{\downarrow \varepsilon} \quad =_{def} \quad X$

(iii) $X \langle T_1 \dots T_k \rangle_{\downarrow \langle i \rangle \circ pos'} \quad =_{def} \quad T_{i \downarrow pos'}$

◊

The following defines the notions of *matching* and *instantiation* of templates.

Definition 3.2.3 (Matching of templates and instantiation at nodes)
We define when a template **matches** a tree by induction on the structure of templates:

(i) X matches t iff $symbol(t) = symbol(X)$

(ii) $X \langle T_1 \dots T_k \rangle$ matches $F(t_1, \dots, t_n)$ iff $F = symbol(X)$, $k = n$, and T_i matches t_i for each $1 \leq i \leq k$.

A template T is defined to be **instantiated** at a node σ iff T matches $subtree(\sigma)$. ◊

As an example, Figure 3.4 on the facing page illustrates the instantiation of the template T having the form $E_0 \langle H \langle E_1 \; E_2 \rangle \rangle$ in a tree. Note that the template can be instantiated at the tree's root node as well.

As already mentioned above, a template carries variables at each node in order to allow specifying semantic aspects (constraints) of transformation specifications by referring to attributes. This is made possible using *attribute occurrences* which are introduced next.

Definition 3.2.4 (Attribute occurrences)
An **attribute occurrence** has the form $X.a$ where $X \in \mathcal{V}ar$ and $a \in \mathcal{A}$. The set of attribute occurrences is denoted by $Attr_Occ$.

Given a template T, $Attr_Occ(T) =_{def} \{X.a \mid X \in Var(T), a \in \mathcal{A}\}$ denotes the set of attribute occurrences of T. ◊

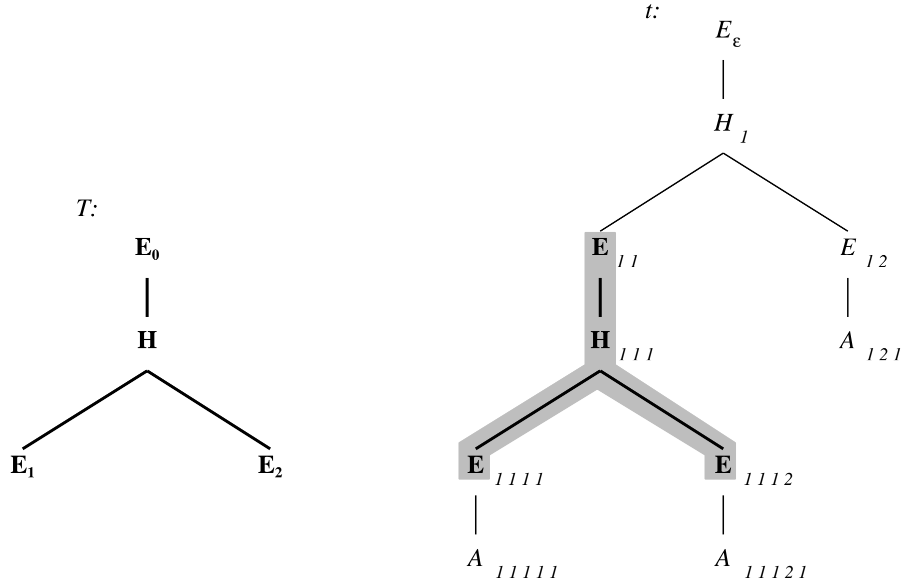


Figure 3.4: Example of the instantiation of a template

A relation between (sequences of) tree nodes is specified using a set of *rules* which each consist of a sequence of *source templates*, a *target template*, a sequence of *rest variables* and a set of *constraints*. Constraints can express that descendant nodes are related, or that predicates on some attribute occurrences hold.

Definition 3.2.5 (Transformation rules, transformation specification)
A **transformation rule** has the form

$$\bar{S} \rightarrow T/\bar{R} \text{ where } \varphi_1, \varphi_2, \ldots, \varphi_k$$

where $\bar{S}$ is a sequence of templates, T is a single template, $\bar{R}$ is a sequence of variables, $k \geq 0$, and $\varphi_1, \ldots, \varphi_k$ are constraints. $\bar{S}$, T, and $\bar{R}$ are called the rule's **source**, **target** and **rest**, respectively.

A **constraint** has either the form

$$\bar{X} \rightarrow Y/\bar{Z}$$

where $\bar{X} \in \mathcal{V}ar^*$, $Y \in \mathcal{V}ar$ and $\bar{Z} \in \mathcal{V}ar^*$, or the form

$$P(\alpha_1, \ldots, \alpha_k)$$

where $k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are attribute occurrences and $P \in Pred_k$. We will call a constraint of the first form a **syntactic constraint** and one of the second form a **semantic constraint**.

A **transformation specification** is defined to be a set of transformation rules. $\diamond$

We now take a look at an example of a transformation specification. In this simple specification the rules' and syntactic constraints' rest is always empty.

Example 3.2.6
Let Σ_T be defined as in Example 3.1.5 on page 27. In addition, let Σ_S be the following set of symbols:

$$\Sigma_S = \{L, S, B\}$$

A combined set of symbols is formed by $\Sigma_{ST} = \Sigma_S \cup \Sigma_T$.

We define an attribution such that Σ_T-trees are attributed as in Example 3.1.8 on page 29 and Σ_S-trees obtain no additional attributes. So the attribution is defined as the tuple $\big(\Sigma_{ST}, \{x\}, \{w\}, \mathbf{Int}, att_{ST}, Dep_{ST}\big)$, where

$$att_{ST}(t_{\downarrow pos}, a) \quad =_{def} \quad \begin{cases} att_T(t_{\downarrow pos}, a) & \text{if} \quad t \in \mathcal{T}_{\Sigma_T} \\ \bot & \text{otherwise} \end{cases}$$

and

$$Dep_{ST} = Dep_T$$

We further introduce the constraint system $CS_{ST} = \big((Pred_k)_{k \in \mathbf{Nat_0}}, I\big)$ over **Int** where

$$\begin{aligned} Pred_2 &= \{P\} \\ Pred_k &= \emptyset \quad \text{for} \quad k \neq 2 \end{aligned}$$

where $P(x, y)$ is satisfied iff $x + y \leq 2$. We will write applications of P in the latter form to increase readability.

We then define the following specification describing a transformation of S-trees to T-trees:

$$spec_{S \to T} = \{r_1, r_2, r_3, r_4\}$$

where

$$\begin{aligned}
r_1 : \quad & S \to T\,\langle\, E\,\rangle \;/\; \varepsilon \\
& \quad \textsf{where} \quad S \to E \;/\; \varepsilon \\
r_2 : \quad & S_0\,\langle\, B\,\langle\, S_1 S_2\,\rangle\,\rangle \to E\,\langle\, H\,\langle\, E_1 E_2\,\rangle\,\rangle \;/\; \varepsilon \\
& \quad \textsf{where} \quad S_1 \to E_1 \;/\; \varepsilon, \\
& \qquad\qquad\quad S_2 \to E_2 \;/\; \varepsilon, \\
& \qquad\qquad\quad E.x + E.w \leq 2 \\
r_3 : \quad & S_0\,\langle\, B\,\langle\, S_1 S_2\,\rangle\,\rangle \to E\,\langle\, V\,\langle\, E_1 E_2\,\rangle\,\rangle \;/\; \varepsilon \\
& \quad \textsf{where} \quad S_1 \to E_1 \;/\; \varepsilon, \\
& \qquad\qquad\quad S_2 \to E_2 \;/\; \varepsilon, \\
& \qquad\qquad\quad E.x + E.w \leq 2 \\
r_4 : \quad & S\,\langle\, L\,\rangle \to E\,\langle\, A\,\rangle \;/\; \varepsilon
\end{aligned}$$

■

A second example defines a transformation specification realizing the approach to line breaking described in Chapter 2.

Example 3.2.7
Let $A_L = (\Sigma, \mathit{Inh}, \mathit{Synth}, \mathcal{D}, \mathit{att}, \mathit{Dep})$ be an attribution where

$$\begin{aligned}\Sigma &= \{\mathit{BoxKerfSeq}, \mathit{Box}, \mathit{Kerf}, \mathit{Glue}, \mathit{LineSeq}, \mathit{Line}, \mathit{Int}\}\\ \mathit{Inh} &= \{\alpha', \mathit{Nat}, \mathit{Plus}, \mathit{Minus}\}\\ \mathit{Synth} &= \{\alpha, w, h, d, \mathit{value}, \mathit{nat}, \mathit{plus}, \mathit{minus}\}\\ \mathcal{D} &= \mathbf{Int}\end{aligned}$$

and *att* and *Dep* correspond to the attribute grammar described in Chapter 2.

We assume a constraint system CS_L to be defined that contains two predicates P_1 and P_2, both of arity 2, where $P_1(x, y)$ is satisfied iff $x + y \geq \mathit{lineWidth}$ and $P_2(x, y)$ is satisfied iff $x - y \leq \mathit{lineWidth}$. Here, *lineWidth* is some given constant integer value. To make things more readable we will use infix notation for these two predicates.

We then define a transformation specification spec_L containing the following transformation rules:

$r_1 :\quad \mathit{BoxKerfSeq}\,\langle\, \varepsilon \,\rangle \rightarrow \mathit{LineSeq}\,\langle\, \varepsilon \,\rangle \;/\; \varepsilon$

$r_2 :\quad \mathit{BoxKerfSeq}_0 \rightarrow \mathit{LineSeq}_0\,\langle\, \mathit{Line}\;\; \mathit{LineSeq}_1 \,\rangle \;/\; \varepsilon$
where $\quad \mathit{BoxKerfSeq}_0 \rightarrow \mathit{Line} \;/\; \mathit{BoxKerfSeq}_1,$
$\mathit{BoxKerfSeq}_1 \rightarrow \mathit{LineSeq}_1 \;/\; \varepsilon$

$r_3 :\quad \mathit{BoxKerfSeq}_0\,\langle\, \mathit{Box}_1\;\; \mathit{BoxKerfSeq}_1 \,\rangle \rightarrow \mathit{Line}_0\,\langle\, \mathit{Box}_2\;\; \mathit{Line}_1 \,\rangle \;/\; \mathit{BoxKerfSeq}_2$
where $\quad \mathit{Box}_1 \rightarrow \mathit{Box}_2 \;/\; \varepsilon,$
$\mathit{BoxKerfSeq}_1 \rightarrow \mathit{Line}_1 \;/\; \mathit{BoxKerfSeq}_2$

$r_4 :\quad \mathit{BoxKerfSeq}_0\,\langle\, \mathit{Kerf}\;\; \mathit{BoxKerfSeq}_1 \,\rangle \rightarrow \mathit{Line}\,\langle\, \varepsilon \,\rangle \;/\; \mathit{BoxKerfSeq}_0$
where $\quad \mathit{Line.Nat} + \mathit{Line.Plus} \geq \mathit{lineWidth}$

$r_5 :\quad \mathit{BoxKerfSeq}_0\,\langle\, \mathit{Kerf}\,\langle\, \mathit{Glue}_1 \,\rangle\; \mathit{BoxKerfSeq}_1 \,\rangle$
$\rightarrow \mathit{Line}_0\,\langle\, \mathit{Glue}_2\;\; \mathit{Line}_1 \,\rangle \;/\; \mathit{BoxKerfSeq}_2$
where $\quad \mathit{Line.Nat} - \mathit{Line.Minus} \leq \mathit{lineWidth},$
$\mathit{Glue}_1 \rightarrow \mathit{Glue}_2,$
$\mathit{BoxKerfSeq}_1 \rightarrow \mathit{Line}_1 \;/\; \mathit{BoxKerfSeq}_2$

We assume that further transformation rules $\mathit{Box}_1 \rightarrow \mathit{Box}_2$ and $\mathit{Glue}_1 \rightarrow \mathit{Glue}_2$ performing structural copying are defined. ■

3.2.2 Semantics of a specification

A sequence of source nodes should by intuition be *related* to a target node and a sequence of rest nodes according to a given specification if a rule's source, target and rest can be

instantiated, and the rule's constraints are satisfied. A semantic constraint should be satisfied if the predicate on the given attributes holds; a syntactic constraint should be satisfied if the nodes referred to by the given variables are related. These ideas are formalized in the following by notions of *variable environments* and *template instantiation by environments*, and the definition of a *transformation relation*.

A *variable environment* defines a mapping from variables to nodes. Undefined variables are mapped to the special symbol $\bot$. It is enforced that a defined variable of some symbol can only be mapped to a tree node of the same symbol.

Definition 3.2.8 (Variable environment)
We define $\mathcal{N}_\bot =_{def} \mathcal{N} \cup \{\bot\}$. A **variable environment** is a function $env\colon \mathcal{V}ar \to \mathcal{N}_\bot$ where for each $X \in \mathcal{V}ar$ and $\sigma \in \mathcal{N}$

$$env(X) = \sigma \quad \Rightarrow \quad symbol(X) = symbol(\sigma)$$

The set of all variable environments is denoted by *Env*.

For a sequence of variables $\bar{X} = X_1 X_2 \ldots X_n$ we will use $env(\bar{X})$ as an abbreviation for the sequence $env(X_1)\ env(X_2)\ \ldots\ env(X_n)$.

The following abbreviations are introduced for convenient addressing of node variables and attribute occurrences, given an environment *env*:

$$\begin{aligned} [\![X]\!]_{env} &=_{def} env(X) \\ [\![X.a]\!]_{env} &=_{def} \begin{cases} att\big([\![X]\!]_{env}, a\big) & \text{if} \quad [\![X]\!]_{env} \neq \bot \\ \bot & \text{otherwise.} \end{cases} \end{aligned}$$

For a k-ary constraint P we write:

$$\Vdash_{env} P(\alpha_1, \ldots, \alpha_k) \quad \Leftrightarrow_{def} \quad \big([\![\alpha_1]\!]_{env}, \ldots, [\![\alpha_k]\!]_{env}\big) \in I(P)$$

$\Diamond$

We now extend the notion of template instantiation introduced before by incorporating an environment: an environment instantiates a template at a node iff it maps each template variable to the corresponding descendant of the node.

Definition 3.2.9 (Instantiation of templates by environments)
Let *env* be an environment.

env **instantiates** a template T at a node σ iff T is instantiated at σ and $[\![T_{\downarrow pos}]\!]_{env} = \sigma_{\downarrow pos}$ for each $pos \in Pos(T)$.

env instantiates a sequence $T_1 T_2 \ldots T_n$ of templates at a sequence $\sigma_1 \sigma_2 \ldots \sigma_m$ of nodes iff $n = m$ and *env* instantiates T_i at σ_i for each $1 \leq i \leq n$. $\Diamond$

Example 3.2.10
As an example, we consider the instantiation of the template T having the form $E_0 \langle H \langle E_1\ E_2 \rangle \rangle$ in the tree t shown in Figure 3.4 on page 33.

T is instantiated at $t_{\downarrow 11}$ by any environment *env* containing the following mappings:

$$\begin{array}{rcl} E_0 & \mapsto & t_{\downarrow 11} \\ H & \mapsto & t_{\downarrow 111} \\ E_1 & \mapsto & t_{\downarrow 1111} \\ E_2 & \mapsto & t_{\downarrow 1112} \end{array}$$

■

We are now ready to formally define the semantics of a specification as a *transformation relation* between sequences of source nodes, a target node and a sequence of rest nodes. The definition of the transformation relation is based upon a notion of *S-derivations* which reflect the application of a transformation rule (each sub derivation corresponds to a syntactic constraint of the applied rule).

Definition 3.2.11 (Semantics of a specification)
Let *spec* be a specification, $\bar{\sigma} \in \mathcal{N}^*$, $\tau \in \mathcal{N}$, $\bar{\rho} \in \mathcal{N}^*$ and $r \in spec$ be a rule of the form

$$\bar{S} \rightarrow T/\bar{R} \text{ where } \varphi_1, \ldots, \varphi_k$$

An **S-derivation** for $\bar{\sigma} \rightarrow_r \tau/\bar{\rho}$ has the form $(\bar{\sigma} \rightarrow_r \tau/\bar{\rho}, env)[D_1, \ldots, D_n]$, where $n \geq 0$, D_i is itself an S-derivation for each $1 \leq i \leq n$, and *env* is a variable environment such that

(i) *env* instantiates $\bar{S}$ at $\bar{\sigma}$

(ii) *env* instantiates T at τ

(iii) $[\![\bar{R}]\!]_{env} = \bar{\rho}$

(iv) each semantic constraint $P(\alpha_1, \ldots, \alpha_m) \in \{\varphi_1, \ldots, \varphi_k\}$ is satisfied, i.e., $\Vdash_{env} P(\alpha_1, \ldots, \alpha_m)$

(v) each syntactic constraint $(\bar{X} \rightarrow Y/\bar{R}) \in \{\varphi_1, \ldots, \varphi_k\}$ is satisfied, i.e., D_i is an S-derivation for $[\![\bar{X}]\!]_{env} \rightarrow_{r'} [\![Y]\!]_{env}/[\![\bar{Z}]\!]_{env}$ for some $1 \leq i \leq n$ and some rule $r' \in spec$; in addition it is required that $[\![Y]\!]_{env}$ is a root node (i.e., has the form $t_{\downarrow\varepsilon}$ for some tree t) if $Y \notin Var(T)$

We will write $\bar{\sigma} \rightarrow_r \tau/\bar{\rho}$ iff an S-derivation for $\bar{\sigma} \rightarrow_r \tau/\bar{\rho}$ exists. Further, we write $\bar{\sigma} \rightarrow \tau/\bar{\rho}$ iff $\bar{\sigma} \rightarrow_r \tau/\bar{\rho}$ for some $r \in spec$.

The semantics of a transformation specification *spec* is declared as

$$\begin{array}{rcl} \mathcal{TR}_r(spec) & =_{def} & \{(\bar{\sigma}, \tau, \bar{\rho}) \mid \bar{\sigma} \rightarrow_r \tau/\bar{\rho}\} \\ \mathcal{TR}(spec) & =_{def} & \{(\bar{\sigma}, \tau, \bar{\rho}) \mid \exists r \in spec \,.\, (\bar{\sigma}, \tau, \bar{\rho}) \in \mathcal{TR}_r(spec)\} \end{array}$$

◊

Example 3.2.12
We now take a look at the transformation of an example tree according to the transformation specification from Example 3.2.6 on page 34. The tree s that is to be transformed is depicted in Figure 3.5 on the next page. Figure 3.6 on page 40 shows three trees t, t' and t'', which are valid transformation results for the input tree. The following shows why t is a valid transformation result:

1. $s_{\downarrow 1111} \rightarrow_{r_4} t_{\downarrow 11111}$. Rule $r_4 : S\langle L \rangle \rightarrow E\langle A \rangle/\varepsilon$ has no constraints, so it is sufficient that source and target templates can be instantiated which is true. Thus, a corresponding S-derivation
$$D_1 = (s_{\downarrow 1111} \rightarrow_{r_4} t_{\downarrow 11111}, env_1)[]$$
exists where env_1 instantiates $S\langle L \rangle$ at $s_{\downarrow 1111}$ and instantiates $E\langle A \rangle$ at $t_{\downarrow 11111}$.

2. In the same way, we can see that $s_{\downarrow 1112} \rightarrow_{r_4} t_{\downarrow 11112}$ and $s_{\downarrow 12} \rightarrow_{r_4} t_{\downarrow 112}$. The corresponding S-derivations shall be denoted by D_2 and D_3, respectively.

3. Using D_1 and D_2 we can then derive that $s_{\downarrow 11} \rightarrow_{r_3} t_{\downarrow 111}$. Recall that r_3 was defined as
$$\begin{aligned} r_3 : \quad & S_0\langle B\langle S_1 S_2 \rangle\rangle \rightarrow E\langle V\langle E_1 E_2 \rangle\rangle/\varepsilon \\ & \textsf{where} \quad S_1 \rightarrow E_1/\varepsilon, \\ & \qquad\qquad S_2 \rightarrow E_2/\varepsilon, \\ & \qquad\qquad E.x + E.w \leq 2 \end{aligned}$$
Source and target templates of this rule can be instantiated as is visualized in Figure 3.8 on page 41, and the semantic constraint $E.x + E.w \leq 2$ is satisfied. So an S-derivation
$$D_4 = (s_{\downarrow 11} \rightarrow_{r_3} t_{\downarrow 111}, env_4)[D_1, D_2]$$
exists where env_4 instantiates $S_0\langle B\langle S_1 S_2 \rangle\rangle$ at $s_{\downarrow 11}$ and also instantiates $E\langle V\langle E_1 E_2 \rangle\rangle$ at $t_{\downarrow 111}$.

4. Similarly, an S-derivation
$$D_5 = (s_{\downarrow \varepsilon} \rightarrow_{r_2} t_{\downarrow 1}, env_5)[D_3, D_4]$$
can be constructed.

5. Finally, an S-derivation
$$D_6 = (s_{\downarrow \varepsilon} \rightarrow_{r_1} t_{\downarrow \varepsilon}, env_6)[D_5]$$
can easily be constructed.

Analogously, $s_{\downarrow \varepsilon} \rightarrow_{r_1} t'_{\downarrow \varepsilon}$ and $s_{\downarrow \varepsilon} \rightarrow_{r_1} t''_{\downarrow \varepsilon}$ can be shown to be valid transformations.

However, the tree shown in Figure 3.7 on page 41 is no valid transformation result. Here, syntactic constraints alone are satisfied, but $s_{\downarrow \varepsilon} \rightarrow_{r_2} t''_{\downarrow 1}$ does not hold because the semantic constraint is not satisfied ($t''_{\downarrow 1}.x + t''_{\downarrow 1}.w = 3 \not\leq 2$).

This example shows that our transformation model really allows to specify a transformation *relation* as three valid target structures exist. ■

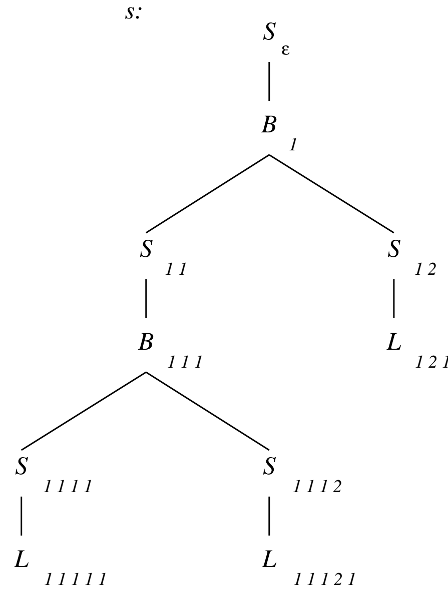


Figure 3.5: Source tree for the example transformation

3.2.3 Optimizing transformation specifications

Optimization can easily be realized by associating an optional *objective function* with each rule of a specification. An objective function assigns an integer value with the result of a transformation measuring the cost or badness of the possible transformation. This is captured in the following definition.

Definition 3.2.13 (Optimizing transformation specification)
An **objective function** has the form

$$f(\alpha_1, \ldots, \alpha_k)$$

where $k \geq 0$, f is a function from $\mathcal{D}^k \rightarrow \mathbf{Int}$, and $\alpha_i \in \mathit{Attr_Occ}$ for $1 \leq i \leq k$. The set of objective functions is denoted by *Obj*.

An **optimizing transformation specification** consists of a transformation specification *spec* and a function

$$obj : spec \rightarrow Obj \ \cup \ \{\perp\}$$

which associates with each rule $r \in spec$ an optional objective function. ◊

In the semantics of an optimizing transformation specification a sequence of source nodes is only related to a target node and sequence of rest nodes if no better target node and sequence of rest nodes exist w.r.t. the objective function of the applied transformation. This idea is formalized in the following definition of *optimal S-derivations*:

Definition 3.2.14 (Semantics of an optimizing specification)
Let *spec* be an optimizing specification, $\bar{\sigma} \in \mathcal{N}^*$, $\tau \in \mathcal{N}$, $\bar{\rho} \in \mathcal{N}^*$, and $r \in spec$.

An S-derivation $(\bar{\sigma} \rightarrow_r \tau/\bar{\rho}, env)[D_1, \ldots, D_n]$ is called **optimal** iff

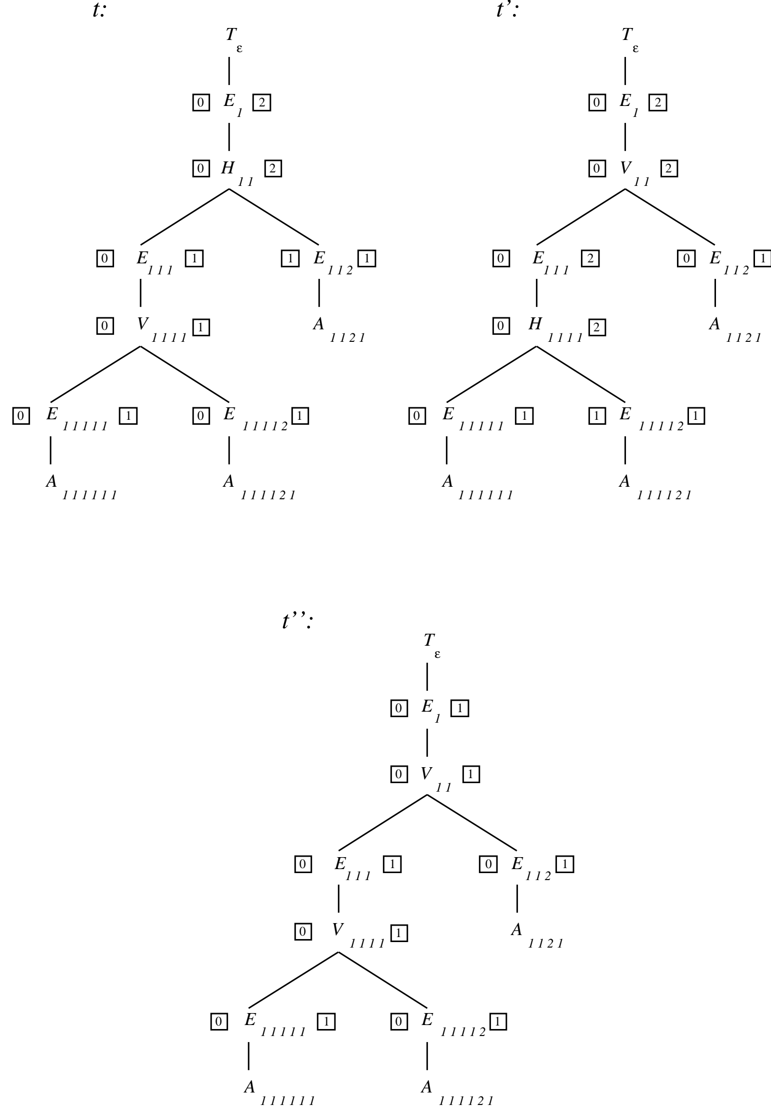


Figure 3.6: Results of the example transformation

(i) $obj(r) = \bot$, or $obj(r)$ has the form $f(\alpha_1, \ldots, \alpha_k)$ and no S-derivation $(\bar{\sigma} \rightarrow_r \tau'/\bar{\rho}', env')[D'_1, \ldots, D'_n]$ exists such that $\tau' = replace(\tau, t')$ for some t' and

$$f([\![\alpha_1]\!]_{env'}, \ldots, [\![\alpha_k]\!]_{env'}) < f([\![\alpha_1]\!]_{env}, \ldots, [\![\alpha_k]\!]_{env})$$

(ii) D_i is optimal for each $1 \leq i \leq n$

We write $\bar{\sigma} \rightarrow_{r,opt} \tau/\bar{\rho}$ iff an optimal derivation for $\bar{\sigma} \rightarrow_r \tau/\bar{\rho}$ exists. Further, we write $\bar{\sigma} \rightarrow_{opt} \tau/\bar{\rho}$ iff $\bar{\sigma} \rightarrow_{r,opt} \tau/\bar{\rho}$ for some $r \in spec$.

The semantics of an optimizing transformation specification *spec* is finally declared as

$$\mathcal{TR}_{opt}(spec) \quad =_{def} \quad \{(\bar{\sigma}, \tau, \bar{\rho}) \mid \bar{\sigma} \rightarrow_{opt} \tau/\bar{\rho}\}$$

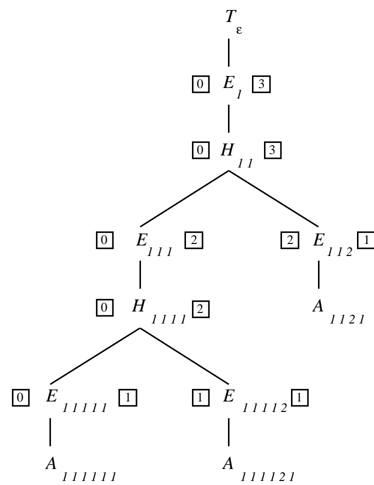


Figure 3.7: A tree which is no result for the example transformation

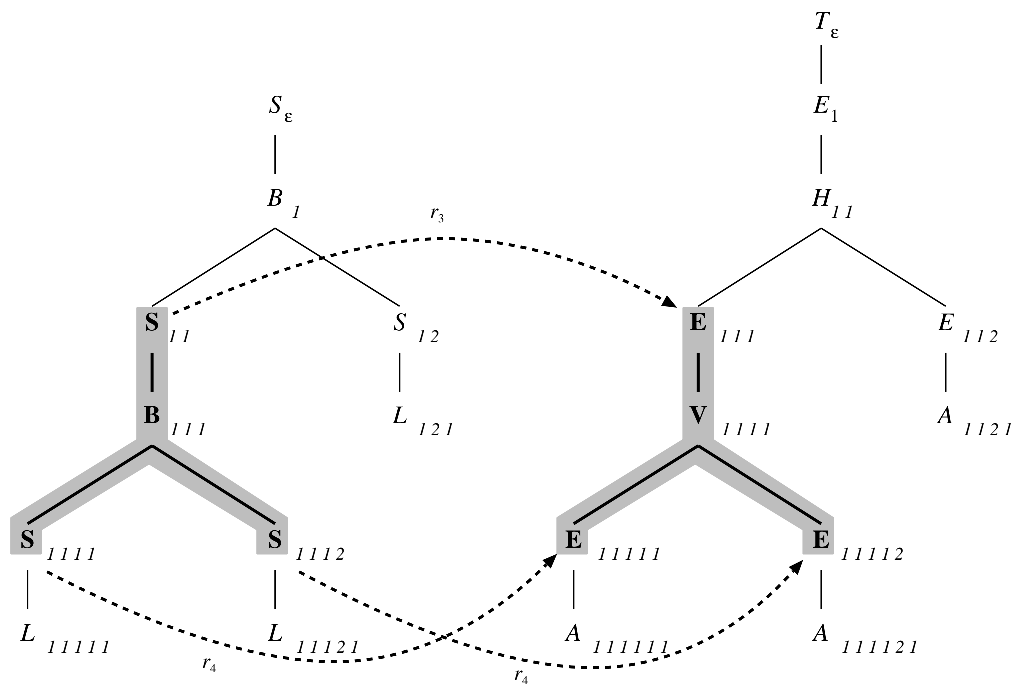


Figure 3.8: Example of the application of a transformation rule

◊

We have now formally defined specifications of optimizing transformations between attributed trees and their semantics. The semantics has been defined in a natural way through a notion of derivations.

However, the semantics is unfortunately not constructive, i.e., it does not yet give any idea of how to effectively compute target and rest nodes given source nodes. The reason for this is that the semantics defines relations to target nodes by means of an existentially quantified environment leaving open how such an environment can be *constructed*.

So we will now investigate the *execution* of transformation specifications, i.e., the task of constructing target and rest nodes given a sequence of source nodes.

3.3 Efficient execution of transformations

In this section an *execution scheme* for the transformation model is developed which describes the construction of transformation results, given source nodes, in an operational way and which can directly be used to implement the model. Omitting optimization at first, the task of performing a transformation is to find, given a sequence of source nodes, all possible target and rest nodes which are related to the source nodes by the means of $\mathcal{TR}(spec)$.

The execution scheme developed in this section works efficiently by pruning search space for transformation results to a great extent using the transformation rules' semantic constraints. During execution the constraints of each rule are processed separately. To make things a little easier, constraints are processed strictly in order from left to right (this will be shown to impose some restrictions on specifications; however, an algorithm will be given later that turns a specification into an equivalent specification satisfying these restrictions, if one exists).

We first describe how target trees are built while a transformation is performed and introduce means to analyze the attribute dependencies affecting transformations. We then formally define the notion of LR specifications which define the restrictions to specifications allowing left to right processing of constraints as mentioned above. Then an execution scheme for LR specifications without optimization is introduced and proven to be sound and complete w.r.t. the semantics given in Section 3.2.2. After that, support for optimization is added, resulting in a system of recursive functions implementing a transformation specification. It is then shown that dynamic programming can be applied to further increase efficiency by avoiding unnecessary re-computations of sub transformation results. Finally, the algorithm already mentioned above, turning transformation specifications into equivalent LR specifications, is given.

3.3.1 Construction of target trees

A crucial decision to make is whether target trees should be built in a bottom-up or top-down fashion. The following will show why top-down construction promises much better efficiency.

In the case of sub transformations, target trees are subtrees of larger trees. The situation for a sub transformation is illustrated in Figure 3.9 on the next page. Transformation rules may involve constraints on inherited attributes of nodes of these subtrees or attributes dependent on them. Such constraints can not be evaluated until enough of the context of the target subtrees is known for the required attribute occurrences to be defined.

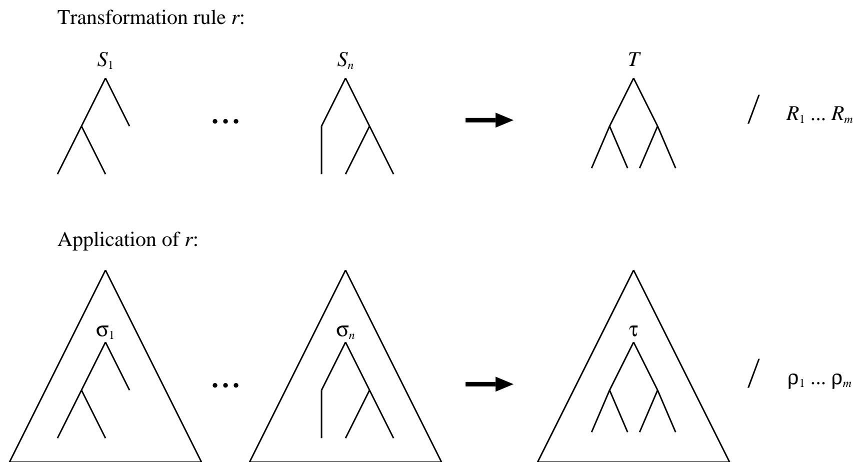


Figure 3.9: A transformation rule and its application

As a consequence, if target trees were built bottom-up, checking semantic constraints would in the worst case have to be delayed until the complete target tree has been built; i.e., in the course of a transformation all syntactically consistent target trees would have to be searched for first, and only then all semantically inconsistent results could be excluded. So this approach is unacceptably inefficient. Semantic constraints should in contrast be used to recognize failing transformations as soon as possible.

In contrast, a better approach is to construct the target tree in a top-down fashion. Then, inherited attributes and attributes dependent on them can be computed already *during* the process of building the target tree, and thus semantic constraints can be checked as early as possible, pruning search space for transformation results to the greatest possible extent. This approach promises highest efficiency and is followed here.

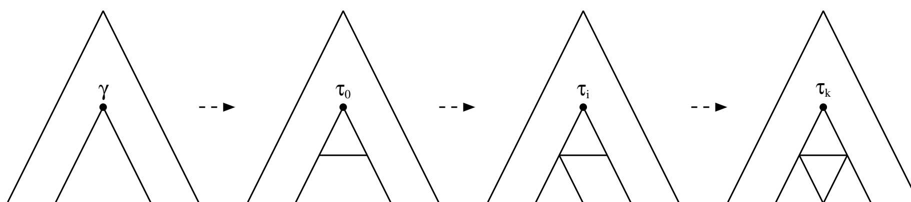


Figure 3.10: Construction of target subtrees

The top-down construction of target trees is realized as follows: for performing a (sub) transformation, besides the sequence of source nodes an additional unexpanded *anchor* node is given. When applying a transformation rule, the target subtree will initially be constructed as a tree fragment having only unexpanded nodes at the target template's leaves; the target subtree will then be expanded by processing syntactic constraints until a complete subtree has been reached. This process is illustrated in Figure 3.10.

3.3.2 Tree fragments and their attribution

The idea to build target trees in a top-down fashion requires a representation of intermediate trees allowing unexpanded subtrees, which we will call *tree fragments*. In the following, we will see how tree fragments can be modeled and introduce a partial order resembling the expansion of tree fragments. Then, we will discuss the attribution of tree fragments.

Representing tree fragments

Intermediate target trees can contain unexpanded subtrees, i.e., subtrees which are not yet known. The idea of tree fragments is accounted for in our model simply by assuming the presence of a special syntactic symbol $\triangle$ representing a yet unknown sequence of subtrees. I.e., we assume that $\triangle \in \Sigma$. An unexpanded subtree of some symbol F is then represented as $F_\triangle =_{def} F\big(\triangle()\big)$. Trees containing no $\triangle$-subtrees are called **complete**.

It would make not much sense to allow the usage of $\triangle$ in specifications since a user would neither want to specify a transformation to nor from (partly) undefined trees. To prevent this, the following simple restriction can be introduced in order to exclude variables of symbol $\triangle$:

$$symbol(\xi) \neq \triangle \quad \text{for all} \quad \xi \in \mathcal{V}ar$$

Expansion of tree fragments

The idea of expanding target trees from an initial tree fragment to a complete subtree motivates the definition of a binary relation expressing that one tree fragment t' is an expansion of another tree fragment t. Thus, we define a relation $\preceq \;\subseteq\; \mathcal{T} \times \mathcal{T}$ as follows.

Definition 3.3.1
The relation $\preceq \;\subseteq\; \mathcal{T} \times \mathcal{T}$ is defined as the least relation satisfying the following conditions, where $F \in \Sigma$, $k, l \geq 0$, and $t_1, \ldots, t_k$ and $t'_1, \ldots, t'_l$ are trees:

(i) $F_\triangle \preceq F(t_1, \ldots t_k)$

(ii) $F(t_1, \ldots, t_k) \preceq F(t'_1, \ldots, t'_l)$, if $t_i \preceq t'_i$ for $1 \leq i \leq k$

$\lozenge$

As indicated by the symbol, $\preceq$ is a *partial order* on tree fragments.

Proposition 3.3.2
The binary relation $\preceq \;\subseteq\; \mathcal{T} \times \mathcal{T}$ is a partial order, i.e., it is reflexive, transitive and antisymmetric.

Proof:
see Appendix B.2. $\square$

We now introduce corresponding partial orders on tree nodes, sequences of nodes and on environments as well. A node is called *less or equally expanded* than another node iff it is the node at the same position in a less or equally expanded tree. A sequence of nodes is called less or equally expanded than another sequence of nodes iff both have the same length and each contained node is less or equally expanded than the corresponding node in the second sequence. An environment is called a *predecessor* of another environment iff the environment maps each variable to $\perp$ or to a node less or equally expanded than the node the other environment maps the variable to.

Definition 3.3.3
Let t and t' be trees, $pos \in Pos(t)$, and $pos' \in Pos(t')$. We define a binary relation $\preceq \;\subseteq \mathcal{N} \times \mathcal{N}$ as follows:

$$t_{\downarrow pos} \preceq t'_{\downarrow pos'} \quad \Leftrightarrow_{def} \quad pos = pos' \;\wedge\; t \preceq t' \;\wedge\; symbol(t_{\downarrow pos}) \neq \triangle$$

If $\sigma \preceq \sigma'$, we say that σ is **less or equally expanded** than σ'.

For sequences of nodes we define:

$$\sigma_1 \ldots \sigma_k \preceq \sigma'_1 \ldots \sigma'_l \quad \Leftrightarrow_{def} \quad \begin{cases} k = l \\ \sigma_i \preceq \sigma'_i \quad \text{for each} \quad 1 \leq i \leq k \end{cases}$$

Let $env, env' \in Env$. A binary relation $\preceq \;\subseteq\; Env \times Env$ is defined as follows:

$$env \preceq env' \quad \Leftrightarrow_{def} \quad [\![\xi]\!]_{env} = \perp \quad \vee \quad [\![\xi]\!]_{env} \preceq [\![\xi]\!]_{env'} \\ \text{for all} \quad \xi \in \mathcal{V}ar$$

We will call env a **predecessor** of env' iff $env \preceq env'$. ◊

Attribution of tree fragments

Tree fragments were introduced in order to evaluate and check semantic constraints already during the process of building target trees. Hence, it is necessary to describe the attribution of tree fragments.

The semantics of attribute grammars can be extended in a natural way to handle tree fragments, simply by assigning tree occurrences of synthesized attributes of unexpanded nodes and all dependent tree attribute occurrences the special undefined value $\perp$. These tree occurrences of attributes will be called *preliminary* as later expansion of the tree fragment may cause those attributes to evaluate to a defined value.[1]

Figure 3.11 on the next page illustrates the attribution of tree fragments, again using the attribute grammar from Example 3.1.8 on page 29. The shown example tree fragment resulted from replacing the node at position 1 1 1 in the tree that was shown in Figure 3.3 on page 30 by an unexpanded subtree.

The extensions to attribute grammars necessary to allow attribution of tree fragments are precisely described in Appendix C. At this point only two properties of the

[1]Note that $\perp$ is also used for attributes which are not associated with a tree node at all (see Section 3.1.1), so it is conversely not true that non-preliminary occurrences of attributes always evaluate to a defined value.

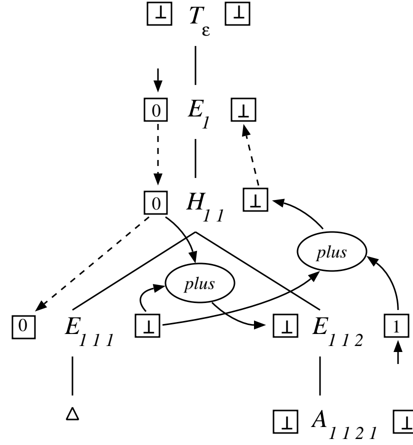


Figure 3.11: Example of a partially attributed tree fragment

attribution of tree fragments that are relevant here are described. Firstly, we expect that preliminary tree attribute occurrences really evaluate to $\perp$, and secondly, we expect that (non-preliminary) occurrences which evaluate to a defined value in a tree fragment evaluate to the same value as the tree fragment expands. The second requirement is crucial to allow a semantic constraint to be checked as soon as all attributes involved become defined, as the result (constraint satisfied or not satisfied) will remain valid while the target tree expands. This monotonicity condition is illustrated in Figure 3.12 where an undefined subtree of a tree t is replaced by a new tree fragment and the value of a non-preliminary attribute a at some node σ_1 does not change while a preliminary attribute b at some other node σ_2 evaluates to a defined value in the expanded tree.

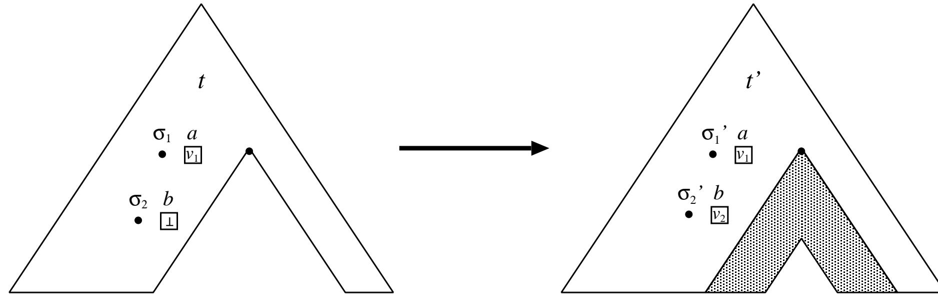


Figure 3.12: Monotonicity condition on the attribution of tree fragments

Regarding our transformation model it is practical to analyze the preliminary attributes of only the part of a target tree instantiating a rule's target template. Since the attribution of a subtree depends only on the inherited attributes at its root and the subtree's structure itself, the preliminary attribute occurrences in a subtree from a node σ can be determined from a given set of inherited attributes at σ already known to be preliminary, and from a set of positions giving the descendants of σ known to be yet unexpanded. Given this information, the preliminary attribute occurrences in a subtree

of σ are:

- attributes at σ which are contained in the given set $I \subseteq \mathit{Inh}$ of inherited attributes already known to be preliminary
- synthesized attributes of descendants of σ whose subtrees are known to be preliminary; the positions of those descendants is given by a set $P \subseteq \mathit{Pos}\big(\mathit{subtree}(\sigma)\big)$
- all attribute occurrences dependent on those given above

The set of final attribute occurrences in a subtree consists of all attribute occurrences which are not preliminary. The formal definition is as follows.

Definition 3.3.4 ($\mathit{Prelim_Attr_Occ}_{I,P}(\sigma)$)
The set of **preliminary tree attribute occurrences** within the subtree of a node σ w.r.t. a set of attributes $I \subseteq \mathit{Inh}$ and set of positions $P \subseteq \mathit{Pos}\big(\mathit{subtree}(\sigma)\big)$ is denoted by $\mathit{Prelim_Attr_Occ}_{I,P}(\sigma)$ and is defined to be the least set of pairs $(\mathit{pos}, a) \subseteq \mathit{Pos}\big(\mathit{subtree}(\sigma)\big) \times \mathcal{A}$ satisfying the following conditions:

(i) $\big\{(\varepsilon, a) \mid a \in I\big\} \subseteq \mathit{Prelim_Attr_Occ}_{I,P}(\sigma)$

(ii) $\big\{(\mathit{pos}, a) \mid \mathit{pos} \in P \ \wedge \ a \in \mathit{Synth}\big\} \subseteq \mathit{Prelim_Attr_Occ}_{I,P}(\sigma)$

(iii) $$\left.\begin{array}{l}(\mathit{pos}, a) \in \mathit{Prelim_Attr_Occ}_{I,P}(\sigma) \\ (\sigma_{\downarrow \mathit{pos}}.a, \sigma_{\downarrow \mathit{pos}'}.a') \in \mathit{Dep}\end{array}\right\} \Rightarrow (\mathit{pos}', a') \in \mathit{Prelim_Attr_Occ}_{I,P}(\sigma)$$

$$\mathit{Final_Attr_Occ}_{I,P}(\sigma) \quad =_{\mathit{def}} \quad \Big(\mathit{Pos}\big(\mathit{subtree}(\sigma)\big) \times \mathcal{A}\Big) \setminus \mathit{Prelim_Attr_Occ}_{I,P}(\sigma) \qquad \diamondsuit$$

In order to help formalize the monotonicity property of attributions given informally above, we introduce a binary relation $\preceq_{I,P}$ on nodes where $I \subseteq \mathit{Inh}$ and $P \subseteq \mathit{Pos}(\sigma)$. $\sigma \preceq_{I,P} \sigma'$ holds iff σ' is an expansion of σ where P is the set positions at which subtrees of σ are preliminary and I is the set of preliminary inherited attributes at σ. I.e., $\sigma \preceq_{I,P} \sigma'$ is true iff:

- σ is less or equally expanded than σ'
- only descendants from positions in P may be preliminary, i.e., descendants of σ and σ' from a position p, where p is not a prefix of any element of P, have equal subtrees
- only attributes from I may be preliminary, i.e., inherited attributes not contained in I have the same value at σ and σ'

The formal definition of $\preceq_{I,P}$ is given next.

Definition 3.3.5

(i) Let σ be a node, $P \subseteq \mathit{Pos}(\sigma)$, and $I \subseteq \mathit{Inh}$.

The set of positions of final subtrees of σ w.r.t. P is denoted by $\mathit{Final_Pos}_P(\sigma)$.

$$\mathit{Final_Pos}_P(\sigma) \quad =_{\mathit{def}} \quad \Big\{\mathit{pos} \in \mathit{Pos}\big(\mathit{subtree}(\sigma)\big) \mid \ \nexists \mathit{pos}' \,.\, \mathit{pos} \circ \mathit{pos}' \in P\Big\}$$

(ii) Let σ and σ' be nodes. Further, let $P \subseteq Pos(subtree(\sigma))$ and $I \subseteq Inh$.

We define

$$\sigma \preceq_{I,P} \sigma' \quad \Leftrightarrow_{def} \quad \begin{cases} \sigma \preceq \sigma' \\ \forall pos \in Final_Pos_P(\sigma) \;.\; subtree(\sigma_{\downarrow pos}) = subtree(\sigma'_{\downarrow pos}) \\ \forall a \in Inh \setminus I \;.\; att(\sigma, a) = att(\sigma', a) \end{cases}$$

◊

The monotonicity property of attributions can now be formalized.

Definition 3.3.6
An attribution $A = (\Sigma, Inh, Synth, \mathcal{D}, att, Dep)$ is called **monotonous** iff the following conditions are satisfied for any nodes σ and σ', $P \subseteq Pos(subtree(\sigma))$, any attribute $a \in \mathcal{A}$, and $I \subseteq Inh$:

(i) $\sigma \preceq_{I,P} \sigma'$
$\Rightarrow \quad \big((pos, a) \in Final_Attr_Occ_{I,P}(\sigma) \quad \Rightarrow \quad att(\sigma_{\downarrow pos}, a) = att(\sigma'_{\downarrow pos}, a)\big)$

(ii) $\sigma \preceq \sigma' \quad \Rightarrow \quad att(\sigma, a) = \bot \;\vee\; att(\sigma, a) = att(\sigma', a)$

◊

The properties of monotonous attributions described above concerning semantic constraints are captured in the following proposition and corollary.

Proposition 3.3.7
Let $A = (\Sigma, Inh, Synth, \mathcal{D}, att, Dep)$ be a monotonous attribution. Then the following is true for any environments env and env', attribute occurrence α, and value $d \in \mathcal{D}$:

$$[\![\alpha]\!]_{env} \neq \bot \;\wedge\; env \preceq env' \quad \Rightarrow \quad [\![\alpha]\!]_{env'} = [\![\alpha]\!]_{env}$$

Proof:
Follows immediately from Definition 3.3.6 (ii). □

Corollary 3.3.8
Let A be a monotonous attribution. Then the following holds for any k-ary predicate P, attribute occurrences $\alpha_1, \ldots, \alpha_k$ and environments env, env':

$$env \preceq env' \;\wedge\; [\![\alpha_i]\!]_{env} \neq \bot \quad \text{for all} \quad 1 \leq i \leq k \quad \Rightarrow$$
$$\Vdash_{env} P(\alpha_1, \ldots, \alpha_k) \quad \Leftrightarrow \quad \Vdash_{env'} P(\alpha_1, \ldots, \alpha_k)$$

□

We will need Corollary 3.3.8 later in Section 3.3.6 when proving the soundness of the then developed execution scheme to show that semantic constraints can be checked as soon as all tree attribute occurrences are defined.

Incremental attribution of tree fragments

We have now completely described and formalized the notion of tree fragments and their attribution. As a target tree fragment is expanded during the invocation of a transformation rule, the question may arise how attributes can be efficiently re-evaluated with each expansion step. A closer look shows that this is just a special case of incremental attribution [7, 39, 40]. In the case of general incremental attribution arbitrary subtrees can be replaced, while in our case only unexpanded subtrees can be replaced and only attributes having the value $\perp$ will ever have to be re-evaluated. Efficient methods for incremental attribution are well-known from the field of language-based programming environments [39, 40]; these methods can be directly applied or slightly refined to exactly fit our purposes (which is, however, not described here).

3.3.3 Attribute dependencies in templates

It will be necessary to analyze the attribute dependencies affecting the building of target tree structures. For this purpose we define a relation describing the possible dependencies between attribute occurrences in a (target) template.

Definition 3.3.9 (Attribute dependencies in templates)
Let T be a template. $\mathit{Dep}(T) \subseteq \mathit{Attr_Occ}(T) \times \mathit{Attr_Occ}(T)$ denotes the relation describing all possible dependencies between attribute occurrences in T:

$$\begin{aligned}&(X.a, Y.b) \in \mathit{Dep}(T) \quad \Leftrightarrow_{def}\\ &\quad X \in \mathit{Var}(T) \ \wedge \ Y \in \mathit{Var}(T) \ \wedge\\ &\quad \exists env \;.\; env \text{ instantiates } T \ \wedge \ \big([\![X]\!]_{env}.a, [\![Y]\!]_{env}.b\big) \in \mathit{Dep}\end{aligned}$$

$\Diamond$

We now define the sets of preliminary and final attribute occurrences in a target template. The set $\mathit{Prelim_Attr_Occ}_{I,V}(T)$ contains the preliminary attribute occurrences in T when the inherited attributes at T's root given in I are assumed to be preliminary and the leaf variables given in V are assumed to be preliminary as well, i.e., have an unexpanded node assigned. The set of final attribute occurrences in a template consists of all attribute occurrences which are not preliminary.

Definition 3.3.10 ($\mathit{Prelim_Attr_Occ}_{I,V}(T)$)
Let T be a template. Further, let $I \subseteq \mathit{Inh}$ and $V \subseteq \mathit{Leaf_Var}(T)$.

We define $\mathit{Prelim_Attr_Occ}_{I,V}(T)$ to be the least set satisfying the following conditions:

(i) $\{T_{\downarrow\varepsilon}.a \mid a \in I\} \subseteq \mathit{Prelim_Attr_Occ}_{I,V}(T)$

(ii) $\{\xi.a \mid \xi \in V \ \wedge \ a \in \mathit{Synth}\} \subseteq \mathit{Prelim_Attr_Occ}_{I,V}(T)$

(iii) $\left.\begin{array}{l}\alpha \in \mathit{Prelim_Attr_Occ}_{I,V}(T)\\ (\alpha, \alpha') \in \mathit{Dep}(T)\end{array}\right\} \quad \Rightarrow \quad \alpha' \in \mathit{Prelim_Attr_Occ}_{I,V}(T)$

$\mathit{Final_Attr_Occ}_{I,V}(T) \quad =_{def} \quad \mathit{Attr_Occ}(T) \setminus \mathit{Prelim_Attr_Occ}_{I,V}(T)$ $\Diamond$

3.3.4 LR specifications

It has already been indicated above that target subtrees are expanded by processing the syntactic constraints while applying a transformation rule. It is however not clear yet in what order the constraints of a rule are to be processed. For the execution scheme developed in this section we will restrict ourselves to process the constraints of a rule—including semantic constraints—strictly from left to right.

Left to right execution imposes some restrictions on transformation specifications; specifications satisfying these properties will be called LR specifications. The main restrictions are:

- when processing a semantic constraint, all of the constraint's attribute occurrences must be final
- when processing a syntactic constraint, the variables on the left hand side must be defined; this means that those variables must occur in source templates or have been assigned a node by a previously processed syntactic constraint
- when processing a syntactic constraint the target variable's inherited attributes necessary to process the constraint must be final (all rules that can possibly be invoked through the constraint must be considered)
- after processing all constraints all leaf variables of the target template must be defined; this ensures that no tree fragment can result from a transformation

The conditions listed above describe properties that must necessarily be satisfied at the time constraints are processed. However, it turns out that these conditions can be guaranteed to be satisfied by certain properties of a transformation specification that can be *statically* checked by analysis of the specification. This analysis of specifications is described next. First, some required preliminary definitions are introduced.

Preliminary definitions

The set of variables that become defined by processing a constraint consists of the variables appearing on the constraint's right hand side in case of a syntactic constraint and is empty for a semantic constraint. The set of variables that are defined after processing the first i constraints of a rule consists of the variables appearing in the rules's source templates and the variables defined through each of the first i constraints.

Definition 3.3.11
The set of variables that become defined by processing a constraint φ is denoted by Def_Var_φ which is defined as follows:

$$Def_Var_\varphi \quad =_{def} \quad \begin{cases} \{Y\} \cup Var(\bar{Z}) & \text{if } \varphi \text{ has the form } \bar{X} \rightarrow Y/\bar{Z} \\ \emptyset & \text{otherwise.} \end{cases}$$

The set of variables that are defined after processing the first i constraints of a rule r of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$ are denoted by $Def_Var_{i,r}$ (where $0 \leq i \leq k$),

defined as follows:

$$Def_Var_{i,r} \quad =_{def} \quad Var(\bar{S}) \cup \left(\bigcup_{1 \leq j \leq i} Def_Var_{\varphi_j} \right)$$

◊

In order to be able to express which transformation rules can possibly be applied at a syntactic constraint, we introduce the notion of *signatures* of rules and syntactic constraints. If a transformation rule can be applied through a syntactic constraint, then the signatures of the rule and the constraint must be the same. Signatures take into account the arities and (root) symbols of source, target and rest variables or templates. As such, signatures of constraints and rules are elements of $\Sigma^* \times \Sigma \times \Sigma^*$.

Definition 3.3.12 (Signature)
The set of signatures is denoted by

$$Sig \quad =_{def} \quad \Sigma^* \times \Sigma \times \Sigma^*$$

The **signature** of a syntactic constraint φ is denoted by $sig(\varphi) \in Sig$:

$$\begin{array}{l} sig(X_1 \ldots X_n \rightarrow Y/Z_1 \ldots Z_m) \quad =_{def} \quad (x_1 \ldots x_n, y, z_1 \ldots z_m) \\ \quad \text{where} \\ \qquad \begin{array}{lcll} x_i & = & symbol(X_i) & \text{for each } 1 \leq i \leq n \\ y & = & symbol(Y) & \\ z_j & = & symbol(Z_j) & \text{for each } 1 \leq j \leq m \end{array} \end{array}$$

The **signature** of a transformation rule r is denoted by $sig(r) \in Sig$:

$$\begin{array}{l} sig(S_1 \ldots S_n \rightarrow T/R_1 \ldots R_m) \quad =_{def} \quad (s_1 \ldots s_n, t, r_1 \ldots r_m) \\ \quad \text{where} \\ \qquad \begin{array}{lcll} s_i & = & symbol(S_{i\downarrow\varepsilon})) & \text{for each } 1 \leq i \leq n \\ t & = & symbol(T_{\downarrow\varepsilon}) & \\ r_j & = & symbol(R_j) & \text{for each } 1 \leq j \leq m \end{array} \end{array}$$

◊

The application of a transformation rule requires that initially enough inherited occurrences of attributes at the anchor node are defined to be able to evaluate all semantic constraints and to be able to perform all sub transformations for syntactic constraints. Further, it is necessary that each time a constraint is processed enough of the target tree is expanded so that all required attributes evaluate to a defined value. Otherwise it would be possible that a transformation cannot proceed because a semantic constraint cannot be checked due to missing attribute values.

The following defines the set $Requ_Attr_Occ(\varphi)$ of attribute occurrences required to be defined in order to be able to evaluate a constraint φ, and the set $Requ_Attr(r)$ of

attributes at a rule r's anchor node required to invoke the rule. The set of attribute occurrences required for processing a semantic constraint is the set of attribute occurrences referenced by the constraint; for a syntactic constraint the required attribute occurrences are occurrences of attributes from sets $Requ_Attr(r')$ at the constraint's target variable which are required to invoke any rule r' matching the syntactic constraint's signature. The attributes required to process a rule are those attributes for whose occurrences at the target template's root variable a dependency to any attribute occurrences needed for any of the rule's constraints exists.

As the sets $Requ_Attr_Occ(\varphi)$ and $Requ_Attr(r)$ mutually depend on each other, those sets are formally defined as a least fixed point.

Definition 3.3.13 ($Requ_Attr(r)$ and $Requ_Attr_Occ(\varphi)$)
Let r be a transformation rule of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$.

$Requ_Attr(r) \subseteq \mathcal{A}$ and $Requ_Attr_Occ(\varphi_i) \subseteq Attr_Occ$ for each $1 \leq i \leq k$ are defined to be the least sets satisfying the following conditions:

(i) $\{\alpha_1, \ldots, \alpha_n\} \subseteq Requ_Attr_Occ\big(P(\alpha_1, \ldots, \alpha_n)\big)$

(ii) $\big\{Y.a \mid \exists r' \in spec \;.\; sig(r') = sig(\bar{X} \rightarrow Y/\bar{Z}) \;\wedge\; a \in Requ_Attr(r')\big\} \subseteq Requ_Attr_Occ(\bar{X} \rightarrow Y/\bar{Z})$

(iii) $\big\{a \in Inh \mid \exists \alpha \;.\; (T_{\downarrow\varepsilon}.a, \alpha) \in Dep(T) \;\wedge\; \alpha \in Requ_Attr_Occ(\varphi_i)\big\} \subseteq Requ_Attr(r)$

◊

We finally need to know which attribute occurrences have their final value after processing the first constraints of a rule up to a given index. These are the non-preliminary attribute occurrences in the target template assuming the following:

- only attribute occurrences at the target template's root node not belonging to the rule's required attributes are preliminary; i.e., the rule's required attributes are final
- the target template's leaf variables that are not contained in $Def_Var_{i,r}$ are preliminary; the leaf variables contained in $Def_Var_{i,r}$ are final

Definition 3.3.14 ($Final_Attr_Occ_{i,r}$)
Let r be a rule of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$ and i be an index $(0 \leq i \leq k)$.

The set $Final_Attr_Occ_{i,r}$ is defined as follows:

$$\begin{aligned} Final_Attr_Occ_{i,r} \quad &=_{def} \quad Attr_Occ \setminus Prelim_Attr_Occ_{R,\mathcal{X}}(T) \\ \text{where} & \\ R \;&=\; Inh \setminus Requ_Attr(r) \\ \mathcal{X} \;&=\; Leaf_Var(T) \setminus Def_Var_{i,r} \end{aligned}$$

◊

Formal definition of LR specifications

We are now ready to formalize the requirements on a specification allowing processing of constraints from left to right. As already mentioned, we call specifications satisfying those restrictions *LR specifications*.

Definition 3.3.15 (LR specification)
A specification is called an **LR specification** iff for all rules r of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \dots, \varphi_k$ the following holds:

(i) for each syntactic constraint φ_i (where $1 \leq i \leq k$) of the form $\bar{X} \rightarrow Y/\bar{Z}$ the following conditions are satisfied:

  (a) $\mathit{Var}(\bar{X}) \subseteq \mathit{Def_Var}_{i-1,r}$

  (b) $\mathit{Def_Var}_{\varphi_i} \cap \mathit{Def_Var}_{i-1,r} = \emptyset$

  (c) $\{Y\} \cap \mathit{Var}(T) \subseteq \mathit{Leaf_Var}(T)$

(ii) for each semantic constraint φ_i (where $1 \leq i \leq k$) of the form $P(X_1.a_1, \dots, X_n.a_n)$ the following holds:

$$\{X_1, \dots, X_n\} \subseteq \mathit{Def_Var}_{i,r} \cup \mathit{Var}(T)$$

(iii) $\mathit{Requ_Attr_Occ}(\varphi_i) \subseteq \mathit{Final_Attr_Occ}_{i-1,r}$ for each constraint φ_i $(1 \leq i \leq k)$

(iv) $\mathit{Leaf_Var}(T) \cup \mathit{Var}(\bar{R}) \subseteq \mathit{Def_Var}_{k,r}$

$\Diamond$

3.3.5 An execution scheme for LR specifications

We are now ready to introduce an execution scheme for LR specifications in the form of a system of execution rules. Let in the sequel *spec* be a fixed, but arbitrary, LR specification.

First, the principle approach to processing constraints and the notion of processing states will be described. Then, auxiliary operations will be defined. Finally, the rules of the execution scheme will be given.

Processing of constraints

While processing a sequence $\varphi_1\varphi_2 \dots \varphi_k$ of constraints when applying a transformation rule, a chain $env_0 \preceq env_1 \preceq env_2 \dots \preceq env_k$ of environments will be constructed, starting from an initial environment env_0 and constructing an environment with more information with each syntactic constraint. However, the application of a transformation rule can also fail. To account for this we introduce the notion of *processing states*. A processing state is either an environment or a special constant *fail* representing an invalid processing state.

Definition 3.3.16 (Processing state)
An **invalid processing state** is represented by the special constant *fail*. The set of **processing states** is defined as

$$\mathcal{S}tate \quad =_{def} \quad Env \cup \{fail\}$$

$\Diamond$

In the following some operations for constructing and manipulating processing states are introduced.

Definition 3.3.17 (Operations on processing states)
By $\bot_{Env}$ we denote the **empty environment** mapping all variables to $\bot$:

$$\bot_{Env}(\xi) \quad =_{def} \quad \bot \quad \text{for all} \quad \xi \in \mathcal{V}ar$$

The **restriction of an environment** *env* to a set $\mathcal{X} \subseteq \mathcal{V}ar$ of variables is denoted by $env_{\mathcal{X}}$ and defined so:

$$env|_{\mathcal{X}}(\xi) \quad =_{def} \quad \begin{cases} env(\xi) & \text{if} \quad \xi \in \mathcal{X} \\ \bot & \text{otherwise} \end{cases}$$

For an environment $env \in Env$ its **domain** is denoted by $Dom(env)$

$$Dom(env) \quad =_{def} \quad \{\xi \in \mathcal{V}ar \mid env(\xi) \neq \bot\}$$

A binary operator $\triangleleft$ for composition of two processing states $env_1, env_2 \in \mathcal{S}tate$ which is strict w.r.t. *fail* is defined as follows:

$$env_1 \triangleleft env_2 \quad =_{def} \quad \begin{cases} fail & \text{if} \quad env_1 = fail \vee env_2 = fail \\ env' & \text{otherwise.} \end{cases}$$

where

$$env'(\xi) = \begin{cases} env_2(\xi) & \text{if} \quad env_2(\xi) \neq \bot \\ env_1(\xi) & \text{otherwise.} \end{cases}$$

An environment mapping a variable ξ to a node σ and all other variables to $\bot$ is denoted by $[\sigma/\xi]$. For sequences of variables and nodes $[\sigma_1 \ldots \sigma_k/\xi_1 \ldots \xi_k]$ will be used an an abbreviation for $[\sigma_1/\xi_1] \triangleleft \ldots \triangleleft [\sigma_k/\xi_k]$. $\Diamond$

The binary operator $\triangleleft$ has some important properties which are shown next.

Proposition 3.3.18 (Properties of $\triangleleft$)
The $\triangleleft$ operator has the following properties:

(i) For any processing states env_1, env_2, env_3:

$$env_1 \triangleleft (env_2 \triangleleft env_3) = (env_1 \triangleleft env_2) \triangleleft env_3$$

(ii) For $env_1, env_2 \in Env$ where $Dom(env_1) \cap Dom(env_2) = \emptyset$:

$$env_1 \lhd env_2 = env_2 \lhd env_1$$

(iii) For any $env \in \mathcal{S}tate$:

$$env \lhd \bot_{Env} = \bot_{Env} \lhd env = env$$

(iv) For any $env_1, env_2 \in Env$ where $Dom(env_1) = Dom(env_2)$:

$$env_1 \lhd env_2 = env_2$$

(v) For any $env_1, env_2 \in \mathcal{S}tate$, and $\mathcal{X} \subseteq \mathcal{V}ar$:

$$(env_1 \lhd env_2)|_{\mathcal{X}} = env_1|_{\mathcal{X}} \lhd env_2|_{\mathcal{X}}$$

(without proof) □

Auxiliary operations

An operation $inst\colon Template^* \times \mathcal{N}^* \to \mathcal{S}tate$ performing the instantiation of templates is now introduced. This operation constructs an environment instantiating a sequence of templates at a sequence of nodes if possible. If instantiation is not possible, however, *inst* yields *fail*.

Definition 3.3.19 (Operation for template instantiation)
Let $\bar{T} = T_1 T_2 \ldots T_k$ be a sequence of templates, X a variable, and $\bar{\sigma} = \sigma_1 \sigma_2 \ldots \sigma_l$ a sequence of nodes. We define inductively the operations $inst\colon Template^* \times \mathcal{N}^* \to \mathcal{S}tate$ and $inst_1\colon Template \times \mathcal{N} \to \mathcal{S}tate$:

$$inst(T_1 \ldots T_k, \sigma_1 \ldots \sigma_l) \quad =_{def} \quad \begin{cases} inst_1(T_1, \sigma_1) \lhd \ldots \lhd inst_1(T_k, \sigma_k) & \text{if } k = l \\ fail & \text{if } k \neq l \end{cases}$$

$$\begin{aligned} inst_1(X, \sigma) \quad &=_{def} \quad [\sigma/X] \\ inst_1(X \langle T_1 \ldots T_k \rangle, \sigma) \quad &=_{def} \quad inst\big(T_1 \ldots T_k, children(\sigma)\big) \lhd [\sigma/X] \end{aligned}$$

◊

The following proposition shows that *inst* correctly implements template instantiation as defined in Definition 3.2.9 on page 36.

Proposition 3.3.20 (Properties of *inst*)
Let $\bar{T}$ be a sequence of templates, $\bar{\sigma}$ a sequence of nodes, and $env \in \mathcal{S}tate$.

Then the following is true:

(i) if $inst(\bar{T}, \bar{\sigma}) = env \neq fail$, then $\bar{T}$ is instantiated at $\bar{\sigma}$ by env and $Dom(env) = Var(\bar{T})$

(ii) if *env* instantiates $\bar{T}$ at $\bar{\sigma}$, then $inst(\bar{T},\bar{\sigma}) = env|_{Var(\bar{T})}$

(iii) if $inst(\bar{T},\bar{\sigma}) = fail$, then no environment *env* exists that instantiates $\bar{T}$ at $\bar{\sigma}$

(without proof) □

We finally need auxiliary operations for building target trees and for creating anchors for sub transformations.

The function $build\colon Template \times Env \rightarrow \mathcal{T}$ defined first is used to build (target) subtree fragments from templates, using unexpanded subtrees for undefined leaf variables and complete subtrees for defined leaf variables.

Definition 3.3.21
Let T be a template and *env* an environment. We define by induction on the structure of T:

$$build(X, env) \quad =_{def} \quad \begin{cases} subtree([\![X]\!]_{env}) & \text{if} \quad [\![X]\!]_{env} \neq \bot \\ F_{\triangle} & \text{otherwise.} \end{cases}$$

where

$$F = symbol(X)$$

$$build(X\,\langle\, T_1 \ldots T_k \,\rangle, env) \quad =_{def} \quad F(t_1,\ldots,t_k)$$

where

$$\begin{aligned} F &= symbol(X) \\ t_i &= build(T_i, env) \quad \text{for each} \quad 1 \leq i \leq k \end{aligned}$$

◊

Proposition 3.3.22 (Properties of *build*)
Let T be a template, τ a tree node, and *env*, *env'* environments.

(i) If $subtree([\![\xi]\!]_{env}) = subtree([\![\xi]\!]_{env'})$ for all $\xi \in Leaf_Var(T)$, then $build(T, env) = build(T, env')$.

(ii) If *env* instantiates T at τ, then $build(T, env) = subtree(\tau)$

Proof:
see Appendix B.3. □

The operation $anchor\colon \mathcal{V}ar \times Env \rightarrow \mathcal{N}$ creates an anchor node for a sub transformation corresponding to a syntactic constraint $\bar{X} \rightarrow Y/\bar{Z}$. If Y is defined in the given environment that node is returned as the anchor, i.e., the target tree constructed by the sub transformation will replace this node. If Y is not defined, the operation yields the root of a new unexpanded node (which means that a new intermediate tree is constructed through the sub transformation).

Definition 3.3.23
Let Y be a variable where $F = \mathit{symbol}(Y)$, and env an environment. We define:

$$\mathit{anchor}(Y, \mathit{env}) \quad =_{def} \quad \begin{cases} [\![Y]\!]_{env} & \text{if} \quad [\![Y]\!]_{env} \neq \bot \\ F_{\triangle\downarrow\varepsilon} & \text{otherwise.} \end{cases}$$

$\Diamond$

A final helper function constructs the target node at each execution step. This is done by replacing the anchor node or previous intermediate target node by the tree built using the target template and the current environment.

Definition 3.3.24
Let T be a template, γ a tree node, and env an environment. We define:

$$\mathit{target}(T, \gamma, \mathit{env}) \quad =_{def} \quad \mathit{replace}\big(\gamma, \mathit{build}(T, \mathit{env})\big)$$

$\Diamond$

Execution rules

The execution of a transformation according to *spec* is described by a relation $\mathcal{TR}_E \subseteq \mathcal{N}^* \times \mathcal{N} \times \mathcal{N} \times \mathcal{N}^*$ relating a sequence of source nodes and an anchor node to a target node and a sequence of rest nodes. Elements of this relation will be written as $\langle \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle$.

$\mathcal{TR}_E$ will be defined by the means of a rule system. For detailed information about rule systems see, e.g., [57]. We will in particular make use of the notion of derivations induced by rule systems and proofs by induction on the structure of such derivations.

$\mathcal{TR}_E$ is defined using two auxiliary relations: transformations through specific rules are described by $\langle r, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle$; processing of a constraint φ_i of a rule r is described by $\langle \varphi_i, \tau_{i-1}, \mathit{env}_{i-1} \rangle \rightarrow^r \langle \tau_i, \mathit{env}_i \rangle$ where τ_i and env_i represent the current target node and state while executing the rule.

Definition 3.3.25 (Execution rule system)
A rule system for the execution of transformations is defined by four execution rules (**E1**) through (**E4**) introduced below. In the rules, r ranges over transformation rules from *spec* of the form $\bar{S} \rightarrow T/\bar{R}$ `where` $\varphi_1, \ldots, \varphi_k$; $\bar{\sigma}$, $\bar{\sigma}'$, $\bar{\rho}$ and $\bar{\rho}'$ are placeholders for sequences of nodes; τ, τ', γ and γ' are placeholders for single nodes; env, env', env_0, env_1, …, stand for processing states.

(**E1**)
$$\frac{\langle r, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle}{\langle \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle}$$

(E2)

$$\frac{\langle \varphi_1, \tau_0, env_0\rangle \rightarrow^r \langle \tau_1, env_1\rangle \;\ldots\; \langle \varphi_k, \tau_{k-1}, env_{k-1}\rangle \rightarrow^r \langle \tau_k, env_k\rangle}{\langle r, \bar{\sigma}, \gamma\rangle \rightarrow \langle \tau_k, \bar{\rho}\rangle}$$

where

$$\begin{array}{lcl} \tau_0 & = & target(T, \gamma, \bot_{Env}) \\ env_0 & = & inst(\bar{S}, \bar{\sigma}) \lhd inst(T, \tau_0) \\ env_0 & \neq & fail \\ \bar{\rho} & = & [\![\bar{R}]\!]_{env_k} \end{array}$$

(E3)

$$\frac{}{\langle P(\alpha_1, \ldots, \alpha_n), \tau, env\rangle \rightarrow^r \langle \tau, env\rangle}$$

where

$$env \neq fail$$
$$\Vdash_{env} P(\alpha_1, \ldots, \alpha_n)$$

(E4)

$$\frac{\langle \bar{\sigma}', \gamma'\rangle \rightarrow \langle \tau', \bar{\rho}'\rangle}{\langle \bar{X} \rightarrow Y/\bar{Z}, \tau, env\rangle \rightarrow^r \langle \hat{\tau}, \widehat{env}\rangle}$$

where

$$\begin{array}{lcl} env & \neq & fail \\ \bar{\sigma}' & = & [\![\bar{X}]\!]_{env} \\ \gamma' & = & anchor(Y, env) \\ \hat{\tau} & = & target(T, \tau, env \lhd [\tau'/Y]) \\ \widehat{env} & = & env \lhd [\tau'/Y] \lhd [\bar{\rho}'/\bar{Z}] \lhd inst(T, \hat{\tau}) \\ \widehat{env} & \neq & fail \end{array}$$

We will write $\vdash_E \langle \bar{\sigma}, \gamma\rangle \rightarrow \langle \tau, \bar{\rho}\rangle$ iff a derivation rooting in rule (**E1**) exists. Similarly, $\vdash_E \langle \varphi, \tau, env\rangle \rightarrow^r \langle \tau', env'\rangle$ will be written iff a derivation rooting in (**E2**) exists. ◊

3.3.6 Soundness of the execution scheme

We will now show that the execution scheme introduced above is sound w.r.t. the semantics of the transformation model defined in Section 3.2.2. Two auxiliary lemmata necessary for the proof are introduced first.

Lemma 3.3.26
Let r be a rule of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$. Let further $\tau_0, \tau_1, \ldots, \tau_k$ be nodes, and $env_0, env_1, \ldots, env_k$ environments.

If $\langle \varphi_i, \tau_{i-1}, env_{i-1}\rangle \rightarrow^r \langle \tau_i, env_i\rangle$ for each $1 \leq i \leq k$, then the following is true:

(i) env_i instantiates T at τ_i for each $0 \leq i \leq k$

(ii) $env_{i-1} \preceq env_i$ for each $1 \leq i \leq k$

(iii) $[\![\xi]\!]_{env_i} = [\![\xi]\!]_{env_{i+j}}$ for each $\xi \in Def_Var_{i,r} \setminus Var(T)$, $0 \leq i \leq k$, and $0 \leq j \leq k-i$

Proof:
see Appendix B.4. □

Lemma 3.3.27
Let $\bar{\sigma}$ and $\bar{\rho}$ be sequences of nodes. Further, let τ and τ' be nodes.

Then the following is true:

$$\bar{\sigma} \rightarrow \tau/\bar{\rho} \wedge \tau \preceq \tau' \quad \Rightarrow \quad \bar{\sigma} \rightarrow \tau'/\bar{\rho}$$

Proof:
see Appendix B.4. □

We can now show that the execution scheme is sound w.r.t. the semantics of the transformation model.

Theorem 3.3.28 (Soundness of the execution scheme)
Let $r \in spec$ be a transformation rule, $\bar{\sigma}$ and $\bar{\rho}$ sequences of nodes, and γ a single node. Then the following holds:

$$\vdash_E \langle r, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle \quad \Rightarrow \quad \bar{\sigma} \rightarrow_r \tau/\bar{\rho}$$

Proof:
The proof uses induction on the proper sub derivation relation of the E-derivation of $\langle r, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle$. Assume that $\langle r, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle$ and that r has the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$. The derivation D of $\langle r, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle$ then looks like follows:

$$\dfrac{\dfrac{\vdots}{\langle \varphi_1, \tau_0, env_0 \rangle \rightarrow \langle \tau_1, env_1 \rangle} \quad \ldots \quad \dfrac{\vdots}{\langle \varphi_k, \tau_{k-1}, env_{k-1} \rangle \rightarrow \langle \tau_k, env_k \rangle}}{\langle \bar{S} \rightarrow T/\bar{R} \text{ where } \varphi_1, \ldots, \varphi_k, \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle}$$

So $\bar{\sigma} \rightarrow_r \tau/\bar{\rho}$ according to Definition 3.2.11 as the following shows:

(i) $\bar{S}$ is instantiated at $\bar{\sigma}$ by env_k: first observe that $env_0 = inst(\bar{S}, \bar{\sigma}) \triangleleft inst(T, \tau_0)$ and $Var(T) \cap Var(\bar{S}) = \emptyset$. So by Proposition 3.3.20 $\bar{S}$ is instantiated at $\bar{\sigma}$ by env_0. As $Var(\bar{S}) \subseteq Def_Var_{0,r}$ we can conclude that $env_k|_{Var(\bar{S})} = env_0|_{Var(\bar{S})}$ using Lemma 3.3.26 (iii). So $\bar{S}$ is instantiated at $\bar{\sigma}$ also by env_k.

(ii) T is instantiated at τ by env_k: by Lemma 3.3.26 (i) T is instantiated at τ_i by env_i for each $0 \leq i \leq k$, so in particular env_k instantiates T at $\tau_k = \tau$.

(iii) $[\![\bar{R}]\!]_{env} = \bar{\rho}$ according to (**E2**).

(iv) Syntactic constraints are satisfied: consider a constraint φ_i of the form $\bar{X} \to Y/\bar{Z}$. The sub derivation D_i for this constraint has the form

$$\cfrac{\cfrac{\cfrac{\vdots}{\langle r', [\![\bar{X}]\!]_{env_{i-1}}, [\![Y]\!]_{env_{i-1}}\rangle \to \langle \tau', \bar{\rho}'\rangle}}{\langle [\![\bar{X}]\!]_{env_{i-1}}, [\![Y]\!]_{env_{i-1}}\rangle \to \langle \tau', \bar{\rho}'\rangle}}{\langle \bar{X} \to Y/\bar{Z}, \tau_{i-1}, env_{i-1}\rangle \to \langle \tau_i, env_i\rangle}$$

for some τ_i and $env_i = env_{i-1} \lhd inst(T, \tau_i) \lhd [\rho'/\bar{Z}]$. As the sub derivation

$$\cfrac{\vdots}{\langle r', [\![\bar{X}]\!]_{env_{i-1}}, [\![Y]\!]_{env_{i-1}}\rangle \to \langle \tau', \bar{\rho}'\rangle}$$

is a proper sub derivation of D, we know by induction hypothesis that $[\![\bar{X}]\!]_{env_{i-1}} \to_{r'} \tau'/\bar{\rho}'$.

As $Var(\bar{X}) \subseteq Def_Var_{i-1,r}$ and $Var(\bar{Z}) \in Def_Var_{i,r}$ it follows by Lemma 3.3.26 (iii) that $[\![\bar{X}]\!]_{env_i} = [\![\bar{X}]\!]_{env_k} = \bar{\sigma}'$ and $[\![\bar{Z}]\!]_{env_i} = [\![\bar{Z}]\!]_{env_k} = \bar{\rho}'$. By Lemma 3.3.26 (ii) we can conclude that $[\![Y]\!]_{env_i} = \tau' \preceq [\![Y]\!]_{env_k}$

Lemma 3.3.27 implies that $[\![\bar{X}]\!]_{env_k} \to_{r'} [\![Y]\!]_{env_k}/[\![\bar{Z}]\!]_{env_k}$ and a corresponding S-derivation $D_S(\varphi_i)$ exists.

(v) Semantic constraints are satisfied: a constraint φ_i of the form $P(\alpha_1, \ldots, \alpha_l)$ is satisfied in env_i according to (**E3**). As $env_i \preceq env_k$ by Lemma 3.3.26 (ii), we can conclude that φ_i is satisfied also in env_k using Corollary 3.3.8.

So $(\bar{\sigma} \to \tau/\bar{\rho}, env_k)[D_S(\varphi_{i_1}), \ldots, D_S(\varphi_{i_m})]$ is an S-derivation for $\bar{\sigma} \to \tau/\bar{\rho}$. □

We now know that our execution scheme is sound w.r.t. the semantics of the transformation model defined in Section 3.2.2. The next question arising is whether the execution scheme is also *complete*, i.e., the execution scheme also finds *all* possible transformations. So completeness is shown next.

3.3.7 Completeness of the execution scheme

For the execution scheme to be complete it is required that for each rule r, if $\bar{\sigma} \to^r \tau/\bar{\rho}$ for some $\bar{\sigma}$, τ and $\bar{\rho}$, then $\langle \bar{\sigma}, \gamma\rangle \to^r \langle \tau, \bar{\rho}\rangle$ for an appropriate anchor node γ. But what is an appropriate anchor node? The first idea coming to mind may be that γ must be τ replaced by an unexpanded subtree; however, this is to restrictive as not all of τ's context in the complete target tree is required to be known in order to execute a transformation. It turns out that it is sufficient that γ is a predecessor of τ (w.r.t. $\preceq$) and the rule's required attributes have the same value at τ and γ. This is captured in the following, where the notion of a *suitable anchor* is defined.

Definition 3.3.29 (Suitable anchor)
A node $\gamma \in \mathcal{N}$ is called a **suitable anchor** w.r.t. transformation rule r and target node τ iff the following is satisfied:

(i) $\gamma \preceq \tau$

(ii) $att(\gamma, a) = att(\tau, a)$ for each $a \in Requ_Attr(r)$

◊

Note that if τ is a root node, any predecessor $\gamma \preceq \tau$ will be a suitable anchor as no inherited attributes are defined at a root node according to Definition 3.1.7 on page 28.

We will show that in the above situation, where γ is a suitable anchor w.r.t. r and τ, $\langle r, \bar{\sigma}, \gamma \rangle \to \langle \hat{\tau}, \bar{\rho} \rangle$ where $\hat{\tau} = replace(\gamma, subtree(\tau))$.

The following two lemmata help in showing the completeness property.

Lemma 3.3.30
Let T be a template, τ a node, and env and env' environments. Further, the following is true:

(i) $subtree(\tau) = build(T, env)$

(ii) env' instantiates T at τ

Then the following holds for all $\xi \in Leaf_Var(T)$:

$$subtree([\![\xi]\!]_{env'}) = build(\xi, env)$$

Proof:
see Appendix B.5. □

Lemma 3.3.31
Let T be a template, γ, τ and $\hat{\tau}$ nodes. Further, let $\mathcal{X}$ be a set of variables, env and $\widehat{env}$ be environments, and $I \subseteq Inh$.

Further, the following is satisfied:

(i) $\gamma \preceq \tau$

(ii) env instantiates T at τ

(iii) $\hat{\tau} = target(T, \gamma, env|_{\mathcal{X}})$

(iv) $\widehat{env} = env|_{\mathcal{X}} \triangleleft inst(T, \hat{\tau})$

(v) $att(\gamma, a) = att(\tau, a)$ for all $a \in Inh \setminus I$

Then for any $\xi \in \mathcal{X} \cup Var(T)$ and $a \in \mathcal{A}$:

$$\xi.a \in Final_Attr_Occ_{I, Leaf_Var(T) \setminus \mathcal{X}}(T) \quad \Rightarrow \quad [\![\xi.a]\!]_{\widehat{env}} = [\![\xi.a]\!]_{env}$$

Proof:
see Appendix B.5 □

Theorem 3.3.32 (Completeness of the execution scheme)
Let r be a rule from *spec* having the form $\bar{S} \to T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$.

Then for any $\bar{\sigma}, \bar{\rho} \in \mathcal{N}^*$ and $\gamma, \tau, \hat{\tau} \in \mathcal{N}$, where $\hat{\tau} = \mathit{replace}\big(\gamma, \mathit{subtree}(\tau)\big)$ and γ is a suitable anchor w.r.t. r and τ, the following holds:

$$\bar{\sigma} \to_r \tau/\bar{\rho} \quad \Rightarrow \quad \langle r, \bar{\sigma}, \gamma\rangle \to \langle \hat{\tau}, \bar{\rho}\rangle$$

Proof:
The proof is by induction on the structure of the S-derivation of $\bar{\sigma} \to_r \tau/\bar{\rho}$ which has, according to Definition 3.2.11, the following form:

$$(\bar{\sigma} \to_r \tau/\bar{\rho}, \mathit{env})[D_1, \ldots, D_n]$$

We define for each $0 \leq i \leq k$:

$$\begin{aligned} \tau_i &=_{def} \mathit{target}(T, \gamma, \mathit{env}|_{Def_Var_{i,r}}) \\ \mathit{env}_i &=_{def} \mathit{env}|_{Def_Var_{i,r}} \lhd \mathit{inst}(T, \tau_i) \end{aligned}$$

Then the following holds:

(i) $\tau_0 = \mathit{target}(T, \gamma, \bot_{Env})$ and $\mathit{env}_0 = \mathit{inst}(\bar{S}, \bar{\sigma}) \lhd \mathit{inst}(T, \tau_0)$:

As defined above, unfolding the definition of *target* yields:

$$\tau_0 = \mathit{replace}\big(\gamma, \mathit{build}(T, \mathit{env}|_{Def_Var_{0,r}})\big)$$

By Proposition 3.3.22 it follows that

$$\mathit{build}(T, \mathit{env}|_{Def_Var_{0,r}}) = \mathit{build}(T, \mathit{env}|_{Def_Var_{0,r} \cap Leaf_Var(T)})$$

As by LR conditions $\mathit{Leaf_Var}(T) \cap \mathit{Def_Var}_{0,r} = \emptyset$, we can conclude that

$$\begin{aligned} \tau_0 &= \mathit{replace}(\gamma, \mathit{build}(T, \bot_{Env})\big) \\ &= \mathit{target}(T, \gamma, \bot_{Env}) \end{aligned}$$

$\mathit{Def_Var}_{0,r} = \mathit{Var}(\bar{S})$, so $\mathit{env}_0 = \mathit{env}|_{Var(\bar{S})}$. As $\bar{S}$ is instantiated at $\bar{\sigma}$ by *env*, we know by Proposition 3.3.20 that $\mathit{env}|_{Var(\bar{S})} = \mathit{inst}(\bar{S}, \bar{\sigma})$. So

$$\mathit{env}_0 = \mathit{inst}(\bar{S}, \bar{\sigma}) \lhd \mathit{inst}(T, \tau_0)$$

(ii) $\langle \varphi_i, \tau_{i-1}, \mathit{env}_{i-1}\rangle \to^r \langle \tau_i, \mathit{env}_i\rangle$ for $1 \leq i \leq k$:

We first show that all attribute occurrences required to process φ are non-preliminary: by applying Lemma 3.3.31 we can follow that $[\![\xi.a]\!]_{env_{i-1}} = [\![\xi.a]\!]_{env}$ for any $\xi.a \in \mathit{Final_Attr_Occ}_{I,\mathcal{X}}(T)$ where $\mathcal{X} = \mathit{Leaf_Var}(T) \setminus \mathit{Def_Var}_{i-1,r}$ and $\xi \in \mathcal{X} \cup \mathit{Var}(T)$. So for $\xi.a \in \mathit{Requ_Attr_Occ}(\varphi)$

$$[\![\xi.a]\!]_{env_{i-1}} = [\![\xi.a]\!]_{env}$$

We then distinguish cases on the form of constraint φ.

(a) φ has the form $P(\alpha_1, \ldots, \alpha_m)$. We know that $\Vdash_{env} P(\alpha_1, \ldots, \alpha_m)$. For each $1 \leq i \leq m$ we know from above that $[\![\alpha_i]\!]_{env_{i-1}} = [\![\alpha_i]\!]_{env}$ as $\alpha_i \in$ *Requ_Attr_Occ*(φ). Thus, $\Vdash_{env_{i-1}} P(\alpha_1, \ldots, \alpha_m)$. Further, $env_{i-1} = env_i$ as $Def_Var_{i-1} = Def_Var_i$.

So by (**E3**)

$$\langle P(\alpha_1, \ldots, \alpha_m), \tau_{i-1}, env_{i-1}\rangle \rightarrow^r \langle \tau_i, env_i\rangle$$

(b) otherwise, φ is a syntactic constraint of the form $\bar{X} \rightarrow Y/\bar{Z}$. In this case observe first that $[\![\bar{X}]\!]_{env_{i-1}} = [\![\bar{X}]\!]_{env}$ as $Var(\bar{X}) \subseteq Def_Var_{i-1,r}$ in an LR specification. Let $\bar{\sigma}' = [\![\bar{X}]\!]_{env}$, $\tau' = [\![Y]\!]_{env}$, $\gamma' = anchor(Y, env_{i-1})$, and $\bar{\rho}' = [\![\bar{Z}]\!]_{env}$.

If $Y \in Var(T)$, then $[\![Y]\!]_{env_{i-1}} \neq \bot$ as env_{i-1} instantiates T, so by definition of *anchor* $\gamma' = [\![Y]\!]_{env_{i-1}}$ and $Y.a \in$ *Requ_Attr_Occ*(φ) for $a \in$ *Requ_Attr*(r'). So we know from above that $att(\gamma', a) = att(\tau', a)$ for $a \in$ *Requ_Attr*(r').

If, otherwise, $Y \notin Var(T)$, then $[\![Y]\!]_{env_{i-1}} = \bot$ and by definition of *anchor* $\gamma' = F_{\triangle\downarrow\varepsilon}$ where $F = symbol(Y)$. We also know by Definition 3.2.11 that in this case τ' has the form $t_{\downarrow\varepsilon}$ for some tree t. So, by Definition 3.1.7, $att(\gamma', a) = att(\tau') = \bot$ for each $a \in$ *Requ_Attr*$(r') \subseteq Inh$.

Thus, γ' is a suitable anchor w.r.t. r' and τ' and we can assume by induction hypothesis that $\langle\bar{\sigma}', \gamma'\rangle \rightarrow \langle\hat{\tau}', \bar{\rho}'\rangle$ where $\hat{\tau}' = replace\big(\gamma, subtree(\tau')\big)$.

For $j = i-1$ and $j = i$ it is true that $subtree(\tau_j) = build(T, env|_{Def_Var_{j,r}})$ and env_j instantiates T at τ_j, so by Lemma 3.3.30

$$subtree([\![\xi]\!]_{env_j}) = build(\xi, env_{Def_Var_{j,r}})$$

for each $\xi \in Leaf_Var(T)$ and $j \in \{i-1, i\}$. Further, it is true that

$$subtree(\hat{\tau}') = subtree(\tau') = subtree([\![Y]\!]_{env_i})$$

Hence, by Proposition 3.3.22

$$(*) \quad build(T, env_i) = build(T, env_{i-1} \triangleleft [\hat{\tau}'/Y])$$

So

$$\begin{aligned} \tau_i &= target(T, \gamma, env_i) \\ &= replace\big(\gamma, build(T, env_i)\big) \\ &= replace\big(\tau_{i-1}, build(T, env_i)\big) \\ &= replace\big(\tau_{i-1}, build(T, env_{i-1} \triangleleft [\hat{\tau}'/Y])\big) \qquad (*) \\ &= target(T, \tau_{i-1}, env_{i-1} \triangleleft [\hat{\tau}'/Y]) \end{aligned}$$

Further, by Proposition 3.3.18 it follows that:

$$\begin{aligned} env_i &= env|_{Def_Var_{i,r}} \triangleleft inst(T, \tau_i) \\ &= env|_{Def_Var_{i-1,r}} \triangleleft [\tau'/Y] \triangleleft [\bar{\rho}'/\bar{Z}] \triangleleft inst(T, \tau_i) \\ &= env|_{Def_Var_{i-1,r}} \triangleleft inst(T, \tau_{i-1}) \triangleleft [\tau'/Y] \triangleleft [\bar{\rho}'/\bar{Z}] \triangleleft inst(T, \tau_i) \\ &= env_{i-1} \triangleleft [\tau'/Y] \triangleleft [\bar{\rho}'/\bar{Z}] \triangleleft inst(T, \tau_i) \end{aligned}$$

So (**E4**) can be applied and indeed $\langle \varphi_i, \tau_{i-1}, env_{i-1} \rangle \rightarrow^r \langle \tau_i, env_i \rangle$.

(iii) $\tau_k = replace\big(\gamma, subtree(\tau)\big) = \hat{\tau}$ and $[\![R]\!]_{env_k} = [\![R]\!]_{env}$:

By LR conditions we know that $Leaf_Var(T) \subseteq Def_Var_{k,r}$, so using Proposition 3.3.22 we can conclude:

$$\begin{aligned} \tau_k &= target(T, \gamma, env|_{Def_Var_{k,r}}) \\ &= replace\big(\gamma, build(T, env|_{Def_Var_{k,r}})\big) \\ &= replace\big(\gamma, subtree(\tau)\big) \\ &= \hat{\tau} \end{aligned}$$

Further, as by LR conditions $Var(\bar{R}) \subseteq Def_Var_{k,r} \setminus Var(T)$, we can conclude that

$$\begin{aligned} [\![\bar{R}]\!]_{env_k} &= [\![\bar{R}]\!]_{env|_{\left(Def_Var_{k,r} \triangleleft inst(T,\tau_i)\right)}} \\ &= [\![\bar{R}]\!]_{env|_{Def_Var_{k,r}}} \\ &= [\![\bar{R}]\!]_{env} \end{aligned}$$

□

3.3.8 Optimization

The execution scheme described so far does not handle optimization yet. In the following a solution using recursive functions is shown fixing this shortcoming.

A transformation function *trafo* compatible with the execution scheme described in Section 3.3.5, but in addition performing optimization, basically has the functionality

$$trafo\colon \mathcal{N}^* \times \mathcal{N} \rightarrow \mathcal{P}ow(\mathcal{N} \times \mathcal{N}^*)$$

with the following property:

$$(\tau, \bar{\rho}) \in trafo(\bar{\sigma}, \gamma) \quad \Rightarrow \quad \langle \bar{\sigma}, \gamma \rangle \rightarrow \langle \tau, \bar{\rho} \rangle$$

Note that in particular for a transformation function producing *optimal* transformation results the reverse direction $\Leftarrow$ is not true as optimization obviously constrains the transformation relation.

The function *trafo* can be recursively defined using functions $trafo_r$ obtaining transformation results through a specific transformation rule r working as follows:

- compute all transformation results obtained from application of r using only optimal sub transformation results (obtained by recursively applying *trafo*)
- if an objective function is given for r, filter for those minimizing r's objective function among all results

The function $trafo_r$ given below computing all transformation results obtained from application of rule r was mechanically derived from the execution scheme's rule (**E2**) (cf. Section 3.3.5) and filtering for optimal results was added. Note that in order to make things a little easier *trafo* has as an additional parameter a signature. Also note that two auxiliary functions $constrs_r$ and $constr_r$ are used to handle the processing of constraints, and the auxiliary function *minimize* handles optimization. Throughout the following function definitions r is a placeholder for a rule of the form $\bar{S} \to T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$.

$$trafo\colon Sig \times \mathcal{N}^* \times \mathcal{N} \to \mathcal{P}ow(\mathcal{N} \times \mathcal{N}^*)$$
$$trafo(sig, \bar{\sigma}, \gamma) \quad =_{def} \quad \{trafo_r(\bar{\sigma}, \gamma) \mid r \in spec \ \wedge \ sig(r) = sig\}$$

$$trafo_r\colon \mathcal{N}^* \times \mathcal{N} \to \mathcal{P}ow(\mathcal{N} \times \mathcal{N}^*)$$
$$trafo_r(\bar{\sigma}, \gamma) \quad =_{def} \quad \big\{(\tau, [\![\bar{R}]\!]_{env}) \mid (\tau, env) \in E)\big\}$$
where
$$\begin{aligned}
E &= minimize(obj(r), E') \\
E' &= \big\{(\tau, env) \in E'' \mid env \neq fail\big\} \\
E'' &= constrs_r(\varphi_1\varphi_2 \ldots \varphi_k, E_0) \\
E_0 &= \{(\tau_0, env_0)\} \\
\tau_0 &= target(T, \gamma, \perp_{Env}) \\
env_0 &= inst(\bar{S}, \sigma) \lhd inst(T, \tau_0)
\end{aligned}$$

The processing of constraints is realized by the following functions.

$$constrs_r\colon Constraint^* \times \mathcal{P}ow(\mathcal{N} \times \mathcal{S}tate) \to \mathcal{P}ow(\mathcal{N} \times \mathcal{S}tate)$$
$$constrs_r(\varphi_1 \ldots \varphi_k, E) \quad =_{def} \quad E_k$$
where
$$E_i = constr_r(\varphi_i, E_{i-1}) \quad \text{for} \quad 1 \leq i \leq k$$

$$constr_r\colon Constraint \times \mathcal{P}ow(\mathcal{N} \times \mathcal{S}tate) \to \mathcal{P}ow(\mathcal{N} \times \mathcal{S}tate)$$
$$constr_r\big(P(\alpha_1, \ldots, \alpha_n), E\big) \quad =_{def}$$
$$\big\{(\tau, env) \in E \mid env \neq fail \wedge \Vdash_{env} P(\alpha_1, \ldots, \alpha_n)\big\}$$
$$constr_r(\bar{X} \to Y/\bar{Z}, E) \quad =_{def}$$
$$\big\{makeResult(\tau'', \bar{\rho}', \tau, env) \mid (\tau, env) \in E \ \wedge \ env \neq fail \ \wedge$$
$$(\tau', \bar{\rho}') \in trafo\big(sig, [\![\bar{X}]\!]_{env}, anchor(Y, env)\big)\big\}$$
where
$$\begin{aligned}
sig &= sig(\bar{X} \to Y/\bar{Z}) \\
makeResult(\tau', \bar{\rho}', \tau, env) &= (\hat{\tau}, \widehat{env}) \\
\hat{\tau} &= target(T, \tau, env \lhd [\tau'/Y]) \\
\widehat{env} &= env \lhd [\tau'/Y] \lhd [\bar{\rho}'/\bar{Z}] \lhd inst(T, \hat{\tau})
\end{aligned}$$

Minimization for an objective function is realized using the following function:

$$
\begin{aligned}
& \mathit{minimize}\colon (\mathit{Obj} \ \cup \ \{\bot\}) \times \mathcal{P}ow((\mathcal{N} \times \mathit{Env})) \to \mathcal{P}ow((\mathcal{N} \times \mathit{Env})) \\
& \mathit{minimize}(\bot, E) \quad =_{def} \quad E \\
& \mathit{minimize}\big(f(\alpha_1, \ldots, \alpha_k), \emptyset\big) \quad =_{def} \quad \emptyset \\
& \mathit{minimize}\big(f(\alpha_1, \ldots, \alpha_k), E\big) \quad =_{def} \\
& \quad \big\{(\tau, \mathit{env}) \in E \mid f([\![\alpha_1]\!]_{env}, \ldots, [\![\alpha_k]\!]_{env}) = \mathit{min}(M)\big\} \\
& \quad \text{where} \\
& \qquad M = \big\{f([\![\alpha_1]\!]_{env}, \ldots, [\![\alpha_k]\!]_{env}) \mid (\tau, \mathit{env}) \in E\big\}
\end{aligned}
$$

The above functions add optimization to our execution scheme. It should be noted that the transformation result obtained by applying *trafo* can be undefined if the evaluation does not terminate. Assuming termination though, and incorporating our results about soundness and completeness, the following holds for given rule r, $\bar{\sigma} \in \mathcal{N}^*$, $\tau \in \mathcal{N}$, and $\bar{\rho} \in \mathcal{N}^*$, and anchor node γ which is suitable w.r.t. r and τ:

$$(\tau, \bar{\rho}) \in \mathit{trafo}_r(\bar{\sigma}, \gamma) \quad \Leftrightarrow \quad \bar{\sigma} \to_{r,opt} \tau/\bar{\rho}$$

The functions as defined above can almost directly be used to implement the transformation model. This is shown in Appendix E where a concrete implementation in the functional programming language `Haskell` is given.

3.3.9 Application of dynamic programming

A remaining problem with the execution scheme described so far is that in the course of an optimizing transformation the situation will often arise where common sub transformations will be executed more than once. To see this, consider as an example the line breaking example from Chapter 2. While computing the sequence of lines for a *BoxKerfSeq*, it might be possible to first choose a shorter first line followed by a longer second line, or choose a longer first line followed by a shorter second line, in both cases leaving the same rest box kerf sequence. The transformation result for the rest box kerf sequence is the same in both cases and it is of course desirable to compute it only once. With the functional solution described above, however, it will be computed twice.

A solution to this problem is the application of **dynamic programming** where computed solutions to sub problems are kept in a table for later reuse. In our case the subtree of a previously computed target node can be used to replace a different anchor node in a later executed transformation.

Some further investigation is needed to see when a transformation result can be reused. So assume that $\bar{\sigma} \to^r \tau_1/\bar{\rho}$ and a second (potential) target node τ_2 where $\mathit{subtree}(\tau_2) = \mathit{subtree}(\tau_1)$. Under what circumstances do we know that also $\bar{\sigma} \to^r \tau_2/\bar{\rho}$? The crucial point here is that a transformation may depend on the attribution of the target subtree, so it is also necessary that the attribution of the subtrees of τ_1 and τ_2 is the same as far as the attributes required for the applied transformation rule are concerned.

So, when is the attribution a subtree the same when occurring in two target trees? With attribute grammars the attribution of a subtree only depends on the values of the

inherited attributes at its root. More precisely, if a subset of the inherited attributes at the root is known to evaluate to the same value in two contexts the attributes depending only on inherited attributes from that set evaluate to the same value in both contexts too. Carried over to the attribution functions used in this chapter, his property is captured by the axiom defined in the following.

Definition 3.3.33
For all σ, σ', pos, a:

(**Attr2**)
$$\begin{array}{l}\big(subtree(\sigma) = subtree(\sigma') \wedge \\ \Big(\forall b \in \mathcal{A} \,.\, (\sigma.b, \sigma_{\downarrow pos}.a) \in Dep \Rightarrow att(\sigma, b) = att(\sigma', b)\big)\Big) \\ \qquad \Rightarrow \quad att(\sigma_{\downarrow pos}, a) = att({\sigma'}_{\downarrow pos}, a)\end{array}$$

$\Diamond$

It is assumed in the following that (**Attr2**) holds. We are now ready to formulate a theorem lying the basics for the application of dynamic programming.

Theorem 3.3.34
Let $\bar{\sigma}$ and $\bar{\rho}$ be sequences of nodes, τ_1 and τ_2 be nodes, and r be a transformation rule of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$.

Further, the following holds:

(i) $\bar{\sigma} \rightarrow^r \tau_1/\bar{\rho}$

(ii) $subtree(\tau_1) = subtree(\tau_2)$

(iii) $att(\tau_1, a) = att(\tau_2, a)$ for $a \in Requ_Attr(r)$

Then the following holds:

(i) $\bar{\sigma} \rightarrow^r \tau_2/\bar{\rho}$.

(ii) if $\bar{\sigma} \rightarrow_{r,opt} \tau_1/\bar{\rho}$, then also $\bar{\sigma} \rightarrow_{r,opt} \tau_2/\bar{\rho}$

Proof:

(i) The proof is by induction on the structure of the S-derivation of $\bar{\sigma} \rightarrow^r \tau_1/\bar{\rho}$. So let $(\bar{\sigma} \rightarrow^r \tau_1/\bar{\rho}, env_1)[D_1, \ldots, D_n]$ be an S-derivation. We define $env_2 =_{def} env_1 \lhd inst(T, \tau_2)$ (Note that $env_2 \neq fail$ as $subtree(\tau_2) = subtree(\tau_1)$ and T is instantiated at τ_1).

Then an S-derivation $(\bar{\sigma} \rightarrow^r \tau_2/\bar{\rho}, env_2)[D'_1, \ldots, D'_n]$ exists as the following shows.

First consider the required attribute occurrences $\xi.a$ for all constraints $\varphi_1, \ldots, \varphi_k$. If $\xi \notin Var(T)$, then $[\![\xi]\!]_{env_1} = [\![\xi]\!]_{env_2}$, thus $[\![\xi.a]\!]_{env_1} = [\![\xi.a]\!]_{env_2}$. Otherwise $T_{\downarrow pos} = \xi$ for some pos. Suppose that $(\tau.b, \tau_{\downarrow pos}.a) \in Dep$ for some attribute b. Then by Definition 3.3.9 $(T_{\downarrow \varepsilon}.b, \xi.a) \in Dep(T)$ and thus by Definition 3.3.13

$b \in \mathit{Requ_Attr}(r)$. By premises it follows that $att(\tau_1, b) = att(\tau_2, b)$ and thus by (**Attr2**) $att(\tau_{1\downarrow pos}, a) = att(\tau_{2\downarrow pos}, a)$, so finally $[\![\xi.a]\!]_{env_1} = [\![\xi.a]\!]_{env_2}$.

We can now construct an S-derivation for $\bar{\sigma} \rightarrow^r \tau_2$ using env_2.

(a) $\bar{S}$ is instantiated at $\bar{\sigma}$ by env_2 as by LR conditions $Var(\bar{S}) \cap Var(T) = \emptyset$ and thus $env_2|_{Var(\bar{S})} = env_1|_{Var(\bar{S})}$

(b) T is instantiated at τ_2 by env_2

(c) $[\![\bar{R}]\!]_{env_2} = [\![\bar{R}]\!]_{env_1}$ as by LR conditions $Var(\bar{R}) \cap Var(T) = \emptyset$, so $env_2|_{Var(\bar{R})} = env_1|_{Var(\bar{R})}$

(d) $\Vdash_{env_2} \varphi$ for any semantic constraint $\varphi \in \{\varphi_1, \ldots, \varphi_k\}$ of the form $P(\alpha_1, \ldots, \alpha_n)$:

We know that $\Vdash_{env_1} P(\alpha_1, \ldots, \alpha_n)$. Further, we know from above that $[\![\alpha_i]\!]_{env_1} = [\![\alpha_i]\!]_{env_2}$ for each $1 \leq i \leq n$. So also $\Vdash_{env_2} P(\alpha_1, \ldots, \alpha_n)$.

(e) for any syntactic constraint $\varphi \in \{\varphi_1, \ldots, \varphi_k\}$ of the form $\bar{X} \rightarrow Y/\bar{Z}$:

Let $\bar{\sigma}' = [\![\bar{X}]\!]_{env_1} = [\![\bar{X}]\!]_{env_2}$ and $\bar{\rho}' = [\![\bar{R}]\!]_{env_1} = [\![\bar{R}]\!]_{env_2}$.

We distinguish two cases.

i. $T_{\downarrow pos} = Y$ for some pos. Then by Definition 3.2.9 $[\![Y]\!]_{env_1} = \tau_{1\downarrow pos}$ and $[\![Y]\!]_{env_2} = \tau_{2\downarrow pos}$. Let $\tau_1' = \tau_{1\downarrow pos}$, and $\tau_2' = \tau_{2\downarrow pos}$.

We know that $\bar{\sigma}' \rightarrow^{r'} \tau_1'/\bar{\rho}'$. From above we further know that $att(\tau_1', a) = att(\tau_2', a)$ for $a \in \mathit{Requ_Attr}(r')$. Obviously it is also true that $subtree(\tau_1') = subtree(\tau_2')$.

So by induction hypothesis $\bar{\sigma}' \rightarrow^{r'} \tau_2'/\bar{\rho}'$ and a corresponding S-derivation D_i' exists.

ii. Otherwise $Y \notin Var(T)$. In this case $[\![Y]\!]_{env_1} = [\![Y]\!]_{env_2}$. Let $\tau' = [\![Y]\!]_{env_1}$.

We know that $\bar{\sigma}' \rightarrow^{r'} \tau'/\bar{\rho}'$ for some rule r'.

(ii) if $\bar{\sigma} \rightarrow_{r,opt} \tau_1/\bar{\rho}$, then the S-derivation of $\bar{\sigma} \rightarrow^r \tau_1/\bar{\rho}$ is optimal. As can easily be checked the S-derivation for $\bar{\sigma} \rightarrow^r \tau_2/\bar{\rho}$ constructed in (i) is then optimal as well.

□

This theorem can be applied in the situation when a transformation $\bar{\sigma} \rightarrow^r \tau/\bar{\rho}$ has been previously performed and a new transformation with the same source nodes and some new anchor node is to be performed. The following corollary allows to decide if τ's subtree can be used as result for the new transformation.

Corollary 3.3.35
Let $\bar{\sigma}$ and $\bar{\rho}$ be sequences of nodes, τ and γ be nodes, and r a transformation rule.

Let the following be satisfied:

(i) $\bar{\sigma} \rightarrow^r \tau/\bar{\rho}$

(ii) $subtree(\gamma) \preceq subtree(\tau)$

(iii) $att(\tau, a) = att(\gamma, a)$ for all $a \in Requ_Attr(r)$

Then the following holds:

(i) $\bar{\sigma} \rightarrow^r \tau'/\rho$

where

$$\tau' = replace\big(\gamma, subtree(\tau)\big)$$

(ii) if $\bar{\sigma} \rightarrow_{r,opt} \tau/\bar{\rho}$, then also $\bar{\sigma} \rightarrow_{r,opt} \tau'/\bar{\rho}$

Proof:
By definition of τ' it immediately follows that $\gamma \preceq \tau'$. So, for any $a \in Requ_Attr(r)$

$$\begin{aligned} att(\tau', a) &= att(\gamma, a) && (\textbf{Attr1}) \\ &= att(\tau, a) && (\text{premise}) \end{aligned}$$

We also know that $subtree(\tau) = subtree(\tau')$.

Thus both (i) and (ii) are direct consequences of Theorem 3.3.34. □

With the application of dynamic programming together with the transformation functions described in Section 3.3.8 we now have a means to efficiently perform optimizing transformations of attributed trees.

3.3.10 Automatic construction of LR specifications

Section 3.3.4 introduced the notion of LR specifications which imposes restrictions on the order of constraints in rules. These restrictions were shown to allow efficient execution of transformations processing constraints strictly from left to right.

In the following an algorithm is given allowing to turn a possibly non-LR specification into an equivalent LR specification, if possible, by reordering each rule's constraints.

The algorithm takes one rule at a time and works iteratively maintaining a set of remaining constraints (starting off with the set of all of the rule's constraints), a set of currently preliminary attribute occurrences and a set of currently defined node variables. In each iteration step a constraint whose required attribute occurrences are non-preliminary is picked. If no such constraint exists an equivalent LR specification does not exist and an error is issued. The algorithm which must be applied to each of a specification's rules is as follows:

Input: a rule r of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$
Output: a new rule $\bar{S} \rightarrow T/\bar{R}$ where $\varphi'_1, \ldots, \varphi'_k$ which satisfies
the LR conditions and where $\{\varphi_1, \ldots, \varphi_k\} = \{\varphi'_1, \ldots, \varphi'_k\}$, or an error

$I := Inh \setminus Requ_Attr(r)$
$C := \{\varphi_1, \ldots, \varphi_k\}$
$DV := Var(\bar{S})$
$PAO := Prelim_Attr_Occ_{I,DV}(T)$

for $i = 1, 2, \ldots, k$
do
  if $\exists \varphi \equiv P(\alpha_1, \ldots, \alpha_n) \in C \,.\, \mathit{Requ_Attr_Occ}(\varphi) \cap \mathit{PAO} = \emptyset$
  then
    $\varphi'_i := \varphi$
    $C := C \setminus \{\varphi\}$
  else if $\exists \varphi \equiv (\bar{X} \rightarrow Y/\bar{Z}) \in C \,.\, \mathit{Requ_Attr_Occ}(\varphi) \cap \mathit{PAO} = \emptyset$
  then
    $\varphi'_i := \varphi$
    $C := C \setminus \{\varphi\}$
    $DV := DV \;\cup\; \mathit{Def_Var}(\varphi)$
    $\mathit{PAO} := \mathit{Prelim_Attr_Occ}_{I,DV}(T)$
  else
    error
  fi
od
output $\bar{S} \rightarrow T/\bar{R}$ where $\varphi'_1, \ldots, \varphi'_k$

Note that the algorithm chooses semantic constraints as early as possible. This way unnecessary sub transformations which are potentially expensive are avoided.

3.4 Comparison with existing approaches

In Chapter 1 it has been pointed out that existing optimizing formatters have up to now only been described in operational manners and have been implemented using general-purpose programming languages. While the new approach described here is more declarative and makes it a lot easier to create such formatters, the question may arise how the execution of a transformation in the new model relates to existing solutions. So, as an example, a comparison of the approach to line breaking described in [9, 35] with the approach taken here will be made.

As described in [9, 35], line breaking with global optimization can be reduced to the problem of finding the shortest path between two nodes of a graph:

- the input is a sequence of boxes and kerfs as already described in Chapter 2
- a graph is built where nodes correspond to kerfs and
  - a starting node S corresponds to the kerf at the beginning of the input box kerf sequence (note, however, that in the description given in Chapter 2 this kerf at the beginning was omitted)
  - an edge between two kerfs k_1 and k_2 exists iff k_2 follows k_1 and the sub box kerf sequence between k_1 and k_2 can be stretched or shrunken such that the target line width can be attained, i.e., a consistent line is represented by the sub sequence
  - the complete graph is built incrementally, beginning with the starting node and adding all edges as necessary

– an end node E exists, corresponding to the kerf terminating the input box kerf sequence

- cost measuring the badness, i.e., the required amount of stretching and shrinking, is associated with each edge

Any path from S to E in such a graph describes a consistent line break in the sense that for each line maximum stretchability or shrinkability are not exceeded.

Figure 3.13 shows an example graph for an imaginary paragraph that can be broken into two variants of three and one variant of two lines. All edges are marked with a label l_{ij} for later reference, and with cost as described above. Assuming that the nodes are arranged horizontally from left to right in the order in which they appear in the input box-kerf sequence, the path l_{11} l_{12} l_{13} corresponds to the first fit line break, and l_{31} l_{32} corresponds to the last fit line break as described in Chapter 2. Taking into account the cost associated with each line, l_{21} l_{22} l_{13} (the shortest path from S to E) corresponds to the globally optimal linebreak. The locally optimal line break results from choosing directly the best next line (or edge) from each point and happens to coincide with the first fit line break in this example.

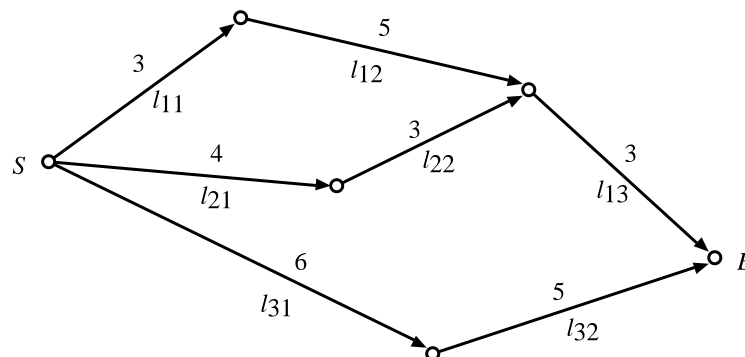


Figure 3.13: Example of a graph describing possible line breaks

We now turn to the corresponding tree transformation according to our new transformation model. The transformation specification describing line breaking was given in Example 3.2.7 on page 35. The source of the transformation is a tree representing the input box kerf sequence (which results from folding a flat input sequence to the right) is depicted in Figure 3.14 on the next page. Intermediate nodes which will not occur as allowable break points have been omitted. Figure 3.15 on page 73 shows the target trees created while executing the transformation specification. Note that the subtrees of $LineSeq_{13}$ and $LineSeq_{23}$ are identical which corresponds to the fact that the two paths in the example graph in Figure 3.13 share the same trailing edge. By using the results that have been obtained in Section 3.3.9 dynamic programming can be applied and this subtree will be computed only once, though, and will be copied when it is needed the second time.

The transformation is performed as an invocation of a transformation $\sigma_0 \rightarrow \tau_0/\varepsilon$ where τ_0 is the root of an unexpanded tree of symbol *LineSeq*. The only rule that can be applied is r_2. This is how the transformation proceeds:

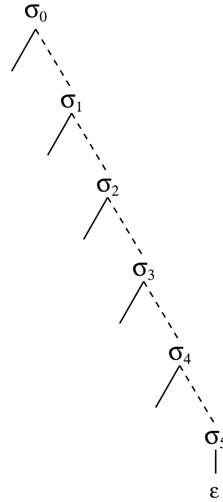


Figure 3.14: Transformation source

- The initial target tree is expanded to the tree (a) shown in Figure 3.16 on page 74 by expanding the target template.
- A sub transformation $\sigma_0 \rightarrow \mathit{Line}_1/\rho$ is invoked yielding three results $(\mathit{Line}_{11}, \sigma_1)$, $(\mathit{Line}_{21}, \sigma_2)$ and $(\mathit{Line}_{31}, \sigma_3)$, each consisting of a possible first line and a remaining input sequence. The results are shown in target trees (b), (c) and (d) in Figure 3.16.
- For each of the three results the transformation proceeds with a second sub transformation $\sigma_i \rightarrow \mathit{LineSeq}_{i2}/\varepsilon$ for $i = 1, 2, 3$. Each of those sub transformations are carried out in a similar way as the root transformation described here:
  - The sub transformation for $i = 1$ proceeds by constructing $(\mathit{Line}_{12}, \sigma_4)$ (only a single consistent alternative exists), and in a further nested sub transformation $(\mathit{Line}_{13}, \sigma_5)$ (again, only one consistent alternative exists).
  - The sub transformation for $i = 2$ proceeds by constructing $(\mathit{Line}_{22}, \sigma_4)$ (only a single consistent alternative exists). At this point, a further sub transformation processing the rest input sequence σ_4 is invoked. However, this sub transformation has already been carried out, so its result will be reused (dynamic programming). The result is $(\mathit{Line}_{23}, \sigma_5)$ where the subtree of Line_{23} is simply copied from Line_{13}.
  - For $i = 3$ the sub transformation creates as the single consistent alternative the line Line_{32}, leaving the empty rest input sequence σ_5.
- The results of the transformation so far are the trees shown in Figure 3.15 on the next page
- Finally, minimization of cost is done. Cost is here attached to nodes of symbol *LineSeq* as a synthesized attribute. The result is $\mathit{LineSeq}_{21}$ shown in Figure 3.16 which corresponds to the result also obtained using the traditional approach.

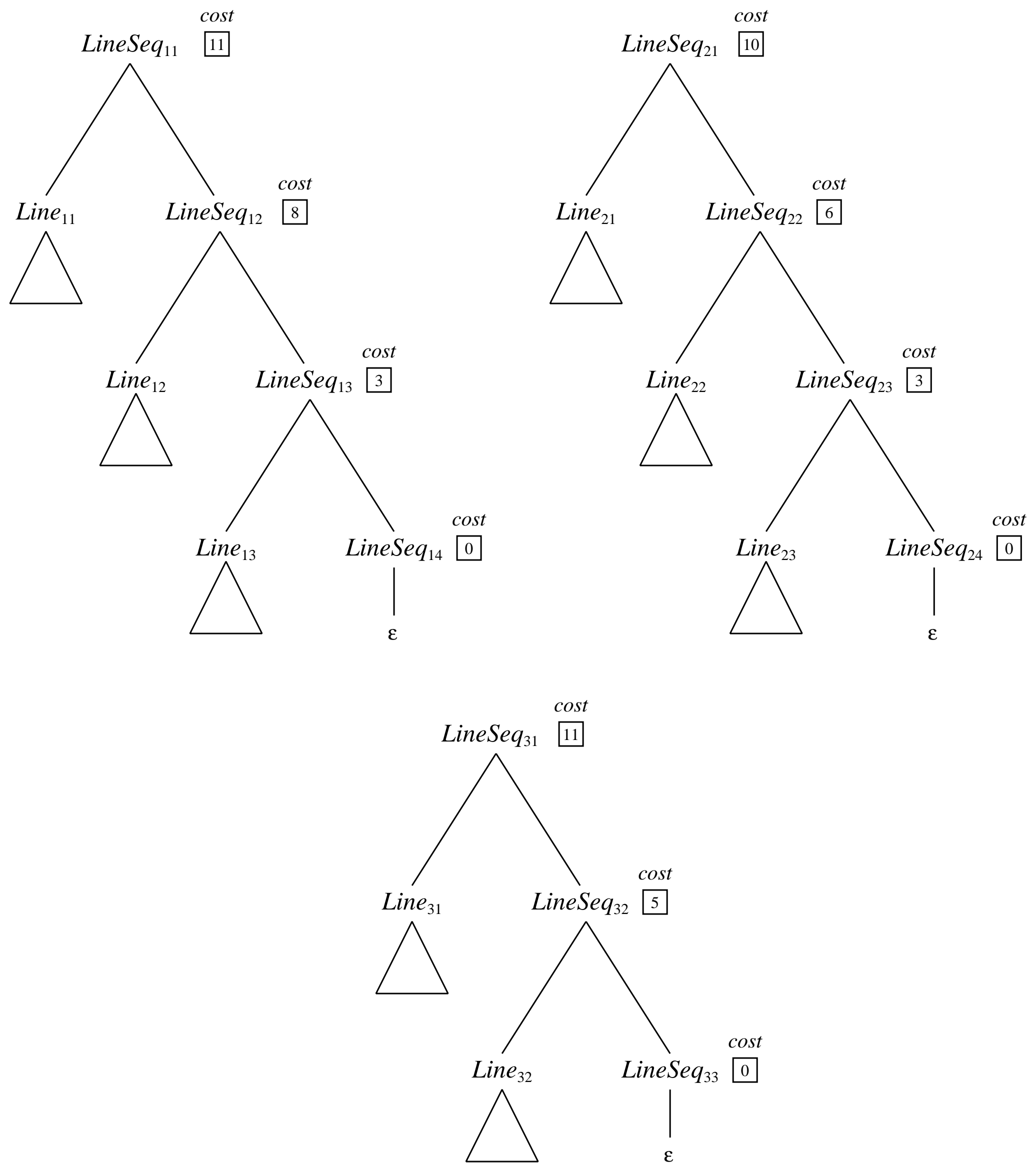


Figure 3.15: Target trees created during transformation

An imaginary graph resembling the one used in the traditional approach actually occurs hidden in the transformation process as well. The nodes in that graph correspond to sub transformations here. The graph for the example transformation is shown in Figure 3.17 on the following page. Implicitly, also the finding of the shortest path happens.

As a last note it should be mentioned that the efficiency of both approaches is basically the same. A little overhead is caused by manipulating trees, though.

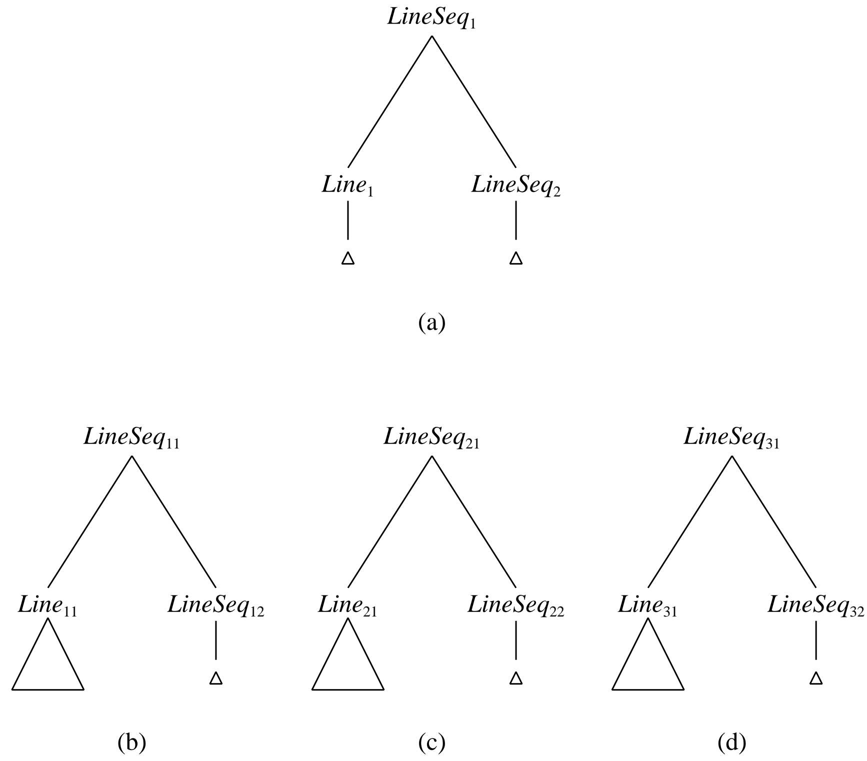


Figure 3.16: Evolution of target trees

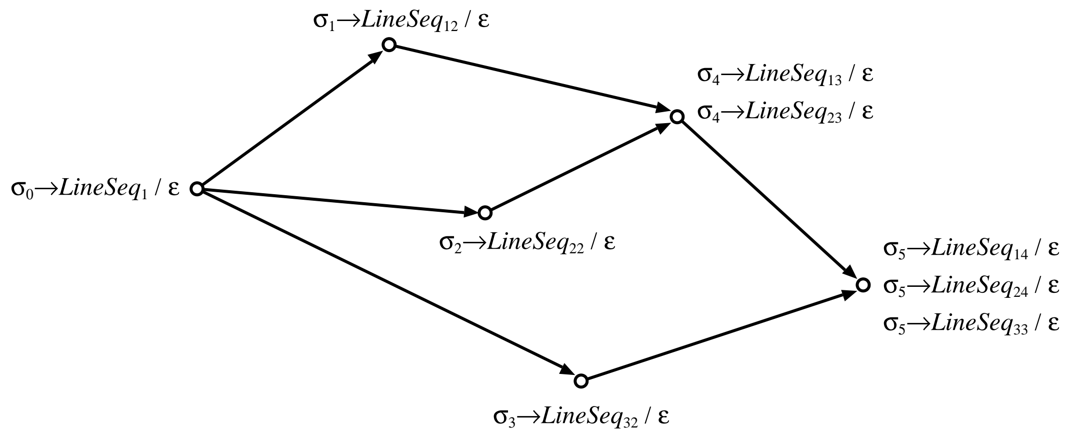


Figure 3.17: Transformation process as a graph

3.5 Conclusion

In this chapter we have developed a formal model for transformations between attributed trees and the model's formal semantics. We have further developed an execution scheme describing how transformations can be efficiently performed for the non-optimizing case.

We have proven this operational semantics to be sound and complete. We have then developed recursive functions performing optimizing transformations and have shown how dynamic programming can be applied further improving the efficiency.

In the following chapters a practical language realizing these theoretical concepts will be introduced and some example applications will be shown.

CHAPTER 4

The Coala system

This chapter presents a new specification language for document formatters based on the theoretical model for optimizing tree transformations developed in Chapter 3, and its runtime environment. The new system is called **Coala** which stands for Constraint-based approach to document layout. Coala has been implemented in Java.

Coala's specification language provides means to specify attribute grammars and optimizing transformation relations. The Coala system allows to load specifications, and to generate interpreters for them. An interpreter is then used to build structures of attributed trees and perform transformations between them. Further, the Coala system provides tools for reading document structures from XML files, and for output of PostScript code for a built-in page layout format, ready for printing and viewing. Thus, Coala provides a complete set of tools for realizing practical document formatters.

In Section 4.1 we will first take a look at the most important elements of Coala's specification language. Then, in Section 4.2, an overview over Coala's runtime system will be given.

4.1 Coala's specification language

Coala provides language elements for specifying attribute grammars (i.e., context-free grammars together with attribute declarations and attribution rules), and rules describing optimizing transformations of attributed trees. Those language elements are described in the following. Note that the complete context-free syntax is summarized in Appendix D.

After introducing the syntax notation used in this chapter, we will first define Coala's lexical conventions in Section 4.1.2. Thereafter, basic syntactic elements of specifications are defined. Section 4.1.4 introduces the syntax and semantics of Coala's expressions.

In Section 4.1.6 through Section 4.1.11 the structure of specifications is then described. Section 4.1.12 shows an example specification.

4.1.1 Syntax notation

In this chapter's description of Coala's specification language a variant of extended `BNF` notation is used:

- keywords (literal sequences of letters) are shown in typewriter font, e.g., **if**, **let**, etc.
- literal character sequences containing special symbols are enclosed in apostrophes, e.g., '`<`', '`>=`', etc.
- nonterminal symbols appear in italic font, e.g., *specification*, *rule*, etc.
- on the right hand side of productions regular expressions composed using the following operators (listed in order of decreasing binding strength) are used:
  - $[r]$ stands for an optional r
  - r^* stands for zero or more repetitions of r
  - r^+ stands for one or more repetitions of r
  - $r_1\, r_2$ stands for r_1 followed by r_2
  - $r_1|r_2$ stands for r_1 or r_2

  where r, r_1 and r_2 are regular expressions. Atomic regular expressions are keywords, literal character sequences and nonterminal symbols. Sub expressions can be enclosed in braces to enforce grouping.

4.1.2 Lexical structure

A specification is at the lowest level a sequence of characters which is in a first step transformed into a sequence of syntactic tokens. Tokens of the following classes are recognized: identifiers, relation identifiers, keywords, constants, operators and separators. White space (blanks, tabulators, newline and carriage-return characters) and comments delimit tokens and are otherwise ignored.

The sequence of tokens is produced in an unambiguous way by following the convention that at a given position in the stream of input characters the next token is always chosen to be the one constituting of the longest sequence of characters. If a sequence of an equal number of characters is matched by multiple rules, the first applicable rule in the following description is used.

In order to ease the definition of some of the token classes, the following definitions of auxiliary lexical classes are first introduced.

letter	::=	'`_`' | '`a`' | '`b`' | ... | '`z`' | '`A`' | '`B`' | ... | '`Z`'
digit	::=	'`0`' | '`1`' | ... | '`9`'
letter_or_digit	::=	*letter* | *digit*
hex_digit	::=	*digit* | '`a`' | '`b`' | ... | '`f`' | '`A`' | '`B`' | ... | '`F`'

Comments. Like in `Java` or `C++`, comments either begin with the characters '`/*`' and end with '`*/`' or begin with the characters '`//`' and extend to the end of the current line. In the first form comments cannot be nested.

Keywords. The following is a list of Coala's keywords: **specification**, **include**, **terminal**, **productions**, **inherited**, **synthesized**, **with**, **fct**, **true**, **false**, **nil**, **if**, **then**, **else**, **let**, **in**, **attr**, **rule**, **where**, and **minimize**.

Natural numbers. Natural numbers including zero are specified in decimal format.

nat_const ::= *digit*$^+$

It is an error to specify a number that exceeds the implementation-defined range of natural numbers.

Floating point numbers. (Positive) floating point numbers can be specified as follows.

real_const ::= *digit** '`.`' *digit*$^+$ [*exponent*]
exponent ::= ('`e`' | '`E`') ['`+`' | '`-`'] *digit*$^+$

If the represented number exceeds the implementation's range or precision of floating point values the number is truncated or rounded as necessary.

Dimension constants. Coala supports dimensions which are specified by a decimal number followed by a unit of measure. Supported units of measure are ☞*point*, centimeter, millimeter and inch.

dim_const ::= *nat_const measure_unit*
| *real_const measure_unit*
measure_unit ::= **pt** | **cm** | **mm** | **in**

String constants. String constants have the following syntax:

string_const ::= '`"`' (*string_char*|*escape_sequence*)* '`"`'
string_char ::= every character except '`\`', '`"`'
escape_sequence ::= '`\`' ('`b`' | '`t`' | '`n`' | '`f`' | '`r`' | '`"`' | '`\`')
| '`\u`' *hex_digit hex_digit hex_digit hex_digit*

String constants can include—besides raw characters—escape sequences for special characters including all ☞*Unicode* characters specified by their 16-bit value in hexadecimal format.

Identifiers. Identifiers are defined as follows.

ident	::=	*letter letter_or_digit**

Relation identifiers. Coala allows, as an extension to the model described in Chapter 3, to assign names to relations, like for instance *ls* or *l* used in the introductory line breaking example described in Section 2.3. Relation identifiers are preceded by an ampersand character. ‘`@@`’ denotes a special default relation identifier.

rel_ident	::=	‘`@`’ *ident*
	|	‘`@@`’

Operators. The following are Coala’s operators: ‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’, ‘`<`’, ‘`>`’, ‘`<=`’, ‘`>=`’, ‘`==`’, ‘`!=`’, ‘`&&`’, ‘`||`’, ‘`!`’.

Separators. The list of Coala’s separators is: ‘`=`’, ‘`,`’, ‘`;`’, ‘`:`’, ‘`|`’, ‘`(`’, ‘`)`’, ‘`{`’, ‘`}`’, ‘`[`’, ‘`]`’, ‘`->`’, ‘`<`’, ‘`>`’, ‘`::=`’, ‘`'`’, ‘`.`’.

This concludes the description of lexical structure. What follows now is Coala’s context-free syntax.

4.1.3 Basic language elements

In the following basic language elements are introduced which can occur in several parts in a specification.

Node variables

Node variables occur in transformation rules’ templates and in attribute occurrences (cf. Chapter 3). A node variable consists of an identifier referring to a syntactic symbol and an optional extension in the form of a number or identifier enclosed in square brackets to disambiguate multiple occurrences of the syntactic symbol in productions or templates.

node_var	::=	*ident*
	|	*ident* ‘`[`’ *natconst* ‘`]`’
	|	*ident* ‘`[`’ *ident* ‘`]`’

Attribute occurrences

Attribute occurrences can appear in attribute equations, semantic constraints and minimize clauses, and consist of a node variable and an attribute name (cf. Chapter 3).

attr_occ	::=	*node_var* ‘`.`’ *ident*

Lists of identifiers

In several places Coala requires comma-separated lists of identifiers.

```
ident_list  ::=  ident
              |  ident_list ‘,’ ident
```

4.1.4 Expressions

Coala supports expressions in attribution rules and transformation rules. The expressions supported are those expected from a simple untyped first-order functional language.

```
expr            ::=  if expr then expr else expr
                  |  let ident ‘=’ expr in expr
                  |  expr where local_def_list
                  |  ident ‘(’ [argument_list] ‘)’
                  |  ident
                  |  attr_occ
                  |  const_expr
                  |  expr binop expr
                  |  unop expr
                  |  ‘(’ expr ‘)’
local_def_list  ::=  local_def
                  |  local_def_list ‘,’ local_def
local_def       ::=  ident ‘=’ expr
argument_list   ::=  expr
                  |  argument_list ‘,’ expr
```

The following operators can be used to construct binary and unary expressions.

```
binop  ::=  ‘*’ | ‘/’ | ‘+’ | ‘-’
         |  ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’
         |  ‘&&’ | ‘||’
unop   ::=  ‘-’ | ‘!’
```

The operators and their associativity are listed in Table 4.1 in the order of decreasing operator precedence.

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>‘*’, ‘/’</td><td>left associative</td></tr>
<tr><td>‘+’, ‘-’</td><td>left associative</td></tr>
<tr><td>‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’</td><td>non-associative</td></tr>
<tr><td>‘&&’, ‘||’, ‘!’</td><td>left associative</td></tr>
</table>

Table 4.1: Coala’s operators in order of decreasing precedence

The following defines Coala’s constant expressions.

const_expr	::=	*string_const*
	|	*nat_const*
	|	*real_const*
	|	*dim_const*
	|	*bool_const*
	|	**nil**

In order to keep the expression grammar unambiguous, within **where** expressions of the form $expr_0$ **where** $ident_1$ '=' $expr_1$ ... $ident_k$ '=' $expr_k$ nested **where** expressions must be enclosed in braces.

Evaluation of expressions

Evaluation of expressions happens in a given *evaluation environment* (or environment for short) and results in a *value* or an evaluation error. An environment provides partial functions from attribute occurrences to values and from identifiers to values.

Also required for the evaluation of expressions is a set of globally defined functions known to the Coala interpreter; Coala provides functions through libraries which are either built-in or plugged in, and allows to define functions in a specification. Libraries will be described in Section 4.2.2, user-defined functions will be described in Section 4.1.5. The application of a function to given argument values results in either a value or an evaluation error.

The Coala interpreter provides a domain of values which includes at least the following:

- the undefined value $\perp$
- the Boolean values *true* and *false*
- integer numbers
- floating point numbers
- strings over the ☞*Unicode* character set
- dimensions

Additional sets of values can be supported by user-provided libraries which will be explained in Section 4.2.2.

In the following the evaluation of each form of expression is described. For compound expressions containing sub expressions the following general rules apply:

- if not otherwise stated, sub expressions are evaluated in the same environment as the compound expression
- generally, if the evaluation of a sub expression leads to an error, the evaluation of the compound expression immediately stops and yields an error too

Constant expressions. All constant expressions evaluate to their represented value. The undefined value $\perp$ is represented by **nil**.

Attribute occurrences. An attribute occurrence evaluates to the value defined by the current environment. If the attribute occurrence's value is undefined an evaluation error results.

References. An identifier referencing a local variable evaluates to the value defined by the current environment. If the referenced local identifier is undefined an evaluation error results.

Function application. An expression of the form *ident* '(' $expr_1$ ',' …',' $expr_k$ ')' is evaluated as follows. First, the expressions $expr_1, \ldots, expr_k$ are evaluated to values $v_1, \ldots, v_k$. The result of the function application is obtained by applying the function denoted by *ident* to the arguments $v_1, \ldots, v_k$. If no function with the given identifier is known to the Coala interpreter an evaluation error results.

If-then-else. An expression $expr_0$ of the form **if** $expr_1$ **then** $expr_2$ **else** $expr_3$ is evaluated in the following fashion. First, $expr_1$ is evaluated. If the result is no Boolean value the result of evaluating $expr_0$ is an evaluation error. Otherwise, if $expr_1$ evaluates to *true*, the result is obtained by evaluating $expr_2$; if $expr_1$ evaluates to *false*, the result is obtained by evaluating $expr_3$.

Let expressions. An expression $expr_0$ of the form **let** *ident* '=' $expr_1$ **in** $expr_2$ is evaluated as follows. First, $expr_1$ is evaluated. A new environment extending the current environment is constructed which maps *ident* to the result of $expr_1$ (any existing mapping of *ident* is overwritten in the new environment). The result of $expr_0$ is the result of evaluating $expr_2$ in this new environment.

Where expressions. An expression of the form $expr_0$ **where** $ident_1$ '=' $expr_1$, …, $ident_k$ '=' $expr_k$ is equivalent to:

let $ident_1$ '=' $expr_1$ **in**
  …
    let $ident_k$ '=' $expr_k$ **in**
      $expr_0$

Unary expressions. Coala supports the two unary operators '-' and '!' which are both abbreviations for applications of predefined unary functions: '-' *expr* is an abbreviation for `neg` '(' *expr* ')'; '!' *expr* is equivalent to `not` '(' *expr* ')'. Those predefined functions will be described in Section 4.2.2.

Sequential OR. An expression of the form $expr_1$ '||' $expr_2$ is equivalent to the expression **if** $expr_1$ **then true else** $expr_2$.

Sequential AND. An expression of the form $expr_1$ '&&' $expr_2$ is equivalent to the expression **if** $expr_1$ **then** $expr_2$ **else false**.

Binary expressions. Binary expressions of the form $expr_1$ *binop* $expr_2$ for all binary operators except ‘`&&`’ and ‘`||`’ are abbreviations for function applications f ‘`(`’ $expr_1$, $expr_2$ ‘`)`’ where f is a predefined function selected according to Table 4.2. Those predefined functions are described in Section 4.2.2.

operator	function		operator	function
‘+’	`plus`		‘-’	`minus`
‘*’	`times`		‘/’	`div`
‘<’	`lt`		‘>’	`gt`
‘<=’	`le`		‘>=’	`ge`
‘==’	`equ`		‘!=’	`notequ`

Table 4.2: Functions corresponding to binary operators

Context conditions imposed on expressions

Expressions must satisfy some context conditions which are checked statically while loading or compiling a Coala specification. These are listed in the following:

- References to local variables must be bound by their context. The variable defined by a **let** expression is bound within the **let** expression's body. Within definitions of user-defined functions, which are described below in Section 4.1.5, the function's parameters are bound within the function's body expression.
- Attribute occurrences must be defined in the context in which they occur. That means that an attribute occurrence's node variable must be bound by an enclosing attribution rule (see Section 4.1.8) or transformation rule (see later in Section 4.1.11) and the addressed attribute must be associated with the node variable's grammar symbol.
- The function referenced in a function application must be defined. A function can either be defined by a built-in or user-provided library (see later in Section 4.2.2), or within the Coala specification as described below. In the latter case the function definition may occur after the place of application.

Note that some errors which have been described to occur during the evaluation of expressions are eliminated if the static context conditions above are satisfied.

4.1.5 User-defined semantic functions

A specification can include definitions of semantic functions.

fct_def ::= **fct** *ident* ‘`(`’ [*ident_list*] ‘`)`’ ‘`=`’ *expr*

A function defined this way has the arity n equal to the number of specified parameter identifiers. When applied to n argument values the result is obtained by evaluating the body expression in an environment that maps the i-th parameter to the i-th argument value for all $1 \leq i \leq n$. An error is raised if the function is applied to other than n arguments.

4.1.6 Structure of specifications

A specification begins with the declaration of the specification's name and is followed by a sequence of declarations. Those declarations can be either the inclusion other specifications, declarations for the specification of an attribute grammar, or transformation rules.

specification	::=	**specification** *ident* ';' *decl**
decl	::=	*include_decl*
	|	*terminal_decl*
	|	*production_list*
	|	*fct_def*
	|	*attr_decl*
	|	*attr_rules*
	|	*trafo_rule*

4.1.7 Inclusion of specifications

In order to increase modularity, parts of large specifications can be put in external specifications and included using a declaration of the form:

include_decl	::=	**include** *ident* ';'

An include declaration instructs the Coala interpreter to load the referenced specification and make the contained declarations available within the current specification. More details about loading specifications are described later in Section 4.2.

4.1.8 Specifying attribute grammars

The language constructs for the specification of attribute grammars are declarations of terminal symbols, productions, attributes and attribution rules. The specification of attribute grammars can be done in a very concise and readable way which is accomplished by the following features:

- attribution rules allow *expressions* on the right hand side, making it unnecessary to define semantic functions explicitly for each rule
- copy rules, i.e, rules where the semantic function is the identity function applied to an occurrence of the same attribute as on the left hand side, are automatically added and can be omitted in most cases
- nonterminal symbols are declared implicitly when occurring on the left hand side of productions

These features will be described in more detail later. In the following we introduce the language constructs for specifying attribute grammars.

Terminal symbols

In the Coala system terminal symbols are treated specially with respect to attribution: no user-defined attributes can be associated with terminal symbols; instead, all terminal symbols have exactly one synthesized attribute `value` which holds a value from Coala's semantic domain.

Coala provides one predefined terminal symbol `Val` representing values from the semantic domain like Booleans, integers, floating point values and string values, but also any values supported through libraries other than the built-in standard libraries (see Section 4.2.2).

The user can specify additional terminal symbols as follows:

terminal_decl	::=	**terminal** *ident_list* '`;`'

The identifiers from the list on the right hand side are declared as terminal grammar symbols. These grammar symbols must not already have been declared. Although declaring additional terminals in this fashion is not strictly necessary (`Val` can always be used instead), they are useful to emphasize that they hold only a specific type of values.

Productions

Productions of context-free grammars can be specified in `BNF` notation:

production_list	::=	**productions** '{' *productions*$^+$ '}'
productions	::=	*ident* '`::=`' *prod_rhs_list* '`;`'
prod_rhs_list	::=	*prod_rhs*
	|	*prod_rhs_list* '`|`' *prod_rhs*
prod_rhs	::=	*node_var**

The identifier on the left hand side of '`::=`' is declared as a nonterminal grammar symbol and the productions given for it on the right hand side (separated by '`|`') are declared. The node variables on the right hand side must refer to grammar symbols defined in the current specification or in an included specification. Only the symbol part of node variables is actually relevant here; however, identifiers can be specified for documentary purposes to distinguish multiple occurrences of the same symbol.

Attribute declarations

Attributes are declared as follows:

attr_decl	::=	**inherited** *ident_list* **with** *ident_list* '`;`'
	|	**synthesized** *ident_list* **with** *ident_list* '`;`'

An attribute declaration declares the attributes given left of **with** to be inherited or synthesized attributes associated with the grammar symbols following **with**.

All identifiers following **with** must denote non-terminal grammar symbols having been defined before. An attribute with a given symbol cannot be declared to be inherited and at the same time synthesized.

Attribution rules

Attribution rules are introduced by specifying a production and a list of attribute equations:

attr_rules	::=	**attr** *node_var* '::=' *node_var** '{' *attr_equ*$^+$ '}'
attr_equ	::=	*attr_occ* '=' *expr* ';'

The production referenced by *node_var* '::=' *node_var** must have been declared before. The following must be true for all attribute equations: the attribute occurrence on the equation's left hand side must refer to either a previously declared synthesized attribute with the production's left hand side symbol, or a previously declared inherited attribute with one of the production's right hand side symbols.

An attribution rule provides a context for the right hand side expressions of its attribute equations in which all attribute occurrences of the referenced production are defined. This context is used to statically check the context conditions that have been listed in Section 4.1.4; further, the expression is always evaluated in an environment that contains mappings for these attribute occurrences.

Built-in copy rules

In attribute grammars many attributes are often simply passed down to child nodes or passed up to parent nodes without modification. To free the user of the tedious task of adding the necessary copy rules manually, Coala adds such rules automatically according to the following rules:

- if in a production the same inherited attribute is associated with the production's left hand side symbol and with a right hand side symbol, and if no rule to compute the latter attribute occurrence is defined, then a copy rule is implicitly added

- if in a production the same synthesized attribute is associated with the production's left hand side symbol and with some right hand side symbols, and if no rule to compute the attribute's occurrence on the left hand side of the production is defined, then a rule copying the the rightmost occurrence on the right hand side is implicitly added

Built-in copy rules allow attribute grammars to be specified very concisely as the example specifications coming up in in Section 4.1.12 and Chapter 5 will show.

4.1.9 Semantics

The semantics of attribute grammar specifications is described in detail in Appendix C. As an extension as compared to the attribute grammar semantics known from the literature [26, 7], support for tree fragments as introduced in Section 3.3.2 has been added. That means that the semantic domain includes the special undefined value $\perp$ which is handled specially w.r.t. attribution: attributes depending on an other attribute evaluating to $\perp$ also evaluate to $\perp$. Besides, the synthesized attributes at unexpanded nodes are ensured to evaluate to $\perp$. Further, attributes for which no rule exists always evaluate to $\perp$.

4.1.10 Attribute evaluation strategy

A note about attribute evaluation strategies is due here. Any evaluation strategy known from the literature [7] could theoretically be employed by Coala. However, in order to achieve highest possible efficiency, Coala's evaluation strategy should have the following properties:

- it should be *lazy*, since during a tree transformation potentially many intermediate trees which are later discarded may be constructed; evaluating only the necessary attributes on demand will avoid unnecessary overhead
- it should be *incremental*, since target trees are constructed in an evolutionary process and evaluating only the changed attributes after replacing an unexpanded subtree will increase efficiency significantly

4.1.11 Transformation rules

Template syntax is basically the same as in Chapter 3. In addition to node variables, however, templates can contain expressions. Appearing on a transformation rule's right hand side an expression is used to construct a terminal node whose attribute `value` contains the result obtained by evaluating the expression. Appearing on the left hand side of a transformation rule, a (constant) expression can be used to constrain the applicability of a rule.

```
template       ::=  leaf_template
                 |  node_var '<' template* '>'
leaf_template  ::=  node_var
                 |  '(' ident ':' expr ')'
                 |  '(' expr ')'
                 |  const_expr
                 |  attr_occ
```

The identifier in a leaf template of the form ‘`(`’ *ident* ‘`:`’ *expr* ‘`)`’ specifies the terminal grammar symbol to use. A leaf template of the form ‘`(`’ *expr* ‘`)`’ is an abbreviation for ‘`(`’ `Val` ‘`:`’ *expr* ‘`)`’. Similarly, *const_expr* is an appreviation for ‘`(`’ `Val` ‘`:`’ *const_expr* ‘`)`’ and *attr_occ* is an abbreviation for ‘`(`’ `Val` ‘`:`’ *attr_occ* ‘`)`’.

The syntax of transformation rules is also basically the same as in Chapter 3 with only slight extensions. A rule is introduced with the keyword **rule** followed by an optional *relation identifier*. Then a sequence of *source templates*, a *target template*, and optional *transformation rest variables*, optional *constraints* and an optional *minimize expression* is specified. Constraints are either semantic constraints specified as (Boolean) expressions enclosed in square brackets, or syntactic constraints. For the latter an optional *relation identifier*, a sequence of *source variables*, a *target variable* and optional *transformation rest variables* are given.

trafo_rule	::=	**rule** [*rel_ident* ':')] *template** '->' *template*
		[*trafo_rest*] [*constraints*] [*minimizer*] ';'
trafo_rest	::=	'/' *node_var**
constraints	::=	**where** *constraint**
constraint	::=	'[' *expr* ']' ';'
	|	[*rel_ident* ':'] *leaf_template** '->'
		leaf_template [*trafo_rest*] ';'
minimizer	::=	**minimize** *expr* ';'

Optional parts (except *minimizer*) are when missing just convenient abbreviations for more verbose syntax and are automatically supplemented as follows:

- missing *rel_ident* ':' is supplemented by '@@' ':'
- missing *trafo_rest* is supplemented by '/' ε
- missing *constraints* is supplemented by **where** ε

In contrast to the definition of transformation rules in Chapter 3, relation identifiers are associated with transformation rules and syntactic constraints. This makes it possible to define more than one transformation relation in a Coala specification and refer to specific relations in syntactic constraints. Relation identifiers mainly increase the readability of specifications and are only a minor extension of the transformation model from Chapter 3. It is straightforward to add support for them.

Minimize expressions—if specified—correspond to the objective functions used in Chapter 3 and can be easily mapped to such functions.

Evaluation of expressions in templates, semantic constraints and minimize expressions

It is required to describe the handling of expressions in templates and semantic constraints as they provide an extension to the model described in Chapter 3.

Expressions in templates and semantic constraints can contain occurrences of attributes of node variables occurring in the rule's source templates, target template and transformation rest. At the point when an expression is evaluated during the process of applying the enclosing transformation rule a variable environment describes the binding of node variable to nodes (cf Section 3.3.5).

Some node variables can be yet undefined when the evaluation takes place. So, for the evaluation of an expression a semantic environment is constructed which maps all attribute occurrences contained in the expression to the attribute's value at the node bound by the attribute occurrence's node variable, or to $\perp$ if the node variable is not yet defined. The evaluation of the expression then proceeds as described in Section 4.1.4.

Semantics of templates

For templates containing no expressions the semantics is the same as has been described in Chapter 3, i.e., Definition 3.2.9 on page 36 directly applies.

In addition, a (leaf) template of the form '(' *ident* ':' *expr* ')' is instantiated at a node iff the following is satisfied:

- the node has the terminal symbol specified by *ident*
- the attribute *value* of the node has the value that results from evaluating *expr*

Semantics of transformation rules

Coala transformation rules are—apart from the minor extensions described above—directly related to rules as defined in Chapter 3. As expressions are allowed to be used as semantic constraints it is necessary to map them to k-ary predicates where k is the number of attribute occurrences in the expression. When evaluating a semantic constraint a runtime error is raised if the result is not a Boolean value. In a similar way, minimize expressions have to be mapped to objective functions. Here, the evaluation result is required to be a number.

4.1.12 Example specification

We will now take a look at a simple example specification. The specified attribute grammar has already been seen in Chapter 3 and is depicted in a graphical way in Figure 3.2 on page 29. The transformation rules are those given in Example 3.2.6 on page 34.

```
specification ExampleSpec;

// Source grammar
productions {
  S ::= B ;
  S ::= L ;
  B ::= S S ;
  L ::= /*empty*/ ;
}

// Target grammar
productions {
  T ::= E ;
  E ::= H ;
  E ::= V ;
  E ::= A ;
  H ::= E E ;
  V ::= E E ;
  A ::= /*empty*/ ;
}

fct max( x, y ) = if x >y then x else y
fct zero()      =0
fct one()       =1

inherited    x with E, H, V;
synthesized  w with E, H, V;

attr T ::= E {
  E.x =zero();
```

```
}

attr E ::= A {
  E.w =one();
}

attr H ::= E[1] E[2] {
  H.w =plus( E[1].w, E[2].w );
  E[2].x = plus( H.x, E[1].x );
}

attr V ::= E[1] E[2] {
  V.w =max( E[1].w, E[2].w );
}

//
// transformation S → T
//

rule S → T< E >
  where S → E;

rule S[0]< B <S[1] S[2] >> → E< H< E[1] E[2] >>
  where S[1] → E[1];
        S[2] → E[2];
        [ (E.x +E.w) ≤ 2 ];

rule S[0]< B <S[1] S[2] >> → E< V< E[1] E[2] >>
  where S[1] → E[1];
        S[2] → E[2];
        [ (E.x +E.w) ≤ 2 ];

rule S< L > → E< A<> >
```

Further examples can be found in the next chapter which examines some practical applications of Coala.

4.2 The Coala runtime system

This section gives an overview over Coala's runtime system. The runtime system provides an API for loading specifications, building and manipulating attributed trees, and performing transformations. Support specific to document processing is provided which allows to read documents from `XML` and to write `Postscript` output.

First, the most important parts of the Coala runtime system are described. Then the implementation of semantic domains using function libraries and some built-in libraries are described. Finally, Coala's support for output of `PostScript` files for a built-in page layout structure are described.

4.2.1 Loading and interpretation of Coala specifications

The most important components of the Coala runtime system are `SpecLoader`s for loading specifications and creating an interpreter, and `Interpreter`s themselves allowing to build and manipulate attributed trees and to perform transformations. Their usage is outlined next.

Loading Coala specifications

A `SpecLoader` provides a generator for Coala interpreters. The basic usage of a `SpecLoader` is as follows.

- necessary function libraries extending the semantic domain (see later in Section 4.2.2) are specified
- Coala specifications are loaded
- an interpreter is generated for the loaded specifications

In the last step the loaded specifications are analyzed and data structures necessary to efficiently perform the attribution of trees and the transformation of attributed trees are generated. The result is an interpreter driven by these data structures. This approach corresponds to the generative approach to document formatters described in Section 1.2.3.

The resulting `Interpreter` can then be used to build tree structures and perform tree transformations. This is described next.

Building attributed tree structures

An `Interpreter` provides functions allowing to create trees and tree fragments in a bottom-up fashion. Only syntactically valid trees with respect to the context-free grammar specified in the loaded specifications are allowed to be created.

Also provided are functions for reading tree structures from `XML`. In order for the necessary `XML` files to be given in a convenient format the following functionality is built in:

- "flat" sequences of `XML` elements are automatically transformed into recursive structures based on the specified context-free grammar
- intermediate nodes are automatically supplemented when missing, also based on the context-free grammar
- if desired, text is transformed into a sequence of words

As an example, assume that the following context-free grammar has been specified:

```
productions {
  Paragraph   ::= WordSeq ;
  WordSeq     ::= /*empty*/
                |  Word WordSeq ;
  Word        ::= Val[text]  ;
}
```

Then the following XML is a valid representation of a *Paragraph* tree:

```
<Paragraph>
  Lorem ipsum dolor sit
</Paragraph>
```

Given this XML the tree shown in Figure 4.1 is created.

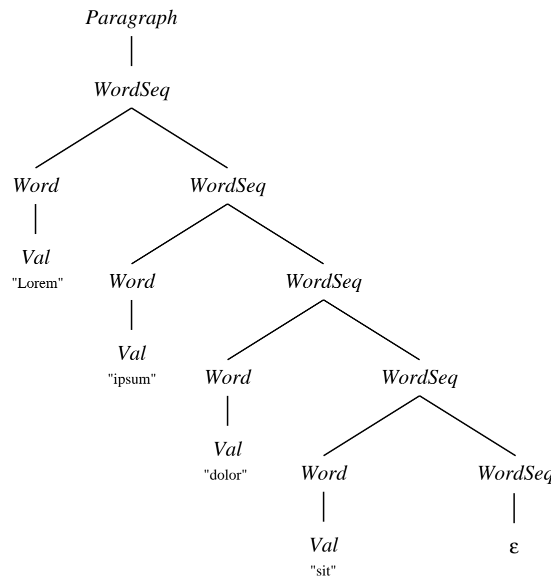


Figure 4.1: Example tree

In order to allow terminal Val nodes with specific values to be specified in XML, mappings from string representations to values are predefined for the built-in basic types and further mappings can be added. This is necessary as XML allows only string attributes. For instance, a terminal Val node with an integer value can be specified as the XML element `<Val type="int" value="10"/>`. A dimension of 10pt can be specified as `<Val type="dim" value="10pt"/>`.

Performing transformations

An Interpreter of course provides functions allowing to perform transformations. The most important variant directly corresponds to the execution scheme developed in Section 3.3 and takes an optional relation identifier, an array of source nodes, and an anchor node as arguments, and returns a list of transformation results. A transformation result consists of a target node and an array of transformation rest nodes.

4.2.2 Function libraries

Coala realizes the semantic domain used for attributing of trees using libraries of functions. Each library defines functions operating on one or more data types. Besides, a library can also define functions converting string representations of values of these data

types to the actual values. The latter functionality is used when reading tree structures from XML as was described in Section 4.2.1.

Coala already provides several libraries, some of which are built in (i.e., are automatically loaded by a SpecLoader), implementing a set of general purpose functions on basic data types.

Some library functions are polymorphic; this is true in particular for the standard arithmetic functions on integers, floating point numbers, and dimensions; e.g., the function plus is defined for all of these data types.

The data types used by Coala's built-in libraries form a hierarchy of classes are and are listed the following.

- Boolean: Boolean values *true* and *false*
- Integer: integer numbers
- Double: (double precision) floating point numbers
- Number: all numbers (integer or floating point)
- Dim: dimensions (stored in some standard unit of measure left to the implementation)
- Str: strings
- Ord: semantic values on which a total order is defined; the types Number, Integer, Double, Dim, and Str described above are all subtypes of Ord
- Type1Font: Postscript ☞*Type 1 Font*s known to the Coala system
- Object: any semantic values including all of the values from the data types above and any data type provided by a user-defined library

Coala's built-in libraries are:

- DimensionLib: provides functions performing arithmetic operations on dimensions and conversion between units of measure
- StdLib: operates on Number, Boolean, Str, and Ord
- PSFontLib: operates on Postscript ☞*Type 1 font*s providing information about their metrics

The libraries are described in more detail next.

DimensionLib

This library provides arithmetic operations on dimensions and operations for converting between units of measure. The provided arithmetic operations on dimensions are:

➢ `Dim plus(Dim x, Dim y)`

➢ `Dim minus(Dim x, Dim y)`

- ➢ `Dim times(Dim x, Number y)`
- ➢ `Dim times(Number x, Dim y)`
- ➢ `Dim div(Dim x, Dim y)`
- ➢ `Dim times(Dim x, Number y)`
- ➢ `Dim neg(Dim x)`

The operations allowing to convert dimensions to certain units of measure are:

- ➢ `Double to_cm(Dim x)`
- ➢ `Double to_mm(Dim x)`
- ➢ `Double to_in(Dim x)`
- ➢ `Double to_pt(Dim x)`

`DimensionLib` also allows to convert string representations of dimensions to actual dimensions. String representations consist of a number followed by an abbreviated unit of measure (one of `cm`, `mm`, `in`, or `pt`).

StdLib

`StdLib` provides several basic functions including arithmetic operations on numbers and dimensions, comparison operations, and operations on strings.

All arithmetic functions on numbers act like follows: if all arguments are integer numbers, an integer operation is performed and the result is an integer number too. Otherwise, all arguments are converted to floating point numbers as necessary, the arithmetic operation is performed on floating point numbers, and the result is a floating point number. These are the provided arithmetic operations:

- ➢ `Number plus(Number x, Number y)`
- ➢ `Number minus(Number x, Number y)`
- ➢ `Number times(Number x, Number y)`
- ➢ `Number div(Number x, Number y)`
- ➢ `Number neg(Number x)`

The following functions compare `Ord`-values.

- ➢ `Boolean lt(Ord x, Ord y)`
- ➢ `Boolean gt(Ord x, Ord y)`
- ➢ `Boolean le(Ord x, Ord y)`
- ➢ `Boolean ge(Ord x, Ord y)`

The function `plus` applied to strings performs string concatenation. If the second argument is no string it will be converted to one before.

➢ `String plus(Str x, Object y)`

The following operations test objects for equality.

➢ `Boolean equ(Object x, Object y)`

➢ `Boolean notequ(Object x, Object y)`

Negation of Boolean values is performed by the function `not`.

➢ `Boolean not(Boolean x)`

`StdLib` also provides functionality for converting string representations of data types `Integer`, `Double`, `Boolean` and `Str` to values.

PSFontLib

This library makes the metric characteristics of `Postscript` ☞*Type 1 Font*s accessible to a Coala specification which is required to compute the geometry of text boxes. The data type `Type1Font` represents fonts. The following functions operate on `Type1Font`s.

➢ `Type1Font makeType1Font(Str name)` finds a font by the given name; returns `nil` if no font with that name is known

➢ `Dim stringWidth(String text, Type1Font f, Number ptSize)` returns the width of `text` when rendered in font `f` at ☞*point size* `ptSize`; substitution of ☞*ligature*s and ☞*kerning* according to the font's specification is taken account for

➢ `Dim ascender(Type1Font f, Number ptSize)` returns the ☞*ascent* of a font at the given size

➢ `Dim descender(Type1Font f, Number ptSize)` returns the ☞*descent* of a font at the given size

4.2.3 A simple page layout format with PostScript language output

In order to complete the set of features needed for realizing working document formatters, Coala defines a simple page layout format and provides functionality converting such page layouts to `Postscript`. The resulting `Postscript` is suitable for viewing, printing, and further conversion to other formats like `PDF`.

The page layout format supports rules and text. Also supported is cross-referencing between boxes. This allows to realize hyperlinks in a viewer application. The context-free grammar productions for the page layout are shown in Figure 4.2 on the next page.

$$
\begin{array}{lcl}
\mathit{PageLayout} & ::= & \mathit{PageSeq} \\
\mathit{PageSeq} & ::= & \varepsilon \\
 & | & \mathit{Page}\ \mathit{PageSeq} \\
\mathit{Page} & ::= & \mathit{Box} \\
\mathit{Box} & ::= & \mathit{HBox} \mid \mathit{VBox} \mid \mathit{TextBox} \mid \mathit{RuleBox} \mid \mathit{EmptyBox} \\
 & | & \mathit{RefBox} \mid \mathit{TargetBox} \\
\mathit{HBox} & ::= & \mathit{Box}\ \mathit{Box} \\
\mathit{VBox} & ::= & \mathit{Box}\ \mathit{Box} \\
\mathit{TextBox} & ::= & \mathit{Val}[\mathit{text}]\ \mathit{Val}[\mathit{fontName}]\ \mathit{Val}[\mathit{ptSize}] \\
\mathit{RuleBox} & ::= & \mathit{Val}[\mathit{width}]\ \mathit{Val}[\mathit{height}]\ \mathit{Val}[\mathit{depth}] \\
\mathit{EmptyBox} & ::= & \mathit{Val}[\mathit{width}]\ \mathit{Val}[\mathit{height}]\ \mathit{Val}[\mathit{depth}] \\
\mathit{RefBox} & ::= & \mathit{Val}[\mathit{label}]\ \mathit{Box} \\
\mathit{TargetBox} & ::= & \mathit{Val}[\mathit{label}]\ \mathit{Box}
\end{array}
$$

Figure 4.2: Grammar of Coala's simple page layout format

4.3 Conclusion

The Coala system provides a specification language and its implementation of the theoretic model of optimizing attributed tree transformations described in Chapter 3. The system includes means to specify attribute grammars in a convenient way and has some minor extensions to the theoretic transformation model, allowing transformation specifications also to be made in a comfortable way.

Coala further provides ready-to-use tools for dealing with `XML` files and `PostScript` output and as such can be used for realizing practical document formatters.

Coala has been implemented using the `Java` programming language. The next chapter will show Coala's practical application realizing several example document formatters. Formatting results produced with this Coala implementation are included.

CHAPTER 5

Example applications

This chapter presents some realistic examples showing the applicability of the new transformation language and its underlying model.

5.1 Line breaking

Line breaking has already been addressed as an introductory example in Chapter 2. In this section it is now shown how a specification of line breaking can be realized using Coala. We first handle a solution using global optimization. We then show how the specification can be modified to achieve variants using local optimization.

5.1.1 Global optimization

As the specification of the source structure and the target structure in the description of line breaking in Chapter 2 is rather straightforward, we focus on the specification of the intermediate structure of sequences of lines and the transformation rules for line breaking.

LineSeq structure

The structure of sequences of lines is described using Coala like follows:

```
productions {
  LineSeq      ::= Val[lineWidth]  LineSeq;
  LineSeq      ::= /*empty*/
                |  Line LineSeq;
  Line         ::= /*empty*/
                |  Box  Line
```

```
                    |   Glue  Line;
  Glue             ::=  Val[nat]  Val[plus]  Val[minus];
}
```

In contrast to Section 2.3 we do not realize line width as a constant but as an inherited attribute of the intermediate structure.

We specify the following attributes.

```
inherited    Nat, Plus, Minus  with Line;
inherited    adjust            with Line;
synthesized  sAdjust           with Line;
synthesized  lineCount         with LineSeq;
synthesized  cost              with Line, LineSeq;
inherited    lineWidth         with LineSeq, Line;
synthesized  nat, plus, minus  with Glue;
inherited    actual            with Glue;
```

Attribution rules

The inherited attribute *lineWidth* is initialized at the root of a *LineSeq* structure. Note that the copy rules necessary to pass down the attribute to all descendants are added automatically for us by Coala and thus need not be specified.

```
attr  LineSeq[0]  ::=  Val[lineWidth]  LineSeq[1] {
  LineSeq[1].lineWidth =Val[lineWidth].value;
}
```

The adjustment factor of each line is computed as follows (cf. Figure 2.5 on page 17).

```
attr  Line[0]  ::=  Box  Line[1] {
  Line[1].Nat =Line[0].Nat +Box.w;
}

attr  Line[0]  ::=  Glue  Line[1] {
  Line[1].Nat   =Line[0].Nat +Glue.nat;
  Line[1].Plus  =Line[0].Plus +Glue.plus;
  Line[1].Minus =Line[0].Minus +Glue.minus;
}

fct  f(lineWidth, nat, plus, minus) =
   if  nat ==lineWidth then 0.0
  else if  lineWidth >nat && plus != 0.0 then
     to_pt(lineWidth  -nat) / to_pt(plus)
  else if  lineWidth <nat && minus != 0.0 then
     to_pt(lineWidth -nat) / to_pt(minus)
  else if  lineWidth >nat then  infinity ()
  else  - infinity ()

attr  Line  ::=  /*empty*/ {
  Line.sAdjust =f(Line.lineWidth, Line.Nat, Line.Plus, Line.Minus);
  Line.sNat =Line.Nat;
}
```

```
attr  LineSeq[0]  ::=  Line  LineSeq[1]  {
  Line.iAdjust  =Line.sAdjust;
  Line.Nat  =0.0pt;
  Line.Plus  =0.0pt;
  Line.Minus  =0.0pt;
  Line.lineWidth  =LineSeq[0].lineWidth;
}

fct g(adjust, nat, plus, minus) =
   if adjust ≥ 0.0 then nat +(adjust * plus)
                   else nat +(adjust * minus)

attr  Line[0]  ::=  Glue  Line[1]  {
  Glue.actual  =g(Line[0].iAdjust, Glue.nat, Glue.plus, Glue.minus);
}
```

Here is how the computation of the cost (badness) of a sequence of lines is specified (cf. Figure 2.11 on page 21).

```
fct lineCost(lineWidth, nat, plus, minus) =
  abs(a)
   where
    a = f(lineWidth, nat, plus, minus)

attr  Line  ::=  /*empty*/ {
  Line.cost  = lineCost(Line.lineWidth, Line.Nat, Line.Plus, Line.Minus);
}

attr  LineSeq  ::=  /*empty*/ {
  LineSeq.lineCount  =0;
  LineSeq.cost  = 0.0;
}

attr  LineSeq[0]  ::=  Line  LineSeq[1]  {
  LineSeq[0].lineCount  =LineSeq[1].lineCount  +1;
  LineSeq[0].cost  =
      (c2 * (n−1) +c1) / n
    where
      n  =LineSeq[0].lineCount,
      c1 =Line.cost,
      c2 =LineSeq[1].cost;
}
```

Transformation rules

Here is the specification of transformation relation l constructing possible single next lines from a (rest) box kerf sequence, described in Section 2.3.

```
rule @l: BoxKerfSeq[0]< Kerf BoxKerfSeq[1] >→ Line<> / BoxKerfSeq[1]
  where
    [ Line.Nat +Line.Plus ≥ Line.lineWidth ];
```

```
rule @l: BoxKerfSeq[0]< Kerf< Val[nat] Val[plus] Val[minus] >BoxKerfSeq[1] >
  → Line[0]<
      Glue< Val[nat] Val[plus] Val[minus] >
      Line[1]
    >/ BoxKerfSeq[2]
  where
    [ Line[1].Nat −Line[1].Minus ≤ Line[1].lineWidth ];
    @l: BoxKerfSeq[1] → Line[1] / BoxKerfSeq[2];

rule @l: BoxKerfSeq[0]< Box[0] BoxKerfSeq[1] >
         → Line[0]< Box[1] Line[1] >/ BoxKerfSeq[2]
  where
    Box[0] → Box[1];
    [ Line[1].Nat −Line[1].Minus ≤ Line[1].lineWidth ];
    @l: BoxKerfSeq[1] → Line[1] / BoxKerfSeq[2];
```

And this is how transformation relation *ls* from Section 2.3 constructing sequences of lines is specified using Coala.

```
rule @ls: BoxKerfSeq<> →LineSeq<>
rule @ls: BoxKerfSeq[0] → LineSeq[0]< Line LineSeq[1] >
  where
    @l:  BoxKerfSeq[0] → Line / BoxKerfSeq[1];
    @ls: BoxKerfSeq[1] → LineSeq[1];
    minimize LineSeq[0].cost;
```

An example formatting result will be shown in Section 5.1.3. Next, we discuss some variants of the line breaking specification.

5.1.2 Variants

Besides global optimization described above some well-known other approaches to line breaking exist [35], as already discussed in Chapter 2. They can be characterized as follows:

- First fit: lines are broken as early as possible
- Local optimization (best fit): next line is always chosen such that its cost is minimal
- Last fit: lines are broken as late as possible

It is quite easy to adapt the line breaking specification developed so far to achieve these variants of formatting. All that has to be done is basically to change the `minimize` clause and move it to a different place. As an example we look at a specification using local optimization (best fit).

```
rule @l_local : BoxKerfSeq[1] → Line / BoxKerfSeq[2]
  where
    @l: BoxKerfSeq[1] → Line / BoxKerfSeq[2];
    minimize Line.cost;

rule @ls_local : BoxKerfSeq<> →LineSeq<>
```

```
rule  @ls_local :  BoxKerfSeq[0] → LineSeq[0]< Line LineSeq[1]  >
  where
    @l_local :   BoxKerfSeq[0] → Line  /  BoxKerfSeq[1];
    @ls_local :  BoxKerfSeq[1] → LineSeq [1];
```

In order to realize first fit and last fit line breaking it is only necessary to minimize or maximize the number of boxes in a line.

Example runs showing the output of our formatter in all discussed variants for an example text are shown next.

5.1.3 Example runs

As an example run, the beginning of Jack London's *The Sea-Wolf* [29] was processed by our formatters. The resulting sequence of lines was converted to a *PageLayout* structure (see Section 4.2.3) containing a single page, and then converted to `Postscript`. The result for each formatting variant is shown in Figure 5.1 on the following page.

5.2 A page formatter

This section shows how page breaking can be specified using Coala. The transformation described takes two streams, one consisting of styled text paragraphs and one consisting of figures, and produces a sequence of pages with the following properties:

- paragraphs are broken into lines using global optimization as described before
- each resulting page contains two areas, one for figures and one for text lines; either of these areas may be empty
- each resulting page is neither underfull nor overfull; stretchable and shrinkable glue is inserted between adjacent figures, between figures and text area (if both are non-empty), and between consecutive paragraphs; as an exception, the last page may be underfull
- no figure appears on a page before the first reference to it
- the resulting sequence of pages is globally optimal w.r.t. some yet to be defined measure of layout quality

The source structure consists of two sequences: one containing text paragraphs (each having its own font), and one containing figures described by a label (for reference), a description, and its size. Text paragraphs can contain references to figures. The following shows how this source structure is specified:

```
productions {
  Article        ::=  ParagraphSeq  FigureSeq;
  FigureSeq      ::=  /*empty*/
                   |  Figure   FigureSeq;
  Figure         ::=  Val[ label ]
                      Val[width]   Val[ height]
                      Val[ description ];
```

I scarcely know where to begin, though I sometimes facetiously place the cause of it all to Charley Furuseth's credit. He kept a summer cottage in Mill Valley, under the shadow of Mount Tamalpais, and never occupied it except when he loafed through the winter months and read Nietzsche and Schopenhauer to rest his brain. When summer came on, he elected to sweat out a hot and dusty existence in the city and to toil incessantly. Had it not been my custom to run up to see him every Saturday afternoon and to stop over till Monday morning, this particular January Monday morning would not have found me afloat on San Francisco Bay.

Not but that I was afloat in a safe craft, for the Martinez was a new ferry-steamer, making ...

First fit

I scarcely know where to begin, though I sometimes facetiously place the cause of it all to Charley Furuseth's credit. He kept a summer cottage in Mill Valley, under the shadow of Mount Tamalpais, and never occupied it except when he loafed through the winter months and read Nietzsche and Schopenhauer to rest his brain. When summer came on, he elected to sweat out a hot and dusty existence in the city and to toil incessantly. Had it not been my custom to run up to see him every Saturday afternoon and to stop over till Monday morning, this particular January Monday morning would not have found me afloat on San Francisco Bay.

Not but that I was afloat in a safe craft, for the Martinez was a new ferry-steamer, making ...

Locally optimal

I scarcely know where to begin, though I sometimes facetiously place the cause of it all to Charley Furuseth's credit. He kept a summer cottage in Mill Valley, under the shadow of Mount Tamalpais, and never occupied it except when he loafed through the winter months and read Nietzsche and Schopenhauer to rest his brain. When summer came on, he elected to sweat out a hot and dusty existence in the city and to toil incessantly. Had it not been my custom to run up to see him every Saturday afternoon and to stop over till Monday morning, this particular January Monday morning would not have found me afloat on San Francisco Bay.

Not but that I was afloat in a safe craft, for the Martinez was a new ferry-steamer, making ...

Globally optimal

I scarcely know where to begin, though I sometimes facetiously place the cause of it all to Charley Furuseth's credit. He kept a summer cottage in Mill Valley, under the shadow of Mount Tamalpais, and never occupied it except when he loafed through the winter months and read Nietzsche and Schopenhauer to rest his brain. When summer came on, he elected to sweat out a hot and dusty existence in the city and to toil incessantly. Had it not been my custom to run up to see him every Saturday afternoon and to stop over till Monday morning, this particular January Monday morning would not have found me afloat on San Francisco Bay.

Not but that I was afloat in a safe craft, for the Martinez was a new ferry-steamer, making ...

Last fit

Figure 5.1: Formatting results for example article

```
  ParagraphSeq  ::= /*empty*/
                 |  Paragraph  ParagraphSeq;
  Paragraph     ::= Font  WordSeq
                 |  WordSeq;
  WordSeq       ::= /*empty*/
                 |  Word  WordSeq;
  Word          ::= Val[text]
                 |  Reference;
  Reference     ::= Val[label];
  Font          ::= Val[fontName]  Val[fontSize];
}
```

Note that this structure can easily be produced from a purely logical article structure where font information is derived from logical structure.

A consecutive number is assigned to each figure which is stored in a symbol table. This number is used when figures are referenced in text.

```
inherited figureCount with FigureSeq, Figure;
inherited iFigureRefTable with FigureSeq, Figure;
synthesized sFigureRefTable with FigureSeq, Figure;
synthesized label with Figure;

attr Article ::= ParagraphSeq FigureSeq {
  FigureSeq.figureCount = 1;
  FigureSeq.iFigureRefTable = symtab_empty_table();
  ParagraphSeq.refTable = FigureSeq.sFigureRefTable;
}

attr FigureSeq ::= /*empty*/ {
  FigureSeq.sFigureRefTable = FigureSeq.iFigureRefTable;
}

attr FigureSeq[0] ::= Figure FigureSeq[1] {
  FigureSeq[1].figureCount = FigureSeq[0].figureCount + 1;
  FigureSeq[1].iFigureRefTable =
    symtab_put(FigureSeq[0].iFigureRefTable,
               Figure.label, FigureSeq[0].figureCount);
}

attr Figure ::= Val[label] Val[width] Val[height] Val[description] {
  Figure.label = Val[label].value;
}

inherited refTable with ParagraphSeq, Paragraph, WordSeq, Word, Reference;
synthesized refText with Reference;

attr Reference ::= Val[label] {
  Reference.refText = "" + symtab_get(Reference.refTable, Val[label].value);
}
```

Symbol tables are realized using a data structure `SymbolTable` on which the following operations are defined:

➢ `SymbolTable symtab_empty_table()` constructs an empty symbol table

➢ `SymbolTable symtab_put(SymbolTable table, Object key, Object value)` puts an entry into a symbol table, constructing a new table

➢ `Object symtab_get(SymbolTable table, Object key)` retrieves the value stored under the given key from a symbol table, or `nil` if the symbol table does not contain the given key

The target structure is again chosen to be the page layout structure as defined in Section 4.2.3 for which Coala provides functionality to produce `Postscript` code.

5.2.1 Area model

An intermediate structure of *areas* is used for the transformation. An area can in general be thought of as a container for vertically arranged boxes, glues, and subareas. Each area's height is constrained by a minimum and a maximum dimension. Two types of areas, *VArea*s and *PageArea*s are used:

- a *VArea* is a sequence of boxes and glues from one of the two input sequences (text lines or figures); a *VArea* can end in a special *End* element indicating that the end of the input sequence has been reached
- a *PageArea* contains two *VArea*s, one for figures and one for lines of text, separated by a glue

The structure of areas is specified by the following Coala productions:

```
productions {
  PageAreaSeq ::= /*empty*/
                |  PageArea PageAreaSeq
                |  Val[pageHeight] PageAreaSeq ;

  PageArea    ::= VArea[figArea] Glue[sep] VArea[textArea] ;
  VArea       ::= /*empty*/
               |  End
               |  Box VArea
               |  Glue VArea;
  End         ::= /*empty*/;
}
```

During transformations we will need to know if a given area is empty (i.e., contains no boxes or glue) and if it is the last area for the corresponding figure or text sequence (i.e., contains an *End* element). For this purpose the synthesized *is_empty* and *is_last* are introduced. The necessary attribution rules are omitted here.

Natural height, stretchability and shrinkability of areas are described using synthesized attributes *nat*, *plus* and *minus*, respectively. Figure 5.2 on page 108 shows the computation of these attributes for *VArea*s. The attribution of *PageArea*s is shown in

Figure 5.3 on the following page, Figure 5.4 on page 109 and Figure 5.5 on page 109. Note that the *Glue* separating figure and text areas only contributes to the *PageArea*'s attribute values if both subareas are non-empty. The corresponding part of the Coala specification looks like follows.

```
attr VArea[0] ::= Box VArea[1] {
  VArea[0].nat   = Box.h + Box.d + VArea[1].nat;
  VArea[0].plus  = 0.0pt + VArea[1].plus;
  VArea[0].minus = 0.0pt + VArea[1].minus;
}

attr VArea[0] ::= Glue VArea[1] {
  VArea[0].nat   = Glue.nat   + VArea[1].nat;
  VArea[0].plus  = Glue.plus  + VArea[1].plus;
  VArea[0].minus = Glue.minus + VArea[1].minus;
}

attr VArea ::= /*empty*/ {
  VArea.nat  = 0.0pt;
  VArea.plus  = 0.0pt;
  VArea.minus = 0.0pt;
}

attr VArea ::= End {
  VArea.nat  = 0.0pt;
  VArea.plus  = 0.0pt;
  VArea.minus = 0.0pt;
}

attr PageArea ::= VArea[figs] Glue[sep] VArea[text] {
  PageArea.nat =
    VArea[figs].nat + VArea[text].nat + glue_nat
    where
      glue_nat =
        if VArea[figs].is_empty || VArea[text].is_empty then 0.0pt
        else Glue[sep].nat;
  PageArea.plus =
    VArea[figs].plus + VArea[text].plus + glue_plus
    where
      glue_plus =
        if VArea[figs].is_empty || VArea[text].is_empty then 0.0pt
        else Glue[sep].plus;
  PageArea.minus =
    VArea[figs].minus + VArea[text].minus + glue_minus
    where
      glue_minus =
        if VArea[figs].is_empty || VArea[text].is_empty then 0.0pt
        else Glue[sep].minus;
}
```

In order to be able to create neither underfull nor overfull pages we introduce the inherited attributes *min* and *max* giving the minimum and maximum height of *VArea*s. For

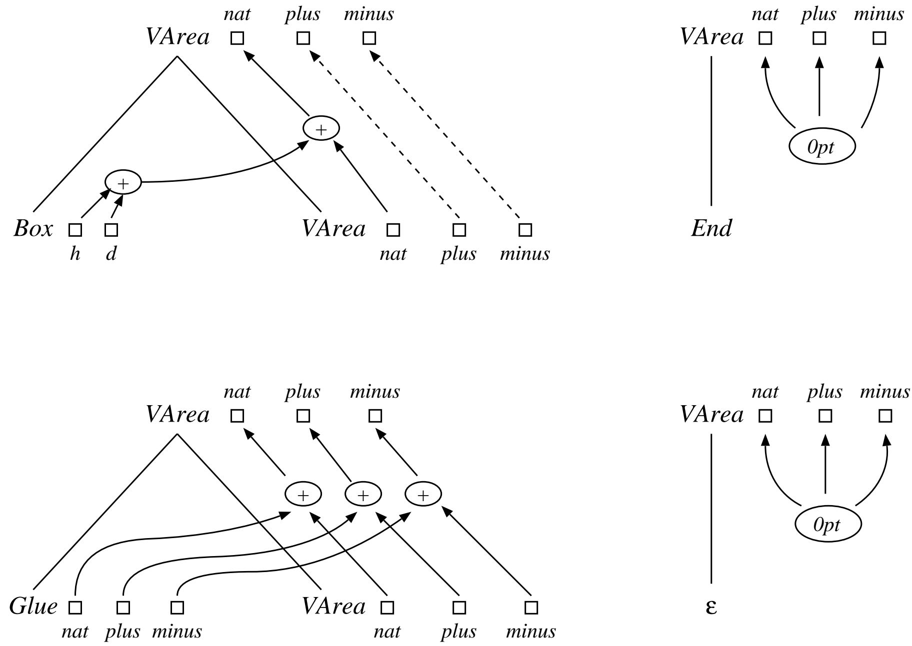


Figure 5.2: Computation of natural height, stretchability and shrinkability of areas (1)

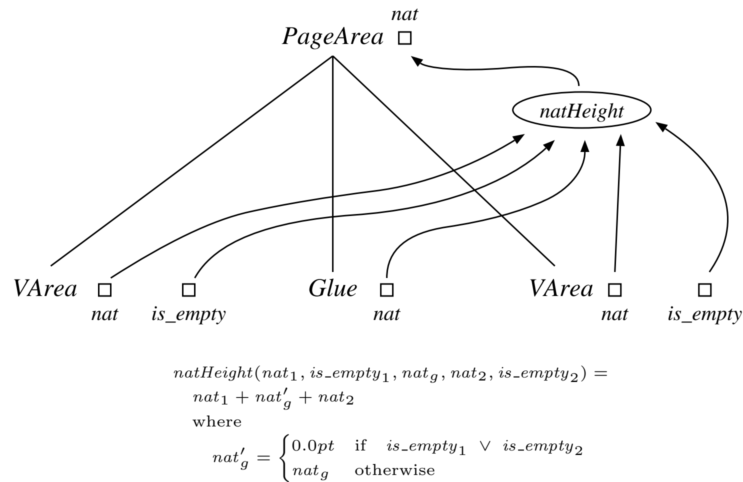


Figure 5.3: Computation of natural height, stretchability and shrinkability of areas (2)

*PageArea*s, the subarea for figures gets a minimum height of *0pt* and a maximum height of the page height; the remaining space on the page is assigned to the subarea for text. The attribution is shown in Figure 5.6 on page 110 and Figure 5.7 on page 111. For *VArea*s consisting of a box or glue and a subarea, the subarea is assigned the *VArea*'s

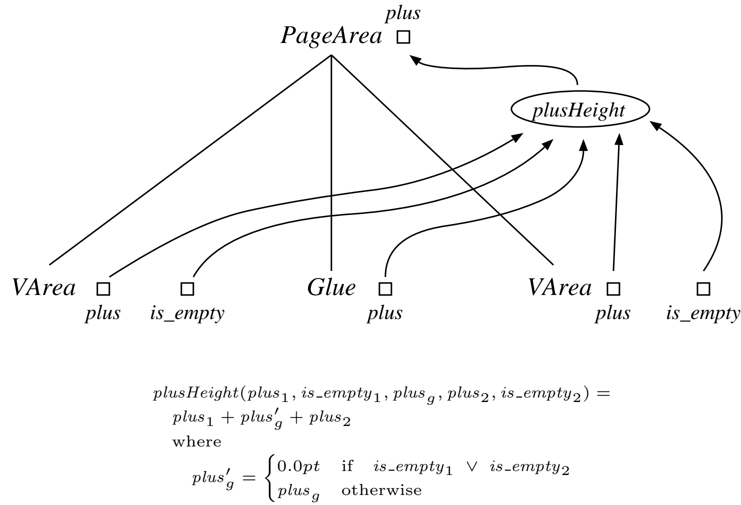


Figure 5.4: Computation of natural height, stretchability and shrinkability of areas (3)

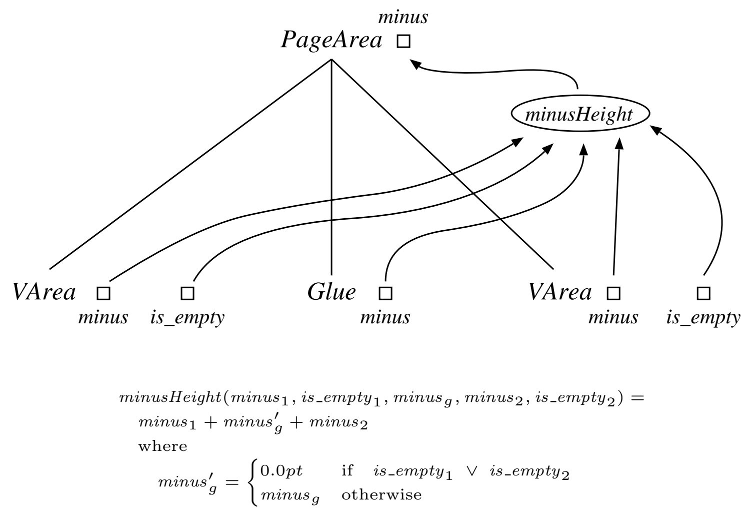


Figure 5.5: Computation of natural height, stretchability and shrinkability of areas (4)

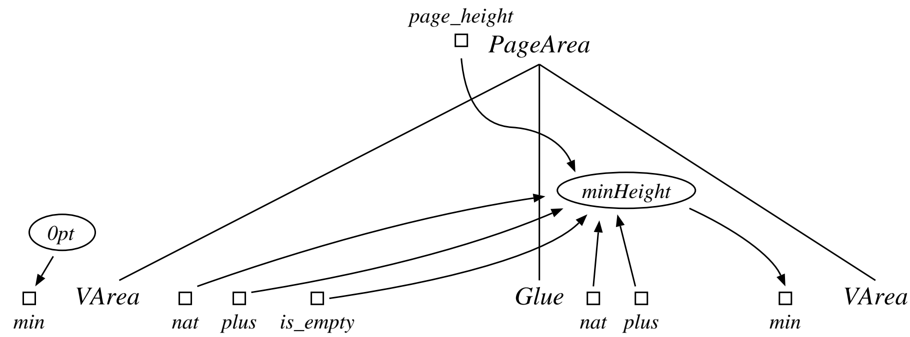
space minus the space used up by the box or glue. The attribution of *VArea*s is depicted in Figure 5.8 on page 111 and specified using Coala as follows.

```
attr PageArea ::= VArea[figs] Glue[sep] VArea[text] {
  VArea[figs].min =0pt;
  VArea[figs].max =PageArea.page_height;
  VArea[text].min =
    PageArea.page_height − VArea[figs].nat − VArea[figs].plus − maxSepHeight
```

```
    where
      maxSepHeight = if VArea[figs]. is_empty then 0pt
                     else Glue[sep]. nat + Glue[sep]. plus;
  VArea[text]. max =
    PageArea.page_height − VArea[figs]. nat + VArea[figs]. minus − minSepHeight
    where
      minSepHeight = if VArea[figs]. is_empty then 0pt
                     else Glue[sep]. nat − Glue[sep]. minus;
}

attr VArea[0] ::= Box VArea[1] {
  VArea[1]. min = VArea[0]. min − Box.h − Box.d;
  VArea[1]. max = VArea[0].max − Box.h − Box.d;
}

attr VArea[0] ::= Glue VArea[1] {
  VArea[1]. min = VArea[0]. min − (Glue.nat + Glue. plus);
  VArea[1]. max = VArea[0].max − (Glue.nat − Glue.minus);
}
```



$$minHeight(pageHeight, nat_1, plus_1, is_empty_1, nat_2, plus_2) = pageHeight - (nat_1 + plus_1) - max_sep_height$$

where

$$max_sep_height = \begin{cases} 0.0pt & \text{if } is_empty_1 \\ nat_2 + plus_2 & \text{otherwise} \end{cases}$$

Figure 5.6: Computation of minimum heights for *PageArea*s

For computation of actual glue sizes, a *PageArea*'s natural height *nat*, stretch factor *plus* and shrink factor *minus*, and the desired page height *page_height* is taken into account and an adjust factor α is computed such that

$$nat + \alpha \times plus = page_height$$

or

$$nat + \alpha \times minus = page_height$$

depending on whether *page_height* is greater than or less than *nat*. If such an α does not exist it is assigned an infinite value. The actual size *act_size* of each *Glue* is then

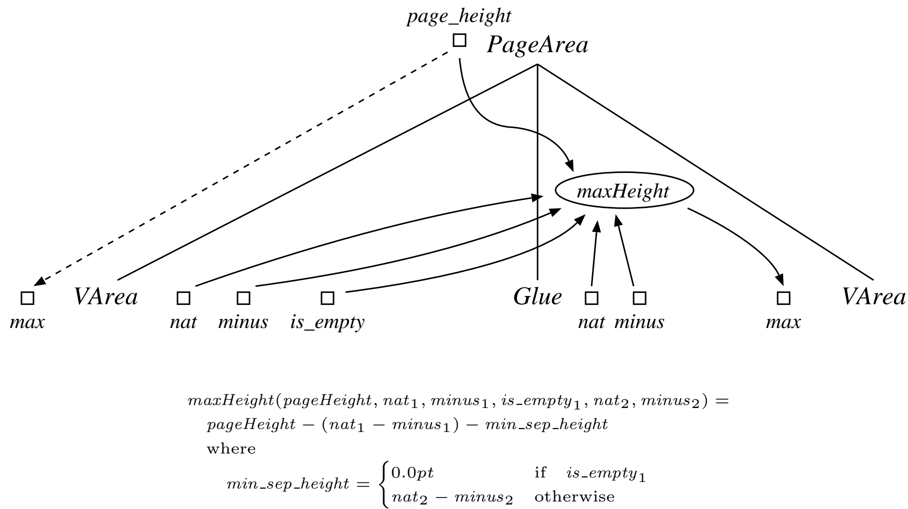


Figure 5.7: Computation of maximum heights for *PageArea*s

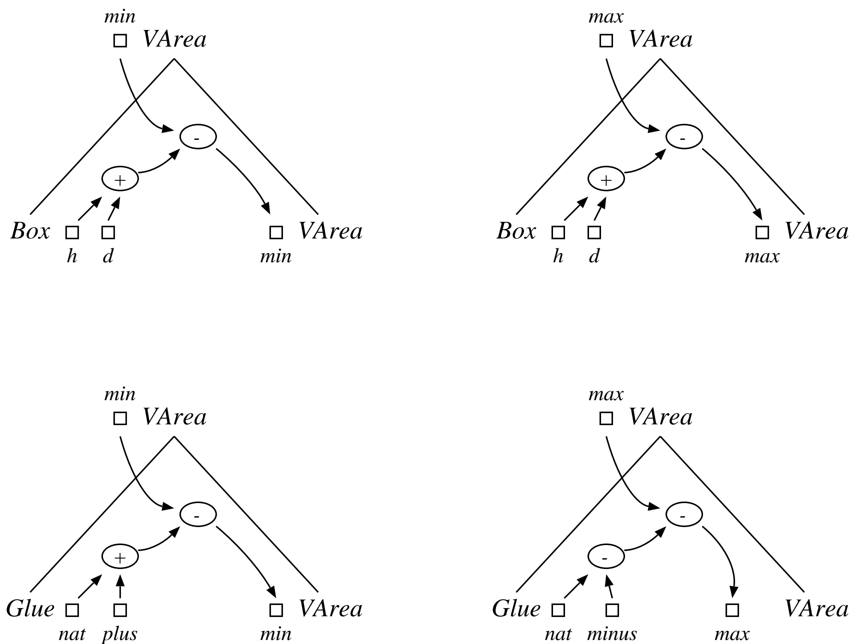


Figure 5.8: Computation of minimum and maximum heights for *VArea*s

computed as $nat + \alpha \times plus$ if $\alpha \geq 0$, or $nat + \alpha \times minus$ otherwise. The attribution realizing this is shown in Figure 5.9 on page 113 and Figure 5.10 on page 114. The corresponding Coala code is as follows.

```
fct adjustFactor(nat, plus, minus, pageHeight) =
  if nat >pageHeight && minus != 0pt then
```

```
    (pageHeight −nat) / minus
  else if nat <pageHeight && plus != 0pt then
    (pageHeight −nat) / plus
  else infinity ()

fct actSize(adjust, nat, plus, minus) =
  if adjust ≥ 0 then nat +(adjust ∗ plus)
                else nat +(adjust ∗ minus)

attr PageAreaSeq[0] ::= PageArea PageAreaSeq[1] {
  PageArea.adjust =
    if PageArea.is_last then 0.0
    else adjustFactor(PageArea.nat, PageArea.plus, PageArea.minus,
                      PageAreaSeq[0].page_height);
}

attr PageArea ::= VArea[figs] Glue[sep] VArea[text] {
  Glue[sep].actual =
    if VArea[figs].is_empty then 0pt
    else actSize(PageArea.adjust, Glue[sep].nat,
                 Glue[sep].plus, Glue[sep].minus);
}

attr VArea[0] ::= Glue VArea[1] {
  Glue.actual =actSize(VArea[0].adjust, Glue.nat, Glue.plus, Glue.minus);
}
```

An appropriate measure for the cost or badness of a *PageArea* is the adjustment factor's absolute value. The cost of a *PageAreaSeq* we simply define to be the average of all *PageAreas*' cost. The cost is stored in the synthesized attribute *layout_cost*. The corresponding attribution rules are straightforward and therefore omitted.

5.2.2 Managing cross references

One of our requirements is consistent cross referencing, i.e., no figures are placed on a page before their first reference in the text. So we need a way to determine if a *PageAreaSeq* being built is consistent w.r.t. this requirement. We realize this using a new user-defined symbol table-like data structure *XRefTable* keeping track of all figures and references to them in a *PageSeq* structure. A cross reference table contains the following information:

- the current page number
- a set of pairs, each consisting of a label and a page number representing the target objects (in our case figures) placed so far
- a set of pairs, each consisting of a label and a page number representing the references placed so far

All operations on cross reference tables are purely functional, i.e., once constructed, cross reference tables never change; instead, operations adding information to an existing table always construct a fresh new table. The following operations are provided.

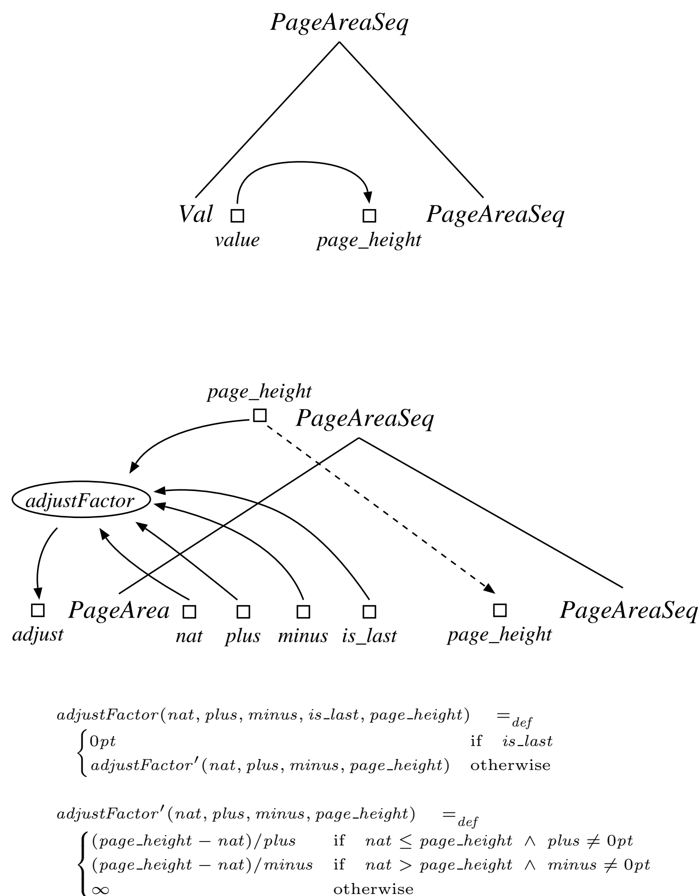


Figure 5.9: Computation of adjustment factor

`XRefTable xref_empty_table()`

Description: Creates a new empty cross reference table with current page number 1

Returns: A new table

`XRefTable xref_add_ref (XRefTable table, String label)`

Description: Adds a cross reference on the current page to an existing table

Parameters:

table: the existing cross reference table

label: the referenced object's label

Returns: A new table with the added reference

`XRefTable xref_add_target(XRefTable table, String label)`

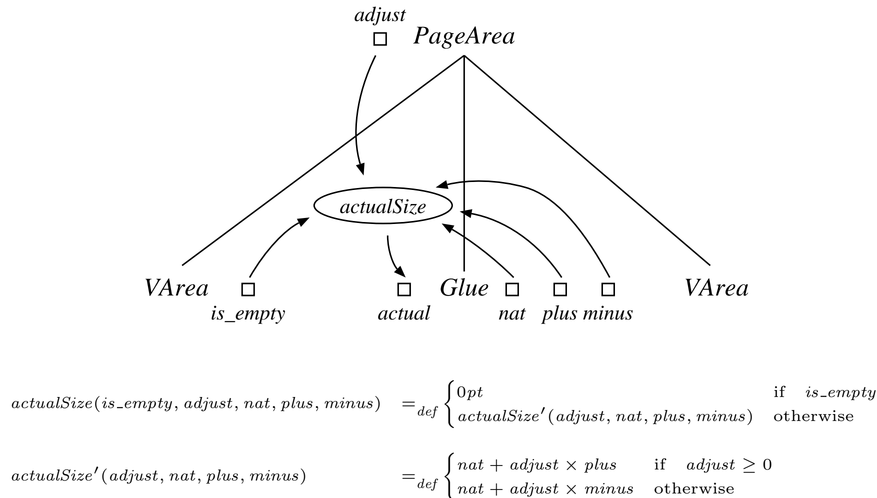


Figure 5.10: Computation of actual glue sizes

Description: Adds a target object (figure) on the current page to an existing table

Parameters:

table: the existing cross reference table

label: the target's label

Returns: A new table with the added target

`XRefTable xref_new_page(XRefTable table)`

Description: Starts a new page

Parameters:

table: an existing cross reference table

Returns: A new table with incremented current page number

`Boolean xref_is_consistent(XRefTable table)`

Description: Determines if a cross reference table is consistent, i.e., contains no target objects (figures) whose page number is less than that of its first reference

Parameters:

table: a cross reference table

Returns: *true* if the table is consistent, *false* otherwise

An inherited attribute *i_xref_table* and a synthesized attribute *s_xref_table* are introduced for passing a cross reference table through a *PageAreaSeq* structure and collecting all figures and references. Figure 5.11 on the facing page shows how the attributes *i_xref_table* and *s_xref_table* are computed. Not shown is that at terminal *Box*es the *i_xref_table* attribute is copied to the *s_xref_table* attribute.

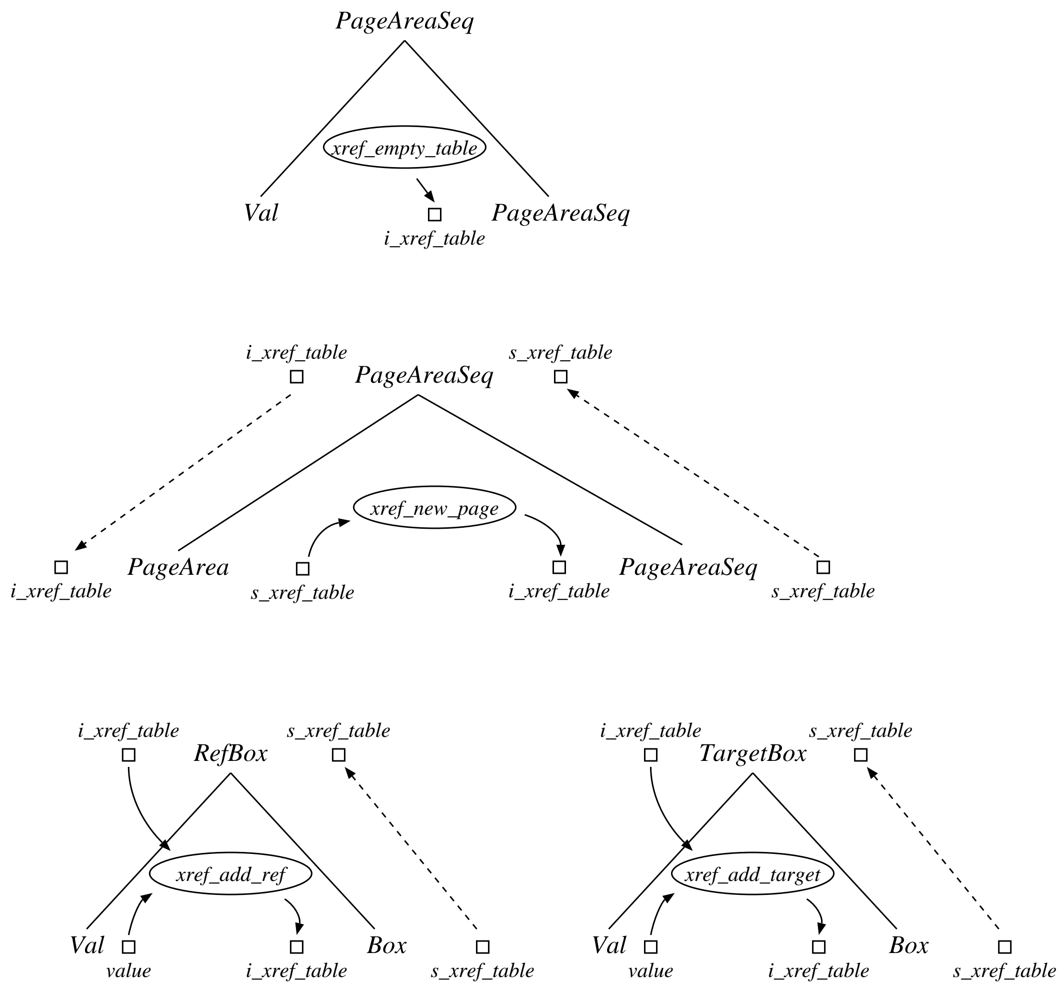


Figure 5.11: Computation of cross referencing information

5.2.3 Transformation rules

We now turn to the transformation rules of our page formatter. The necessary rules can be roughly categorized as follows:

1. creation of (vertical) *BoxKerfSeq*s for text and figures
2. construction of single pages
3. construction of optimal *PageAreaSeq*s
4. transformation of *PageAreaSeq* to a *PageLayout*

In order to save space, Coala rules are only listed for the essential rules (items 2. and 3. in the listing above). The other parts are described in verbal fashion only.

Construction of *BoxKerfSeq*s

The *BoxKerfSeq* for the text is constructed as follows:

- a box for each text line is constructed as described in Section 5.1; for each paragraph a sequence of boxes results
- for each paragraph a box-kerf sequence is constructed such that ☞*widow*s and ☞*orphan*s are avoided; i.e., kerfs are inserted between all adjacent boxes but between the first and the second, and between the second last and the last
- a box-kerf sequence for all text paragraphs results by concatenating all box-kerf sequences for each paragraph, inserting a kerf with a somewhat stretchable and shrinkable glue in between

The box kerf sequence for the document's figures is constructed by creating a placeholder box for each figure (a framed box of the figure's size containing the figure's caption) and inserting a kerf between adjacent figures.

Construction of *PageArea*s

We now define rules constructing a *PageArea* from two source *BoxKerfSeq*s leaving two rest *BoxKerfSeq*s.

As a *PageArea* contains two *VArea*s, we first specify how a single *VArea* is to be constructed from a single box kerf sequence leaving a remaining rest box kerf sequence. We have to make sure that the resulting *VArea*'s height is within its minimum and maximum height, and that if the end of the box-kerf sequence has been reached the resulting *VArea* ens with an *End* element. Here are the corresponding Coala rules.

```
rule @vbreak: BoxKerfSeq<> →VArea<End<>> /BoxKerfSeq

rule @vbreak: BoxKerfSeq[0]<Kerf BoxKerfSeq[1]>
  → VArea<> /BoxKerfSeq[0]
  where
    [ VArea.min ≤ 0.0pt && 0.0pt ≤ VArea.max ];

rule @vbreak: BoxKerfSeq[0]<Kerf BoxKerfSeq[1]>
  → VArea[0]<Glue VArea[1]> / BoxKerfSeq[rest]
  where
    [ VArea[0].max >0.0pt ];
    Kerf → Glue;
    @vbreak: BoxKerfSeq[1] → VArea[1] / BoxKerfSeq[rest];

rule @vbreak: BoxKerfSeq[0]<Box[1] BoxKerfSeq[1]>
  → VArea[0]<Box[2] VArea[1]> / BoxKerfSeq[rest]
  where
    Box[1] → Box[2];
    [ VArea[1].max >0.0pt ];
    @vbreak: BoxKerfSeq[1] → VArea[1] / BoxKerfSeq[rest];

rule Kerf<Val[nat] Val[plus] Val[minus]>
  → Glue<Val[nat] Val[plus] Val[minus]>
```

One note should be made about the second rule which implements breaking at a kerf. The rest box-kerf sequence is in this case the complete input box-kerf sequence including the initial kerf. The reason for this is that if the initial kerf were removed the next page would necessarily contain the next box as there would be no more kerf to break at. That means, e.g., that all figures would be placed on consecutive pages which is clearly not intended.

Using the above `@vbreak` rules it is now easy to specify the construction of pages.

```
rule @PageArea: BoxKerfSeq[1] BoxKerfSeq[2]
  → PageArea< VArea[1] Glue[sep]< 20.0pt 20.0pt 2.0pt > VArea[2] >/
     BoxKerfSeq[3] BoxKerfSeq[4]
  where
    @vbreak: BoxKerfSeq[1] → VArea[1] / BoxKerfSeq[3];
    @vbreak: BoxKerfSeq[2] → VArea[2] / BoxKerfSeq[4];
    [ !PageArea.is_empty ];
```

Construction of optimal *PageAreaSeq*s

We are now ready to define the complete transformation of two box-kerf sequences to an optimal sequence of pages. The basic procedure is as follows:

1. construct new *PageArea* (as described above)
2. check that the cross reference table is still consistent
3. continue formatting with remaining box kerf sequences
4. choose resulting *PageAreaSeq* with minimal layout cost

This is specified using Coala like follows.

```
rule @PageAreaSeq: Val[page_height] BoxKerfSeq[1] BoxKerfSeq[2]
  → PageAreaSeq[0]< Val[page_height] PageAreaSeq[1] >
  where
    @PageAreaSeq: BoxKerfSeq[1] BoxKerfSeq[2] →PageAreaSeq[1];

rule @PageAreaSeq: BoxKerfSeq[1]<> BoxKerfSeq[2]<> →PageAreaSeq<>

rule @PageAreaSeq: BoxKerfSeq[1] BoxKerfSeq[2]
  → PageAreaSeq[0]< PageArea PageAreaSeq[1] >
  where
    [ !(BoxKerfSeq[1].isEmpty && BoxKerfSeq[2].isEmpty) ];
    @PageArea: BoxKerfSeq[1] BoxKerfSeq[2]
             → PageArea /BoxKerfSeq[3] BoxKerfSeq[4];
    [ xref_is_consistent (PageArea. s_xref_table ) ];
    @PageAreaSeq: BoxKerfSeq[3] BoxKerfSeq[4] →PageAreaSeq[1];
    minimize PageAreaSeq[0].layout_cost;
```

Construction of a *PageLayout*

In order to obtain a *PageLayout* from a *PageAreaSeq* additional rules are necessary. The main task is to convert each glue element to empty boxes of the glue's actual size.

Further it is necessary to create the margins of each page using empty boxes. And, for the purpose of this chapter, each page box is surrounded with a frame constructed from rule boxes. This straightforward procedure is omitted here.

5.2.4 Example run

We now take a look at an example run of our formatter. As an example document serves a slightly modified article about Murphy's Law from Wikipedia [53]. The document contains some paragraphs of text—including a title, subtitle and section headers using different fonts—and three figures referenced in the text.

The *PageLayout* structure resulting from the formatting process was written to a `Postscript` file using Coala's built-in converter (see Section 4.2.3). The resulting article is shown in Figure 5.12 on the next page and Figure 5.13 on page 120.

5.3 Ideas for further applications

Coala and its transformation model were designed to solve problems from the field of document processing. However, the new transformation model is rather universal and as such applicable in other fields as well. To give an idea, this section briefly outlines a problem from the field of compiler construction that can be approached using Coala and ideas for application in the field of user interfaces.

5.3.1 Code generation optimizing order of execution

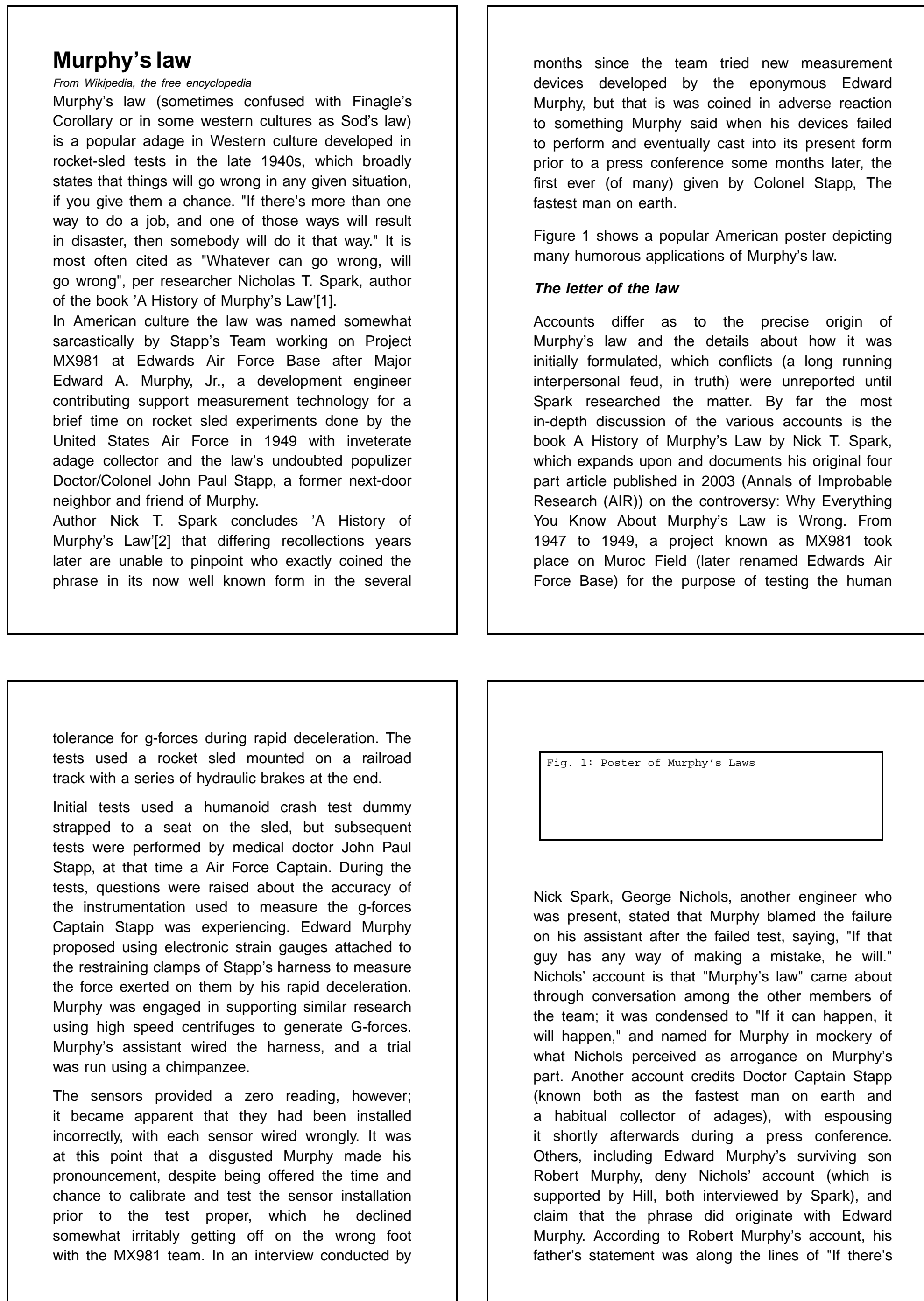
The first example from the field of compiler construction that will be addressed is that of generating optimal machine instruction sequences for binary arithmetic expressions.

Consider a simple stack-based target machine whose instructions are shown in Table 5.1 where for each instruction the change at the top of the stack is shown.

instruction	stack change		
push x	$\cdots$	$\rightarrow$	$\cdots\ val(x)$
add	$\cdots\ v_1\ v_2$	$\rightarrow$	$\cdots\ (v_1 + v_2)$
sub	$\cdots\ v_1\ v_2$	$\rightarrow$	$\cdots\ (v_1 - v_2)$
sub2	$\cdots\ v_1\ v_2$	$\rightarrow$	$\cdots\ (v_2 - v_1)$

Table 5.1: Instructions of simple target machine; x stands for a (symbolic) memory location, $val(x)$ for the numeric value at that location, and v_1, v_2 for numeric values

For a given arithmetic expression containing binary operators multiple target code sequences exist, as the two sub expressions of a binary expression can be evaluated in any order. As an example, consider the expression $(a + b) - \big(c - (d + e)\big)$. Table 5.2 on page 120 shows two possible sequences of target machine instructions implementing this expression (such that the expression's result is on the top of the stack after executing the instruction sequences). The instruction sequence on the left hand side always evaluates the left hand side expression of each binary expression first, while the sequence on the right hand side always evaluates sub expressions in reverse order.



Murphy's law

From Wikipedia, the free encyclopedia

Murphy's law (sometimes confused with Finagle's Corollary or in some western cultures as Sod's law) is a popular adage in Western culture developed in rocket-sled tests in the late 1940s, which broadly states that things will go wrong in any given situation, if you give them a chance. "If there's more than one way to do a job, and one of those ways will result in disaster, then somebody will do it that way." It is most often cited as "Whatever can go wrong, will go wrong", per researcher Nicholas T. Spark, author of the book 'A History of Murphy's Law'[1].

In American culture the law was named somewhat sarcastically by Stapp's Team working on Project MX981 at Edwards Air Force Base after Major Edward A. Murphy, Jr., a development engineer contributing support measurement technology for a brief time on rocket sled experiments done by the United States Air Force in 1949 with inveterate adage collector and the law's undoubted populizer Doctor/Colonel John Paul Stapp, a former next-door neighbor and friend of Murphy.

Author Nick T. Spark concludes 'A History of Murphy's Law'[2] that differing recollections years later are unable to pinpoint who exactly coined the phrase in its now well known form in the several months since the team tried new measurement devices developed by the eponymous Edward Murphy, but that is was coined in adverse reaction to something Murphy said when his devices failed to perform and eventually cast into its present form prior to a press conference some months later, the first ever (of many) given by Colonel Stapp, The fastest man on earth.

Figure 1 shows a popular American poster depicting many humorous applications of Murphy's law.

The letter of the law

Accounts differ as to the precise origin of Murphy's law and the details about how it was initially formulated, which conflicts (a long running interpersonal feud, in truth) were unreported until Spark researched the matter. By far the most in-depth discussion of the various accounts is the book A History of Murphy's Law by Nick T. Spark, which expands upon and documents his original four part article published in 2003 (Annals of Improbable Research (AIR)) on the controversy: Why Everything You Know About Murphy's Law is Wrong. From 1947 to 1949, a project known as MX981 took place on Muroc Field (later renamed Edwards Air Force Base) for the purpose of testing the human tolerance for g-forces during rapid deceleration. The tests used a rocket sled mounted on a railroad track with a series of hydraulic brakes at the end.

Initial tests used a humanoid crash test dummy strapped to a seat on the sled, but subsequent tests were performed by medical doctor John Paul Stapp, at that time a Air Force Captain. During the tests, questions were raised about the accuracy of the instrumentation used to measure the g-forces Captain Stapp was experiencing. Edward Murphy proposed using electronic strain gauges attached to the restraining clamps of Stapp's harness to measure the force exerted on them by his rapid deceleration. Murphy was engaged in supporting similar research using high speed centrifuges to generate G-forces. Murphy's assistant wired the harness, and a trial was run using a chimpanzee.

The sensors provided a zero reading, however; it became apparent that they had been installed incorrectly, with each sensor wired wrongly. It was at this point that a disgusted Murphy made his pronouncement, despite being offered the time and chance to calibrate and test the sensor installation prior to the test proper, which he declined somewhat irritably getting off on the wrong foot with the MX981 team. In an interview conducted by

Fig. 1: Poster of Murphy's Laws

Nick Spark, George Nichols, another engineer who was present, stated that Murphy blamed the failure on his assistant after the failed test, saying, "If that guy has any way of making a mistake, he will." Nichols' account is that "Murphy's law" came about through conversation among the other members of the team; it was condensed to "If it can happen, it will happen," and named for Murphy in mockery of what Nichols perceived as arrogance on Murphy's part. Another account credits Doctor Captain Stapp (known both as the fastest man on earth and a habitual collector of adages), with espousing it shortly afterwards during a press conference. Others, including Edward Murphy's surviving son Robert Murphy, deny Nichols' account (which is supported by Hill, both interviewed by Spark), and claim that the phrase did originate with Edward Murphy. According to Robert Murphy's account, his father's statement was along the lines of "If there's

Figure 5.12: Example article (1)

Table 5.2 on the next page also shows the number of temporary values on the stack after each instruction. Observe that the instruction sequence on the left hand side requires one more temporary cell on the stack. If, for instance, the target machine uses

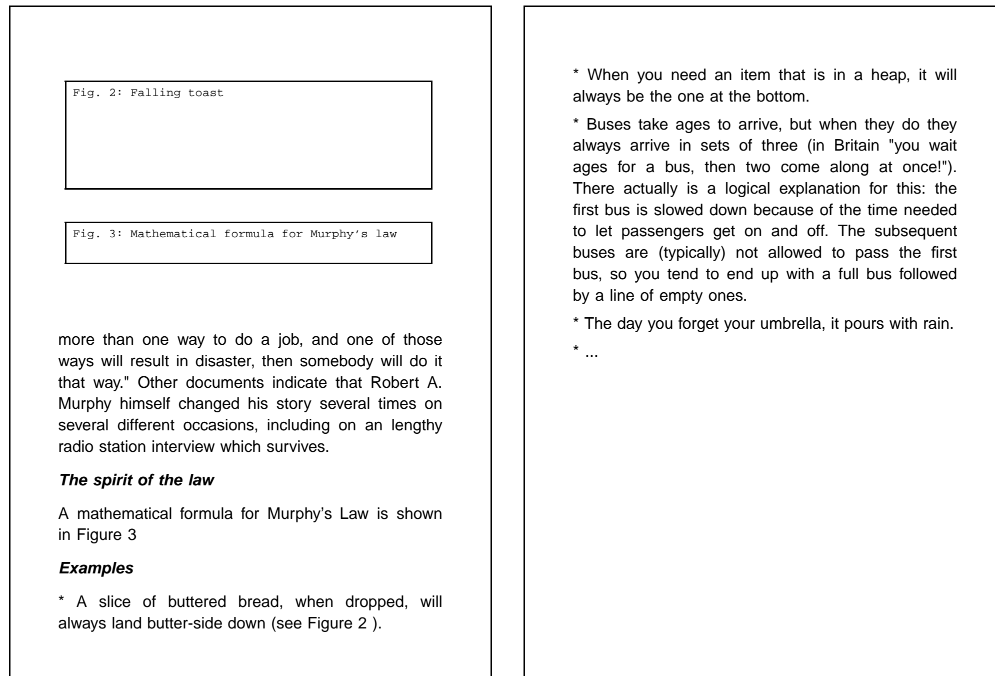


Fig. 2: Falling toast

Fig. 3: Mathematical formula for Murphy's law

more than one way to do a job, and one of those ways will result in disaster, then somebody will do it that way." Other documents indicate that Robert A. Murphy himself changed his story several times on several different occasions, including on an lengthy radio station interview which survives.

The spirit of the law

A mathematical formula for Murphy's Law is shown in Figure 3

Examples

* A slice of buttered bread, when dropped, will always land butter-side down (see Figure 2).

* When you need an item that is in a heap, it will always be the one at the bottom.

* Buses take ages to arrive, but when they do they always arrive in sets of three (in Britain "you wait ages for a bus, then two come along at once!"). There actually is a logical explanation for this: the first bus is slowed down because of the time needed to let passengers get on and off. The subsequent buses are (typically) not allowed to pass the first bus, so you tend to end up with a full bus followed by a line of empty ones.

* The day you forget your umbrella, it pours with rain.

* ...

Figure 5.13: Example article (2)

instruction sequ. 1	stack level		instruction sequ. 2	stack level
push a	1		push d	1
push b	2		push e	2
add	1		add	1
push c	2		push c	2
push d	3		sub2	1
push e	4		push a	2
add	3		push b	3
sub	2		add	2
sub	1		sub2	1

Table 5.2: Two target instruction sequences for the expression $(a+b) - \big(c - (d+e)\big)$

a register stack the machine could run out of registers and be required to use slower memory if excessive temporary storage is used. This can have a significant impact on performance, so it is desired to use as little temporary storage on the stack as necessary.

It is possible to decide which sub expression to evaluate first by computing the amount of required temporary storage for each expression first (as described e.g. in [56]). The amount of temporary storage for a binary expression is then minimized by first evaluating the sub expression requiring more space than the other (if both sub expressions require the same space evaluation order does not matter).

However, code generation minimizing the required temporary storage can also be

specified using Coala as is described in the following. First, source and target grammar is specified.

```
terminal Loc;

productions {
  Expr      ::= Plus | Minus | Loc;
  Plus      ::= Expr[1]  Expr [2];
  Minus     ::= Expr[1]  Expr [2];
}

terminal Add, Sub, Sub2;

productions {
  Code      ::= Conc | Push | Add | Sub | Sub2;
  Conc      ::= Code[1]  Code[2];
  Push      ::= Loc;
}

synthesized requ_storage with Code;
```

The required storage is here realized as a synthesized attribute of the target instruction sequence (the attribution rules are straightforward and have been omitted). The transformation of a source expression to target code minimizing the amount of required temporary storage can be specified in Coala like so:

```
rule @code: Expr< Loc >→Code< Push< Loc >>

rule @code: Expr[0]< Plus< Expr[1] Expr[2] >>
  → Code[0]< Conc[1]< Code[a1]< Conc[2]< Code[1] Code[2] >>Code[a2]< Add >>>
  where
    @code: Expr[1] → Code[1];
    @code: Expr[2] → Code[2];
    minimize Code[0].requ_storage;

rule @code: Expr[0]< Plus< Expr[1] Expr[2] >>
  → Code[0]< Conc[1]< Code[a1]< Conc[2]< Code[1] Code[2] >>Code[a2]< Add >>>
  where
    @code: Expr[2] → Code[1];
    @code: Expr[1] → Code[2];
    minimize Code[0].requ_storage;

rule @code: Expr[0]< Minus< Expr[1] Expr[2] >>
  → Code[0]< Conc[1]< Code[a1]< Conc[2]< Code[1] Code[2] >>Code[a2]< Sub >>>
  where
    @code: Expr[1] → Code[1];
    @code: Expr[2] → Code[2];
    minimize Code[0].requ_storage;

rule @code: Expr[0]< Minus< Expr[1] Expr[2] >>
  → Code[0]< Conc[1]< Code[a1]< Conc[2]< Code[1] Code[2] >>Code[a2]< Sub2 >>>
  where
```

```
@code: Expr[2] → Code[1];
@code: Expr[1] → Code[2];
minimize Code[0].requ_storage;
```

This raises an interesting point: while it is a known fact that the optimal order of evaluation can be determined by analyzing the source expression, in the Coala approach all possible target instruction sequences are generated; cost (required space) is associated with *target* structures which is used to select optimal results. The Coala approach is more declarative, but at the same time less efficient. Thus, it would be very interesting if one could decide in a more general setting which transformation rule's application leads to an optimal result without actually trying all applicable rules.

5.3.2 User interface layout

Another interesting application of Coala would be for optimizing layout of graphical user interfaces. The key aspect here is that many logical elements of user interfaces can be presented to the user in multiple ways. For instance, selecting an item from a given list can be realized using a group of radio buttons or a pop-up menu. Depending on factors like the number of items in the list and the context within the rest of the user interfaces each of the two alternatives can be chosen to be better.

Using Coala it would be possible to globally optimize layouts of user interfaces by specifying all alternatives and applying some measure of visual appearance and usability.

Besides that, user interface layout raises similar problems as occur with document processing. Consider for example a large dialog for editing user preferences where items are organized on pages of a tabbed pane. This resembles the problem of page breaking described before.

While it is certainly generally desirable to have user interfaces optimized for visual appearance and usability, this is particularly important for devices with small displays and limited interaction facilities such as cell phones or PDAs. Coala could be used to automatically produce optimal user interface layouts given the individual characteristics of a target device.

CHAPTER 6

Related Work

In this chapter we take a look at some existing document processing systems and their approach to document formatting. These systems are:

- ODA which is of historical importance
- SGML/DSSSL and XML/XSL which apply an approach to document formatting being widely in use today, and
- Agenda which has up to now the most advanced formatting model known from research

It will be shown that Coala's new approach to document formatting developed in this work has significant advantages as compared to these existing systems.

6.1 ODA

ODA (Office Document Architecture, or more recently Open Document Architecture) was developed in the 1980s in order to facilitate the creation, processing, and interchange of documents in heterogeneous computing systems [19]. Although ODA is only of minor practical relevance today, it is historically important as one of the first approaches applying the structured document model.

6.1.1 ODA's document model

ODA uses an object-oriented model to describe both logical and layout document structures. A document is described as a tree of objects where leaves represent the actual document content. Inner nodes may be tagged with a *user visible name* which is helpful

while creating and editing a document and which allows to define generic document structures as described below. Each child of a node carries an unambiguous sequence number which makes it possible to identify every node of a document as a sequence of numbers. The content nodes at a document tree's leaves must conform to a predefined content architecture and may contain text, raster graphics and vector graphics, but also special content like movie and sound clips, etc. Content nodes may appear only as children of a special type of nodes (*basic logical objects*).

ODA allows to describe the generic structure of classes of documents by specifying the composition of allowable children for given logical objects. This is done in a way comparable with productions of a context-free grammar allowing regular right hand sides.

A limited form of attribution is supported by ODA. Attributes can be specified for nodes (either in the specific or generic document structure) and may be inherited by descendant nodes, comparable with inherited attributes of an attribute grammar using only copy rules.

As opposed to logical document structures, ODA imposes some restrictions on layout document structures. Layout structures may only be composed of the predefined elements *page set*, *composite page*, *basic page*, *frame*, and *block*, nested in the given order.

6.1.2 Formatting of documents

ODA also defines the process of creating layouts from logical documents. Here, content objects are first transformed into blocks, taking into account the content architectures being involved and *presentation styles* (given as attributes). A layout structure is then built from the created blocks, incorporating the generic layout document structure and some attributes from the logical document structure.

6.1.3 Comparison with Coala

In comparison with Coala the following main differences are recognized:

- as opposed to Coala, ODA's support for the attribution of document structures is very limited and does not allow to describe complex semantic properties
- ODA imposes restrictions on the target structure; Coala allows to specify transformation to arbitrary target structures
- an essential part of the formatting process is predefined through given content architectures; using Coala even basic formatting tasks can be precisely specified
- the formatting process is limited at higher levels in ODA as well; for instance, logical objects are always visited strictly sequentially (it is however possible to use multiple streams like for floating figures)

6.2 SGML/DSSSL, XML/XSL

An approach to document formatting in common use today is followed by SGML/DSSSL and, more recently, XML/XSL. SGML and XML provide means to describe the generic and

specific structure of documents; DSSSL and XSL define a universal intermediate language of *formatting object trees* describing document layouts in an abstract way, and a tree transformation language allowing to describe transformations from logical document structures to the intermediate structure. The intermediate language represents layout still in an abstract way, so the transformation from a logical document structure to it is usually quite simple. A processor for the intermediate language, usually implemented in a general-purpose programming language, is then used to create concrete layout structures.

6.2.1 SGML's and XML's document model

SGML and XML use a similar approach to the structured document model. Documents are described as a hierarchy of nested elements and character content. Each element has a name assigned and has an arbitrary set of attributes given as pairs of an attribute name and a string value. The linear form of a document appropriate for storage and interchange is given in a fully bracketed way where each element is enclosed in a *start tag* containing the element's name and attributes and an *end tag* again containing the element's name. SGML allows start or end tags to be omitted as long as a document's structure can be still recreated in an unambiguous way.

The generic structure of documents can be defined in both SGML and XML using a *document type definition.* A document type definition allows to define the allowable sub elements and character content of an element referenced by its name using context-free productions with regular right hand sides, and allows to specify which attributes can be associated with elements. Besides document type definitions, more powerful languages for defining the generic structure of XML exist, as for example the XML Schema language [47].

Many languages based on SGML and XML describing both logical and layout document structures exist. To name just a few, XHTML [46] allows to describe the logical structure of World Wide Web pages, DocBook is a language for book-like documents and reports, targeted at technical documentation; SVG [44] is a language for documents containing vector graphics.

6.2.2 Flow object trees

Both DSSSL and XSL use an intermediate representation of *formatting object trees* during the formatting process. A formatting object tree describes a document layout in an abstract way, i.e., the complex tasks of formatting take place in a subsequent transformation to a concrete layout document structure. A formatting object tree in XSL contains (slightly simplified):

- A *layout master set* defining the page geometries to be used. It is possible to specify the sequencing of pages, like, e.g., the repetition of a combination of left and right hand side pages for book-like documents. Each page master defines a set of *regions* into which the flows of the document's content are distributed.
- One or more *page sequences* defining the document's content. Each page sequence references a page (sequence) master defined in the layout master set and contains one or more *flows.* A flow is assigned a name referencing the region of a master

page into which the flow's content is to be distributed. A flow finally contains basic content such as *blocks* (textual content like paragraphs with information about layout styles) and tables.

Figure 6.1 shows a simple example formatting object tree defining only a single page master and a single flow of text with two paragraphs.

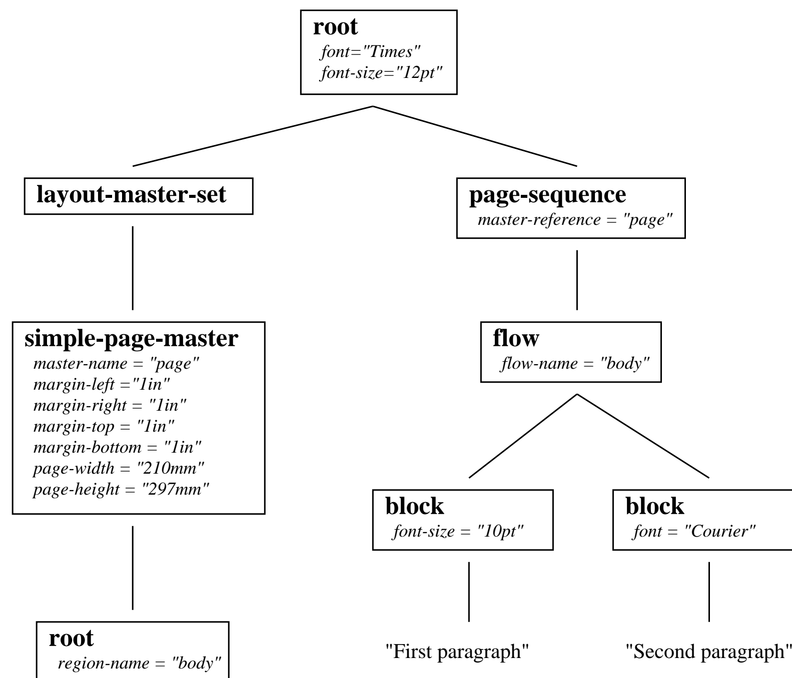


Figure 6.1: Example formatting object tree

6.2.3 Transformation to target structures

XSL and DSSSL use *area trees* as target document structures which describe the formatting result in a geometrically structured way; each area represents a rectangular portion on the output medium. The transformation from a formatting object tree to an area tree is specified for XSL and DSSSL in a mostly verbal fashion.

6.2.4 Comparison with Coala

SGML and XML do not provide means to describe semantic properties of structured documents. However, Coala has built-in support for handling XML documents. So the semantics of XML documents can be specified using an attribute grammar in Coala.

As the transformation from formatting object trees to area trees is specified in a mostly verbal fashion for both XSL and DSSSL, known implementations are available in general-purpose programming languages only. This has the following shortcomings:

- as the underlying specifications are complex, it is difficult to create implementations at all

- it is hard to verify that implementations fully conform to the specifications
- implementations are difficult to maintain and extend

Coala allows to specify such transformations declaratively in an executable way. No implementation needs to be created by hand, so the shortcomings given above are eliminated.

Coala would also allow to specify transformations from logical document structures to flow object trees and as such replace DSSSL's and XSL's transformation languages. Here, the transformation specification could benefit from using attribute grammars to specify semantic properties of logical document structures like, e.g., section numbering.

6.3 Agenda

Agenda [43] is a system allowing to specify document formatters on a very high level.

Agenda uses attribute grammars to specify generic document structures and their semantic properties.

Agenda's formatting model is based on the one described in [34] which does not apply the structured document model. In this model, formatting source and target are described as a hierarchy of objects which can contain one or more streams of subordinate objects. During a formatting process objects from the source streams are poured into layout objects specified by *layout templates* which are conceived as containers of constrained capacity.

Agenda extends this formatting model allowing the application of a structured document model where document classes are specified using attribute grammars. *Multistream type definitions* are used to identify the hierarchy of streams of sub documents in both source and target documents. *Layout references* describe the mapping from source sub documents to target sub documents and correspond to the layout templates used in the original formatting model. A Boolean attribute is used to realize the constrained capacity of target sub documents. A target sub document is extended as long as this attribute indicates that the target sub document is still consistent.

Agenda allows to specify document formatters in a declarative way on a very high level. Document formatters are automatically generated from these specifications.

In comparison with Coala the following similarities and differences can be observed:

- Coala and Agenda share the same document model: in both systems an attribute grammar based structured document model is used.
- Both Coala and Agenda do not impose any restriction on source and target structures. As opposed to ODA and DSSSL/XSL any document structure can be the target of a transformation process. Document formatters are not specified using document processing specific terms and concepts which makes the systems more universally usable exceeding the realm of document processing. Another similarity of Coala and Agenda is that target structures can serve as source structures of subsequent transformations, i.e., transformation processes can be specified using intermediate structures.

- Due to its underlying pouring model Agenda formatters are restricted to produce "last-fit" type results (cf. Chapter 2 and Chapter 5). It is not possible to specify formatters optimizing for any given measures. It would also not be easy to extend Agenda by such a feature. As opposed to that, Coala allows to specify formatters optimizing for any semantic property that can be described using an objective function applied to attributes specified through an attribute grammar.

- Formatters are specified in Agenda on a very abstract level using streams in source and target structures and layout references between them. While allowing very concise specifications, a minor downside of this approach is that sub documents can be processed in a fixed sequential order only. Coala requires to specify the mapping from document structures to streams and back. This approach allows more freedom in processing of document structures.

6.4 (Constraint) Logic Programming

In this section some interesting similarities of logic programming and its extension constraint logic programming with the new tree transformation model introduced in this work are pointed out.

First, logic programs and their relationship to tree transformation specifications without semantic constraints are described. Then, it will be shown how constraint logic programs relate to transformation specifications with semantic constraints.

6.4.1 Logic programs

A logic program, as e.g. known from `Prolog`, consists of a finite set of clauses, each of which having the form

$$p_0(\bar{t}_0) \leftarrow p_1(\bar{t}_1) \ \wedge \ p_2(\bar{t}_2) \ \wedge \ \ldots \ \wedge \ p_n(\bar{t}_n)$$

where $n \geq 0$, p_i are predicate symbols, and $\bar{t}_i$ are sequences of terms over a set of function symbols, a set of constants, and a set of variables. The interpretation of a clause is that the left hand side formula can be derived to be true if the right hand side formula can be derived to be true. A clause with $n = 0$ is called a fact.

For example, given a constant *nil*, function symbol *cons*, predicate symbol *append* and variables W, X, Y, Z, the following is a simple logic program describing a ternary relation between lists:

$$\begin{aligned}
&append(nil, Y, Y) \leftarrow \\
&append\big(cons(W, X), Y, cons(W, Z)\big) \leftarrow append(X, Y, Z)
\end{aligned}$$

The intended meaning is that $append(X, Y, Z)$ is true whenever the list Z is a concatenation of lists X and Y.

A *query* has the form

$$\leftarrow p(\bar{t})$$

and can be interpreted as the question "is there a substitution for the variables in $\bar{t}$ with ground terms such that $p(\bar{t})$ can be derived form the logic program's clauses". A logic programming system typically answers to such a query with an enumeration of all such substitutions, or with yes or no if $\bar{t}$ contains no variables.

Logic programs are executed using a mechanism called *resolution* which finds answers using unification and backtracking.

Transformation specifications without semantic constraints can be implemented as logic programs in a quite straightforward manner. To see this, consider as an example the following transformation specification (which is similar to the one given in Example 3.2.6 on page 34 with semantic constraints removed)[1]:

$$
\begin{aligned}
&S_0 \langle\, B \langle\, S_1 S_2 \,\rangle\,\rangle \to E \langle\, H \langle\, E_1 E_2 \,\rangle\,\rangle \\
&\quad \textsf{where} \quad S_1 \to E_1, \\
&\quad\phantom{\textsf{where}} \quad S_2 \to E_2 \\
&S_0 \langle\, B \langle\, S_1 S_2 \,\rangle\,\rangle \to E \langle\, V \langle\, E_1 E_2 \,\rangle\,\rangle \\
&\quad \textsf{where} \quad S_1 \to E_1, \\
&\quad\phantom{\textsf{where}} \quad S_2 \to E_2 \\
&S \langle\, L \,\rangle \to E \langle\, A \,\rangle
\end{aligned}
$$

This transformation specification can be turned into the following logic program:

$$
\begin{aligned}
&\mathit{trafo}\big(s(b(S_1,S_2)), e(h(E_1,E_2))\big) \leftarrow \mathit{trafo}(S_1,E_1) \ \wedge\ \mathit{trafo}(S_2,E_2) \\
&\mathit{trafo}\big(s(b(S_1,S_2)), e(v(E_1,E_2))\big) \leftarrow \mathit{trafo}(S_1,E_1) \ \wedge\ \mathit{trafo}(S_2,E_2) \\
&\mathit{trafo}\big(s(l), e(a)\big) \leftarrow
\end{aligned}
$$

Here, s, b, e, h and v are function symbols representing the corresponding syntactic symbols with capital letters; l and a are constants representing the terminal syntactic symbols L and A. Variables are denoted by capital letters. Note, however, that variables in terms cannot be constrained to only be substituted with terms constructed with a given function symbol or constant.

Given the query

$$\leftarrow \mathit{trafo}(s(b(s(b(s(l),s(l))),s(l))),X)$$

the system would emit four terms corresponding to the trees depicted in Figure 3.6 on page 40 and Figure 3.6 on page 40 (without the root nodes of symbol T). Note that the semantic constraints in the original example ruled out one of the four trees.

So, transformation specifications without semantic constraints have a strong relationship to logic programs. Note that rests in transformation rules and syntactic constraints, which are not used in the above simple examples, can be realized in logic programs as well by using a ternary transformation relation.

However, semantic constraints and optimization cannot be realized using a logic program—at least not in a straightforward manner.

[1]Note that the empty rest parts have been omitted.

6.4.2 Constraint logic programs

The objects a logic program as described above deals with are (ground) terms with no special meaning. Constraint logic programming (CLP) languages [20] extend a logic programming language by a semantic domain $\mathcal{D}$ and constraints over $\mathcal{D}$. Rules in a constraint logic program then have the form

$$p_0(\bar{t}_0) \leftarrow p_1(\bar{t}_1) \ \wedge \ p_2(\bar{t}_2) \ \wedge \ \ldots \ \wedge \ p_n(\bar{t}_n) \ \wedge \ c_1 \ \wedge \ c_2 \ldots \ \wedge \ c_m$$

where all c_i are constraints. For example, [20] introduces CLP(R) which provides equations and inequations such as $X + Y > 0$ or $Y = Z$ on real numbers.

Given a query, in extension to the search for solutions as described above for pure logic programs, a CLP system manages a constraint store. Constraints are always solved as far as possible. If the constraints in the constraint store are determined not to be satisfiable the CLP system backtracks like a normal logic programming system. This way constraints are used to prune the search space as much as possible.

In comparison with Coala and its underlying new tree transformation model introduced in this work the following can be noticed:

- like in CLP, Coala uses semantic domains and constraints in addition to dealing with term/tree structures
- while in a constraint logic program extra variables are used to hold semantic values, with Coala semantic properties are attached to tree nodes as attributes; this can be seen as an interesting new variant of CLP
- while a CLP system can deal with unsolved or only partly solved sets of constraints, Coala uses semantic constraints only to check if a possible transformation is valid (or on its way to become valid) which is sufficient for transformations of trees required by document formatters
- due to its simpler use of constraints, Coala allows to specify user-defined semantic domains and constraints, which is achieved simply by defining functions returning Boolean results; CLP systems have fixed built-in semantic domains and adding more requires to implement and integrate in most cases complex constraint solvers

6.5 Conclusion

We have compared Coala's approach to document formatting with important existing approaches.

We have seen that Coala is based upon the same powerful structured document model that is also used in Agenda where the generic structure and semantic properties of documents are specified using attribute grammars. This document model is significantly more powerful than that used in ODA, SGML and XML.

Coala's formatting model has the following advantages in comparison with ODA, XSL and DSSSL:

- transformations to arbitrary target structures are possible; the possible target structures in ODA, XSL and DSSSL are fixed or significantly restricted

- Coala allows to specify formatting processes in a formal and executable way; thus, a formatting process is always specified in a precise way and it is not necessary to manually create implementations in a general-purpose programming language

In comparison with Agenda, Coala allows to specify more powerful formatters optimizing formatting results for given measures. This comes at the price of a slightly less abstract level. As a possible future extension, Coala would greatly benefit from carrying over Agenda's stream model.

Coala could be used in the scope of document processing systems currently used in practice. As an example, it would be possible to specify the transformation process from formatting object trees to area trees described in Section 6.2 using Coala.

We have finally seen an interesting similarity of Coala with (constraint) logic programming.

CHAPTER 7

Summary and outlook

7.1 Summary

This work has introduced a new model for optimizing transformations between attributed trees, targeted at the declarative specification of document formatters. The transformation model is based on rules incorporating the following:

- source and target tree templates
- transformation rest for handling breaking problems often occurring with document formatters
- syntactic constraints describing required relationships between source subtrees and target subtrees
- semantic constraints on attribute occurrences at source and target tree nodes describing required conditions on semantic properties of source and target trees
- optional objective functions allowing to minimize for given semantic properties

The transformation model and its semantics have been defined in a completely formal manner, the semantics being concise and descriptive, yet not constructive. The notion of LR specifications has been introduced which imposes restrictions on the order of constraints in rules, which allows efficient execution of transformations processing constraints strictly from left to right. An execution scheme for LR specifications without handling optimization has been defined whose soundness and completeness w.r.t. the transformation model's semantics has been formally proved. Handling of optimization has been realized by turning this execution scheme into a set of recursive transformation functions returning optimal results for each transformation rule's application. It has

been shown how efficient execution can be achieved by making use of dynamic programming. Further, it has been shown how standard transformation specifications can be automatically turned into LR specifications if possible.

A new system called Coala has been developed consisting of a specification language based on the new transformation model and a set of tools specific to document processing. The specification language allows convenient definition of attribute grammars and transformation specifications. Coala allows to realize complete document formatters by supporting the reading of XML documents and output of Postscript code for a predefined document layout structure. Coala has been implemented in the Java programming language.

The applicability of the new approach to document formatting has been shown by realizing line breaking in several variants and page breaking using Coala. It has further been shown that Coala can be applied in other fields beyond document processing: an example application from the field of compiler construction realizing code generation for binary arithmetic expressions optimizing the order of evaluation has been given; in addition, some promising ideas for applying Coala in the field of user interfaces have been outlined.

The new approach has been compared with previous approaches to document formatting both being of historical importance and being in practical use today. In addition an interesting relationship with (constraint) logic programming has been shown.

7.2 Contributions

The main contribution of this work is a new model for transformations between attributed trees. It has been developed on a very strong formal basis. The model's semantics is formally defined in a very concise and natural way. The correctness of its scheme for execution of transformations w.r.t. the model's semantics is almost completely formally proved. More over, Coala, a system based on the new transformation model has been developed, implemented using the Java programming language. Examples implemented using this new system show that the new approach is not only of theoretical interest but also applicable in practice.

Optimizing document formatters which were difficult to create with conventional approaches using general-purpose programming languages can now be specified on a high level in a declarative, concise and readable way using the new system. Besides the obvious advantages this has the following benefits.

- As formatting constraints and optimization can be specified in a declarative and concise way using the new approach, very difficult formatting problems can now be tackled that could previously not be approached at all; such very hard formatting problems arise with documents such as newspapers (optimal page planning), magazines and product catalogs. In these kinds of documents very complex constraints on the placement of objects occur and optimization in various ways is desired. In today's practice a lot of manual work is still involved which can be approached to be automated using the new approach.

- The new approach allows to declaratively specify layout style guides on a completely new level; existing solutions only allow to specify relatively simple aspects

of layout styles.

Besides document formatting the new approach can be used in other fields as well. Some ideas have been given in this work, but many more possible uses should be explored.

7.3 Possible extensions and improvements

Improvement of Coala is possible in many aspects, two of which are outlined in the following.

As described in Section 6.3 the existing Agenda system allows to specify formatting problems on a very high level, but has little support for formatting constraints and does not allow optimization at all. It would be interesting, yet certainly not easy, to investigate if Agenda's approach based on document streams could be carried over to Coala.

Another idea is to extend Coala's support for constraints and optimization. As has been shown in other work about constraint programming [3, 51], it can be beneficial to have constraints that are not strictly required but desirable to be fulfilled. Such constraints are usually called *preferential*. Preferential constraints arise naturally in document processing; for instance, a desired property for page breaking with footnotes would be to not break footnote bodies across multiple pages if possible. Preferential constraints can already be simulated in Coala by integrating them in to objective functions, i.e., cost rises with each unsatisfied preferential constraint. However, this approach is less declarative and makes it harder to define objective functions.

Bibliography

[1] Adobe Systems Inc.: *PDF Reference, Fifth Edition, Version 1.6*, 2004

[2] A. Aho, R. Sethi, J. Ullman: *Compilerbau, Teile 1 und 2*, Addison-Wesley, 1988

[3] A. Borning, B. Freeman-Benson, M. Wilson: *Constraint Hierarchies*, LISP and Symbolic Computation, 5(3), 1992

[4] K. P. Brooks: *A Two-View Document Editor with User-defineable Document Structure*, SRC Report Nr. 33, DEC, 1988

[5] A. Brüggemann-Klein, R. Klein, S. Wohlfeil: *Pagination Reconsidered*, Electronic Publishing, VOL. 8(2 & 3), 1995

[6] D. Day, M. Priestley, D. Schell: *Introduction to the Darwin Information Typing Architecture*, `http://www.ibm.com/developerworks/xml/library/x-dita1/`, 2005

[7] P. Deransart, M. Jourdan, B. Lorho: *Attribute Grammars: Definitions, Systems and Bibliography*, vol 323 of LNCS, Springer-Verlag, 1988

[8] J. Eickel: *Logical and layout structures of documents*, Computer Physics Communications, 1990

[9] J. Eickel: *Dokumentarchitektur*, Lecture at Technical University of Munich, 1996/97

[10] B. Freeman-Benson, A. Borning: *Integrating Constraints with an Object-Oriented Language*, ECOOP 92, 1992

[11] T. Freter: *XML: Document and Information Management*, Sun Microsystems Research, 1998

[12] C. F. Goldfarb: *The SGML Handbook*, Oxford University Press, 1990

[13] M. Goossens, F. Mittelbach, A. Samarin: *The LaTeX Companion*, Addison-Weseley, 1994

[14] R. Heckmann, R. Wilhelm: *A Functional Description of TeX's Formula Layout*, Journal of Functional Programming, Vol. 7(5), 1997

[15] B. Heikkinen: *Generalization of Document Structure Assembly*, University of Helsinki, Department of Computer Science, Report A-2000-2, 2000

[16] Denis Howe (Ed.): *Free On-line Dictionary of Computing*, `http://www.foldoc.org/`, 1993

[17] International Organization for Standardization / International Electrotechnical Commission: *Information technology—Processing languages—Document Style Semantics and Specification Language (DSSSL)*, International Standard ISO/IEC 10179:1996, 1996.

[18] International Organisation for Standardization: *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, International Standard ISO 8879, 1986

[19] International Organisation for Standardization: *Office Document Architecture (ODA) and Interchange Format*, International Standard ISO 8613, 1988

[20] J. Jaffar, J.L. Lassez: *Constraint Logic Programming*, in Proceedings of ACM Symposium on Principles of Programming Languages Conference, 1987

[21] M. Jazayeri, W. Ogden, W. Rounds: *The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars*, in Communications of the ACM 18, 1975

[22] Wolfram Kahl: *Beyond Pretty-Printing: Galley Concepts in Document Formatting Combinators*, in G. Gupta (Ed.): PADL '99, LNCS 1551, Springer, 1998

[23] U. Kastens: *Ordered Attribute Grammars*, in Acta Informatica 13(3), 1980

[24] K. Kennedy, S.K. Warren: *Automatic Generation of Efficient Evaluators for Attribute Grammars*, in Proc. 3rd ACM Symp. on Principles of Programming Languages, 1976

[25] J.H. Kingston: *The Design and Implementation of the Lout Docutment Formatting Language*, Software – Practice and Experience 23(9), 1993

[26] D.E. Knuth: *Semantics of Context-free Languages*, Math. Systems Theory, 1968

[27] D.E. Knuth: *The TEXbook*, Addison-Wesley, 1986

[28] D.E. Knuth: *The Complete TEX Source*

[29] J. London: *The Sea-Wolf*, 1904

[30] G. Lopez, B. Freeman-Benson, A. Borning: *Kaleidoscope: A Constraint Imperative Programming Language*, Technical Report 93-09-04, Department of Computer Science and Engineering, University of Washington, 1993

[31] G. Lopez, B. Freeman-Benson, A. Borning: *Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine*, Technical Report 94-07-07, Department of Computer Science and Engineering, University of Washington, 1994

[32] A. Malton: *The Denotational Semantics of a Functional Tree-Manipulation Language*, Computer Languages, 19(3), 1993

[33] F. Mittelbach, C. Rowley: *The Pursuit of Quality*, Proc. of *Electronic Publishing* '92, 1992

[34] M. Murata, K. Hayashi: *Formatter hierarchy for structured documents*, in EP92 Text processing and document manipulation, eds., C. Vanoirbeek and G. Corey, Cambridge University Press, 1992

[35] M. F. Plass, D. E. Knuth: *Choosing better line breaks*, In Nievergelt et al., editor, Document Preparation Systems, North-Holland Publishing Company, 1982

[36] M. F. Plass: *Optimal pagination techniques for automatic typesetting systems*, Technical Report STAN-CS-81-870, Department of Computer Science, Stanford University, 1981

[37] P. Prescod: *Introduction to DSSSL*, `http://www.prescod.net/dsssl/`, 1997

[38] A. Reinhardt: *Managing the New Document*, Byte Magazine, 1994

[39] T. Reps, T. Teitelbaum, A. Demers: *Incremental Context-Dependent Analysis of Language-Based Editors*, ACM Transactions on Programming Languages and Systems, 5(3), July 1983

[40] T. Reps, T. Teitelbaum: *The Synthesizer Generator*, in Proc. of the ACM SIGSOFT-SIGPLAN Software Engineering Symposium on Practical Software Development Environment, ACM SIGPLAN Notices, 19(5), May 1984

[41] C. Roisin, I. Vatton: *Formatting Structured Documents*, Technical Report 2044, INRIA, 1993

[42] C. Roisin, I. Vatton: *Merging Logical and Physical Structures in Documents*, Electronic Publishing, Vol. 1(1), 1993

[43] W. Schreiber: *Generierung von Dokumentverarbeitungssystemen aus formalen Spezifikationen von Dokumentarchitekturen*, Herbert Utz Verlag Wissenschaft, 1996

[44] World Wide Web Consortium: *Scalable Vector Graphics (SVG) 1.0 Specification*, `http://www.w3.org/TR/2000/CR-SVG-20001102/index.html`, 2000

[45] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Second Edition)*, `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000

[46] World Wide Web Consortium: *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*, `http://www.w3.org/TR/xhtml1/`, 2002

[47] World Wide Web Consortium: *XML Schema Part 0: Primer*, `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`, 2001

[48] World Wide Web Consortium: *Extensible Stylesheet Language (XSL) Version 1.0*, `http://www.w3.org/TR/2000/CR-xsl-20001121/`, 2000

[49] World Wide Web Consortium: *XSL Transformations (XSLT) Version 1.0*, `http://www.w3.org/TR/1999/REC-xslt-19991116/`, 1999

[50] Philip Wadler: *A prettier printer*, `http://cm.bell-labs.com/cm/cs/who/wadler/topics/recent.html`, 1998

[51] M. Wilson: *Hierarchical Constraint Logic Programming*, Technical Report 93-05-01, University of Washington, 1993

[52] N. Walsh, L. Muellner: *DocBook: The Definitive Guide*, 1st edition, O'Reilly, 1999

[53] Wikipedia: *Murphy's law*, `http://en.wikipedia.org/wiki/Murphy%27s_law`, 2006

[54] S. Wohlfeil: *On the Pagination of Complex, Book-Like Documents*, Dissertation, Department of Computer Science, FernUniversität Hagen, 1997

[55] R. Wilhelm, R. Heckmann: *Grundlagen der Dokumentverarbeitung*, Addison-Wesley, 1996

[56] R. Wilhelm, D. Maurer: *Übersetzerbau* (2. Auflage), Springer, 1997

[57] G. Winskel: *The Formal Semantics of Programming Languages*, MIT Press, 1993

APPENDIX A

Glossary

This glossary describes common terms from the field of document processing which are used in examples in this work. The information here is gathered from various sources, e.g., [16, 55]. No claim to provide a complete listing of terms used in the field of document processing is made.

Ascent. The amount by which a ☞*glyph* extends above the ☞*baseline*. See also ☞*descent*.

Baseline. The imaginary line at which characters on a line of text are aligned. Portions of some characters may extend below the baseline (see ☞*descent*).

Character. A logical atomic element of text. Characters are arranged in ☞*character sets*.

Character set. A collection of ☞*character*s where each individual element is identified by a code position (a positive integer number).

Descent. The amount by which a ☞*glyph* extends below the ☞*baseline*. See also ☞*ascent*.

Encoding. A mapping of binary values to code positions of a ☞*character set*.

Ex-height. A typographical unit of measure which is based on the ☞*font* currently in use. 1*ex* originally used to equal the ☞*ascent* of the ☞*glyph* for the ☞*character* ‘x’ in a ☞*font* at a given ☞*point size*. Today, this unit of measure is a rather arbitrary property of a ☞*font*. See also ☞*Em-width*.

Em-width. A typographical unit of measure which is based on the ☞*font* currently in use. 1*em* orriginally used to equal the width of the ☞*glyph* for the character 'M' in a ☞*font* at a given ☞*point size*. Today, this unit of measure is a rather arbitrary property of a ☞*font*. See also ☞*Ex-height*.

Font. A font defines a set of ☞*glyph*s. For scalable fonts glyphs are defined by their outline ☞*path* at some normalized ☞*point size* (usually 1pt) and must be scaled to the size needed in documents.

Glyph. The presentation of a single ☞*character* or a sequence of more than one consecutive ☞*character*s in a ☞*font* at some ☞*point size*. See also ☞*Ligature*.

Kerning. The process of adjusting the spacing between certain pairs of adjacent ☞*glyph*s on a line of text. This is done in order to improve the visual appearance or to adjust the general tightness of text. In the first case, for instance, the two letters VA are usually moved closer together (as opposed to VA), so that the bounding boxes of both individual ☞*glyph*s overlap. In the latter case, the space between all pairs of ☞*glyph*s are adjusted equally. Kerning is guided by the specification of the metrics specified for a ☞*font*.

Leading. The vertical space separating two adjacent lines of text. This space was in the time of manual typesetting realized using pieces made out of lead.

Ligature. A single ☞*glyph* representing two or more consecutive ☞*character*s instead of only one. Ligatures are intended to give a better visual appearance than the sequence of ☞*glyph*s for the single characters. Common ligatures include 'ﬁ' and 'ﬃ' which look better than 'fi' and 'ffi'.

Orphan. First line of a paragraph appearing alone at the end of a page while the rest of the paragraph is placed on the following page. Orphans are in general undesired and avoided whenever possible. See also ☞*widow*.

Path. A sequence of straight and curved line segments. Paths are used to define the outline and interior of graphic objects, e.g., ☞*glyph*s. More details are beyond the scope of this work and can be found, e.g, in [1].

Point. Important unit of measure in the area of document processing. Varying definitions have developed over time. Today a `Postscript` point is referred to in the majority of cases, which equals 1/72 of an inch or $0.352\bar{7}mm$.

Point size. Measure for the size of ☞*glyph*s of a ☞*font* given in ☞*point*s. The point size equals the height of a line of text, i.e., the sum of maximum ☞*ascent*s and ☞*descent*s of all ☞*glyph*s and is measured in points (see ☞*point*).

Type 1 font. A `Postscript` language ☞*font* format which defines ☞*glyph*s as the interior of outline ☞*path*s. This allows such fonts to be freely scalable to any desired ☞*point size* without loss of quality.

Unicode. A ☞*character set* defining a large collection of ☞*character*s covering most contemporary and historic languages.

UTF-8. A popular ☞*encoding* for ☞*Unicode* which uses byte sequences of varying length to encode ☞*character*s. Most characters used in western languages are encoded as a single byte.

Widow. Last line of a paragraph appearing alone on a new page. Usually undesired and thus avoided whenever possible. See also ☞*orphan*.

APPENDIX B

Missing proofs from Chapter 3

B.1 Auxiliary lemmata

Lemma B.1.1
For any trees t, t'_1, t'_2, and a position $pos \in Pos(t)$:

$$\big(t[t'_1/pos]\big)[t'_2/pos] \;=\; t[t'_2/pos]$$

Proof (by induction on pos):

(i) $pos = \varepsilon$. Then

$$\begin{aligned}\big(t[t'_1/pos]\big)[t'_2/pos] &= t'_1[t'_2/\varepsilon] \\ &= t'_2 \\ &= t[t'_2/pos]\end{aligned}$$

(ii) $pos = \langle i \rangle \circ pos'$. Suppose $t = F(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n)$, then

$$\begin{aligned}\big(t[t'_1/pos]\big)[t'_2/pos] &= F\big(\ldots, t_i[t'_1/pos'], \ldots\big)[t'_2/pos] \\ &= F\big(\ldots, (t_i[t'_1/pos'])[t'_2/pos'], \ldots\big) \\ &= F\big(\ldots, t_i[t'_2/pos'], \ldots) && \text{(Ind. Hyp.)} \\ &= t[t'_2/pos]\end{aligned}$$

□

Lemma B.1.2
For any tree t and position $pos \in Pos(t)$ the following holds:

$$t\big[subtree(pos,t)/pos\big] \;=\; t$$

Proof (by induction on pos):

(i) $pos = \varepsilon$. Then $t\big[subtree(pos,t)/pos\big] = t[t/\varepsilon] = t$.

(ii) $pos = \langle i\rangle \circ pos'$. Suppose $t = F(t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n)$. Then

$$\begin{aligned}
& t\big[subtree(pos,t)/pos\big] \\
&\quad = \; F\big(\ldots,t_i\big[subtree(\langle i\rangle \circ pos',t)/pos'\big],\ldots\big) \\
&\quad = \; F\big(\ldots,t_i\big[subtree(pos',t_i)/pos'\big],\ldots\big) \\
&\quad = \; F(\ldots,t_i\ldots) && \text{(Ind. Hyp.)} \\
&\quad = \; t
\end{aligned}$$

□

Lemma B.1.3
For any tree node σ and trees t_1, t_2:

$$replace\big(replace(\sigma,t_1),t_2\big) \;=\; replace(\sigma,t_2)$$

Proof:
Suppose $\sigma = t_{\downarrow pos}$. Then:

$$\begin{aligned}
replace\big(replace(\sigma,t_1),t_2\big) &= \big((t[t_1/pos])[t_2/pos]\big)_{\downarrow pos} \\
&= t[t_2/pos]_{\downarrow pos} && \text{(Lemma B.1.1)} \\
&= replace(\sigma,t_2)
\end{aligned}$$

□

Lemma B.1.4
For any tree node σ:

$$replace\big(\sigma,subtree(\sigma)\big) \;=\; \sigma$$

Proof:
Suppose $\sigma = t_{\downarrow pos}$. Then

$$\begin{aligned}
replace\big(\sigma,subtree(\sigma)\big) &= \big(t[subtree(pos,t)/pos]\big)_{\downarrow pos} \\
&= t_{\downarrow pos} && \text{(Lemma B.1.2)} \\
&= \sigma
\end{aligned}$$

□

Lemma B.1.5
Let T be a template, and env, env' environments.

If $subtree([\![\xi]\!]_{env}) \preceq subtree([\![\xi]\!]_{env'})$ for each $\xi \in Leaf_Var(T)$, then $build(T, env) \preceq build(T, env')$.

Proof:
The simple proof by induction on the structure of T is omitted. □

Lemma B.1.6
Let T be a template, and τ, τ' tree nodes.

If T can be instantiated at both τ and τ', then the following is true:

$$\tau \preceq \tau' \quad \Rightarrow \quad inst(T, \tau) \preceq inst(T, \tau')$$

Proof:
The proof by induction on the structure of T is omitted. □

Lemma B.1.7
Let T be a template, $\xi \in Leaf_Var(T)$, and pos the position of ξ in T, i.e., $T_{\downarrow pos} = \xi$. Further, let env be an environment and $t = build(T, env)$.

Then $subtree(t_{\downarrow pos}) = build(\xi, env)$.

Proof (by induction on the structure of T):

(i) T has the form X. Then $\xi = X$ and $pos = \varepsilon$. So $subtree(t_{\downarrow pos}) = t = build(T, env)$.

(ii) T has the form $X \langle T_1 \dots T_k \rangle$. Then $pos = \langle i \rangle \circ pos'$ for some i and pos'. By definition of $build$, $build(T, env) = F(t_1, \dots, t_k)$ where $F = symbol(X)$ and $t_i = build(T_i)$ for $1 \le i \le k$.

So, by induction hypothesis, $subtree(t_{\downarrow pos}) = subtree(t_{i \downarrow pos'}) = build(\xi, env)$.

□

Lemma B.1.8
Let r be a rule of the form $\bar{S} \to T/\bar{R}$ where $\varphi_1, \dots, \varphi_k$. Further, let $\gamma, \tau_0, \tau_1, \dots, \tau_k$ be nodes, and $env_0, env_1, \dots, env_k$ environments.

If $\tau_0 = target(T, \gamma, \bot_{Env})$, $env_0 = inst(\bar{S}, \bar{\sigma}) \triangleleft inst(T, \tau_0)$, and $\langle \varphi_i, \tau_{i-1}, env_{i-1} \rangle \to^r \langle \tau_i, env_i \rangle$ for each $1 \le i \le k$, then

(i) $\tau_i = target(T, \gamma, env_i)$ for each $0 \le i \le k$

(ii) for any $0 \le i \le k$ and $\xi \in Leaf_Var(T) \setminus Def_Var_{i,r}$ the following holds:

$$subtree([\![\xi]\!]_{env_i}) = F_{\triangle} \quad \text{where} \quad F = symbol(\xi)$$

(iii) $\tau_{i-1} \preceq \tau_i$ for each $1 \le i \le k$

(iv) $[\![\xi]\!]_{env_i} = \bot$ for any $\xi \in \mathcal{V}ar \setminus \big(Def_Var_{i,r} \ \cup \ Var(T)\big)$, $0 \leq i \leq k$

Proof:
Note that the execution rules (**E1**) through (**E4**) that are referred to below were defined in Section 3.3.5.

(i) We prove this by induction on i.

(a) $i = 0$. Here, $\tau_0 = target(T, \gamma, env_0)$ by premises.

(b) $i \mapsto i + 1$. We distinguish two cases:

i. If φ_{i+1} is a semantic constraint, then $\tau_{i+1} = \tau_i$ and $env_{i+1} = env_i$ (cf. (**E3**)). So $\tau_{i+1} = target(T, \gamma, env_{i+1})$ by induction hypothesis.

ii. If φ_{i+1} is a syntactic constraint, then $\tau_{i+1} = target(T, \tau_i, env_i \lhd [\tau''/Y])$ for some t'' and Y (cf. (**E4**)). By induction hypothesis we can assume that $\tau_i = target(T, \gamma, env_i)$, so we know by definition of *target* and Lemma B.1.3 that $\tau_{i+1} = replace(\gamma, t)$ for some tree t.

Further, $env_{i+1} = \ldots \lhd inst(T, \tau_i + 1) \neq fail$. So, by Proposition 3.3.20, env_{i+1} instantiates T at τ_{i+1} and $subtree(\tau_{i+1}) = build(T, env_{i+1})$ by Proposition 3.3.22.

Thus, we can conclude using Lemma B.1.4 that

$$\begin{aligned} \tau_{i+1} &= replace\big(\gamma, build(T, env_{i+1})\big) \\ &= target(T, \gamma, env_{i+1}) \end{aligned}$$

(ii) The proof is again by induction on i.

(a) $i = 0$. We have, by unfolding the definition of *target*, $\tau_0 = replace(\gamma, t)$ where $t = build(T, \bot_{Env})$. By Lemma B.1.7 it follows that $subtree(t_{\downarrow pos}) = build(\xi, \bot_{Env}) = F_\triangle$.

As, according to (**E2**), $env_0|_{Var(T)} = inst(T, \tau_0) \neq fail$, env_0 instantiates T at τ_0. So by Definition 3.2.9 $[\![\xi]\!]_{env_0} = \tau_{0\downarrow pos}$. Hence, $subtree([\![\xi]\!]_{env_0}) = subtree(\tau_{0\downarrow pos}) = subtree(t_{\downarrow pos}) = F_\triangle$.

(b) $i \mapsto i + 1$. As $Def_Var_{i+1,r} \supseteq Def_Var_{i,r}$, we know by induction hypothesis that $subtree([\![\xi]\!]_{env_i}) = F_\triangle$ for $\xi \in Leaf_Var(T) \setminus Def_Var_{i+1,r}$.

We then have to distinguish two cases. If φ_{i+1} is a semantic constraint, then $env_{i+1} = env_i$, so this case is trivial.

Otherwise, φ_{i+1} is a syntactic constraint of the form $\bar{X} \rightarrow Y/\bar{Z}$ and by (**E4**) and definition of *target* we have $\tau_{i+1} = replace(\gamma, t)$ where $t = build(T, env_i \lhd [\tau''/Y])$ for some τ''.

We can then conclude using Lemma B.1.7 that $subtree(t_{\downarrow pos}) = build(\xi, env_i \lhd [\tau''/Y])$. As $\xi \notin Def_Var_{i+1,r}$, but $Y \in Def_Var_{i+1,r}$ by definition, this becomes $subtree(t_{\downarrow pos}) = build(\xi, env_i)$.

In the same way as in the proof of the induction basis we know again that env_{i+1} instantiates T at τ_{i+1}. So by Definition 3.2.9, it follows that $[\![\xi]\!]_{env_{i+1}} = \tau_{i+1\downarrow pos}$.

Thus, finally, $subtree([\![\xi]\!]_{env_{i+1}}) = subtree(\tau_{i+1\downarrow pos}) = subtree(t_{\downarrow pos}) = build(\xi, env_i) = F_\triangle$.

(iii) If φ_i is a semantic constraint, then $\tau_i = \tau_{i+1}$, so this case is trivial. Otherwise, φ_i is a syntactic constraint of the form $\bar{X} \to Y/\bar{Z}$. In this case we have by (**E4**)

$$\begin{aligned} \tau_i &= target(T, \tau_{i-1}, env_{i-1} \triangleleft [\tau''/Y]) \\ &= replace\big(\tau_{i-1}, build(T, env_{i-1} \triangleleft [\tau''/Y])\big) \\ &= replace\big(\gamma, build(T, env_{i-1} \triangleleft [\tau''/Y])\big) \end{aligned}$$

for some τ''.

On the other hand, we know by (i) and the definition of *target* that $\tau_{i-1} = replace\big(\gamma, build(T, env_{i-1})\big)$. So it is sufficient to show that $subtree([\![\xi]\!]_{env_{i-1}}) \preceq subtree([\![\xi]\!]_{env_{i-1}\triangleleft[\tau''/Y]})$ for each $\xi \in Leaf_Var$, because then $build(T, env_{i-1}) \preceq build(T, env_{i-1} \triangleleft [\tau''/Y])$ by Lemma B.1.5.

So, let $\xi \in Leaf_Var(T)$. As $Y \notin Def_Var_{i-1,r}$ by LR conditions, we know by (ii) that $subtree([\![Y]\!]_{env_{i-1}}) = F_\triangle$ where $F = symbol(Y)$. So it is indeed true that $subtree([\![\xi]\!]_{env_{i-1}}) \preceq subtree([\![\xi]\!]_{env_{i-1}\triangleleft[\tau''/Y]})$.

(iv) We prove this again by induction on i.

(a) $i = 0$. By (**E2**) we have $env_0 = inst(\bar{S}, \bar{\sigma}) \triangleleft inst(T, \tau_0)$ for some τ_0. Thus, $Dom(env_0) = Var(\bar{S}) \cup Var(T) = Def_Var_{0,r} \cup Var(T)$.

(b) $i \mapsto i+1$. The case where φ_{i+1} is a semantic constraint is trivial. If φ_{i+1} is a syntactic constraint of the form $\bar{X} \to Y/\bar{Z}$, then $env_{i+1} = env_i \triangleleft [\tau''/Y] \triangleleft [\bar{\rho}'/\bar{Z}] \triangleleft inst(T, \tau')$ for some τ'', $\bar{\rho}'$, and τ'.

Consider a variable $\xi \in \mathcal{V}ar \setminus \big(Def_Var_{i+1,r} \cup Var(T)\big)$. As $Def_Var_{i+1,r} = Def_Var_{i,r} \cup \{Y\} \cup Var(\bar{Z})$, we can assume by induction hypothesis that $[\![\xi]\!]_{env_i} = \bot$. Hence, as $\xi \notin \{Y\} \cup Var(\bar{Z}) \cup Var(T)$, we have $[\![\xi]\!]_{env_{i+1}} = \bot$.

□

Lemma B.1.9
Let T be a template and τ, τ' tree nodes.

If T can be instantiated at τ and $\tau \preceq \tau'$, then T can be instantiated also at τ'.

Proof:
The proof by induction on the structure of T is omitted. □

Lemma B.1.10
Let T be a template instantiated at a node σ. Further, let $\mathcal{X} \subseteq Leaf_Var(T)$ and $P = \{pos \mid T_{\downarrow pos} \in \mathcal{X}\}$.

Then the following is true:

(i) $\mathit{Prelim_Attr_Occ}_{I,P}(\sigma) \subseteq \{(pos, a) \mid (T_{\downarrow pos}.a) \in \mathit{Prelim_Attr_Occ}_{I,V}(T)\}$

(ii) $T_{\downarrow pos}.a \in \mathit{Final_Attr_Occ}_{I,V}(T) \quad \Rightarrow \quad (pos, a) \in \mathit{Final_Attr_Occ}_{I,P}(\sigma)$

Proof:

(i) Let $(pos, a) \in \mathit{Prelim_Attr_Occ}_{I,P}(\sigma)$. Then a chain (pos_0, a_0), (pos_1, a_1), ..., (pos_k, a_k) exists where

- $(pos_0, a_0) \in \{(\varepsilon, a') \mid a' \in I\} \cup \{(pos', a') \mid pos' \in P \wedge a' \in \mathit{Synth}\}$
- $(\sigma_{\downarrow pos_{i-1}}.a_{i-1}, \sigma_{\downarrow pos_i}.a_i) \in \mathit{Dep}$
- $(pos_k, a_k) = (pos, a)$

σ instantiates T, so

$$(T_{\downarrow pos_{i-1}}.a_{i-1}, T_{\downarrow pos_i}, a_i) \in \mathit{Dep}(T) \quad \text{for all} \quad 1 \le i \le k$$

Further, if $(pos_0, a_0) \in \{(\varepsilon, a') \mid a' \in I\}$, then $T_{\downarrow pos_0}.a_0 \in \mathit{Prelim_Attr_Occ}_{I,\mathcal{X}}(T)$. Otherwise, $(pos_0, a_0) \in \{(pos', a') \mid pos' \in P \wedge a' \in \mathit{Synth}\}$. In this case $T_{\downarrow pos_0} \in \mathcal{X}$, so also $T_{\downarrow pos_0}.a_0 \in \mathit{Prelim_Attr_Occ}_{I,\mathcal{X}}(T)$.

So $(T_{\downarrow pos_i}.a_i)_{0 \le i \le k}$ is a chain where

- $T_{\downarrow pos_0}.a_0 \in \{T_{\downarrow \varepsilon}.a' \mid a \in I\} \cup \{\xi.a' \mid \xi \in \mathcal{X} \wedge a' \in \mathit{Synth}\}$
- $(T_{\downarrow pos_{i-1}}.a_{i-1}, T_{\downarrow pos_i}.a_i) \in \mathit{Dep}(T)$
- $T_{\downarrow pos_k}.a_k = T_{\downarrow pos}.a$

So $T_{\downarrow pos}.a \in \mathit{Prelim_Attr_Occ}_{I,\mathcal{X}}(T)$.

(ii) follows directly from the definitions of $\mathit{Final_Attr_Occ}_{I,V}(T)$ and $\mathit{Final_Attr_Occ}_{I,P}(\sigma)$

□

Definition B.1.11
The operation $\mathit{subtempl}(pos, T)$ yields the sub template of a template T from a position $pos \in \mathit{Pos}(T)$ and is inductively defined so:

$$\begin{aligned} \mathit{subtempl}(\varepsilon, T) &=_{def} T \\ \mathit{subtempl}(\langle i \rangle \circ pos', X\langle T_1 \dots T_k \rangle) &=_{def} \mathit{subtempl}(pos', T_i) \end{aligned}$$

◊

Lemma B.1.12
Let T be a template, $pos \in Pos(T)$, and env be an environment. Further, let $t = build(T, env)$.

Then the following holds:

$$subtree(pos, t) \;=\; build\big(subtempl(pos, T), env\big)$$

Proof (by induction on pos):

(i) If $pos = \varepsilon$, then the contention is obviously true.

(ii) Otherwise, pos has the form $\langle i \rangle \circ pos'$ and T has the form $X \langle T_1 \ldots T_k \rangle$. Then

$$\begin{aligned} subtree(pos, t) &= subtree\big(pos', build(T_i, env)\big) \\ &= build\big(subtempl(pos', T_i), env\big) \qquad \text{(Ind. Hyp.)} \\ &= build\big(subtempl(pos, T), env\big) \end{aligned}$$

□

Lemma B.1.13
Let T be a template, σ a node, and env an environment. T is instantiated at σ by env.

Then for any $pos \in Pos(T)$

$$subtree(\sigma_{\downarrow pos}) \;=\; build\big(subtempl(pos, T), env\big)$$

Proof (by induction on pos):

(i) $pos = \varepsilon$. Then

$$\begin{aligned} subtree(\sigma_{\downarrow pos}) &= subtree(\sigma) \\ &= build(T, env) \qquad \text{(Lemma B.1.12)} \\ &= build\big(subtempl(pos, T), env\big) \end{aligned}$$

(ii) $pos = \langle i \rangle \circ pos'$. Then T must have the form $X \langle T_1, \ldots, T_n \rangle$ where $n \geq i$. As T is instantiated at σ by env we know that also T_i is instantiated at $\sigma_{\downarrow i}$ by env. Thus

$$\begin{aligned} subtree(\sigma_{\downarrow pos}) &= subtree\big((\sigma_{\downarrow i})_{\downarrow pos'}\big) \\ &= build\big(subtempl(pos', T_i), env\big) \qquad \text{(Ind. Hyp.)} \\ &= build\big(subtempl(\langle i \rangle \circ pos', T), env\big) \end{aligned}$$

□

B.2 Proofs from Section 3.3.2

We first introduce a lemma needed for proving that $\preceq$ is a partial order.

Lemma B.2.1
For any trees t and t', and symbol F the following holds:

(i) $t \preceq t' \;\wedge\; t' = F_\triangle \quad\Rightarrow\quad t = F_\triangle$

(ii) if $t \preceq t'$ and t has the form $F(t_1, \ldots, t_k) \neq F_\triangle$, then t' has the form $F(t'_1, \ldots, t'_k)$ where $t_i \preceq t'_i$ for $1 \leq i \leq k$

(iii) $t \preceq t' \quad\Rightarrow\quad \mathit{symbol}(t) = \mathit{symbol}(t')$

Proof:
These properties can be shown by rule induction (see, e.g., [57]). Note that $\preceq$ is the least set defined by a set of rules which take the two forms given in Definition 3.3.1 on page 44. The details are left to the reader. □

Proposition 3.3.2

The binary relation $\preceq \;\subseteq\; \mathcal{T} \times \mathcal{T}$ is a partial order, i.e., it is reflexive, transitive and antisymmetric.

Proof:

(i) $\preceq$ is reflexive, i.e., $t \preceq t$ for all $t \in \mathcal{T}$:

We use induction on the structure of t which has the form $F(t_1, \ldots, t_k)$. By induction hypothesis, $t_i \preceq t_i$ for $1 \leq i \leq k$, so immediately $t \preceq t$ according to the definition of $\preceq$.

(ii) $\preceq$ is antisymmetric, i.e., $t \preceq t' \;\wedge\; t' \preceq t \quad\Rightarrow\quad t = t'$.

We use induction on the structure of t distinguishing two cases:

(a) t has the form $F_\triangle$. As $t \preceq t'$, by Lemma B.2.1 (iii) t' must have the form $F(t_1, \ldots, t_k)$. But at the same time $t' \preceq t$, so by Lemma B.2.1 (i) $t' = F_\triangle = t$.

(b) t has the form $F(t_1, \ldots, t_k) \neq F_\triangle$. Lemma B.2.1 (ii) implies that t' has the form $F(t'_1, \ldots, t'_k)$ where $t_i \preceq t'_i$ for $1 \leq i \leq k$. As $t' \preceq t$, for the same reason $t'_i \preceq t_i$ for $1 \leq i \leq k$. So by induction hypothesis $t_i = t'_i$ for $1 \leq i \leq k$, so $t = t'$.

(iii) $\preceq$ is transitive, i.e., $t \preceq t' \;\wedge\; t' \preceq t'' \quad\Rightarrow\quad t \preceq t''$:

We use induction on the structure of t distinguishing two cases again:

(a) t has the form $F_\triangle$. Then, as $t \preceq t'$ and $t' \preceq t''$, by Lemma B.2.1 (iii) $\mathit{symbol}(t') = \mathit{symbol}(t'') = F$. So $t \preceq t''$ by definition of $\preceq$.

(b) t has the form $F(t_1, \ldots, t_k) \neq F_\triangle$. $t \preceq t'$, so by Lemma B.2.1 (ii) t' has the form $F(t'_1, \ldots, t'_k)$ where $t_i \preceq t'_i$ for $1 \leq i \leq k$. Further, $t' \preceq t''$, so again by Lemma B.2.1 (ii) t'' has the form $F(t''_1, \ldots, t''_k)$ where $t'_i \preceq t''_i$ for $1 \leq i \leq k$. By induction hypothesis $t_i \preceq t''_i$, so $t \preceq t''$ by definition of $\preceq$.

□

B.3 Proofs from Section 3.3.5

Proposition 3.3.22

Let T be a template, τ a tree node, and env, env' environments.

(i) If $subtree([\![\xi]\!]_{env}) = subtree([\![\xi]\!]_{env'})$ for all $\xi \in Leaf_Var(T)$, then $build(T, env) = build(T, env')$.

(ii) If env instantiates T at τ, then $build(T, env) = subtree(\tau)$

Proof:
The proof for the first fact by simple induction on the structure of T is omitted. We only show the proof for the second contention which is also by induction on the structure of T.

(i) T has the form X. Then, $build(T, env) = subtree([\![X]\!]_{env})$ by definition of $build$. As env instantiates T at τ we know that $[\![X]\!]_{env} = \tau$.

(ii) T has the form $X \langle T_1 \ldots T_k \rangle$. Then $build(T, env) = F(t_1, \ldots, t_k)$ where $t_i = build(T_i, env)$ and $F = symbol(X)$ for each $1 \leq i \leq k$.

As env instantiates T at τ, env also instantiates T_i at $\tau_{\downarrow i}$, and thus, by induction hypothesis, $build(T_i, env) = subtree(\tau_{\downarrow i})$ for each $1 \leq i \leq k$.

We also know that $child_count(\tau) = k$ and $symbol(\tau) = F$. So, $F\big(subtree(\tau_{\downarrow 1}), \ldots, subtree(\tau_{\downarrow k})\big) = subtree(\tau)$.

□

B.4 Proofs from Section 3.3.6

Lemma 3.3.26

Let r be a rule of the form $\bar{S} \rightarrow T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$. Let further $\tau_0, \tau_1, \ldots, \tau_k$ be nodes, and $env_0, env_1, \ldots, env_k$ environments.

If $\langle \varphi_i, \tau_{i-1}, env_{i-1} \rangle \rightarrow^r \langle \tau_i, env_i \rangle$ for each $1 \leq i \leq k$, then the following is true:

(i) env_i instantiates T at τ_i for each $0 \leq i \leq k$

(ii) $env_{i-1} \preceq env_i$ for each $1 \leq i \leq k$

(iii) $[\![\xi]\!]_{env_i} = [\![\xi]\!]_{env_{i+j}}$ for each $\xi \in Def_Var_{i,r} \setminus Var(T)$, $0 \leq i \leq k$, and $0 \leq j \leq k - i$

Proof:

(i) The proof is by induction on i.

(a) $i = 0$. By (**E2**) $env_0 = inst(\bar{S}, \bar{\sigma}) \lhd inst(T, \tau_0) \neq fail$. Thus, env_0 instantiates T at τ_0 by Proposition 3.3.20.

(b) $i \mapsto i+1$. If φ_{i+1} is a semantic constraint, then env_{i+1} instantiates T at τ_{i+1} directly by induction hypothesis, as $env_{i+1} = env_i$ and $\tau_{i+1} = \tau_i$. Otherwise, φ_{i+1} is a syntactic constraint of the form $\bar{X} \to Y/\bar{Z}$. Here we have $env_{i+1} = env \lhd inst(T, \tau_{i+1})$ for some env. Thus, again by Proposition 3.3.20, env_{i+1} instantiates T at τ_{i+1}.

(ii) If φ_i is a semantic constraint, then $env_i = env_{i+1}$, so $env_i \preceq env_{i+1}$ by reflexivity of $\preceq$.

Otherwise, φ_i is a syntactic constraint of the form $\bar{X} \to Y/\bar{Z}$. In this case we have by (**E4**)

$$\begin{aligned} \tau_i &= target(T, \tau_{i-1}, env_{i-1} \lhd [\tau''/Y]) \\ &= replace\big(\tau_{i-1}, build(T, env_{i-1} \lhd [\tau''/Y])\big) \end{aligned}$$

and $env_i = env_{i-1} \lhd [\tau''/Y] \lhd [\bar{\rho}'/\bar{Z}] \lhd inst(T, \tau_i)$ for some τ'' and $\bar{\rho}'$. Note that the execution rules enforce that $env_i \neq fail$.

We show that $[\![\xi]\!]_{env_{i-1}} \preceq [\![\xi]\!]_{env_i}$ for each $\xi \in \mathcal{V}ar$ by distinction of the following cases:

(a) $\xi \in Var(\bar{Z})$. As by LR conditions $\xi \notin Def_Var_{i-1,r}$, we know by Lemma B.1.8 (iv) that $[\![\xi]\!]_{env_{i-1}} = \bot$.

So obviously $[\![\xi]\!]_{env_{i-1}} \preceq [\![\xi]\!]_{env_i}$.

(b) $\xi \in Var(T)$. We have by (**E4**) $env_i|_{Var(T)} = inst(T, \tau_i)$ and by (i) and by Proposition 3.3.20 $env_{i-1}|_{Var(T)} = inst(T, \tau_{i-1})$.

By Lemma B.1.8 we know that $\tau_{i-1} \preceq \tau_i$. By Lemma B.1.6 it follows that $inst(T, \tau_{i-1}) \preceq inst(T, \tau_i)$.

Thus, $[\![\xi]\!]_{env_{i-1}} \preceq [\![\xi]\!]_{env_i}$.

(c) $\xi \in \mathcal{V}ar \setminus \big(Var(T) \cup Var(\bar{Z})\big)$. For $\xi \neq Y$ obviously $[\![\xi]\!]_{env_{i-1}} = [\![\xi]\!]_{env_i}$. If $\xi = Y$ (and $Y \notin Var(T)$), then by Lemma B.1.8 (iv) $[\![\xi]\!]_{env_{i-1}} = \bot$, so $[\![\xi]\!]_{env_{i-1}} \preceq [\![\xi]\!]_{env_i}$.

(iii) The proof is by induction on j. The induction base $j = 0$ is trivial. For the induction step, if φ_{i+j+1} is a semantic constraint, then $env_{i+j+1} = env_{i+j}$ according to (**E3**), so this case is trivial as well.

Otherwise, φ_{i+j+1} is a syntactic constraint of the form $\bar{X} \to Y/\bar{Z}$ and by (**E4**) $env_{i+j+1} = env_{i+j} \lhd [\tau''/Y] \lhd [\bar{\rho}'/\bar{Z}] \lhd inst(T, \tau')$ for some τ'', $\bar{\rho}'$, and τ'. By LR conditions $\big(\{Y\} \cup Var(\bar{Z})\big) \cap Def_Var_{i+j,r} = \emptyset$ and obviously

$Def_Var_{i,r} \subseteq Def_Var_{i+j,r}$. So $[\![\xi]\!]_{env_{i+j+1}} = [\![\xi]\!]_{env_{i+j}}$, and thus, by induction hypothesis, $[\![\xi]\!]_{env_{i+j+1}} = [\![\xi]\!]_{env_i}$ for $\xi \in Def_Var_{i,r} \setminus Var(T)$.

□

Lemma 3.3.27

Let $\bar{\sigma}$ and $\bar{\rho}$ be sequences of nodes. Further, let τ and τ' be nodes.

Then the following is true:

$$\bar{\sigma} \to \tau/\bar{\rho} \ \wedge \ \tau \preceq \tau' \quad \Rightarrow \quad \bar{\sigma} \to \tau'/\bar{\rho}$$

Proof:
Let the S-derivation D for $\bar{\sigma} \to_r \tau/\bar{\rho}$ have the form $(\bar{\sigma} \to_r \tau/\bar{\rho}, env)[D_1, \ldots, D_n]$, where r is a transformation rule of the form $\bar{S} \to T/\bar{R}$ where $\varphi_1, \ldots, \varphi_k$. Let further $env' = env \triangleleft inst(T, \tau')$. We show by induction on the structure of D that an S-derivation $(\bar{\sigma} \to_r \tau'/\bar{\rho}, env')[D'_1, \ldots, D'_n]$ exists as well.

First observe that $inst(T, \tau') \neq fail$ as env instantiates T at τ and $\tau \preceq \tau'$ by Lemma B.1.9 and Proposition 3.3.20. Consequently, env' instantiates T at τ'.

We are now ready to check the conditions in Definition 3.2.11.

(i) $\bar{S}$ is instantiated by env': as in an LR-specification $Var(T) \cap Var(\bar{S}) = \emptyset$ we know that $env'|_{Var(\bar{S})} = env|_{Var(\bar{S})}$ which entails that $\bar{S}$ is instantiated by env'.

(ii) T is instantiated by env' as already shown above.

(iii) $[\![\bar{R}]\!]_{env'} = \bar{\rho}$ because $[\![\bar{R}]\!]_{env} = \bar{\rho}$ and $Var(\bar{R}) \cap Var(T) = \emptyset$.

(iv) each semantic constraint $P(\alpha_1, \ldots, \alpha_n) \in \{\varphi_1, \ldots, \varphi_k\}$ is satisfied: $\tau \preceq \tau'$, so $env \preceq env'$. Thus, $\Vdash_{env} P(\alpha_1, \ldots, \alpha_n)$ entails $\Vdash_{env'} P(\alpha_1, \ldots, \alpha_n)$ by Corollary 3.3.8.

(v) each syntactic constraint $(\bar{X} \to Y/\bar{Z}) \in \{\varphi_1, \ldots, \varphi_k\}$ is satisfied: let D_i be the sub S-derivation for $[\![\bar{X}]\!]_{env} \to [\![Y]\!]_{env}/[\![\bar{Z}]\!]_{env}$. $[\![\bar{X}]\!]_{env'} = [\![\bar{X}]\!]_{env}$ as $Var(\bar{X}) \cap Var(T) = \emptyset$. Likewise $[\![\bar{Z}]\!]_{env'} = [\![\bar{Z}]\!]_{env}$. $[\![Y]\!]_{env} \preceq [\![Y]\!]_{env'}$, so by induction hypothesis an S-derivation D'_i for $[\![\bar{X}]\!]_{env'} \to [\![Y]\!]_{env'}/[\![\bar{Z}]\!]_{env'}$ exists.

□

B.5 Proofs from Section 3.3.7

Lemma 3.3.30

Let T be a template, τ a node, and env, env' environments. Further, the following is true:

(i) $subtree(\tau) = build(T, env)$

(ii) env' instantiates T at τ

Then the following holds for all $\xi \in \mathit{Leaf_Var}(T)$:

$$\mathit{subtree}([\![\xi]\!]_{env'}) = \mathit{build}(\xi, \mathit{env})$$

Proof:
Let pos be the position of ξ in T, i.e., $T_{\downarrow pos} = \xi$. Then

$$\begin{aligned} \mathit{subtree}([\![\xi]\!]_{env'}) &= \mathit{subtree}(\tau_{\downarrow pos}) && (env' \text{ inst. } T \text{ at } \tau\text{, Def 3.2.9}) \\ &= \mathit{subtree}(pos, \mathit{build}(T, env)) \\ &= \mathit{build}(\xi, env) && (\text{Lemma B.1.7}) \end{aligned}$$

□

Lemma 3.3.31

Let T be a template, γ, τ and $\hat{\tau}$ nodes. Further, let $\mathcal{X}$ be a set of variables, env and $\widehat{env}$ be environments, and $I \subseteq \mathit{Inh}$.

Further, the following is satisfied:

(i) $\gamma \preceq \tau$

(ii) env instantiates T at τ

(iii) $\hat{\tau} = \mathit{target}(T, \gamma, env|_{\mathcal{X}})$

(iv) $\widehat{env} = env|_{\mathcal{X}} \lhd \mathit{inst}(T, \hat{\tau})$

(v) $\mathit{att}(\gamma, a) = \mathit{att}(\tau, a)$ for all $a \in \mathit{Inh} \setminus I$

Then for any $\xi \in \mathcal{X} \cup \mathit{Var}(T)$ and $a \in \mathcal{A}$:

$$\xi.a \in \mathit{Final_Attr_Occ}_{I,\mathit{Leaf_Var}(T)\setminus\mathcal{X}}(T) \quad \Rightarrow \quad [\![\xi.a]\!]_{\widehat{env}} = [\![\xi.a]\!]_{env}$$

Proof:
If $\xi \in \mathcal{X} \setminus \mathit{Var}(T)$, then directly $[\![\xi.a]\!]_{\widehat{env}} = [\![\xi.a]\!]_{env|_{\mathcal{X}}} = [\![\xi.a]\!]_{env}$.

Otherwise, $\xi \in \mathit{Var}(T)$. Let pos be the position of ξ in T and $P = \{pos(T, \xi') \mid \xi' \in \mathit{Leaf_Var}(T) \setminus \mathcal{X}\}$.

(i) By Lemma B.1.13 it follows that $\mathit{subtree}\big(\tau_{\downarrow pos'}\big) = \mathit{build}\big(\mathit{subtempl}(pos', T), env\big)$ and that $\mathit{subtree}(\hat{\tau}_{\downarrow pos'}) = \mathit{build}(\mathit{subtempl}(pos', T), env)$.

(ii) We can now show that the requirements for the monotonicity property of att are fulfilled, i.e., for any $pos' \in \mathit{Final_Pos}_P(\hat{\tau})$

$$(*) \quad \mathit{subtree}(\hat{\tau}_{\downarrow pos'}) = \mathit{subtree}(\tau_{\downarrow pos'})$$

and

$$(**) \quad (pos, a) \in \mathit{Final_Attr_Occ}_{I,P}(\hat{\tau})$$

Assume that $\mathit{subtree}(\hat{\tau}_{\downarrow pos'})) \neq \mathit{subtree}(\tau_{\downarrow pos'}))$ for a position pos'. Then from (i) it follows that

$$\mathit{build}\big(\mathit{subtempl}(pos', T), env\big) \neq \mathit{build}\big(\mathit{subtempl}(pos', T), env|_{\mathcal{X}}\big)$$

So $\mathit{Leaf_Var}\big(\mathit{subtempl}(pos', T)\big) \not\subseteq \mathcal{X}$, which implies that a position pos'' exists such that $pos' \circ pos'' \in P$. Thus, $pos' \notin \mathit{Final_Pos}_P(\hat{\tau})$. So $(*)$ is fulfilled.

$(**)$ follows from the premise $\xi.a \in \mathit{Final_Attr_Occ}_{I,\mathit{Leaf_Var}(T)\setminus\mathcal{X}}(T)$ and Lemma B.1.10.

So the requirements given in Definition 3.3.6 are satisfied and we know that

$$att(\hat{\tau}_{\downarrow pos}) = att(\tau_{\downarrow pos})$$

(iii) env ($\widehat{env}$) instantiates T at τ (τ'), so by Definition 3.2.9:

$$[\![\xi]\!]_{env} = \tau_{\downarrow pos}$$

and

$$[\![\xi]\!]_{\widehat{env}} = \tau'_{\downarrow pos}$$

(iv) Thus, the following is true:

$$\begin{array}{rcll} [\![\xi]\!]_{\widehat{env}} & = & att(\hat{\tau}_{\downarrow pos}, a) & \text{(iii)} \\ & = & att(\tau_{\downarrow pos}, a) & \text{(ii)} \\ & = & [\![\xi]\!]_{env} & \text{(iii)} \end{array}$$

□

APPENDIX C

Extensions to attribute grammars and their semantics

We now formalize the attribute grammar model used in this dissertation. The model is very close to the original model introduced in [26, 7]. As an extension, an attribute grammar allows to attribute tree fragments which were introduced in Section 3.3.2 in a special way.

An undefined semantic value $\perp$ exists and attribution behaves in a strict way w.r.t. this value, i.e., attributes depending on an other attribute with value $\perp$ also evaluate to $\perp$. Further, synthesized attributes of unexpanded nodes evaluate to $\perp$.

As compared with the traditional semantics of attribute grammars, it is also ensured that attributes in a tree always evaluate to an unambiguous value.

C.1 Context-free grammars

A context-free grammar is given by a tuple $\Gamma = (\Sigma, T, \Pi, S)$, where

(i) Σ is a set of grammar symbols containing a special symbol $\triangle$

(ii) $T \subseteq \Sigma$ is a set of *terminal symbols*; $\triangle \in T$

(iii) $\Pi \subseteq (\Sigma \setminus T) \times \Sigma^*$ is a set of *productions* where a production is written as $X ::= X_1 \ldots X_n$

(iv) $S \in \Sigma \setminus T$ is a *start symbol*

Note that the transformation model described in this work uses more than one tree structure (at least a source structure and a target structure). To avoid unnecessary

complexity all structures are described using a single (attribute) grammar. So in our case the start symbol S is of no significance.

C.2 Attributes

An attribute grammar associates attributes with grammar symbols. Attributes are divided into *inherited* and *synthesizes* attributes. Thus, let $\mathcal{A} = \mathit{Inh} \cup \mathit{Synth}$ be a set of *attribute symbols*.

For each production attribution rules (equations on attribute occurrences of the production) are defined describing relationships between attributes. On the left hand side of each rule is a production attribute occurrence; on the right hand side is a *semantic function* applied to other production attribute occurrences.

A production occurrence of an attribute in a production p of the form $X ::= X_1 \dots X_n$ is specified by its symbol a and the position $\nu \in \{\varepsilon, 1, 2, \dots, n\}$, where ε denotes the position of X and i denotes the position of X_i for $1 \leq i \leq n$:

$$\mathit{Attr}(p) \quad =_{def} \quad \big\{(a, \nu) \mid a \in \mathcal{A} \text{ and } \nu \in \{\varepsilon, 1, 2, \dots, n\}\big\}$$

An occurrence of an attribute in a production can be either *defined*, i.e., computed by rules belonging to this production, or otherwise *applied*. Defined occurrences are synthesized attributes of the production's left hand side symbol X and inherited attributes of the production's right hand side symbols X_i $(1 \leq i \leq n)$:

$$\begin{aligned} \mathit{Def}(p) \quad &=_{def} \big\{(a, \varepsilon) \mid a \in \mathit{Synth}\big\} \cup \\ &\qquad \big\{(a, \nu) \mid a \in \mathit{Inh} \ \wedge \ \nu \in \{1, 2, \dots, n\}\big\} \\ \mathit{Appl}(p) \quad &=_{def} \mathit{Attr}(p) \setminus \mathit{Def}(p) \end{aligned}$$

C.3 Attribution rules

Let $\mathcal{D}$ be a *semantic domain* not containing the special element $\bot$, i.e., $\bot \notin \mathcal{D}$. An *attribution rule* is an equation of the form

$$(a_0, \nu_0) = f_{p,a_0,\nu_0}\big((a_1, \nu_1), \dots, (a_m, \nu_m)\big)$$

where

(i) $p \in \Pi$ is a production $X ::= X_1 \dots X_n$ which does not have the form $X ::= \triangle$

(ii) $(a_0, \nu_0) \in \mathit{Def}(p)$

(iii) $(a_i, \nu_i) \in \mathit{Attr}(p)$ for $1 \leq i \leq m$

(iv) f_{p,a_0,ν_0} is a function from $\mathcal{D}^m \to \mathcal{D} \ \cup \ \{\bot\}$

The set of attribution rules is denoted by *Attr_Rule*.

Note that the attribution of unexpanded nodes is handled in a special way as described below. So no attribution rules are allowed to be given for a production with $\triangle$ on the right hand side.

C.4 Initialization of synthesized attributes at terminal nodes

Attribution rules allow to specify the computation of inherited attributes (except those at a tree's root) and synthesized attributes of nonterminal nodes. What is missing is a way to provide values for synthesized attributes at terminal nodes. Among several possible solutions described in the literature [7, 56] we choose to assume the presence a function $\mathit{init}\colon T \times \mathit{Synth} \to \mathcal{D} \cup \{\bot\}$ for this purpose.

C.5 Attribute grammars

An *attribute grammar* is a tuple

$$AG = \big(\Gamma = (\Sigma, T, \Pi, S), \mathcal{A} = \mathit{Inh} \mathbin{\dot\cup} \mathit{Synth}, \mathit{Rule}, \mathcal{D}, \mathit{init}\big)$$

where

(i) Γ is a context-free grammar

(ii) $\mathcal{A}$ is a set of inherited and synthesized attribute symbols

(iii) $\mathit{Rule}\colon \Pi \to \mathcal{P}\mathit{ow}(\mathit{Attr_Rule})$ is a mapping from productions to sets of attribution rules

(iv) $\mathcal{D}$ is a semantic domain

(v) $\mathit{init}\colon T \times \mathit{Synth} \to \mathcal{D} \cup \{\bot\}$ is a function providing values of synthesized attributes for terminal symbols

C.6 Semantics

C.6.1 Syntax trees

Trees over a signature Σ were already introduced in Chapter 3. Trees could there be built in a completely free way. Given a context-free grammar we now have to make sure that trees are built only using existing productions.

Let $\sigma \in \mathcal{N}$ be a tree node having n children nodes ($\mathit{child_count}(\sigma) = n \in \mathbf{Nat_0}$). The production used at σ is given by $\mathit{prod}(\sigma) =_{\mathit{def}} (X ::= X_1 \ldots X_n)$ where $X = \mathit{symbol}(\sigma)$ and $X_i = \mathit{symbol}\big(\sigma_{\downarrow i}\big)$ for $1 \leq i \leq n$. A tree t is a Γ-*syntax tree* iff for all $\sigma \in \mathit{Node}(t)$ either $\mathit{prod}(\sigma) \in \Pi$, $\mathit{symbol}(\sigma) \in T$ and $\mathit{child_count}(\sigma) = 0$, or $\mathit{subtree}(\sigma)$ has the form $F_{\triangle}$.

C.6.2 Tree attribute equations

We can now construct the system of attribute equations for a given tree t by instantiating all production occurrences of attributes with tree attribute occurrences in all used productions' attribution rules. The instantiation of an attribution rule r of the form $(a_0, \nu_0) = f_{p,a_0,\nu_0}\big((a_1, \nu_1), \ldots, (a_m, \nu_m)\big)$ at a node σ is given by

$$\hat{r}(\sigma) \quad =_{\mathit{def}} \quad \big(\sigma_{\downarrow\nu_0}.a_0 = f_{p,a_0,\nu_0}(\sigma_{\downarrow\nu_1}.a_1, \ldots, \sigma_{\downarrow\nu_m}.a_m)\big)$$

The complete system of equations for a syntax tree t is then given by:

$$\widehat{Rule}(t) \quad =_{def} \quad \Big\{\hat{r}(\sigma) \mid \sigma \in Node(t) \ \wedge \ prod(\sigma) \in \Pi \ \wedge \ r \in Rule\big(prod(\sigma)\big)\Big\}$$

C.6.3 Consistent attributions

A *consistent attribution* associates values from $\mathcal{D}$ or the undefined value $\bot$ with all attribute occurrences in a way such that all attribute equations are satisfied and attribute occurrences for which no rule exists and dependent attributes are undefined. In Chapter 3 attributions were modeled by mappings $att\colon \mathcal{N} \times \mathcal{A} \to \mathcal{D} \ \cup \ \{\bot\}$. Formally such an attribution is valid iff for each node $\sigma \in Node(t)$ of a tree t the following is satisfied:

$$att(\sigma, a) = \begin{cases} init\big(symbol(\sigma), a\big) & \text{if } symbol(\sigma) \in T \ \wedge \ a \in Synth \\ f\big(att(\alpha_1), \ldots, att(\alpha_m)\big) & \text{if } \big(\sigma.a = f(\alpha_1, \ldots, \alpha_m)\big) \in \widehat{Rule}(t) \ \wedge \\ & \quad att(\alpha_i) \neq \bot \text{ for all } 1 \leq i \leq m \\ \bot & \text{otherwise.} \end{cases}$$

Note that no attribution rules were allowed to be given for productions of the form $X ::= \triangle$. This ensures that synthesized attributes of unexpanded nodes always evaluate to $\bot$.

Also note that it is enforced that all attributes in a tree evaluate to an unambiguous value which is a requirement raised in Chapter 3. Traditional attribute grammars in contrast generally allow ambiguous attribution in cases of underspecification.

C.6.4 Attribute dependencies

In Chapter 3 a binary relation *Dep* describes the functional dependencies between attributes. For attribute grammars this relation can be defined as the transitive closure of a relation describing the direct dependencies induced by tree attribute equations:

$$\begin{aligned} (t_{1\downarrow pos_1}.a_1, t_{2\downarrow pos_2}.a_2) \in Dep_1 \quad &\Leftrightarrow_{def} \quad t_1 = t_2 =: t \ \wedge \\ & \qquad \big(t_{\downarrow pos_2}.a_2 = f(\ldots, t_{\downarrow pos_1}.a_1, \ldots)\big) \in \widehat{Rule}(t) \\ Dep \quad &=_{def} \quad Dep_1^+ \end{aligned}$$

Dep is required to be irreflexive in Chapter 3 which means that Dep_1 does not contain any cycles. Note that the test for a attribute grammar to be free of circular attribute dependencies generally requires exponential complexity [21]. However, for certain classes of attribute grammars the test can be performed efficiently (see, e.g., [23] and [24]). In the following we will assume that no circular attribute dependencies occur.

C.6.5 Monotonicity property

In Chapter 3 attributions were required to behave monotonically w.r.t. the continuous expansion of the tree being attributed. This basically means that attributes having a final value in a tree where certain subtrees are yet unexpanded have the same value when any of these unexpanded subtrees have been replaced by new subtrees. The property was

defined formally in Definition 3.3.6. We now show that attribute grammars as defined above satisfy this property.

Let $A = (\Sigma, \mathit{Inh}, \mathit{Synth}, \mathcal{D}, \mathit{att}, \mathit{Dep})$ be an attribution where *att* has the consistency property defined in Section C.6.3 and *Dep* is the non-circular relation defined in Section C.6.4.

Theorem C.6.1
A is monotonous, i.e., the following conditions are satisfied for any nodes $\sigma = t_{\downarrow pos_0}$ and σ', $P \subseteq \mathit{Pos}(\mathit{subtree}(\sigma))$, any attribute $a \in \mathcal{A}$, and $I \subseteq \mathit{Inh}$:

(i) $\sigma \preceq_{I,P} \sigma'$

$$\Rightarrow \quad \big((pos, a) \in \mathit{Final_Attr_Occ}_{I,P}(\sigma) \quad \Rightarrow \quad \mathit{att}(\sigma_{\downarrow pos}, a) = \mathit{att}(\sigma'_{\downarrow pos}, a)\big)$$

(ii) $\sigma \preceq \sigma' \quad \Rightarrow \quad \mathit{att}(\sigma, a) = \bot \ \vee \ \mathit{att}(\sigma, a) = \mathit{att}(\sigma', a)$

Proof:

(i) We prove this property using well-founded induction.

Let $< \subseteq \mathit{Final_Attr_Occ}_{I,P}(\sigma) \times \mathit{Final_Attr_Occ}_{I,P}(\sigma)$ where

$$(pos_1, a_1) < (pos_2, a_2) \quad \Leftrightarrow_{def} \quad (\sigma_{\downarrow pos_1}.a_1, \sigma_{\downarrow pos_2}.a_2) \in \mathit{Dep}$$

Dep has no cycles, so $<$ is a well-founded relation.

We introduce a new predicate Φ on attribute occurrences.

$$\Phi(pos, a) \quad \Leftrightarrow_{def} \quad \big((pos, a) \in \mathit{Final_Attr_Occ}_{I,P}(\sigma) \\ \Rightarrow \quad \mathit{att}(\sigma_{\downarrow pos}) = \mathit{att}(\sigma'_{\downarrow pos})\big)$$

Using well-founded induction we show that $\Phi(pos, a)$ for all (pos, a), if $\sigma \preceq_{I,P} \sigma'$. It is necessary to show that

$$\big(\forall (pos', a') \,.\, (pos', a') < (pos, a) \quad \Rightarrow \quad \Phi(pos', a')\big) \quad \Rightarrow \quad \Phi(pos, a)$$

for all (pos, a).

To show this let $(pos, a) \in \mathit{Final_Attr_Occ}_{I,P}(\sigma)$. We distinguish the following cases.

(a) $pos = \varepsilon$ and $a \in \mathit{Inh}$. Then $a \notin I$ as $(pos, a) \in \mathit{Final_Attr_Occ}_{I,P}(\sigma)$. So $\mathit{att}(\sigma_{\downarrow pos}.a) = \mathit{att}(\sigma'_{\downarrow pos}.a)$ follows directly from $\sigma \preceq_{I,P} \sigma'$.

(b) otherwise

i. $\big(\sigma_{\downarrow pos}.a = f(\sigma_{\downarrow pos_1}.a_1, \ldots, \sigma_{\downarrow pos_n}.a_n)\big) \in \widehat{Rule}(t)$.

In this case $(pos_i, a_i) \in \mathit{Final_Attr_Occ}_{I,P}(\sigma)$ for each $1 \leq i \leq n$, as otherwise $\sigma_{\downarrow pos}.a$ would not be final.. By $\sigma \preceq \sigma'$ we also know that $\big(\sigma'_{\downarrow pos}.a = f(\sigma'_{\downarrow pos_1}.a_1, \ldots, \sigma'_{\downarrow pos_n}.a_n)\big) \in \widehat{Rule}(t')$, if $\sigma' = t'_{\downarrow pos_0}$.

$\Phi(pos', a')$ for all $(pos', a') < (pos, a)$ implies that $att(\sigma_{\downarrow pos_i}.a_i) = att(\sigma'_{\downarrow pos_i}.a_i)$ for all $1 \leq i \leq n$. Thus

$$\begin{aligned} att(\sigma_{\downarrow pos}.a) &= f\big(att(\sigma_{\downarrow pos_1}.a_1), \ldots, att(\sigma_{\downarrow pos_n}.a_n)\big) \\ &= f\big(att(\sigma'_{\downarrow pos_1}.a_1), \ldots, att(\sigma'_{\downarrow pos_n}.a_n)\big) \\ &= att(\sigma'_{\downarrow pos}.a) \end{aligned}$$

ii. $symbol(\sigma_{\downarrow pos}) = F \in T$ and $a \in Synth$. Then also $symbol(\sigma'_{\downarrow pos}) = F$ as $\sigma \preceq \sigma'$, so

$$\begin{aligned} att(\sigma_{\downarrow pos}.a) &= init(F, a) \\ &= att(\sigma'_{\downarrow pos}.a) \end{aligned}$$

Thus, $\Phi(pos, a)$ holds.

iii. otherwise

$$\begin{aligned} att(\sigma_{\downarrow pos}.a) &= \bot \\ &= att(\sigma'_{\downarrow pos}.a) \end{aligned}$$

Thus, $\Phi(pos, a)$ holds.

(ii) This property is shown in a similar way also using well-founded induction. Let $\sigma' = t'_{\downarrow pos_0}$ where $t \preceq t'$.

As *Dep* contains no cycles, $Dep(t) \subseteq Dep$ is a well-founded relation.

A new predicate $\tilde{\Phi}$ is defined as follows.

$$\tilde{\Phi}(pos, a) \quad \Leftrightarrow_{def} \quad att(t_{\downarrow pos}, a) = \bot \ \vee \ att(t_{\downarrow pos}, a) = att(t'_{\downarrow pos}, a)$$

In order to show that $\tilde{\Phi}(pos, a)$ for all (pos, a) using well-founded induction it is necessary to show that

$$\Big(\forall (pos', a') \ . \ \big((pos', a'), (pos, a)\big) \quad \Rightarrow \quad \tilde{\Phi}(pos', a')\Big) \quad \Rightarrow \quad \tilde{\Phi}(pos, a)$$

We show this distinguishing the following cases.

(a) $pos = \varepsilon$ and $a \in Inh$. Then $att(t_{\downarrow pos}, a) = \bot$, so $\tilde{\Phi}(pos, a)$ follows immediately.

(b) otherwise:

i. $\big(t_{\downarrow pos}.a = f(t_{\downarrow pos_1}.a_1, \ldots, t_{\downarrow pos_n}.a_n)\big) \in \widehat{Rule}(t)$.

By $t \preceq t'$ we know that $\big(t'_{\downarrow pos}.a = f(t'_{\downarrow pos_1}.a_1, \ldots, t'_{\downarrow pos_n}.a_n)\big) \in \widehat{Rule}(t')$.

$\forall (pos', a') \ . \ \big((pos', a'), (pos, a)\big) \ \Rightarrow \ \tilde{\Phi}(pos', a')$ implies that for each $1 \leq i \leq n$ either $att(t_{\downarrow pos_i}, a_i) = \bot$ or $att(t_{\downarrow pos_i}, a_i) = att(t'_{\downarrow pos_i}, a_i)$.

If $att(t_{\downarrow pos_i}, a_i) = \bot$ for some i, then $att(t_{\downarrow pos}, a) = \bot$, so $\tilde{\Phi}(pos, a)$ holds. Otherwise

$$\begin{aligned} att(t_{\downarrow pos}.a) &= f\big(att(t_{\downarrow pos_1}.a_1), \ldots, att(t_{\downarrow pos_n}.a_n)\big) \\ &= f\big(att(t'_{\downarrow pos_1}.a_1), \ldots, att(t'_{\downarrow pos_n}.a_n)\big) \\ &= att(t'_{\downarrow pos}.a) \end{aligned}$$

So $\tilde{\Phi}(pos, a)$ holds in this case as well.

ii. $symbol(t_{\downarrow pos}) = F \in T$ and $a \in Synth$. Then also $symbol(t'_{\downarrow pos}) = F$ as $t \preceq t'$, so

$$\begin{aligned} att(t_{\downarrow pos}.a) &= init(F, a) \\ &= att(t'_{\downarrow pos}.a) \end{aligned}$$

Thus, $\tilde{\Phi}(pos, a)$ holds.

iii. otherwise

$$\begin{aligned} att(t_{\downarrow pos}.a) &= \bot \\ &= att(t'_{\downarrow pos}.a) \end{aligned}$$

Thus, $\tilde{\Phi}(pos, a)$ holds.

□

A also satisfies the property (**Attr2**) described in Section 3.3.9 which is required for the application of dynamic programming while performing a transformation.

Theorem C.6.2
A satisfies (**Attr2**), i.e, for all σ, σ', pos, a the following holds:

$$\begin{aligned} &\big(subtree(\sigma) = subtree(\sigma') \wedge \\ &\forall b \in \mathcal{A} \,.\, (\sigma.b, \sigma_{\downarrow pos}.a) \in Dep \Rightarrow att(\sigma, b) = att(\sigma', b)\big) \\ &\quad \Rightarrow \quad att(\sigma_{\downarrow pos}, a) = att(\sigma'_{\downarrow pos}, a) \end{aligned}$$

(without proof) □

C.7 Construction of an attribute grammar from a Coala specification

The syntactic elements for the specification of an attribute grammar were described in Section 4.1.8. In the following we describe how an attribute grammar as defined in C.5 can be constructed from those elements of a Coala specification.

C.7.1 Context-free grammar

Given all the terminals and productions declared in a Coala specification a context-free grammar $\Gamma = (\Sigma, T, \Pi, S)$ as described in C.1 can easily be constructed where

- Σ is the set of all defined terminal and non-terminal symbols
- $T \subseteq \Sigma$ is the set of terminal symbols
- $\Pi \subseteq \Sigma \times \Sigma^*$ is the set of declared productions
- S is one of the defined non-terminal symbols (we will not make use of it, so it does not matter)

C.7.2 Attributes

The attribute declarations contained in a Coala specification naturally induce a set $\mathcal{A} = \mathit{Inh} \,\dot\cup\, \mathit{Synth}$ of attribute symbols.

C.7.3 Attribution rules

Coala attribution rules can be mapped to a function $\mathit{Rule}\colon \Pi \to \mathcal{P}ow(\mathit{Attr_Rule})$ as defined in C.3 as follows. Consider a Coala attribution rule of the form

attr $\mathit{node_var}_0$ '::=' $\mathit{node_var}_1 \ldots \mathit{node_var}_n$ '{'
    ...
    $\mathit{attr_occ}_0$ '=' *expr*
    ...
'}'

Further, assume the following:

- let p be the production referred to by $\mathit{node_var}_0$ '::=' $\mathit{node_var}_1 \ldots \mathit{node_var}_n$
- let $\{\mathit{attr_occ}_1, \ldots, \mathit{attr_occ}_k\}$ be the set of attribute occurrences appearing in *expr*.
- for $0 \le i \le k$, let ν_i be the position of $\mathit{attr_occ}_i$'s node variable in the attribution rule's production where the production's left hand side has position ε and $\langle j \rangle$ is the position of the j-th right hand side node variable.
- let a_i be the attribute referenced by $\mathit{attr_occ}_i$ for $0 \le i \le k$.

We construct a semantic function $f_{p,a_0,\nu_0}(x_1, \ldots, x_k)$ which returns the result of evaluating *expr* in an environment that binds $\mathit{attr_occ}_i$ to the value of x_i for each $1 \le i \le k$. Using f_{p,a_0,ν_0}, an attribution rule as defined in C.3 is then given by

$$(a_0, \nu_0) = f_{p,a_0,\nu_0}\big((a_1, \nu_1), \ldots, (a_k, \nu_k)\big)$$

The complete mapping *Rule* for an attribute grammar can be created by collecting all attribution rules contained in a Coala specification including implicit copy rules.

APPENDIX D

Summary of Coala syntax

1.	*specification*	::=	**specification** $ident_{41}$ ‘;’ $decl_{2}{}^{*}$
2.	*decl*	::=	$include_decl_{3}$
		|	$terminal_decl_{4}$
		|	$production_list_{5}$
		|	fct_def_{12}
		|	$attr_decl_{9}$
		|	$attr_rules_{10}$
		|	$trafo_rule_{13}$
3.	*include_decl*	::=	**include** $ident_{41}$ ‘;’
4.	*terminal_decl*	::=	**terminal** $ident_list_{32}$ ‘;’
5.	*production_list*	::=	**productions** ‘{’ $productions_{6}{}^{+}$ ‘}’
6.	*productions*	::=	$ident_{41}$ ‘::=’ $prod_rhs_list_{7}$ ‘;’
7.	*prod_rhs_list*	::=	$prod_rhs_{8}$
		|	$prod_rhs_list_{7}$ ‘|’ $prod_rhs_{8}$
8.	*prod_rhs*	::=	$node_var_{35}{}^{*}$
9.	*attr_decl*	::=	(**inherited** | **synthesized**) $ident_list_{32}$ **with** $ident_list_{32}$ ‘;’

10. *attr_rules* ::= **attr** $node_var_{35}$ ‘::=’ $node_var_{35}{}^{*}$
    ‘{’ $attr_equ_{11}{}^{+}$ ‘}’
11. *attr_equ* ::= $attr_occ_{38}$ ‘=’ $expr_{22}$ ‘;’
12. *fct_def* ::= **fct** $ident_{41}$ ‘(’ [$ident_list_{32}$] ‘)’ ‘=’ $expr_{22}$
13. *trafo_rule* ::= **rule** [rel_ident_{42} ‘:’] $template_{18}{}^{*}$ ‘->’ $template_{18}$
    [$trafo_rest_{14}$] [$constraints_{15}$] [$minimizer_{17}$] ‘;’
14. *trafo_rest* ::= ‘/’ $node_var_{35}{}^{*}$
15. *constraints* ::= **where** $constraint_{16}{}^{*}$
16. *constraint* ::= ‘[’ $expr_{22}$ ‘]’ ‘;’
    | [rel_ident_{42} ‘:’]
    $leaf_template_{19}{}^{*}$ ‘->’ $leaf_template_{19}$ [$trafo_rest_{14}$] ‘;’
17. *minimizer* ::= **minimize** $expr_{22}$ ‘;’
18. *template* ::= $leaf_template_{19}$
    | $node_var_{35}$ ‘<’ $template_{18}{}^{*}$ ‘>’
19. *leaf_template* ::= $node_var_{35}$
    | ‘(’ $ident_{41}$ ‘:’ $expr_{22}$ ‘)’
    | ‘(’ $expr_{22}$ ‘)’
    | $const_expr_{28}$
    | $attr_occ_{38}$

Expressions

22. *expr* ::= **if** $expr_{22}$ **then** $expr_{22}$ **else** $expr_{22}$
    | **let** $ident_{41}$ ‘=’ $expr_{22}$ **in** $expr_{22}$
    | $expr_{22}$ **where** $local_def_list_{23}$
    | $ident_{41}$ ‘(’ [$argument_list_{25}$] ‘)’
    | $ident_{41}$
    | $attr_occ_{38}$
    | $const_expr_{28}$
    | $expr_{22}$ $binop_{26}$ $expr_{22}$
    | $unop_{27}$ $expr_{22}$
    | $expr_{22}$ ‘’’ $ident_{41}$
    | ‘(’ $expr_{22}$ ‘)’

23. *local_def_list* ::= $\mathit{local_def}_{24}$
  | $\mathit{local_def_list}_{23}$ ‘,’ $\mathit{local_def}_{24}$

24. *local_def* ::= ident_{41} ‘=’ expr_{22}

25. *argument_list* ::= expr_{22}
  | $\mathit{argument_list}_{25}$ ‘,’ expr_{22}

26. *binop* ::= ‘*’ | ‘/’ | ‘+’ | ‘-’
  | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’
  | ‘&&’ | ‘||’

27. *unop* ::= ‘!’ | ‘-’

28. *const_expr* ::= $\mathit{string_const}_{52}$
  | $\mathit{nat_const}_{47}$
  | $\mathit{real_const}_{48}$
  | $\mathit{dim_const}_{50}$
  | $\mathit{bool_const}_{29}$
  | **nil**

29. *bool_const* ::= **true** | **false**

Lists of identifiers

32. *ident_list* ::= ident_{41}
  | $\mathit{ident_list}_{32}$ ‘,’ ident_{41}

Node variables

35. *node_var* ::= ident_{41}
  | ident_{41} ‘[’ $\mathit{nat_const}_{47}$ ‘]’
  | ident_{41} ‘[’ ident_{41} ‘]’

Attribute occurrences

38. *attr_occ* ::= $\mathit{node_var}_{35}$ ‘.’ ident_{41}

Lexical tokens

41. *ident* ::= letter_{43} $\mathit{letter_or_digit}_{45}{}^{*}$

42. *rel_ident* ::= ‘@’ ident_{41}

		|	‘`@@`’
43.	*letter*	::=	‘`_`’ | ‘`a`’ | ‘`b`’ | ... | ‘`z`’ | ‘`A`’ | ‘`B`’ | ... | ‘`Z`’
44.	*digit*	::=	‘`0`’ | ‘`1`’ | ... | ‘`9`’
45.	*letter_or_digit*	::=	$letter_{43}$ | $digit_{44}$
46.	*hex_digit*	::=	$digit_{44}$
		|	‘`a`’ | ‘`b`’ | ... | ‘`f`’ | ‘`A`’ | ‘`B`’ | ... | ‘`F`’
47.	*nat_const*	::=	${digit_{44}}^{+}$
48.	*real_const*	::=	${digit_{44}}^{*}$ ‘`.`’ ${digit_{44}}^{+}$ [$exponent_{49}$]
49.	*exponent*	::=	(‘`e`’ | ‘`E`’) [‘`+`’ | ‘`-`’] ${digit_{44}}^{+}$
50.	*dim_const*	::=	nat_const_{47} $measure_unit_{51}$
		|	$real_const_{48}$ $measure_unit_{51}$
51.	*measure_unit*	::=	**pt** | **cm** | **mm** | **in**
52.	*string_const*	::=	‘`"`’ ($string_char_{53}$ | $escape_sequence_{54}$)* ‘`"`’
53.	*string_char*	::=	every character except ‘`"`’, ‘`\`’
54.	*escape_sequence*	::=	‘`\`’ (‘`b`’ | ‘`t`’ | ‘`n`’ | ‘`f`’ | ‘`r`’ | ‘`"`’ | ‘`\`’)
		|	‘`\`’ ‘`u`’ hex_digit_{46} hex_digit_{46} hex_digit_{46} hex_digit_{46}

APPENDIX E

Haskell implementation

The following lists a `Haskell` implementation of the transformation model presented in this dissertation. This implementation was designed to be as close as possible to the solution given in Chapter 3. As such it serves as a proof of concept, but has also in no way been tuned towards efficiency. Note also that the implementations of some functions having little relevance to the transformation model have been omitted. A full version of the program including test cases can be obtained from the author.

In Chapter 3 variables denoting sets and functions yielding sets have been given names with capitalized first letters. In `Haskell`, however, names beginning with capital letters always represent types. So, in order to stay as close as possible to the notation used in Chapter 3, the convention has been followed that variables and functions giving sets are always prefixed with an underscore character. For example, the function *Var* giving the variables occurring in a template is denoted here by `_Var`.

```
-- ---------------------------------------------
-- ---------------------------------------------
--
-- Haskell implementation of Coala's
-- attributed tree transformation model
--
-- ---------------------------------------------
-- ---------------------------------------------

-- ---------------------------------------------
-- Syntactic symbols and semantic domain
-- ---------------------------------------------

type Sym = String
```

```
unexpsym :: Sym
unexpsym = "_"

type D = Int

-- -----------------------------------------------
-- Trees
-- -----------------------------------------------

data Tree = Tree Sym [Tree]

instance Show Tree where
  show (Tree s []) = s
  show (Tree s ts) = s ++ "(" ++ (showsubtrees ts) ++ ")"
    where
      showsubtrees :: [Tree] → String
      showsubtrees [] = ""
      showsubtrees (t:[]) = (show t)
      showsubtrees (t:ts) = (show t) ++ "," ++ (showsubtrees ts)

instance Eq Tree where
  (Tree s ts) == (Tree s' ts') = s == s' && ts == ts'

type Pos = [Int]

symbol :: Tree → Sym
symbol (Tree s _) = s

subtree :: Pos → Tree → Tree
subtree [] t = t
subtree (i:is) (Tree _ ts) = subtree is (elementAt ts i)
subtree _ _ = error "subtree: invalid pos"

treplace :: Tree → Tree → Pos → Tree
treplace t1 t2 [] = t2
treplace (Tree s ts) t2 (i:is) = Tree s (replaceElem ts t3 i)
  where
    t3 = treplace (elementAt ts i) t2 is

subtree_count :: Tree → Int
subtree_count (Tree _ ts) = length ts

unexptree :: Sym →Tree
unexptree s =
  Tree s [Tree unexpsym []]

-- -----------------------------------------------
-- Tree nodes
-- -----------------------------------------------

data Node = Tree :@: Pos
```

```
instance Eq Node where
    (t :@: pos) == (t' :@: pos')  =  t == t' && pos == pos'

nsubtree :: Node → Tree
nsubtree (t :@: pos) = subtree pos t

child :: Int → Node → Node
child i (t :@: pos) = t :@: (pos ++ [i])

desc :: Node → Pos → Node
desc (t :@: pos1) pos2 = t :@: (pos1 ++ pos2)

nsymbol :: Node → Sym
nsymbol (t :@: pos) = symbol (subtree pos t)

replace :: Node → Tree → Node
replace (t1 :@: pos) t2 = (treplace t1 t2 pos) :@: pos

child_count :: Node → Int
child_count n = subtree_count (nsubtree n)

children :: Node → [Node]
children n = [child i n | i ← [1 .. (child_count n)]]

parent :: Node → Node
parent (t :@: pos) = t :@: (init pos)

-- ------------------------------------------
-- Theory chapter's example attribution
-- ------------------------------------------

att :: Node → String → Maybe Int

-- (implementation omitted)

-- ------------------------------------------
-- Theory chapter's example constraint
-- system
-- ------------------------------------------

sat :: String → [Maybe Int] → Bool

-- (implementation omitted)

-- ------------------------------------------
-- Node variables and templates
-- ------------------------------------------

data Var = Sym :#: Int

instance Eq Var where
```

```
  (s1 :#: i1) == (s2 :#: i2)  =  s1 == s2 && i1 == i2

instance Show Var where
  show (s :#: nr) = (show s) ++ "[" ++ (show nr) ++ "]"

vsymbol :: Var → Sym
vsymbol (s :#: id) = s

data Template = LeafVar Var
              | CTemplate Var [Template]

_Var :: Template → [Var]
_Var (LeafVar x) = [x]
_Var (CTemplate x ts) =
  [x] ++ (concat $ map _Var ts)

_LeafVar :: Template → [Var]
_LeafVar (LeafVar x) = [x]
_LeafVar (CTemplate x ts) =
  concat (map _LeafVar ts)

root_var :: Template → Var
root_var (LeafVar x) = x
root_var (CTemplate x ts) = x

-- ---------------------------------------------
-- Attribute occurrences
-- ---------------------------------------------

data AttrOcc = Var :.: String

-- ---------------------------------------------
-- Transformation rules and specifications
-- ---------------------------------------------

data Rule = Rule [Template] Template [Var]
                 [Constraint] Obj
data Constraint = SynConstr [Var] Var [Var]
                | SemConstr String [AttrOcc]
type Obj = Maybe ([Maybe D] →Int, [AttrOcc])
type Spec = [Rule]
type Sig = ([Sym], Sym, [Sym])

rsignature :: Rule → Sig
rsignature (Rule _Ss _T _Rs _ _) = (ss, t, rs)
  where
    ss = [vsymbol $ root_var _S | _S ← _Ss]
    t  = vsymbol $ root_var _T
    rs = [vsymbol _R | _R ← _Rs]

csignature :: Constraint → Sig
csignature (SynConstr _Xs _Y _Zs) = (xs, y, zs)
```

```
  where
    xs = [vsymbol _X | _X ← _Xs]
    y  = vsymbol _Y
    zs = [vsymbol _Z | _Z ← _Zs]

-- ---------------------------------------------
-- Variable environments
-- ---------------------------------------------

type Env = Var →Maybe Node

attenv :: Env → AttrOcc→ Maybe D
attenv env (x :.: a) =
  case env x of
  Nothing →Nothing
  Just n → att n a

satenv :: Env → Constraint → Bool
satenv env (SemConstr p aos) = sat p vs
  where
    vs = map (attenv env) aos

empty_env :: Env
empty_env v = Nothing

envget :: Env → Var → Node
envget env v = checkdef (env v)
  where
    checkdef (Just n) = n
    checkdef _ = error $ "envget: undef (" ++ (show v) ++ ")"

-- ---------------------------------------------
-- Processing states
-- ---------------------------------------------

data State = Fail
           | State Env

-- ---------------------------------------------
-- Auxiliary operations
-- ---------------------------------------------

empty_state :: State
empty_state = State empty_env

getenv :: State → Env
getenv (State env) = env
getenv Fail = error "env: Fail"

restrict_env :: Env → [Var] → Env
restrict_env env xs = env'
  where
```

```
    env' x = if x `elem` xs then env x
             else Nothing

(<:) :: State → State → State
Fail <: _     = Fail
_    <: Fail  = Fail
(State env1) <: (State env2) =
  State env3
  where
    env3 x = case env2 x of
               Just n → Just n
               Nothing →env1 x

(//) :: Node →Var → State
n // v = State env
  where
    env x = if x == v then Just n
            else Nothing

(///) :: [Node] → [Var] → State
[] /// [] = State empty_env
(n:ns) /// (v:vs) = (n // v) <: (ns /// vs)
_ /// _ = error "///: differing lengths"

inst1 :: Template → Node →State
inst1 (LeafVar x) n =
  if vsymbol x == nsymbol n then
    n // x
  else Fail
inst1 (CTemplate x ts) n =
 (inst ts (children n)) <: (inst1 (LeafVar x) n)

inst :: [Template] → [Node] → State
inst [] [] = empty_state
inst [] _ = Fail
inst _ [] = Fail
inst (t:ts) (n:ns) =
  (inst1 t n) <: (inst ts ns)

build :: Template → State → Tree
build (LeafVar x) (State env) =
  case env x of
  Nothing →unexptree (vsymbol x)
  Just n  → nsubtree n
build (CTemplate x ts) s@(State env) =
  Tree f [build t s | t ← ts]
  where
    f = vsymbol x
build _ Fail = error "build: Fail"

anchor :: Var → Env → Node
anchor x env =
```

```
  case env x of
  Just n → n
  otherwise → (unexptree f) :@: []
  where
    f = vsymbol x

-- ---------------------------------------------
-- Generic constraint processing
-- ---------------------------------------------

type GConstraint a = a → [a]

process1 :: (GConstraint a) → [a] → [a]
process1 c ss = concat $ map c ss

process :: [GConstraint a] → [a] → [a]
process [] states = states
process (c:cs) states =
  process cs $ process1 c states

-- ---------------------------------------------
-- Transformation functions
-- ---------------------------------------------

spec :: Spec

type Result = (Node, [Node])
type PState = (Node, State)

trafo :: Sig → [Node] → Node → [Result]
trafo sig sigmas gamma =
  concat [ trafo_r r sigmas gamma |
           r ← spec, rsignature r == sig]

trafo_r :: Rule → [Node] → Node → [Result]
trafo_r r@(Rule _Ss _T _Rs phis obj) sigmas gamma =
  makeResult r ss'
  where
    ss'  = minimize obj ss
    ss   = process (makeGConstrs r) [(tau0, env0)]
    tau0 = replace gamma (build _T empty_state)
    env0 = (inst _Ss sigmas) <: (inst1 _T tau0)

makeResult :: Rule → [PState] → [Result]
makeResult _ [] = []
makeResult r ((_, Fail ):ss) = makeResult r ss
makeResult r ((tau, State env):ss) = x:xs
  where
    x = makeResult' r tau env
    xs = makeResult r ss

makeResult' :: Rule → Node →Env → Result
```

```
makeResult' (Rule _ _T _Rs _ _) tau env =
  (tau, [envget env r | r ← _Rs])

minimize :: Obj → [PState] → [PState]
minimize Nothing ss = ss
minimize obj@(Just (f, aos)) ss =
  [s | s ← ss, cost obj s == c]
  where
    c = minimum [cost obj s' | s' ← ss]

cost :: Obj → PState → Int
cost Nothing _ = error "cost: no obj fct"
cost _ (_, Fail) = error "cost: fail"
cost (Just (f, aos)) (_, State env) = f vs
  where
    vs = map (attenv env) aos

makeGConstrs :: Rule → [GConstraint PState]
makeGConstrs r@(Rule _Ss _T _Rs phis obj) =
  [makeGConstr r phi | phi ← phis]

makeGConstr :: Rule → Constraint →
                              GConstraint PState
makeGConstr r phi@(SemConstr s aos) = c
  where
    c (_, Fail) = []
    c s@(_, State env ) =
      if satenv env phi then [s]
      else []

makeGConstr (Rule _Ss _T _Rs phis obj)
            phi@(SynConstr _Xs _Y _Zs) = c
  where
    c (_, Fail) = []
    c (tau, State env) =
      [makePState _T _Rs _Y env tau result |
        result ← _Result]
      where
        _Result =
          trafo (csignature phi)
                [envget env _X | _X ← _Xs]
                (anchor _Y env)

makePState :: Template → [Var] → Var → Env → Node → Result
                                                  → PState
makePState _T _Rs _Y env tau (tau', rhos') = (_tau_, _env_)
  where
    _tau_ = replace tau (build _T (State env <:
                                   (tau' // _Y )))
    _env_ = State env <: (tau' // _Y) <:
              (rhos' /// _Rs) <: (inst1 _T _tau_)
```

```
-- ---------------------------------------------
-- Utility functions
-- ---------------------------------------------

-- get element from list where index starts at 1
elementAt :: [a] → Int → a

-- replace element at index starting at 1
replaceElem :: [Tree] → Tree → Int → [Tree]

-- (implementations omitted)

-- ---------------------------------------------
-- Example specification
-- ---------------------------------------------

-- spec = [ r1, r2, r3, r4 ] where
spec = [ r1, r2, r3, r4 ] where
 r1 = Rule
       [LeafVar ("S":#:0)]
       -- →
       (CTemplate ("T":#:0) [LeafVar ("E":#:0)])
       []
       -- where
       [SynConstr ["S":#:0] ("E":#:0) []]
       -- minimizing
       Nothing

 r2 = Rule
       [CTemplate ("S":#:0)
        [CTemplate ("B":#:0)
         [LeafVar ("S":#:1), LeafVar ("S":#:2)]
         ]]
       -- →
       (CTemplate ("E":#:0)
        [CTemplate ("H":#:0)
         [LeafVar ("E":#:1), LeafVar ("E":#:2)]
         ])
       []
       -- where
       [SynConstr ["S":#:1] ("E":#:1) [],
        SynConstr ["S":#:2] ("E":#:2) [],
        SemConstr "P" [("E":#:0 :.: "x"),
                       ("E":#:0 :.: "w")]
        ]
       -- minimizing
       Nothing

 r3 = Rule
       [CTemplate ("S":#:0)
        [CTemplate ("B":#:0)
         [LeafVar ("S":#:1), LeafVar ("S":#:2)]
```

```
          ]]
        -- →
        (CTemplate ("E":#:0)
         [CTemplate ("V":#:0)
           [LeafVar ("E":#:1), LeafVar ("E":#:2)]
           ])
        []
        -- where
        [SynConstr ["S":#:1] ("E":#:1) [],
         SynConstr ["S":#:2] ("E":#:2) [],
         SemConstr "P" [("E":#:0 :.: "x"),
                        ("E":#:0 :.: "w")]
         ]
        -- minimizing
        Nothing

r4 = Rule
        [CTemplate ("S":#:0)
                   [LeafVar ("L":#:0)]]
        -- →
        (CTemplate ("E":#:0)
                   [CTemplate ("A":#:0) []])
        []
        -- where
        []
        -- minimizing
        Nothing
```

Index

Symbols

$T_{\downarrow pos}$, *32*
$[\sigma/\xi]$, *54*
$[\sigma_1 \ldots \sigma_k/\xi_1 \ldots \xi_k]$, *54*
$Attr_Occ$, *32*
$Attr_Occ(T)$, *32*
Def_Var_{φ}, *50*
$Def_Var_{i,r}$, *51*
$Dep(T)$, *49*
$Dom(env)$, *54*
Env, *36*
$Final_Attr_Occ_{I,P}(\sigma)$, *47*
$Final_Attr_Occ_{I,V}(T)$, *49*
$Final_Attr_Occ_{i,r}$, *52*
$Final_Pos_P(\sigma)$, *47*
$Leaf_Var$, *32*
$Pos(T)$, *32*
$Prelim_Attr_Occ_{I,P}(\sigma)$, *47*
$Prelim_Attr_Occ_{I,V}(T)$, *49*
$Requ_Attr(r)$, *52*
$Requ_Attr_Occ(\varphi)$, *52*
Sig, *51*
$Template$, *31*
$Var(T)$, *31*
$anchor(Y, env)$, *57*
$build(T, env)$, *56*
$env_{\mathcal{X}}$, *54*
$inst(\bar{T}, \bar{\sigma})$, *55*
$target(T, \gamma, env)$, *57*
$\mathcal{N}_{\perp}$, *36*
Σ-tree, *26*
$\mathcal{S}tate$, *54*
$\mathcal{TR}(spec)$, *37*
$\mathcal{TR}_r(spec)$, *37*
$\mathcal{TR}_{opt}(spec)$, *40*
$fail$, *54*
$\vdash_E$, *58*
$\perp_{Env}$, *54*
Obj, *39*
$\preceq_{I,P}$, *48*
$\preceq$, *44*
$\triangleleft$, *54*
TeX, 11–13, 20
ascent, **141**
baseline, **141**
character set, **141**
character, **141**
descent, **141**
em-width, **142**
encoding, **141**
ex-height, **141**
font, **142**
glyph, **142**
kerning, **142**
leading, **142**
ligature, **142**
orphan, **142**
path, **142**
point size, **142**
point, **142**
type 1 font, **142**
utf-8, **143**
unicode, **142**
widow, **143**

A

Agenda, 8, 123, 127–128, 130, 131, 135
attribute, *28*
attribute grammar, 4, **4**, 6, 8, 9, 11, 12, 15, 20, 29, 45, 66, 77, 97, 124, 126–128, 130, 134, **159–166**
attribute occurrence, *32*
attribution, *28*

B

BNF, 78, 86

C

C++, 79
Coala, 9, *77*, 75–97, 99, 100, 102, 103, 106, 107, 109, 111, 115–118, 121–124, 126–128, 130, 131, 134, 135, 165–167
compiler construction, 4
complete tree, *44*
concatenation, 9
constraint, 1, 5, 6, **6**, 7, 8, 18, 19, 22, 30–33, *33*, 88, **129–130**, 133
Constraint Logic Programming, **128–130**
constraint system, 31, *31*, 34, 35
context-free grammar, 3, **3**, 4, 11, 12, 25, 28, 124, 159, 161

D

DITA, 3
DocBook, 125
document, 1, **1**, 2–5, 11, 22, 23, 116, 118, **123–131**, 142
  structured, 8
document class, 2, **2**, 3
document formatter, 1, 5, **5**, 6–8, 24, 77, 92, 96, 97
document processing, 1, 91, 118, 122, 141, 142
document processing system, 1, **2**, 123
document structure, 11, **11–12**, 13, 15, 17, 77, 123–125
Document Type Definition, 3
domain, *28*, *54*
DSSSL, 8, 123–127, 130
dynamic programming, 25, 42, **66–69**, 71, 72, 75, 134, 165

E

empty environment, *54*

F

formatter, *see* document formatter
formatting object tree, 125

G

glue, 14
grammar symbol, 3

H

Haskell, 66, 171
HTML, 8

I

incremental attribution, 49
inherited attribute, *28*
instantiation (of a template), *32*
instantiation by environment, *36*
invalid processing state, *54*

J

Java, 77, 79, 97, 134

L

less or equally expanded, *45*
line breaking, 13
Logic Programming, **128–130**
Lout, 8, 13
LR specification, *53*

M

matching (of a template), *32*
monotonous attribution, *48*

N

node, *27*
node variable, *31*

O

objective function, *39*
ODA, 123–124, 130
optimal S-derivation, *39*
optimizing transformation specification, *39*

P

PDF, 8, 96
Portable Document Format, 4
position, *26*
position (in a template), *32*
PostScript, 77, 91, 96, 97
Postscript, 91, 94, 96, 103, 106, 118, 134, 142
power set, 9
predecessor, *45*
prefix, *9*
preliminary tree attribute occurrence, *47*
processing state, *54*

production, 3
Prolog, 128

R
rest, *33*
restriction (of a function), 9
restriction of an environment, *54*
rule system, 57

S
S-derivation, *37*
Scalable Vector Graphics, 4
semantic constraint, 25, 30, *33*, 80, 89, 90, 133
sequence, 9
SGML, 2, 3, 123–127, 130
signature (of syntactic constraint), *51*
signature (of transformation rule), *51*
source, *33*
structured document model, 2, **2**, 4, 123, 125, 127
suitable anchor, *60*
SVG, 125
syntactic constraint, *33*, 88, 89, 133
synthesized attribute, *28*

T
target, *33*
template, *31*
transformation rule, *33*
transformation specification, *33*
tree attribute occurrence, *28*

V
variable environment, *36*

W
Wikipedia, 118

X
XHTML, 125
XML, 2, 3, 77, 91–94, 97, 123–127, 130, 134
XML Schema, 3, 125
XSL, 123–127, 130
XSL-FO, 8
XSLT, 8