# A Pattern-based Foundation for Language-Driven Software Engineering

Tim Reichert

A thesis submitted in partial fulfilment
of the requirements of the
University of Northumbria at Newcastle
for the degree of
Doctor of Philosophy

Research undertaken in the
School of Computing, Engineering
and Information Sciences

August 2011

# Abstract

This work brings together two fundamental ideas for modelling, programming and analysing software systems. The first idea is of a methodological nature: engineering software by systematically creating and relating languages. The second idea is of a technical nature: using patterns as a practical foundation for computing. The goal is to show that the systematic creation and layering of languages can be reduced to the elementary operations of pattern matching and instantiation and that this pattern-based approach provides a formal and practical foundation for language-driven modelling, programming and analysis.

The underpinning of the work is a novel formalism for recognising, deconstructing, creating, searching, transforming and generally manipulating data structures. The formalism is based on typed sequences, a generic structure for representing trees. It defines basic pattern expressions for matching and instantiating atomic values and variables. Horizontal, vertical, diagonal and hierarchical operators are different ways of combining patterns. Transformations combine matching and instantiating patterns and they are patterns themselves. A quasiquotation mechanism allows arbitrary levels of meta-pattern functionality and forms the basis of pattern abstraction. Path polymorphic operators are used to specify fine-grained search of structures. A range of core concepts such as layering, parsing and pattern-based computing can naturally be defined through pattern expressions.

Three language-driven tools that utilise the pattern formalism showcase the applicability of the pattern-approach. *Concat* is a self-sustaining (meta-)programming system in which all computations are expressed by matching and instantiation. This includes parsing, executing and optimising programs. By applying its language engineering tools to its own meta-language, Concat can extend itself from within. *XMF (XML Modeling Framework)* is a browser-based modelling- and meta-modelling framework that provides flexible means to create and relate modelling languages and to query and validate models. The pattern functionality that makes this possible is partly exposed as a schema language and partly as a JavaScript library. *CFR (Channel Filter Rule Language)* implements a language-driven approach for layered analysis of communication in complex networked systems. The communication on each layer is visible in the language of an “abstract protocol” that is defined by communication patterns.

# Contents

# List of Figures

# Source Code Listings

# List of Tables

# Acknowledgements

My deepest gratitude goes to my external supervisor, mentor and friend Prof. Dr. Dominikus Herzberg. I am forever indebted to him for his unwavering support through all the ups and downs, for sharing with me his deep insights into informatics and for always being there when I needed his help. Without Dominikus, this document would not exist.

I am very grateful to Dr. Nick Rossiter, my supervisor at Northumbria University, who gave me the chance to do highly interesting research, supported me throughout the years and gave valuable feedback on a draft of this thesis. Prof. Dr. Ahmed Bouridane, Prof. Dr. Maia Angelova and David Livingstone at Northumbria University were very supportive at different stages of my project and I am very thankful for that.

During active research and write up, I could always rely on the sharp mind of my friend and great software engineer Robert Skarwecki to produce tremendously valuable feedback on all levels, from big picture stuff to source code details and wording issues. Thank you so much for all the unbilled hours you spent on my project ;)

When studying Software Engineering at Heilbronn University, I became deeply interested in programming languages and the "big" questions in computing. After graduation, it was Prof. Dr. Nicola Marsden who encouraged me to follow my passion and who helped me with finding a suitable PhD programme. I am deeply grateful to her for all the support she gave me over the years.

Aaron Müller, Florian Eitel, Wolfgang Schoch and Edmund Klaus helped validate and inspire my own research on Concat and CFR through their excellent bachelor and diploma thesis work. Manuel Sontag and Markus Willinger gave useful comments on a draft version of this document. Benjamin Sommerfeld assisted me with proofreading.

For support in the form of scholarships, I would like to thank the Robert Bosch Foundation, Gustav Berger-Stiftung and Thomas Gessmann-Stiftung. The XMF project was partially funded by the Ministry of Education Baden Württemberg, Germany.

Socialising with a PhD student can be quite tough, as my family and friends have experienced over the years and especially in the last few months. All the more I would like to thank all of you for the love and support I received.

## Declaration

I declare that the work contained in this thesis has not been submitted for any other award and that it is all my own work.

Name: Tim Reichert

Signature:

Date: 24 August 2011

# Chapter 1

# Introduction

This PhD thesis combines two fundamental ideas for modelling, programming and analysing software systems. The first idea is of a methodological nature: engineering software by systematically creating and relating languages. The second idea is of a technical nature: using patterns as a practical foundation for computing. The result is a unified approach to software engineering that methodologically and technically scales from machine language to user interface. The research described in this thesis is both theoretical and practical. It formally defines a core of pattern functionality and a precise notion of language layering and then applies this foundation by creating three tools: a sustainable programming framework for experimenting with language designs, a web-based meta-modelling environment and a protocol language for analysing modern automotive networks in layers of languages.

## 1.1 Problem Description

Millions of lines of source code that are hard to maintain and out of date with respect to the newest implementation technologies appear to be a quasi-standard in large software projects [96]. Language-driven approaches to software engineering promise a drastic reduction in program size, more readable code and less platform-dependence [30, 35, 149]. The key to achieve this is a shift of focus in software development from programs to languages [49, 50]. Software engineering becomes an activity driven by the quest for the "right" means of expression [61]. While this view on software engineering has gained prominence in recent years and several approaches exist that can be called language-driven, there is still a need for a solid foundation for language-driven engineering.

Patterns are a promising candidate for providing this foundation. Pattern matching has been studied for a long time in different fields of computer science, see Chapter 2. Re-

cently, patterns have been investigated as a practical foundation for programming [9, 89]. This thesis is an attempt to capture the essence of Language-Driven Software Engineering (LDSE) and to define a pattern-based foundation for LDSE that is both formally sound and useful in practice.

Combining the unifying power of patterns and the unifying power of languages promises to bridge a wide gap between the theoretical possibilities and practical actualities of the software field. Theoretically, computing can be captured by a few deduction rules of a calculus [33] and the ability to create abstractions based on other abstractions is a source of nearly unlimited expressive power [1, 22, 51]. Practically, mainstream programming languages require hundreds of pages of specifications [60] and applications consist of vast amounts of unreadable and redundant source code [92, 149].

## 1.2 Background

This section provides an introduction to the core concepts underlying the thesis and discusses challenges for supporting a language-driven approach.

### 1.2.1 Computing with Patterns

Patterns play an important role in different fields of computing [166]. Regular expressions use patterns for manipulating text [158]. Functional programming languages construct and deconstruct algebraic data types and express conditional execution with patterns [122]. Logic programming languages use patterns – terms with variables – and their unification to query knowledge bases and for implementing deduction [153]. Pattern-based schema languages for XML restrict valid documents to those that match the schema [126].

The defining property of patterns is that they describe operations on particular data structures, e.g., strings, algebraic types, terms and XML documents, in a declarative manner. Patterns mix elements of a data language with elements of a meta-language to define these operations. The simplest patterns are expressions of the data language, i.e., patterns that do not contain meta-language elements. More interesting are patterns that intersperse data language expressions with variables. More expressive power is provided by patterns that use operators to express optionality, recurrence or alternatives of patterns in a structure [178]. Pattern operators can be used to define deeper concepts such as traversing, transforming or querying data [9]. Ultimately, the amount of expressive power and control a pattern language requires is application-dependent.

The declarative nature of patterns allows their interpretation in different ways. Recognition uses a pattern to validate that data corresponds to a schema. Matching combines

recognition with decomposition of structures through variable bindings. Instantiation creates data structures based on a pattern and variable bindings. Pattern-based transformation is defined by first matching a pattern that is the source of the transformation and then instantiating a pattern that is the target of the transformation. Querying a structure with patterns involves searching for occurrences of a pattern along paths in the structure. Refining patterns involves replacing a subset of meta-elements with data elements.

It shall be noted that in software engineering the term pattern is also used in a different sense, namely in the context of design patterns [57]. Design patterns are reusable solutions to common problems in designing and programming systems. In contrast to that, the patterns in this work are expressions in a pattern language that are interpreted as operations on data structures.

### 1.2.2 Language-Driven Software Engineering

Meta-linguistic abstraction – the creation of new languages through the use of existing ones – is a recognised principle for controlling complexity in engineering [1]. Throughout science, engineering and mathematics, specialised languages are used and invented to tackle problems more efficiently; examples include architectural diagrams, electric circuit models and logics [50]. Specialist terminology is introduced constantly in natural language to make communication effective and efficient. In software development, the systematic creation and use of languages by engineers is still not an established principle. This is surprising to some degree, as software engineering is inherently concerned with language: all software is description based on language. This includes programs and, therefore, also compilers and interpreters. In a wider sense, it includes all communication with a computer or between computers via a syntactic interface.

Software engineers have the power to not only invent but also implement languages. In the early days of computing, it was necessary for engineers to manually *translate* to machine language solutions expressed in natural language, mathematics or through diagrams. The introduction of high-level languages abstracted practical computing from the computer itself. Programming in a high-level language does (at least in theory) not require knowledge about the underlying machine, machine code being executed and the relationship between expressions and machine code. The general principle is that of hiding the actualities of a system behind a syntactic interface and to substitute them for a conceptual model that explains the system behaviour.

This approach *scales* because a language resulting from the abstraction process can be utilised to implement a new language that again provides its own conceptual model [177]. The principle can be illustrated using the example of one general purpose programming

language implementing another: a functional programming language implemented using a compiler to machine code abstracts from the actual machine. Users of the language think in terms of a conceptual model of functions and their application to arguments. Using this language to implement an object-oriented language again abstracts from this implementation and replaces it with a model of objects sending messages [59].

The abstraction principles just described are powerful in the sense that every new language not only introduces its own notation, but also its own way of thinking [7]. A syntactic interface must be designed in such a way that it reflects the conceptual model and not the actual implementation. For example, if objects are implemented as dispatch tables [123], this detail must not be visible at the syntactic interface.

The need to find suitable software languages is not limited to programming. Modern software systems are more and more interconnected and communicate using different protocols. This is true not only for personal computers and mobile devices but also for embedded systems. For example, modern cars contain up to 100 microcomputers that communicate via several bus systems [23]. The communication behaviour exposed by these systems is complex. Analysing this behaviour in a meaningful way requires means to render it in a language that reflects how application developers think about the problems.

What distinguishes language abstractions from other abstraction techniques, e.g., procedural or object-oriented, is the focus on the syntactic interface and the ability to create conceptual models (execution models) independent of an implementation language. The concept of language also has associated with it a larger scope than that of function or object. By applying meta-linguistic abstraction to the creation of software, engineers can build languages that reflect the way they think about (parts of) a system and that – at the same time – are executable. This also blurs the boundaries between programming languages and user interfaces [84]: the system itself becomes the implementation of the language with its user interface being the syntax. The creation of a system can, therefore, be understood entirely in terms of creating and relating languages.

The overall goal of introducing new languages is to provide engineers with the most suitable means for creating, viewing and exploring a system [148]. What the most suitable language is depends on the particular system. This entails that new languages have to be created constantly using a combination of existing ones. All these observations lead to a view on software engineering that not only emphasises the importance of language but also considers the constant creation of new languages as the driving force of development. Software engineering becomes language engineering and thus is *language-driven*.

This work defines language layering (Section 1.3) and patterns as the methodological and technical foundation for LDSE and introduces tools for engineering languages buillt upon this foundation. In Section 2.6, several existing approaches that can be considered

language-driven will be discussed and contrasted to this work. All these approaches make the constant creation of new computer languages with the goal of achieving better means of expression a central theme and provide necessary methodologies and tools. Examples include Language-Oriented Programming [177] and Generative Development [35].

### 1.2.3 Challenges and Guiding Principles

One of the great challenges for realising a language-driven approach is the provision of tools for creating and relating languages. Key requirements are means for defining syntax and dealing with syntactic ambiguities. At the same time, the tools have to be highly flexible and must not unnecessarily restrict freedom of expression, e.g., by defining strict syntactic boundaries. Mainstream programming languages are not designed to be such tools. They are based on a design philosophy that strictly separates between language- and user-abstractions and, therefore, between language designer and language user.

While it is theoretically possible to introduce new languages by creating an interpreter or compiler [110], this approach is impractical when new languages are created, related and combined constantly, as in LDSE. Research on Domain-Specific Languages (DSLs) [163] has provided numerous examples of how expressive specialised computer languages with their own syntax and semantics can be and how generative techniques can be used to implement them [35]. However, the ability to create DSLs alone is not what LDSE is about. A language should not be a dead-end, but a tool that can be used to create new languages based on it [61].

Systems that are highly flexible may provide guiding principles for implementing language-driven tools. Examples are Smalltalk [59], Lisp [61] and Forth [21]. While these languages are not widely used in mainstream programming, the mere fact that after 30-50 years they are still in use and have the ability to adapt to new paradigms is a sign that they have strong means to sustain themselves [56]. The design philosophy behind these systems is to provide a small kernel of powerful abstractions and strong means of extensibility. This approach can also be applied to development environments and application software: Squeak, a Smalltalk IDE (Integrated Development Environment) written in Smalltalk, provides highly flexible means for evolution from within [85]. TEX [106], a typesetting system designed by Donald E. Knuth in the 1980s, provides a high level of typesetting quality and is widely used in academia and publishing. TEX is based on a language kernel with primitives for typesetting and it can be extended via its macro system. Besides bug fixes, the TEX kernel has been kept stable for almost 30 years. Nonetheless, the system adapted constantly via its macro system to growing demands and new technologies.

Another design principle that provides a promising model for how to implement language-driven systems can be found in the telecommunications domain. Telecommunication systems are among the largest, most reliable and scalable systems that have ever been built [71]. It is a well known fact that telecommunication systems, like computer networks, are designed in layers [69]. The protocols which are used on each layer constitute a language. This enables engineers to view the operation of a system at different levels of abstraction through different forms of expression. A key design principle is that the encodings of these languages in the form of protocols have to be conflict free [68]. Applying layering does not necessarily mean defining ever *higher* levels of abstraction. It rather ensures that communication is at the *right* level of abstraction, i.e., the level of abstraction that reflects how an engineer thinks about a system.

### 1.2.4 Concatenative Programming

Recently, concatenative programming languages have been attracting research interest because they provide a particularly small core of primitive functionality and a simple, yet powerful, extension mechanism [45, 70]. Programs are formed by concatenating smaller programs based on a set of primitive programs. The only abstraction mechanism is that of associating an atomic program with a program that is its implementation; there are no variables involved. Concatenative programming can be seen as a purely functional version of stack programming [169]. Programs denote functions and the concatenation of programs denotes the composition of the respective functions they denote. Primitive programs of a concatenative language are literals and operations that re-arrange data. The same notation is used for programs and data. The functions denoted by literals and operations map a sequence rather than individual elements.

For example, the program `dup` maps a sequence with at least one element to a copy of that sequence in which the last element of the original sequence is duplicated. Although not entirely correct with regards to the functional semantics, it is nonetheless intuitive to think of the behaviour `dup` in an imperative manner, i.e., to state that `dup` duplicates the last element in a sequence. Accordingly, the program `drop` removes the last element and `swap` re-arranges the order of the last two elements in a sequence. Literals, such as numbers and strings, append their representation to a sequence. For understanding the execution of a compound program, it is useful to think about its execution in terms of the individual programs it consists of. For instance, the result of the concatenative program `1 2 drop` can be determined by applying the functions denoted by individual program parts to an initially empty sequence in the order in which they appear in the program. The literal `1` denotes a function that appends `1` to a sequence. Given the empty sequence `[]`,

the result of `1` is `[1]`. Applying the function denoted by `2` to this result yields `[1 2]` and subsequently applying `drop` yields `[1]`.

Because programs and data have the same representation, appending the elements of the current result sequence and the remaining program always results in a valid program and applying the function denoted by that program to an empty sequence always produces the same result as applying the function denoted by the remaining program to the current result sequence. In other words, the execution state can always be represented by a program. This makes program rewriting of concatenative programs particularly easy [170]. Literals remain in place and the leftmost operation in the program is applied to the data sequence proceeding it. For example, the execution of the program `1 2 3 drop swap dup` consists of a sequence of steps that define the application of the leftmost operation to data. The first step is the application of `drop` which produces the new state `1 2 swap dup`. Next, `swap` is applied which results in state `2 1 dup`. The application of `dup` produces the final state `2 1 1`. In effect, the literals before the leftmost operation in the program serve as a stack.

Given a primitive program `*` that multiplies two numbers, a program that calculates the square of a number can be defined by concatenating the program `dup` with the program `*` to form the new program `dup *`. The square program is abstracted by the definition `square = dup *`. The program `square` is atomic, because it is not formed by concatenating other programs, i.e., it consists of a single element. It is, however, not primitive, as its behaviour can be expressed by program `dup *`. Indeed, the semantics are that every occurrence of `square` can be replaced with `dup *`. Accordingly, the execution of program `3 square` consists of the state `3 dup *`, the state `3 3 *` and the final result `9`. The basic idea is to replace all atomic, non-primitive programs with their definition, until the resulting compound program consists of primitive programs only. The resolution process entails that the order of individual programs in the program text of a compound program is transferred to the implementation level. Even after an arbitrary number of resolution steps, sub-programs on a lower-level of abstraction can directly be associated with the atomic high-level programs they implement.

In concatenative programming, *quotations* serve as both data structures and code containers. They have the same syntax as programs but are surrounded by brackets. The semantics of quotations are the same as for all data: each quotation denotes a function that appends the quotation to a sequence. For example, the execution of the program `[2 *] 3 swap` that contains a quotation as the first element has the result `3 [2 *]`. The interesting part of quotations is implemented by the program `call` that "unquotes" a quotation by removing the surrounding brackets. For example, the concatenative program `[2 *] 3 swap call` has the execution state `3 [2 *] call`, followed by `3 2 *`

and the result `6`. When combined with `call` and operations to manipulate their internal structure, quotations provide a particularly simple form of meta-programming. A Turing-complete concatenative programming system can be built on quotations, a call mechanism, an operation for testing equality and a few primitive operators for re-arranging elements such as `dup` and `swap` [99].

For this work, concatenative programming is of interest because concatenative program execution, meta-programming and abstraction can be expressed particularly well through pattern-based transformations. The reason for this is that (1) the complete state of the program execution can be represented as a program, (2) the effect of individual operations is local, i.e., operations affect data elements in their proximity in the program text, (3) the quotation mechanism provides a structural distinction between passive and active code and (4) the variable-free abstraction and resolution mechanisms can be expressed through simple substitution operations. Concat, the programming and meta-programming system that will be presented in Chapter 5, is based on a combination of concatenative programming and pattern matching.

## 1.3 Methodology: Language Layering

Language-driven software engineering (LDSE) is based on the notion of creating languages using other languages. This idea can be captured by the methodological framework of language layering. In this section and in the remainder of this work, it will be demonstrated that layering, and more generally LDSE, can be built upon a simple methodological principle: *behavioural equivalence*. The approach is based on the notion of *refinement*; the idea dates back to the late 1960s and was formalised in the early 1970s ( [77,91,121]). Refinement in its most general form is also called *behavioural refinement*. It states that a specification $\mathrm{S}_1$ is behaviourally refined by a specification $\mathrm{S}_2$ (both having the same syntactic interface) if the denotation of $\mathrm{S}_2$ implies the denotation of $\mathrm{S}_1$. The following presentation is based on the FOCUS formalism (a stream-based algebra for components) as described by Broy and Stølen [25]:

$$(\mathrm{S}_1 \rightsquigarrow \mathrm{S}_2) \Leftrightarrow (\llbracket \mathrm{S}_2 \rrbracket \Rightarrow \llbracket \mathrm{S}_1 \rrbracket)$$

In other words, $\mathrm{S}_2$ refines $\mathrm{S}_1$, if an and only if the implementation $\mathrm{S}_2$ guarantees to be a valid behavioural substitute for the implementation of $\mathrm{S}_1$. Behavioural equivalence $\mathrm{S}_1 \leftrightsquigarrow \mathrm{S}_2$ demands that

$$(\mathrm{S}_1 \leftrightsquigarrow \mathrm{S}_2) \Leftrightarrow (\mathrm{S}_1 \rightsquigarrow \mathrm{S}_2) \wedge (\mathrm{S}_2 \rightsquigarrow \mathrm{S}_1)$$

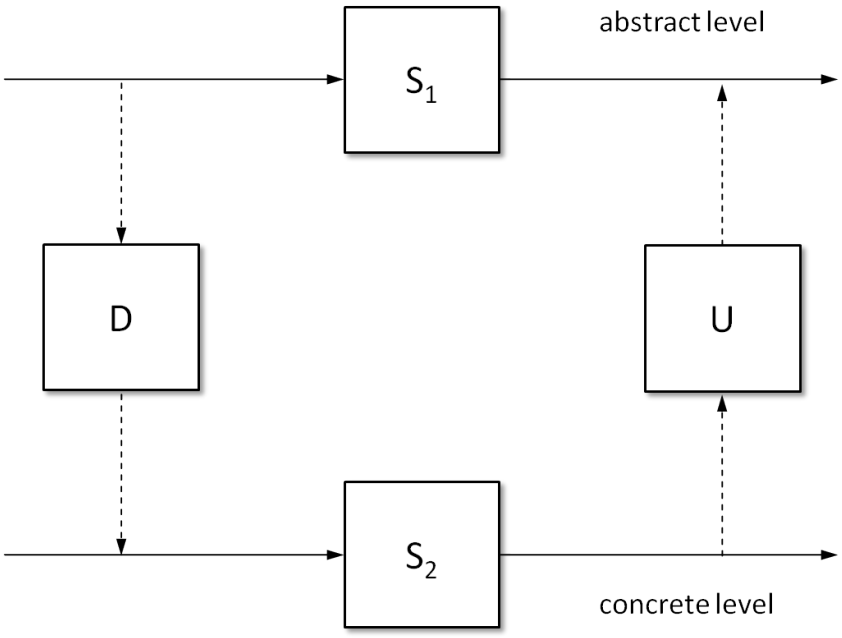


Figure 1.1: Interface Refinement as a Commuting Diagram

A more interesting form of behavioural refinement is *interface refinement*. Interface refinement generalises behavioural refinement since it allows changes to the external interfaces of the specifications. Given two more specifications, namely $\mathrm{D}$ (the so-called *downwards relation*) and $\mathrm{U}$ (*upwards relation*), interface refinement is defined as

$$\mathrm{S}_1 \rightsquigarrow \mathrm{D} \succ \mathrm{S}_2 \succ U$$

The composite specification $\mathrm{D} \succ \mathrm{S}_2 \succ U$ is a behavioural refinement of $\mathrm{S}_1$. Figure 1.1 is a graphical representation of interface refinement. The operator "$\succ$" denotes piped composition, see [25]; a more precise definition will be given below.

As is shown by Herzberg and Broy [69], a variant of interface refinement called *communication refinement* is the basis of layering as it is used as the predominant design principle in telecommunication systems and computer networks. Communication refinement constrains interface refinement by the requirement that the downwards and upwards relation in combination behave transparently. With $\mathrm{Id}$ being the specification of the identity function, it holds that

$$\mathrm{D} \succ \mathrm{U} \Leftrightarrow \mathrm{Id}$$

Engineers of distributed communicating systems have impressively demonstrated that layering is an important and fundamental design principle for the systematic and robust design of large-scale systems [68, 71]. The key insight in the context of this work is that

protocols constitute languages and that protocol layers constitute language layers. It is all about protocols (read "languages") and their interrelation. The basis of a protocol or language-driven approach is rooted in communication refinement and – less restrictively – in interface refinement.

For relating languages in programming, modelling and analysis, the primary interest is in equivalence relationships, not in refinement relationships. Protocol engineering brings with it unreliability, loss of messages and jitter – aspects which are irrelevant when a software system is viewed as a "stack" of languages. Another reason being that the key idea of abstraction and substitution in computing demands behavioural equivalence and not behavioural refinement. However, the *methodological aspect* of how to decompose behaviour and its description remains unaffected. This work reuses the compositional approach of interface refinement and communication refinement for the purpose of relating computational processes and their descriptions. Interface relations can be seen as language relations because any computational process can be seen as an interpreter of a language with regards to its syntactic interface. Therefore, a computational process is at the same time an interpreter of a description and a description being interpreted. This fundamental duality of computations needs to be reflected by language layering.

To ease the definition of language layering, the assumption will be made that both the descriptions of computational processes and the data processed by computational processes are based on a uniform representation. As described in the previous section, this is the case for concatenative languages. Let $\Sigma$ be an alphabet and let $V \subseteq \Sigma^*$, where $^*$ denotes the Kleene closure, be a vocabulary of words. Let $S$ be the set of all sequences that can be formed of words from $V$, including nested sequences up to an arbitrary depth. In the following, any description of a computational process, say $\mathrm{F}$, is given by a sequence $\mathrm{F} \in S$, with the function $f : S \to S$ denoting the computational process itself, $[\![\mathrm{F}]\!] = f$. Functions representing computational processes work on sequences as well. Concatenation of sequences is denoted by "$\succ$", which above was called "piped composition". On a descriptional level, the meaning of concatenation is given by function composition: $[\![\mathrm{F} \succ \mathrm{F}']\!] = f \circ f'$ with $[\![\mathrm{F}]\!] = f$ and $[\![\mathrm{F}']\!] = f'$.

**Definition 1** (Language Layering)**.** *Let* $\mathrm{Cmp} \in S$ *and* $\mathrm{Cmp}' \in S$ *be two descriptions of two computational processes (functions)* $cmp : S \to S$ *and* $cmp' : S \to S$ *with* $[\![\mathrm{Cmp}]\!] = cmp$ *and* $[\![\mathrm{Cmp}']\!] = cmp'$*. Furthermore, let* $\mathrm{Int} \in S$ *and* $\mathrm{Ext} \in S$ *be descriptions of two computational processes* $int : S \to S$ *and* $ext : S \to S$ *called* internalisation *and* externalisation*; it holds that* $[\![\mathrm{Int}]\!] = int$ *and* $[\![\mathrm{Ext}]\!] = ext$*. Given the following arrangement, the languages realised by* $\mathrm{Cmp}$ *and* $\mathrm{Cmp}'$ *are said to be* layered*:*

$$(\mathrm{Cmp} \leftrightsquigarrow \mathrm{Int} \succ \mathrm{Cmp}' \succ \mathrm{Ext}) \Leftrightarrow (cmp = int \circ cmp' \circ ext)$$

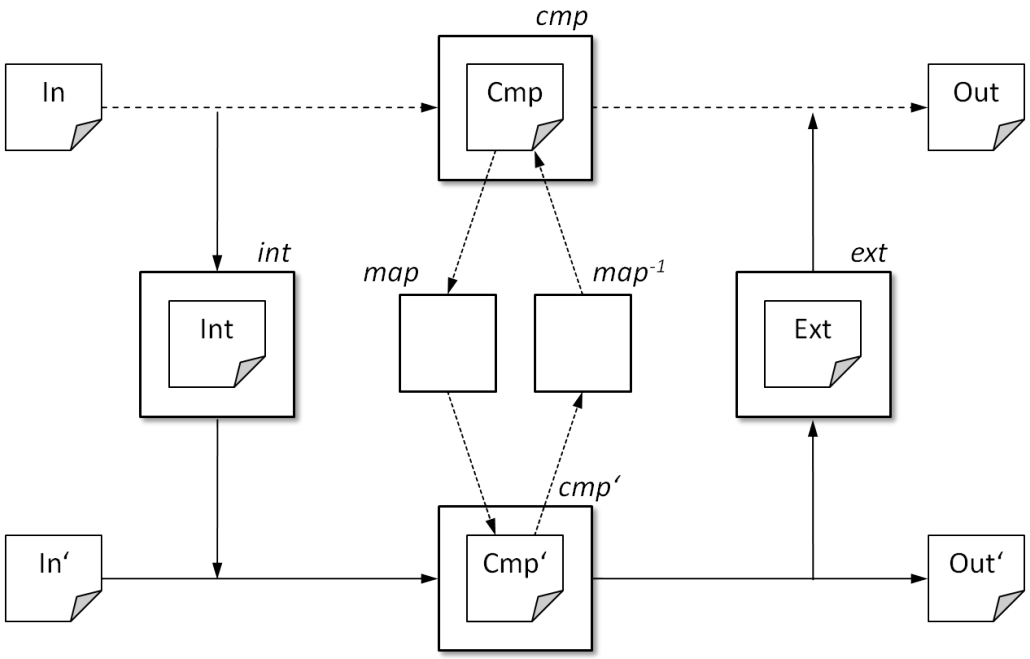


Figure 1.2: Language Layering

The description on the "upper" computational layer is represented by $\mathrm{Cmp}$ whereas $\mathrm{Cmp}'$ represents the description on the "lower" computational layer. Descriptions $\mathrm{Int}$ and $\mathrm{Ext}$ describe layer adaptions, an internalisation process and an externalisation process.

Similarly to communication refinement, language layering can be restricted by the requirement that $int \circ ext = id$ ($id$ stands for the *identity function*). In this case, internalisation and externalisation define a bidirectional encoding relation called *view*.

The relation between $\mathrm{Cmp}$ and $\mathrm{Cmp}'$ in language layering can also be defined by two relations $map$ and $map^{-1}$ with $\mathrm{Cmp} \circ map = \mathrm{Cmp}'$ and $\mathrm{Cmp}' \circ map^{-1} = \mathrm{Cmp}$, thus $map \circ map^{-1} = id$. The mappings $map$ and $map^{-1}$ can be described by corresponding descriptions. It is possible to set up language layering in such a way that $map = int$ and $map^{-1} = ext$.

The principle of language layering is pictured in Figure 1.2. Boxes represent computational processes, document icons represent descriptions. A document icon inside a box represents a description of a computational process. Language layering can be recursively applied to any box in Figure 1.2.

Language layering provides the methodical framework for language-driven software engineering. It will be investigated further in the context of pattern-based computing (Section 4.5), language engineering (Section 5.4), modelling (Section 6.1 ) and system analysis (Section 7.3).

## 1.4 Hypothesis and Approach

This work is based on two foundational approaches. The goal of Language-Driven Software Engineering is to control complexity by making the theoretical potential of syntactic and semantic abstraction a practical reality in software engineering. Pattern-based computing aims to define a unified foundation for LDSE by focusing on the *formal* nature of computing [120]: a computing system processes symbols and through an interface it implies a model of computation by means of syntactic encodings of symbolic structures and operational input/output relations between symbolic structures. Patterns are natural tools to define, recognise, (de-)construct, transform and search symbolic structures [9].

**Hypothesis** *The systematic creation and layering of languages can be reduced to the elementary operations of pattern matching and instantiation. This pattern-based approach provides a formal and practical foundation for language-driven modelling, programming and analysis.*

This hypothesis is impossible to prove in general as there is no accepted consensus of what a language-driven foundation for the problem domains should be. What can be done, however, is (1) to assure that the pattern foundation is solid, (2) to support the hypothesis through the application of the foundation to the three problem domains and (3) to critically evaluate the results. The soundness of the pattern basis is assured by defining precise operational semantics for pattern matching and instantiation. The application part of the research consists of the implementation of three tools that support the language-driven approach. The evaluation is performed based on a range of examples to which these tools are applied.

The research approach taken can be summarised as follows:

1. Analysis of the problem domain and hypothesis building (Chapter 1)
2. Literature study (Chapter 2)
3. Formalisation of a framework that can support the hypothesis (Chapters 3 and 4)
4. Application of the approach to different domains (Chapters 5, 6, 7)
5. Evaluation of the results (Chapter 8)

## 1.5 Scientific Contribution and Novelty

In the last few years, several approaches have emerged that can be considered language-driven, including Generative Programming [34], Model-Driven Engineering [151] and Domain-Specific Languages [55]. The novelty of the approach presented in this thesis is to provide a foundation for language-driven engineering based on patterns and language layering. The ensuing chapters demonstrate the applicability of this approach to core aspects of software engineering: modelling, programming and analysis. The concrete outcomes of this work are three tools that are novel in themselves. All three tools are based on the formal foundation of patterns and the methodology of language layering:

**Concat** A self-sustaining, language-driven programming system based on tagged structures and a mechanism for hiding these structures behind a user-definable syntactic interface. All aspects of the system – parsing its own character-representation into an internal structure, program execution, meta-programming, rendering of output – is based on pattern matching and instantiation and on language layering. This includes not only the program-level but also the meta-level: Concat is defined using itself by means of a meta-circular implementation and thus is its own engine of evolution.

**XMF (XML Modeling Framework)** A web-based modelling- and meta-modelling framework that demonstrates how language-driven concepts can be applied to modelling and tooling. The framework uses patterns to define modelling languages, relate different models, create model instances, query models and to implement constraints. Bidirectional transformations map between models and their external representation. This view mechanism provides visual means that can be used to create, edit and display models.

**CFR (Channel Filter Rule Language)** The realisation of a language-driven approach for the layered analysis of communication behaviour in complex networked systems. The system extends traditional protocol analysis in order to capture communication at the level of abstraction that reflects application design. This is achieved by abstracting communication patterns on an existing protocol layer as messages on a new abstract protocol layer. The abstract protocol provides its own language for describing the communication of the system. This technique can be used to specify, monitor and test complex communication scenarios.

**Theoretical Contribution** In addition to these three tools, the theoretical contribution of the work comprises:

- A unifying foundation for language-driven software engineering through patterns
- A formal definition of operations for matching and instantiating (meta-)patterns in horizontal, vertical and diagonal manner
- A novel way of defining path polymorphic operations through pattern operations
- A pattern-based approach for defining execution models using structural and temporal views
- A pattern-based meta-architecture for modelling and meta-modelling
- Application of a single formalism to define, transform, query, constrain and relate models
- A methodological approach for layering languages that guided the design process for the above mentioned applications
- A novel approach for the specification of analysers for recovering communication design intentions and communication scenarios

With a mathematically defined set of primitive patterns and pattern combinators, as well as a range of tools and techniques based on them, the work provides theoretical and practical contributions that are relevant for research on self-sustaining systems, (meta-)programming, (meta-)modelling and analysing complex systems.

## 1.6 Relevant Publications

The language layering techniques underlying this thesis and the protocol re-engineering approach of Chapter 7 are based on the author's research on telecommunication systems that was published as a book chapter in the *Wiley Encyclopedia of Computer Science and Engineering* [71]. The research on concatenative programming and a purely concatenative version of Concat are described in a paper that was presented at the *International Conference on Software and Data Technologies (ICSOFT)* [70]. Two publications describe CFR and its application in the automotive domain: a full paper presented at the *International Conference on Software Engineering (ICSE)* [144] and a short paper presented at the *Software Engineering (SE)* conference [142]. XMF, its underlying pattern language and its utilisation for teaching language-driven software engineering are detailed in a paper presented at the *International Conference of Education, Research and Innovation (ICERI)* [143]. The meta architecture underlying XMF is formalised in an article that was published in a research report of Heilbronn University [72].

## 1.7 Overview

This section provides a short overview of the chapters that make up the rest of this thesis.

**Chapter 2** The work presented in this thesis is related and compared to similar work by other researchers. The chapter surveys approaches to software engineering that can be seen as language-driven. It compares this work to other work on pattern matching and related subjects such as recognition systems. Concat is related to other self-sustaining systems, metacircular interpreters and highly flexible programming languages. The pattern language underlying XMF is compared with schema, transformation and query languages and the approach underlying CFR is related to work on forward engineering protocols.

**Chapter 3** The chapter is the formal foundation of the thesis. It defines operational semantics of the core pattern formalism underlying Concat, CFR and XMF. Pattern matching is defined in a unified manner on typed sequences. Horizontal combination of patterns includes sequencing, repetition and choice. Vertical combination defines staged application of patterns. Diagonal combination enables the combination of patterns and results to provide stack-like matching semantics. Transformations are based on the combination of matching and instantiation. Searching, querying and collectively transforming structures can be expressed by path polymorphic operators. A quasiquotation mechanism allows arbitrary meta-levels and is the foundation for pattern abstraction and self-referential definition of the pattern system.

**Chapter 4** This chapter connects the core pattern approach to its application for programming. It defines rewriting as a composition of transformations and patterns that express a strategy for applying transformations. Restrictions to rewriting systems are discussed that produce a concatenative system. The notion of pattern-based computing is formalised based on rewriting. Conditional rules are defined as transformations that can initiate computational processes. Structural and temporal views define mechanisms to hide internal representations and certain computational steps. Grammars are defined as meta-transformations on a unified representation of strings and trees. Elliptical patterns are a practical extension for programs that manipulate nested structures.

**Chapter 5** Concat and its programming and meta-programming features are introduced as a practical application of the pattern approach. Concat is a highly-extensible framework for creating and relating software languages. Concat defines a program-level and a meta-level and allows arbitrary extensions of both levels. Core Concat provides operations,

productions and macros for defining computation, parsing and program transformation. New computational concepts can be introduced by applying the meta-language not only to the program-level, but also to itself.

**Chapter 6** XMF, a tool and tool infrastructure for modelling and meta-modelling based on patterns is introduced. XMF applies the ideas of Concat to XML and Web-based technologies. Partial instantiation is utilised to query and constrain models. Class and object models are introduced as examples and a mechanism based on HTML that provides visual means of creating, editing and viewing models is discussed.

**Chapter 7** The chapter presents a language-driven approach for the analysis of communication behaviour in automotive networks. Roots of complexity and challenges in automotive applications are introduced. The presented approach is based on filtering to find dispersed communication patterns and on the abstraction of these patterns. Abstract protocols are used to define new language layers. A domain-specific language is presented that implements the approach and it is discussed how the language can be formalised with patterns.

**Chapter 8** The final chapter evaluates the theoretical and practical results of the work and shows directions for future research.

# Chapter 2

# Related Work

This chapter relates and compares the research described in this thesis to existing research. Pattern matching has been a subject of study in many sub-fields of computing (Section 2.1) and is the basis of several program transformation systems (Section 2.2). Recently, patterns have been investigated as a unifying framework for programming (Section 2.3). The pattern approach presented in this work is based on a uniform representation for programs and data, a feature found in homoiconic languages (Section 2.4). The uniform representation makes it possible to naturally express various structural transformations. These includes parsing which gives pattern an interpretation as grammars that define recognisers (Section 2.5). The goal is to utilise patterns for language engineering. Several approaches have emerged in recent years that view language creation as the driving force of software development (Section 2.6). Concat is a pattern-based implementation of a language-driven system inspired by concatenative languages (Section 2.7). Its extensibility mechanisms supports change from within, which make it a self-sustaining system (Section 2.8). XMF is based on the XPLT language which is at the same time schema, transformation and query languages for XML (Section 2.9). The abstract protocols in CFR and their bottom-up definition describe a protocol-re-engineering approach based on layering (Section 2.10)

## 2.1 Pattern Matching

Matching and instantiating patterns is certainly not a new idea. It underlies the replacement rules in rewriting systems [12, 16, 46] and has been extensively studied in computer science for various data structures, including strings [107], trees [78] and objects [48]. Algebraic datatypes lend themselves naturally to pattern matching because values are described as terms built from data constructors [93, 109]. In fact, pattern-matching on algebraic data types is a common technique in functional programming languages [108] where it is sometimes referred to as "ML-style pattern matching" because of its importance in the ML programming language [122].

For instance, in ML a type for possibly empty binary trees with integers as leafs can be defined as follows:

```
datatype itree =
  Empty | leaf of int | node of itree * itree;
```

The 0-ary constructors `Empty` creates an empty tree. The unary constructor `leaf` creates a leaf node and the binary constructor `node` creates a node with two children. For example, the term `node (node (leaf 2, leaf 3), leaf 4)` represents the tree in Figure 2.1.

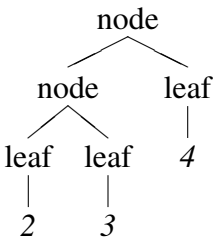


Figure 2.1: Binary Tree Example

Pattern matching is defined on the nested structure of compound values. The type constructors can be used to distinguish cases when defining functions. For instance, the following function definition adds up the numbers at the leafs of a tree recursively:

```
fun addup Empty = 0
  | addup (leaf n) = n
  | addup (node (n, m)) = addup n + addup m;
```

The three cases of the function are defined by patterns on the left-hand side. The patterns make use of the three data constructors as well as variables `n` and `m` in order to extract parts of the compound value. The resulting variable bindings are used in computations on the right-hand side.

Concat's operation definitions are similar to ML function definitions on tuples of algebraic types. Nevertheless, Concat provides a more expressive pattern language and implements operator definitions as a syntactic layer on rewriting. The view mechanism described in this work can be seen as an application of the ideas outlined by Wadler [175] to layered computation and syntactic extension.

Logic programming languages such as Prolog [153] use patterns (terms with variables) to implement a procedural semantics for derivation. The conditional rules of Concat are similar to productions in Prolog, but provide a functional rather than relational semantics and define the evaluation of conditionals as invocations of a rewriting system. Macro systems in various languages are based on patterns, either on strings, as in C preprocessor macros [100], or on structured representations as in Lisp [61]. The hygienic macro system of Scheme provides an elliptical pattern operator that serves as a foundation for intelligently transforming nested structures containing repeated occurrences of patterns [44]. The operator ° (see Section 4.6) implements elliptical pattern matching. It is the basis of structural transformations in XMF.

Various Lisp dialects provide a quasiquotation mechanism to construct lists from templates. A quasiquote expression describes a list as a mix of static structures (quoted parts) and dynamic computations (unquoted parts). For example, the following S-Expression uses the backquote to declare the structure as a template and two commas for unquoting:

```
`(node (leaf ,n) (leaf ,(+ n 1)))
```

If this S-Expression is evaluated, only the two parts that are unquoted by the commas are executed while the surrounding parts of the expression are treated as list structures. During evaluation, the unquoted parts of the expression are replaced with the results of the respective computations. For instance, evaluation of the example in an environment where `n` is bound to `2` yields the result `(node (leaf 2) (leaf 3))`.

The quasiquotation mechanism defined in the pattern core and utilised in Concat is inspired by that of Scheme [98], but provides functionality for instantiation *and* matching. It is utilised not only to create patterns using meta-patterns, but also to match the internal structure of patterns – including arbitrary quasiquoted and unquoted patterns.

Attempts have been made to integrate pattern matching into object oriented programming languages that do not provide built-in pattern support [167]. The recently developed Scala programming language incorporates a match statement that allows prioritised matching on objects and types [127]. Concat follows a different approach that does not start with a language and integrates pattern matching, but starts with pattern matching and builds a language with its own syntax and semantics on top of it. The language designer controls how much of its pattern-based foundation is exposed by the language and which

parts are replaced with a direct semantics [101].

## 2.2 Program Transformation Systems

Program transformation systems define the manipulation of programs as data [10,74,166]. Transformations are typically specified by a set of rules. ASF+SDF [41] is a generic toolset for defining programming language manipulation tools such as parsers, transformers, and analysers based on the Syntax Definition Formalism (SDF) [66]. SDF is also used by the Stratego/XT [165] framework for specifying parsers. A separate language in Stratego defines pattern-based rewriting rules and strategies for application of these rules. Concat on the other hand defines a single pattern formalism for parsing, rewriting and defining rewriting strategies. This provides a more elementary and yet highly flexible basis for programming and meta-programming.

Concat allows the extension of its pattern-language to include concrete syntax of programs with which these patterns are matched. Sellink and Verhoef call patterns that are based on the concrete syntax of the data they transform "native patterns" and describe an approach that generates such patterns from context free grammars [147]. Similar meta-language extensibility is also supported by Stratego/XT and the TXL [32] source transformation language. Concat goes a step further systems in that it allows not only the use of literal syntax in patterns and the definition of patterns based on literal syntax, but a complete replacement of the meta-language syntax.

OMeta is an object-oriented language for pattern matching [180] designed for experimentation with programming languages [178]. It generalises Parsing Expression Grammar (see Section 2.5) so that patterns can operate on list structures and defines a rule-based language for transformations. In addition to that, it implements memoization and directly supports left-recursion [179]. Concat and the pattern formalism presented in this work provide similar operations to that in OMeta. However, details of the core semantics, especially result combination in horizontal matching are different. More profoundly, this work is based on typed sequences while OMeta is based on S-Expression-like structures. The formalism in this work is based on matching and instantiation and transformations are pattern expressions. In OMeta, transformations are implemented using semantic actions defined in a host language. For example, the following OMeta parser defined by Warth [178] binds the results of individual parsers `exp` and `fac` to variables `x` and `y`; the

bindings are then used in semantic actions to create JavaScript arrrays:

```
exp = exp:x '+' fac:y -> ['add', x, y]
    | exp:x '-' fac:y -> ['sub', x, y]
    | fac
```

OMeta's goal of language experimentation is similar to that of Concat, but OMeta relies primarily on a transformational approach, where a language is transformed into a target language for which an implementation exists. Concat on the other hand attempts to define a self-contained computational framework, which is reflected in the design of the core pattern formalism with its vertical and diagonal operators.

In Context-Oriented Programming (COP) [76], parametric method dispatch is generalised to arbitrary contexts so that code executed may depend on arbitrary system state. In Concat the range of computational concepts can be the entire program state, which makes Concat a natural choice for COP. In Core Concat, concepts such as operations and macros are implemented by separating transformation rules into an operational part (rules) and a contextual part (operator symbol, start- and end-tags) in the user interface.

## 2.3 Pattern Calculus

While patterns have been studied in many different sub-fields, there are few approaches that attempt to define a foundation for practical computing on patterns. The most comprehensive work in this direction is that of Barry [9]. It is an attempt to provide a unifying framework for programming through pattern matching according to the principle "computation is pattern matching". The work is based on a set of pattern calculi and a programming language named *bondi* based on it. Patterns are defined to match on arbitrary data structure and *path polymorphism* supports traversal of these data structure for searching or querying. *Pattern polymorphism* allows free variables in patterns for dynamically assembling and evaluating them.

All these techniques are implemented in the pattern system presented in this work as operators on typed sequences. Hierarchical matching defines pattern operations on the type and content component of typed sequences which are a uniform data representation for strings and trees. Traversal of typed sequences, as defined by path polymorphism, is expressed by the $find$, $findall$ and $find\ in$ operators that allow fine-grained control over the type of data items that contain or surround the searched data. Pattern polymorphism is implemented by partial instantiation and used extensively in XMF to modularise schemas, querying data and defining constraints. While the pattern calculus seeks a foundation for

programming by providing means to express programming paradigms such as OO and functional programming using patterns, the focus of this thesis is to provide a foundation for language-driven engineering. Therefore, the focus is on means for creating syntax and defining systems in layers of languages.

## 2.4 Uniform and Homoiconic Languages

Typed sequences and their notation define a uniform syntax for representing programs and data. This greatly simplifies the specification of structural patterns. Bachrach and Playford use the term "Skeleton Syntax Tree Representations" for such uniform syntaxes for programs [6]. Uniform representations are the basis of homoiconic languages, i.e., languages that use the same notation for programs and data and thus provide means for program manipulation [95, 116]. S-Expression are primarily used in Lisp [114] and Scheme [98]. In contrast to typed sequences, S-Expressions specify untyped lists structures and use encoding *conventions* to express typing.

For example, in the S-Expression `'(node (leaf 2) (leaf 3))`, the symbols `node` and `leaf` are used in a prefix position to indicate the type of data. However, this is merely a convention, i.e. the "special" meaning of the first element in the list is defined by the functions interpreting the data. Therefore, a postfix syntax such as `((2 leaf) (3 leaf) node)` would be conceivable if the functions operating on the data follow that schema. Similarly, in concatenative languages, words and quotation provide a nested sequence representation for programs and data.

Compound terms in Prolog consist of a functor and a fixed number of arguments that are atomic terms or compound terms. The functor can be used to encode the type of data. In this respect, terms in Prolog define compound values in a structurally similar way to algebraic datatypes. For example, the tree in Figure 2.1 can be represented by the term `node(node(leaf(2), leaf(3)), leaf(4))`. The use of lists as arguments allows the representation of arbitrary length data, as is exemplarily shown by the term `leaves([leaf(1),leaf(2),leaf(3)])`. Therefore, a Prolog term with a single list argument closely corresponds to a typed sequence. XML (Extensible Markup Language) defines a structured representation that, at its core, is based on a mix of element tags and text in between. A subset of XML with elements that do not contain attributes corresponds very closely to typed sequences (see Section 3.1).

The advantage of uniform syntaxes is a syntactic layer that is shared by all languages based on the uniform syntax that can be used for structural manipulation. The drawback is the resulting lack of syntactic freedom. Common Lisp *reader macros* provide a controlled way to break out of the syntactic restrictions by defining Lisp functions that transform

character data into S-Expressions during parsing [111]. For example, as defined by Graham [61], the following reader macro allows to write quoatations using a quote character as prefix instead of using the quote macro explicitly.

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list 'quote (read stream t nil t))))
```

The above reader macro defines a function that is called by the Lisp reader whenever the quote character appears in the stream. This function then calls `read` on the character stream without the quote character and wraps the result in a list structure. This list structure represents code that explicitly makes use of the quote macro. For example, reading `'x` with the reader macro activated yields the result `(quote x)`.

The Factor programming language [45] provides a similar mechanism based on *parse words*. A parse word is a stack-based function that can explicitly manipulate program text. It gains control over the parser when a certain trigger is detected in the program text. The example

```
SYNTAX: ADD
  scan-word suffix!
  scan-word suffix!
  \ + suffix! ;
```

defines a parse word `ADD` that allows a simple form of prefix notation to be used for addition instead of the standard postfix notation. Whenever the substring `ADD` is encountered in the character stream the general parser calls the parse word which then reads in the next two words, adds them to the end of the parse result using the `suffix!` word and finally also adds the symbol `+`. This means that the program `ADD 2 3` is transformed into `2 3 +` during parsing.

The view mechanism in Concat provides a more comprehensive approach that hides an internal representation behind a syntactic interface by defining transformations for internalisation and externalisation of programs and meta-programs. In Concat, even basic literals and the notations for lists are not hard-coded into the parser, but defined by the view mechanism.

## 2.5 Recognition Systems

This work defines a pattern-based foundation for various manipulation tasks for structures of which character strings are one kind. When interpreted as grammars, pattern

expressions define unambiguous parsers for strings. Hence, this work is related to parsing techniques and grammar formalisms in general and *analytical* or *recognition-based* formalisms and parsing techniques in particular.

Regular expressions [158] recognise strings based on operators that express concatenation, repetition and alternation. For example, the expression `a(b|c)*` matches all strings starting with `a` followed by an arbitrary number of `b`s and `c`s. Parsing Expression Grammars (PEGs) [52] can be regarded as an extension of regular expressions to a more expressive parsing language. Prioritised choice (expressed using the `/`-operator) and greedy repetition make PEG grammars unambiguous. For example, the PEG grammar fragment `Exp <- ('ab' / 'a') Rest` clearly defines a parsing strategy for a string such as "abc". Even though both choices `'ab'` and `'a'` match the beginning of the string, it is the lefttmost choice `'ab'` that is always tried first. In case `'ab'` succeeds `'a'` will never be tried – not even if `Rest` fails.

The unambiguity of PEGs make the formalism especially suitable for defining analytical (rather than generative) grammars [64]. The theory of PEG parsing is based on the parsing algorithms described by Birman and Ullman [14]. First practical implementations of PEG-like operators go back at least to Schorre's META II system in which the basic operations of PEGs are compiled into recursive-descent parsers [146].

Restricting the pattern core of Chapter 3 to horizontal operators, atoms, negation and references to define production rules yields the core PEG formalism. The use of a PEG-style grammar formalism to parse not only strings, but also nested sequences, is described by Baker [8]: the META II approach is used to parse S-Expression. Concat applies this approach to typed sequences. Because characters are also encoded as typed sequences, there is no conceptual difference between parsing and structural pattern matching.

A similar unification of parsing strings and pattern matching on nested structures can be realised with Definite Clause Grammars (DCGs) [19]. DCGs are a grammar formalism built into Prolog and other logic programming languages [150]. DCGs are syntactic sugar for regular Prolog rules. For example, the DCG grammar

```
as --> [a].
as --> [a], as.
```

recognises atomic sequences of `as`, such as `[a]` or `[a,a,a]`. It is equivalent to the following standard Prolog definition:

```
as([a|R],R).
as([a|Ri],Ro) :- as(Ri,Ro).
```

In contrast to PEGs, DCGs undo choices and try alternatives in case parsing fails. For instance, by forcing Prolog to backtrack, a query `as([a,a,a],R)` produces three different result values `R`: the sequences `[a,a]`, `[a]` and the empty sequence `[]`.

DCGs can be used to parse strings, which in Prolog are lists of atoms, and structured terms. DCGs do not provide syntactic meta-operators for repetition or alternation. However, those can be implemented using Prologs abstraction mechanism [11]. In contrast to PEGs and the pattern expressions presented in this work, DCGs support full backtracking and ambiguous grammars.

Parser combinators are a technique for implementing top-down parsers in functional languages [38, 82, 176]. Parsers are implemented as functions and larger parsers are built from smaller parsers by functional composition using higher-order functions [83]. For example, the following Scheme code defines a parser `a-or-b` that recognises either token `a` or token `b`.

```
(define a-or-b
  (alternate
   (token "a")
   (token "b")))
```

A parser is a function that when applied to a string yields a pair containing the parse result and the rest of the string. The function `token` yields such a parser that recognises a single token. For example, the application `((token "a") "abc")` yields `("a" . "bc")`. The higher-order function (parser combinator) `alternate` combines two parsers. The result is a parser that attempts the first parser and in case of success yields its result, otherwise it attempts the second parser and yields its result.

Because `a-or-b` is a parser, it can again be combined using parser combinators. Parser combinators typically implement basic features such as sequencing, repetition or look-ahead but might provide more advanced functionality. An important feature of most combinator libraries is to give users control over the combination of parse results. This typically involves host-language interaction – at least on the level of result data types. For example, in a Lisp parser combinator library parse trees are represented by list structures and the user must have a way to express how results of individual parsers are arranged in the list structure.

While the sematincs of the operators presented in this work can be implemented using combinatory techniques the goal is to define an extensible formalism with stand-alone syntax, semantics and meta-functionality. This goes beyond what parser combinator libraries are used for. The *into* combinator defined by Hutton [82] allows results of previous parsers to be passed on to following parsers as arguments. Result passing is implemented

by the vertical operator presented in this work. However, the vertical operator passes the result of a match to the following pattern as the input sequence. The passed result is thus treated in the same way as any other parsing input. This allows the vertical combination of parser that were not specifically designed for receiving parameters.

One of the core features of Concat is a unfication of all stages of program execution through pattern-based transformations. Piumatra uses a pattern-based transformation language is used to define front-, middle- and backend of a simple compiler that generates machine code for a Lisp-like programming language [137]. In Concat such a compiler could be realised using internalisation macros and computation macros with typed sequences being the representation for program text, abstract syntax trees and machine code.

## 2.6 Language-Driven Approaches

Several modern approaches to software development are language-driven in the sense that they focus on the importance of languages and their systematic creation and use. Favre [49, 50] proposes two new fields of research for the systematic study and engineering of languages: Software Linguistics and Software Language Engineering. Generative Programming [34, 35] divides the software development process into two stages: domain engineering and application engineering. Languages are created during domain engineering and used during application engineering. Software Factories [62] is a generative approach where languages are used to describe the variations in product families. The approach is neutral with respect to methodology and process and can be seen as a contribution to these areas.

Model Driven Development [104,151] aims at generating executable code by the stepwise transformation of models that are typically expressed using visual languages. Model Driven Architecture (MDA) can be seen as a particular realisation of Model Driven Development around a set of technologies that were developed in the context of the Unified Modelling Language (UML) [13], in particular Meta Object Facility (MOF) [129]. Clark et al. propose meta-modelling as a foundation for language driven development and discuss syntax and semantics based on meta-modelling [30].

Several model-driven tools exist that generate code for the .NET and Java platforms [26]. Intentional Software [149] focuses on technologies for creating and integrating languages with natural notations, so do several other approaches to language extension [182]. Tools for language prototyping focus on generating programming language implementations from higher-level specifications [41]. Language workbenches [43, 94, 105] are tools for systematically creating languages and development environments.

Many of the existing tools and frameworks mentioned above are rather large and complex. One aim of the work presented in this thesis is to provide simpler, more lightweight tooling based on an extensible core of pattern functionality. This is eminent in all three tools that will be presented in the following chapters. Based on a core of pattern functionality, Concat provides means for parsing, programming, meta-programming and is highly extensible. Similarly, XMF uses a small pattern core for defining schemas, transformations, querying, views and even constraints. In CFR, analysis and scenario specification is based on just three concepts derived from the pattern approach.

The STEPS project [96] undertaken by Kay et al. takes the question of how large software systems need to be literally. The goal is to build a complete personal computer operating system with applications in under 20.000 lines of source code. The systematic use of languages and tools for creating and relating languages is the fundamental approach that underlies this endeavour.

Concat and CFR (Channel Filter Rule Language) provide means to build or analyse systems in layers of languages. The notion of language layering is propagated in several other works. Graham advocates a *bottom-up* approach in which new languages are created incrementally inside a host-language using existing languages – with the user interface of the system forming the topmost language layer [61]. Language-oriented programming [177] on the other-hand propagates a "middle-out" approach where development starts with the definition of a domain-specific language that forms the middle-layer that links an application with an implementation language.

Domain Specific Languages (DSLs) [55] are a topic of growing research interest and questions such as how to create, maintain, and integrate languages are studied in the context of DSLs. In the Lisp community, the idea of developing specialised languages that are embedded in Lisp is widespread [61]. The use of DSLs has a tradition in the Unix community where they are called "little languages" [145]. A discussion of popular DSLs available under Unix is given by Raymond [141]. There is a plethora of literature that describes the implementation and use of DSLs, an annotated Bibliography is provided by Deursen et al. [163]. Much of the research on DSLs is concerned with how to build DSLs using specific technologies, e.g., specialised parsing techniques and frameworks [20, 133, 134], code generation [67] or the meta-programming facilities of specific languages [37, 47, 103, 160]. Less work exists on techniques for creating and relating DSLs. Methodologies for developing DSLs are proposed in [117] and [113]. The former includes a broad study of DSLs, the latter focuses on formal methods for creating DSLs. Concat and XMF are tools that can be used for creating domain specific programming, specification and modelling languages.

## 2.7 Concatenative Programming

The execution mechanism of Core Concat restricts rewriting in such a way that it conforms to a concatenative programming language. Concatenative languages can be regarded as a purely functional version of stack languages. The interesting aspect of concatenative languages for this work is that these languages provide a simple computational foundation that – unlike lambda calculus – is not based on the concept of variables and their substitution. The simple structure and the non-existence of variables makes transformations of concatenative programs particularly simple [70].

Much of the foundational work on concatenative languages was performed by Manfred von Thun in conjunction with the development of the Joy programming language [169]. Cat [42] is a concatenative language that – unlike Joy – supports static type checking. Factor [45] is a programming language designed for use in practice. It has a concatenative core, but also supports object-oriented programming and a macro system.

Concatenative languages are closely related to stack languages. The former are characterised by the homomorphic relationship between programs and functions, the latter by the use of a stack as the central concept in the execution model. A language may be both stack-based and concatenative, but this must not necessarily be the case. Forth [140] and PostScript [2] are popular "high-level" stack-based languages that are not concatenative. Several assembly and intermediate languages also use a stack-based model of execution.

In a concatenative language, even those words that may intuitively be perceived as data, for example numbers and strings, denote functions. Thus, concatenative languages are not only functional in the sense that functions have no side effects, but also in the sense that "everything is a function". This form of purity and the non-existence of variables relates them closely to function-level programming as defined by Backus [7] and the point-free style of functional programming [58].

## 2.8 Self-Sustaining Systems

Concat provides means to evolve its syntax and semantics. In the extreme case this allows Concat to replace itself by its own means. Systems that enable such change from within are studied as *self-sustaining systems* [56]. Examples include classic programming languages with small kernels and strong extension mechanisms such as Squeak/Smalltalk [59,85], Klein/Self [161,162] and Lisp [115], self-implementations of scripting languages such as PyPy (Python) [15] and highly flexible frameworks for creating language paradigms such as the Combined Object Lambda Architecture (COLA) [136].

Metacircular definitions of programming languages are common in the Lisp family

of languages [115] and discussed extensively by Abelson and Sussman [1]. Meta Object Protocols (MOPs) [103] provide means to extend and overwrite *selected aspectes* of the semantics of a programming language by implementing objects and methods. This can be seen as a more gradual form of language evolution compared to metacircular interpreters. The ability to introduce new syntax and semantics of *individual* pattern operations in Concat is in effect a meta-pattern protocol.

## 2.9 XML Validation and Manipulation

The XML Pattern Language for Transformations (XPLT) which was developed as a basis for XMF can be seen as a schema language that defines a set of valid XML documents. Popular schema languages for XML include Document Type Definition (DTD), XML Schema and RELAX NG (REgular LAnguage for XML Next Generation), see [126] for a taxonomy based on formal language theory. Relax NG is closest to XPLT. It is based on regular expression operators that are represented as elements and mixed with regular elemenents to form patterns. Like XPLT, it also provides a reference mechanism for defining schemas in a modular way [168].

Nevertheless, XPLT is more than a schema language. Using variables, it defines schemas in such a way that given two schemas that follow certain name conventions, (bi-directional) transformations can be derived. XSLT [172] is the W3C standard language for transformation of XML documents. Its syntax is like that of XPLT based on XML, but transformations are described explicitly in a procedural manner using constructs such as loops and branching. Closer to XPLT are transformation languages based on grammar formalisms. Huang et al. [81] use the pattern calculus-based *bondi* programming language for transforming XML. Pierce formalises pattern matching on XML structures based on regular expression operators [80].

XPLT provides a declarative formalism for defining bidirectional transformations even over recurring patterns. However, because of the expressiveness of the formalism, bidirectional transformations cannot be guaranteed. Foster et al. present a formalism that can only express transformations for which bidirectional transformation can be guaranteed [53]. It would be interesting further research to investigate if a subset of XPLT exists for which bi-directional transformations can be guaranteed or at least checked statically.

The query facility of XPLT relates it also to query languages. XQuery (XML Query) [174] provides advanced querying capabilities based on path expressions and predicates. XPLT on the other hand uses pattern refinement to query and search documents for instances of a pattern. More advanced mechanisms of XQuery include result ordering or predicates such as selecting all nodes with a value smaller than a threshold. Such features

are currently not implemented in the pattern core of XPLT but can be expressed using the provided JavaScript library. For example, by implementing complex predicates as functions. Overall, XPLT does not claim to provide all the features of specialised, feature-rich schema, transformation and query languages. Instead, it provides a single pattern language based on a small core and JavaScript as an extension mechanism. The advantage of using XPLT instead of separate language for schemas, transformations and querying is not only its simplicity, but the amount of reuse possible: a single pattern can be used for schema validation, as a source or target for a transformation, in a query or as a constraint.

## 2.10 Layered Analysis and Protocol Re-Engineering

The basic idea of the CFR language is to extend protocol analysis to the abstract layers of a communication system so that communication scenarios can be described in a specialised language that bridges heterogeneity gaps. This approach can be related to the large body of work done in *forward engineering* and analysing protocols. Indeed, protocol analysers are a standard technique for monitoring, testing and reverse engineering systems and many academic and commercial implementations are available [181], e.g., Ethereal [132]. The aim of the CFR approach is not to implement another protocol analyser that understands a set of predefined protocols. Instead, the aim is to provide an elegant specification mechanism that makes protocol analysers programmable and extends their scope to include abstract protocols and scenarios – not only standard protocols.

Extending a protocol analyser with features to capture abstract protocols is essentially a reverse engineering activity. It is, in principle, the reverse of approaches such as OSI [87]. The attempt to use just three basic concepts to reverse engineer protocol-based systems is closely related to approaches in forward engineering protocols, protocol stacks and protocol-based services that aim to base formalisms on a few basic concepts. For instance, Herzberg and Broy [69] provide a formal approach to modelling layered distributed communication systems with a small number of concepts only.

## 2.11 Conclusions

This thesis brings together two topics that are of current interest in computer science research: attempts to define a pattern-based foundation for programming (“computing is pattern matching”) and language-driven engineering techniques (“computing is creating languages”). The approach draws on experience from and contributes to disparate areas of research, including analytical grammars, self sustaining systems, program transformation, concatenative programming and layering.

# Chapter 3

# Pattern Core

This chapter lays the foundation for the rest of the work. It formally defines a core of application-neutral pattern functionality for recognising, deconstructing, creating, searching and generally manipulating data structures. The approach is based on composition which applies to both the way data structures and patterns are defined. Typed sequences compose data to form compound data, starting with the elementary atom type. Operators compose patterns to form compound patterns, starting with elementary patterns (Section 3.1). Patterns are interpreted in two fundamental ways: as recognisers that deconstruct data and as templates that construct data. A formal operational semantics for matching and instantiation defines these interpretations (Sections 3.2 and 3.3). Transformations combine matching and instantiation by first matching data with a recogniser pattern and then instantiating a template pattern to produce a result (Section 3.4). Based on core operators, path polymorphic traversal of structures can be expressed in a flexible way. This is the basis for querying and searching data with patterns (Section 3.5). Meta-patterns are patterns that can manipulate patterns. They are based on a quasiquotation mechanism that allows the separation of "active" and "passive" patterns (Section 3.6). Based on meta-transformations, parameterised references define a pattern abstraction mechanism that makes the pattern core extensible (Section 3.7).

## 3.1 Fundamentals

This work builds a pattern-based foundation for language-driven programming, modelling and analysis. The approach is to (1) define a generic data representation and a core of pattern functionality that is a basis for all application domains, (2) add strong extensibility mechanisms to the core and (3) extend the pattern foundation to satisfy the requirements of each application domain. The advantage of this approach is that a wide range of application-specific techniques can be formalised using a small kernel of functionality. To provide a sound foundation, the core functionality and the extensibility mechanisms are defined using formal operational semantics. The definitions are based on a term notation for data and patterns. Concat defines an alternative character-based notation in Chapter 5.

### 3.1.1 Data Language

This work is concerned with a wide range of data types, including programs, models and messages. This requires a data representation mechanism that is highly flexible and and at the same time suitable for pattern matching. The arrangement of atomic values in sequences is the fundamental technique of data encoding found basically everywhere in computing, from describing a memory as a sequence of binary values to encoding a program as a sequence of characters. Explicit nesting can be expressed by allowing sequences inside sequences. This leads to a uniform data representation that distinguishes between elementary data (atomic values) and compound data (sequences) [9]. Types classify, define or restrict data structures [27]. Typed sequences encode the association between structures and types explicitly. Typed sequences consist of a sequence of data (the content) and an atom (the type).

$$
\begin{aligned}
exp &::= atom \mid tseq \\
seq &::= [\ exp^*\ ] \\
type &::= atom \\
tseq &::= \tau(seq, type)
\end{aligned}
$$

Figure 3.1: Data Language

The grammar in Figure 3.1 defines the term notation for typed sequences. The constructor $\tau$ constructs a typed sequence from a sequence and an atomic type identifier. A sequence consists of zero or more atoms or typed sequences surrounded by brackets. In the following, concrete notations for atoms will be introduced when required. The basic idea underlying typed sequences is to (1) provide a means for defining nested structures

and to (2) unambiguously declare how these structures should be interpreted. For example, if $a$ and $char$ are atoms, the typed sequence $\tau([a], char)$ explicitly states that its content $[a]$ is a character encoding or, in other words, that the data is of type character. An example for nested typed sequences is the encoding of the string "ab" with the atom $string$ as the type: $\tau([\tau([a], char), \tau([b], char)], string)$. Typed sequences represent tree structures in which all nodes except the atomic leafs carry type information. The patterns defined in the following can be applied to both the type and the content component. In Section 4.5 a view mechanism will be introduced in order to define arbitrary syntax based on typed sequences.

### 3.1.2 Pattern Language

Pattern expressions define structural operations on data by mixing concrete data with meta-language expressions. The data language introduced in the previous section forms a subset of the pattern language. Thus, all expressions of the data language are also valid patterns. This section defines the meta-expressions that can be interspersed with the data. Pattern expressions are either primitive patterns or compound patterns composed from simpler ones by means of an operator. Table 3.1 gives an overview of the pattern language. The pattern expressions are divided into categories according to their functionality. The rightmost column defines pattern expression recursively. Let $p$ denote a pattern expression and let $P$ denotes a sequence of pattern expressions $[p_1, ..., p_n]$. Let $n$ denote an atom representing the name of a variable. More convenient infix notations are defined for sequencing (concatenation), choice ($|$) and vertical composition ($\rightarrow$) as these operators will be used regularly in the definitions that follow.

As previously stated, the data language defined in the previous section forms a subset of the pattern language: atoms are primitive pattern expressions and there is an operator to construct typed sequence patterns from a sequence of patterns and a type pattern. This entails that every data expression is also a valid pattern expression.

Pattern expressions have two fundamental interpretations: *matching* and *instantiation*. Matching is the process of recognising and deconstructing data. Instantiation is the process of creating and assembling data. Let $Seq$ be the set of all data sequences as defined by the non-terminal $seq$ in Figure 3.1 and let $\epsilon$ denote the empty sequence. Let $A$ be the set of all atoms and let $Tseq$ be the set of all typed sequences as defined by $tseq$ in Figure 3.1. The set of all values is defined as $Val = A \cup Tseq \cup Seq$. Let the store $Sto$ be a set of pairs $(n, v)$ with $n \in A$ and $v \in Val$, each pair representing a variable binding, and let the empty store $\varepsilon$ denote the empty set. Let $Ptn$ be the set of all pattern expressions as defined in Table 3.1. Ignoring the failure case for the moment, the matching operation

| Category | Pattern | Syntax |
|---|---|---|
| Basic | atom | $a$ |
| | anything | $\alpha$ |
| | variable | $n{:}p$ |
| | reference | $ref(p)$ |
| Hierarchical | typed sequence | $\tau(P, p)$ |
| Modifying | negation | $!p$ |
| | maybe | $p?$ |
| | all | $all(p)$ |
| | ignore | $ign(p)$ |
| Horizontal | sequencing | $\sim(P)$ or $p\ p$ |
| | choice | $or(P)$ or $p \mid p$ |
| | repetition | $p^*$ |
| | repetition $> 0$ | $p^+$ |
| Two dimensional | vertical | $\Lambda(P)$ or $p \to p$ |
| | vertical repetition | $p^{*_v}$ |
| | diagonal | $\Delta(p)$ |
| | diagonal' | $\Delta'(p)$ |
| Transformative | unconditional transformation | $p \Rightarrow p$ |
| Quoting | quote | $quote(p)$ |
| | i-activate | $i\text{-}unquote(p)$ |
| | quasiquote | $qq(p)$ |
| | unquote | $uq(p)$ |
| Searching | find | $find\,p\,among\,p$ |
| | findall | $findall\,p\,among\,p$ |
| | replaceall | $replaceall\,p\,among\,p$ |
| | findall nested | $find\,p\,among\,p\,in\,p$ |

Table 3.1: Pattern Language

has the following signature:

$$match : Ptn \times Seq \times Sto \to Val \times Seq \times Sto$$

A pattern expression and a data sequence are matched in the context of variable bindings that are contained in a store. Matching yields a result value, a data sequence and a store. The *operational* semantics is that matching may change the data sequence and the store: values might be removed from or added to the data sequence; the store may be extended with bindings created during the matching. In the following, the matching and instantiation semantics of selected pattern operations will be discussed informally before full formal semantics will be presented in the next section.

An atom only matches a data sequence if a syntactically equal atom is the first element of the sequence. Accordingly, matching the primitive atom pattern $a$ with sequence

of atoms $[a, b, c]$ and the empty store $\varepsilon$ yields result $a$, remaining sequence $[b, c]$ and store $\varepsilon$. The notation $\langle a, [a, b, c], \varepsilon\rangle \xrightarrow{m} \langle a, [b, c], \varepsilon\rangle$ will be used to express a successful match with these inputs and outputs. The pattern $\sim([a, b])$ is a composition of two atom patterns $a$ and $b$ using the sequence operator $\sim$. The operator has the semantics that the patterns in its argument list are matched with the data sequence in the order in which they appear. The individual matching results are combined into a sequence. Matching this pattern with the inputs from above is described by: $\langle\sim([a, b]), [a, b, c], \varepsilon\rangle \xrightarrow{m} \langle[a, b], [c], \varepsilon\rangle$. The variable operator : combines an atom representing a variable name with an arbitrary pattern expression. Its semantics is to match the pattern expression with the data sequence, return the result and to add to the store a binding from the variable name to the result. Matching a variable pattern is illustrated by the example $\langle x{:}\sim([a, b]), [a, b, c], \varepsilon\rangle \xrightarrow{m} \langle[a, b], [c], \{(x, [a, b])\}\rangle$ in which the atom $x$ is the variable name. The matching semantics of typed sequences demands that the first element of the data sequence is a typed sequence. The result is constructed from the individual matching results of the type and content parts. Accordingly, $\langle\tau([a, x{:}\alpha], t), [\tau([a, b], t)], \varepsilon\rangle \xrightarrow{m} \langle\tau([a, b], t), \epsilon, \{(x, b)\}$ where $\alpha$ is a pattern that matches any atom or typed sequence and $t$ is an atom.

The instantiation semantics of patterns can be defined by an operation with the signature:

$$instantiate : Ptn \times Seq \times Sto \rightarrow Seq$$

A pattern expression is instantiated in the context of a data sequence and a store. It yields an updated data sequence. The *operational* semantics is that the data sequence is modified by the instantiation operation. An example is the instantiation of the primitive atom pattern $c$ with the sequence of atoms $[a, b]$ and the empty store $\varepsilon$. An atom is instantiated by adding it to the input sequence. Accordingly, the result is $[a, b, c]$, which is expressed by the notation $\langle c, [a, b], \varepsilon\rangle \xrightarrow{i} \langle[a, b, c]\rangle$. The instantiation semantics of the sequence operator is to instantiate the patterns in its argument list in order, with each instantiation working on the result of the previous one, as shown in the following example: $\langle\sim([b, c]), [a], \varepsilon\rangle \xrightarrow{i} \langle[a, b, c]\rangle$. While the matching interpretation of variables is to add a name-value binding to the store, the instantiation semantics is to look up the value from the store and replace the variable with the bound value. For example, $\langle\sim([a, x{:}\alpha]), \epsilon, \{(x, b)\}\rangle \xrightarrow{i} \langle[a, b]\rangle$. Instantiating a typed sequence pattern yields the result of instantiating the type and content parts and combining them with the $\tau$ operator. For example, $\langle\tau([a], x{:}\alpha), \epsilon, \{(x, t)\}\rangle \xrightarrow{i} \langle[\tau([a], t)]\rangle$.

### 3.1.3 Operational Semantics

The matching and instantiation of patterns is defined by an operational semantics based on the structure of pattern expressions [138]. The behaviour of a compound pattern is described in terms of the behaviour of its parts in the form of a set of transition relations. The transition relations are defined as inference rules that have the following form:

$$\frac{premise_1 \; ... \; premise_n}{conclusion} \; \textsc{Name}$$

The name of a rule is used for reference purposes. By convention, rules resulting in failure contain the symbol $\perp$ in their name. Each primitive pattern and each operator is defined by a set of such rules. The rules describe the operation of abstract machines for matching and instantiation. Each conclusion describes the *overall* result of a matching or instantiation operation.

A premise or conclusion of the form $\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out} \rangle$ denotes a successful match of pattern $p \in Ptn$ with sequence $s_{in} \in Seq$ and store $\sigma_{in} \in Sto$, the outcomes of which are result $r \in Val$, sequence $s_{out} \in Seq$ and store $\sigma_{out} \in Sto$. Accordingly, $\langle p, s_{in}, \sigma \rangle \xrightarrow{i} \langle s_{out} \rangle$ denotes a successful instantiation of pattern $p \in Ptn$ given sequence $s_{in} \in Seq$ and store $\sigma \in Sto$, the result of which is the sequence $s_{out} \in Seq$. The failure state $\perp$ indicates failure of matching or instantiation. Let $result$ denote the outcome of matching and instantiation. For all $s \in Seq$ and $\sigma \in Sto$ the following is true: $\langle \perp, s, \sigma \rangle \xrightarrow{m} \perp$ and $\langle \perp, s, \sigma \rangle \xrightarrow{i} \perp$.

Let $\sigma[n := v]$ denote the set $\sigma \cup \{(n, v)\}$. Let $\sigma[n]$ denote $v$ if $(n, v) \in \sigma$ and $\perp$ otherwise. Let the operator $=$ denote syntactic equality and let the operator $\neq$ denote syntactic inequality. Two atoms are syntactically equal if they have the same symbolic representation. Two typed sequences are syntactically equal, if they have a syntactically equal type and syntactically equal content. Two sequences are syntactically equal if all of the atoms or typed sequences they contain are syntactically equal.

The infix operator :: denotes a function that expects an element on the left and a possibly empty sequence on the right and prepends the element to the sequence. For instance, $v{::}s$ denotes a sequence with $v$ as the first element followed by the elements of sequence $s$. The binary function $append$ maps arguments $s_1$ and $s_2$ to a new sequence that contains the elements of $s_1$ followed by the elements of $s_2$. The unary predicate $atom?$ is true if its argument is an atom and false otherwise. The unary predicate $seq?$ is true if its argument is a sequence (not a typed sequence) and false otherwise. The prefix-operator $not$ negates a truth value. Table 3.2 gives an overview of the notation just introduced and defines

| Notation | Denotation |
|---|---|
| $\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out}\rangle$ | successful match of pattern $p$ |
| $\langle p, s, \sigma\rangle \xrightarrow{m} \bot$ | failure matching pattern $p$ |
| $\langle p, s, \sigma\rangle \xrightarrow{m} result$ | success or failure matching pattern $p$ |
| $\langle p, s_{in}, \sigma\rangle \xrightarrow{i} \langle s_{out}\rangle$ | successful instantiation of pattern $p$ |
| $\langle p, s, \sigma\rangle \xrightarrow{i} \bot$ | failure instantiating pattern $p$ |
| $\langle p, s, \sigma\rangle \xrightarrow{i} result$ | success or failure instantiating pattern $p$ |
| $f(args) \mapsto v$ | successful application of function $f$ |
| $f(args) \mapsto \bot$ | failure applying function $f$ |
| $(n, v)$ | a binding of name $n$ to value $v$ |
| $\sigma[n := v]$ | adding a binding $(n, v)$ to store $\sigma$ |
| $\sigma[n]$ | the value of $n$ in store $\sigma$ |
| $v::s$ | a sequence with first element $v$ |
| $[x_1,x_2, ...]$ | a sequence of elements $x_1,x_2, ...$ |
| $v_1 = v_2$ | syntactic equality of $v_1$ and $v_2$ |
| $v_1 \neq v_2$ | syntactic inequality of $v_1$ and $v_2$ |
| $not\ pred(..)$ | negation of predicate $pred$ |
| $a$ or $a_x$ | an atom $\in A$ |
| $s$ or $s_x$ | a sequence $\in Seq$ |
| $\epsilon$ | the empty sequence |
| $p$ or $p_x$ | a pattern $\in Ptn$ |
| $P$ or $P_x$ | a sequence of patterns |
| $\sigma$ or $\sigma_x$ | a store $\in Sto$ |
| $\varepsilon$ | the empty store |
| $r$ or $r_x$ | a result of a matching operation $\in Val$ |
| $n$ or $n_x$ | a variable name $\in A$ |
| $v$ or $v_x$ | a value $\in Val$ |
| $e$ or $e_x$ | a value $\in A \cup Tseq$ |

Table 3.2: Overview of the Notation

what kind of data is represented by symbols with and without subscript in the following definitions, unless it is stated otherwise.

## 3.2 Matching Semantics

In this section, the matching semantics of patterns will be defined. Matching is the operation of deconstructing and recognising data. Matching operates on a data sequence and a store that contains variable bindings. The semantics define the modification of both and the derivation of a matching result.

### 3.2.1 Helper Functions

**Consistent Binding** The function $cbind$ (for consistently bind) ensures that the mapping from name to value defined by a store is injective. The parameters of $cbind$ are a name, a value and a store $\sigma$. Function $cbind$ fails if $\sigma$ already contains a binding with the same name and different value.

$$\frac{\sigma[n] = \bot}{cbind(n, v, \sigma) \mapsto \sigma[n := v]}\ \textsc{Cbind New}$$

$$\frac{\sigma[n] = v}{cbind(n, v, \sigma) \mapsto \sigma}\ \textsc{Cbind Equal}$$

$$\frac{\sigma[n] \neq v \qquad \sigma[n] \neq \bot}{cbind(n, v, \sigma) \mapsto \bot}\ \textsc{Cbind } \bot$$

**Combination** The function $combine$ defines the combination of data into a sequence. This functionality is required as part of several of the following definitions, for example to define the combination of individual matching results into a compound result. In contrast to $append$, which requires both its arguments to be sequences, $combine$ defines combination also for non-sequence arguments, i.e., atoms and typed sequences. For example, if $a$ and $b$ are atoms, $combine(a, [b])$ has the result $[a, b]$. In addition to that, $combine$ performs an implicit flattening if its first argument is a sequence. For example, $combine([a], [b])$ has the result $[a, b]$. This entails that empty sequences disappear when being combined: $combine(\epsilon, [b])$ has the result $[b]$.

$$\frac{seq?(r_1) \qquad seq?(r_2)}{combine(r_1, r_2) \mapsto append(r_1, r_2)}\ \textsc{Combine Sequences}$$

$$\frac{not\ seq?(r_1) \qquad seq?(r_2)}{combine(r_1, r_2) \mapsto append([r_1], r_2)}\ \textsc{Combine Prepend}$$

$$\frac{seq?(r_1) \qquad not\ seq?(r_2)}{combine(r_1, r_2) \mapsto append(r_1, [r_2])}\ \textsc{Combine Append}$$

$$\frac{not\ seq?(r_1) \qquad not\ seq?(r_2)}{combine(r_1, r_2) \mapsto append([r_1], [r_2])}\ \textsc{Combine Values}$$

### 3.2.2 Fundamental Pattern Expressions

**Atom** An atom matches an input sequence if the first element of that sequence is the same atom. In this case, the result is the atom and the output sequence is the input sequence without the first element. Matching fails if the first element is different from the atom or if the input sequence is empty. Matching an atom has no effect on the store.

$$\frac{}{\langle a, a::s, \sigma\rangle \xrightarrow{m} \langle a, s, \sigma\rangle}\ \textsc{Atom Success}$$

$$\frac{a \neq e}{\langle a, e::s, \sigma\rangle \xrightarrow{m} \bot}\ \textsc{Atom Different}\ \bot$$

$$\frac{}{\langle a, \epsilon, \sigma\rangle \xrightarrow{m} \bot}\ \textsc{Atom Empty}\ \bot$$

**Anything** The pattern expression $\alpha$ matches an input sequence if that sequence contains at least one value. The result is the first value of the input sequence and the output sequence is the input sequence with the first value removed. Matching fails for empty input sequences. Matching $\alpha$ has no effect on the store.

$$\frac{}{\langle \alpha, e::s, \sigma\rangle \xrightarrow{m} \langle e, s, \sigma\rangle}\ \textsc{Any Success}$$

$$\frac{}{\langle \alpha, \epsilon, \sigma \rangle \xrightarrow{m} \bot} \text{ANY EMPTY } \bot$$

**Negation** The negation operator ! implements a form of negation as failure [29]. Matching $!p$ succeeds if $p$ fails, and fails if $p$ succeeds. In case of success the result is $\epsilon$ and the data sequence and store remain unchanged.

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \bot}{\langle !p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \langle \epsilon, s_{in}, \sigma_{in} \rangle} \text{NEGATION SUCCESS}$$

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out} \rangle}{\langle !p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \bot} \text{NEGATION } \bot$$

The rule NEGATION SUCCESS defines that in the case of failure of $p$, $!p$ yields an empty result and an unmodified sequence and store. This behaviour can be utilised to express lookahead, i.e., to *test* if a pattern matches [52]. To achieve this, the pattern must be negated twice: the pattern $!!p$ succeeds if $p$ succeeds but ignores its effect [178].

**Variable** The variable pattern $n{:}p$ adds to the store a binding from $n$ to the result of successfully matching $p$. Matching fails if $p$ fails or a binding already exists for $n$ with a value that is not equal to the result of matching $p$. In case the binding is equal the store remains unchanged.

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \langle r, s_{out}, \sigma_m \rangle \quad cbind(n, r, \sigma_m) \mapsto \sigma_{out}}{\langle n{:}p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out} \rangle} \text{VARIABLE SUCCESS}$$

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \bot}{\langle n{:}p, s_{in}, \sigma_{in} \rangle \xrightarrow{m} \bot} \text{VARIABLE MATCH } \bot$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_{out}, \sigma_m\rangle \quad cbind(n, r, \sigma_m) \mapsto \bot}{\langle n{:}p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Variable Inconsistent}\ \bot$$

The variable binding behaviour allows the expression of syntactic equality of values by using syntactically equal variable names. For instance, the pattern $\sim([x{:}\alpha, x{:}\alpha])$ matches an input sequence only if that sequence starts with two identical values. The atom $x$ is a variable name. The pattern uses the sequential matching operator that will be defined in Section 3.2.4.

**All** The pattern $all(p)$ succeeds if the data sequence is empty after matching $p$, otherwise it fails. The intent of $all$ is to express that input must be matched completely by a pattern.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, \epsilon, \sigma_{out}\rangle}{\langle all(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, \epsilon, \sigma_{out}\rangle}\ \textsc{All Success}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_m, \sigma_m\rangle \qquad s_m \neq \epsilon}{\langle all(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{All Incomplete}\ \bot$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle all(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{All Pattern}\ \bot$$

**Ignore** The pattern $ign(p)$ (for ignore $p$) succeeds if $p$ succeeds. In this case, the result is $\epsilon$. However, only the result of $p$ is ignored, not its effect on the store and the data sequence.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out}\rangle}{\langle ign(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{out}, \sigma_{out}\rangle}\ \textsc{Ignore Success}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle ign(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Ignore}\ \bot$$

Operators that construct compound results from the results of matching individual patterns discard empty sequences. Therefore, the empty sequence result of $ign$ actually disappears when combined with other results.

### 3.2.3 Hierarchical Matching

**Typed Sequence** The typed sequence pattern consists of two patterns that are to be matched with the type and content parts of a typed sequence. The result is a typed sequence constructed from the results of each part. Matching fails if matching either the content or the type pattern fails. It also fails if the type pattern yields a value that is not an atom. The following definition uses the sequential horizontal operator $\sim$ that applies a sequence of pattern in order and combines their results. A definition of this operator is part of the following section.

$$\frac{\begin{array}{c}\langle p, [a], \sigma_{in}\rangle \xrightarrow{m} \langle r_t, \epsilon, \sigma_t\rangle \\ atom?(r_t) \\ \langle {\sim}(P), s_c, \sigma_t\rangle \xrightarrow{m} \langle r_c, \epsilon, \sigma_{out}\rangle\end{array}}{\langle \tau(P,p), \tau(s_c,a){::}s_{out}, \sigma_{in}\rangle \xrightarrow{m} \langle \tau(r_c, r_t), s_{out}, \sigma_{out}\rangle}\ \textsc{TSeq Success}$$

$$\frac{\langle p, [a], \sigma_{in}\rangle \xrightarrow{m} \langle r_t, \epsilon, \sigma_t\rangle \qquad not\ atom?(r_t)}{\langle \tau(P,p), \tau(s_c,a){::}s, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{TSeq Atom}\ \bot$$

$$\frac{\langle all(p), [a], \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle \tau(P,p), \tau(s_c,a){::}s, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{TSeq Type}\ \bot$$

$$\frac{\begin{array}{c}\langle p, [a], \sigma_{in}\rangle \xrightarrow{m} \langle r_t, \epsilon, \sigma_t\rangle \\ atom?(r_t) \\ \langle all(\sim(P)), s_c, \sigma_t\rangle \xrightarrow{m} \bot\end{array}}{\langle \tau(P, p), \tau(s_c, a)::s, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \text{TSEQ CONTENT } \bot$$

Together, the primitive atom pattern introduced in the previous section and the typed sequence operator make it possible to use any typed sequence as a pattern for matching data literally. Because typed sequences may contain other typed sequences, the combinator defines hierarchic matching on arbitrary tree structures.

For example, the pattern $\tau([\tau([c], char), (\tau([\alpha], char))^*], string)$ matches typed sequences of type $string$ that contain at least one typed sequence of type $char$. The first typed sequence must contain an atom $c$. When this pattern is matched with the input sequence $[\tau([\tau([c], char), \tau([a], char), \tau([t], char)], string)]$, the type pattern $string$ is successfully matched with $[string]$. The content part of the typed sequence pattern is matched sequentially with the content part of the typed sequence. The third premise in rule TSEQ SUCCESS creates the pattern $\sim([\tau([c], char), (\tau([\alpha], char))^*])$ which is matched with $[\tau([c], char), \tau([a], char), \tau([t], char)]$. This process of matching type and content is continued recursively for the characters contained in the string. For both characters and strings, the results of matching the type and the content are reassembled to form a typed sequence. Therefore, the overall result of the example is the single element of the input sequence.

### 3.2.4 Horizontal Matching

The patterns described so far process data sequences in a linear manner, removing zero or more elements but never adding elements. That is, if $\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out}\rangle$ then there is an $s_r$ so that $append(s_r, s_{out}) = s_{in}$. The horizontal operators preserve this linear consumption property if all their arguments fulfil this property. This entails that the horizontal operators match their arguments in such a way that the result of one match is invisible to the next match. This is illustrated by Figure 3.2: the input elements $a$, $b$ and $c$ on level $L_0$ are transformed in a stepwise fashion into results $x$, $y$ and $z$ on $L_1$. The parsing processes (illustrated by numbered circles) have no access to the results on $L_1$. In general, all input is processed on the same horizontal level $L_n$.

**Sequencing** The sequential operator $\sim$ expects a sequence of pattern expressions and matches the expressions in the order in which they appear in the sequence. The first pattern

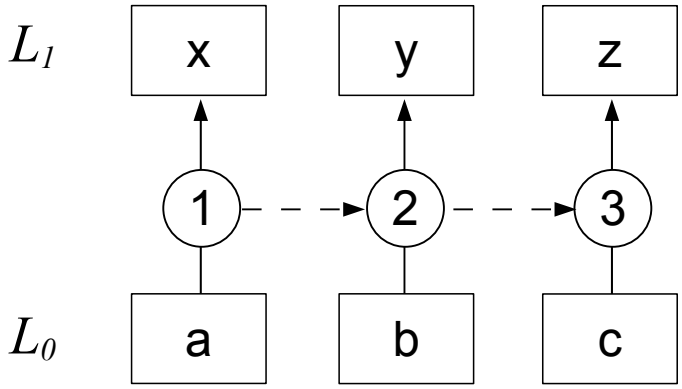


Figure 3.2: Illustration of Horizontal Matching

is matched to the input data sequence and each following pattern to the data sequence and bindings produced by the previous match. The results of all matches are combined into a sequence.

$$\frac{\begin{array}{c}\langle p, s_{in}, \sigma_{sin}\rangle \xrightarrow{m} \langle r_p, s_p, \sigma_p\rangle \\ \langle {\sim}(P), s_p, \sigma_p\rangle \xrightarrow{m} \langle r_P, s_{out}, \sigma_{out}\rangle \\ combine(r_p, r_P) \mapsto r_{out}\end{array}}{\langle {\sim}(p{::}P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_{out}, s_{out}, \sigma_{out}\rangle}\ \textsc{Sequence Success}$$

$$\frac{}{\langle {\sim}(\epsilon), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{in}, \sigma_{in}\rangle}\ \textsc{Sequence Empty}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle {\sim}(p{::}P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Sequence Match} \bot$$

The sequencing operator creates a compound result from individual results by using the helper function $combine$ defined in Section 3.2.1. The advantage of combining results in this way is that in most cases there is no need to express explicit flattening of results in the pattern expressions. If nesting is explicitly desired, it can be expressed using typed sequences which are not automatically flattened. Therefore, $combine$ can distinguish between an intended nesting or a nesting that is used to pass a sequence of results. Wrapping of a result into a typed sequence can be expressed using transformations, as defined in Section 3.4.

For example, if $a$, $b$, $c$ and $d$ are atoms, matching the pattern $\sim([\sim([a,b]),\sim([c,d])])$ with input $[a,b,c,d]$ produces intermediate results $[a,b]$ and $[c,d]$ for the two inner sequential patterns. However, the overall result is the sequence $[a,b,c,d]$ in which the nesting is invisible. A pattern that expresses result nesting explicitly is the following: $\sim([\sim([a,b]) \Rightarrow \tau([a,b],list),\sim([c,d]) \Rightarrow \tau([c,d],list)])$. Matching it with the input from above produces the result $[\tau([a,b],list),\tau([c,d],list)]$ in which the nesting is explicit.

**Choice** The choice operator $or$ matches its arguments in order until one of them succeeds. Changes made by unsuccessful matching attempts have no effects on the result, the data sequence or the store. Only the first successful match has an effect. Matching a choice pattern fails if there is no choice that can be matched successfully.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out}\rangle}{\langle or(p::P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_{out}, \sigma_{out}\rangle}\ \textsc{Choice Success}$$

$$\frac{\begin{array}{c}\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot \\ \langle or(P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result\end{array}}{\langle or(p::P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \textsc{Choice Recurse}$$

$$\frac{}{\langle or(\epsilon), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Choice None} \bot$$

The semantics of the choice operator defines a clear left-to-right order in which patterns are matched. Although definitions may be ambiguous, i.e., more than one pattern of the choice can match an input, the result is always unambiguously defined to be that of the first pattern that matches starting from the left. For instance, if $b$ is an atom the pattern $or([b,\alpha])$ has a clear matching semantics even though both choices match a sequence that starts with $b$: choice $\alpha$ is only tried if matching $b$ fails. Once the matching of one of the choices is successful, there will be no backtracking. This semantics is crucial when patterns are used to express computations and grammars (see Chapter 4). For example, putting the base case of a recursive definition before the recursive case in the choice pattern ensures that the recursive case is only tried if the base case fails. In the following, the alternative notation $p_1|...|p_n$ will be used where appropriate to denote the choice $or([p_1,...,p_n])$.

**Repetition** The matching semantics of the pattern $p^*$ is to match $p$ repeatedly to the input until matching $p$ fails. The result is the combined result of all successful matches and an empty result if there are no successful matches. This means that $p^*$ never fails.

$$\frac{\begin{array}{c}\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_p, s_p, \sigma_p\rangle \\ \langle p^*, s_p, \sigma_p\rangle \xrightarrow{m} \langle r_{p*}, s_{out}, \sigma_{out}\rangle \\ combine(r_p, r_{p*}) \mapsto r_{out}\end{array}}{\langle p^*, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_{out}, s_{out}, \sigma_{out}\rangle}\ \textsc{Repeat Greedy}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle p^*, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{in}, \sigma_{in}\rangle}\ \textsc{Repeat No Match}$$

The $^*$ operator has greedy semantics as it attempts to consume as many elements from the data sequence as possible. Care must be taken when using the operator with patterns that do not consume from the data sequence as this leads to infinite regress, as in the case of the pattern $(\sim(\epsilon))^*$.

### 3.2.5 Vertical and Diagonal Matching

The patterns defined so far consume the input sequence from left to right; results of previous matches are invisible to following matches. Data can be mapped onto a different representational level, e.g., by transformations, but the result cannot be further processed and thus matching is restricted to a single level. The following operators define forms of matching in which the result of a previous match is available to the following match.

**Vertical** The vertical operator $\Lambda$ matches a sequence of patterns. Only the first pattern is matched with the data sequence. All other patterns are matched with the results of their predecessors and they have to consume that result entirely. If all patterns match successfully the result of vertical matching is the result produced by the last pattern in the sequence. This is illustrated by the left-hand side of Figure 3.3. Each process transforms between $L_n$ and $L_{n+1}$ and has no access to data on other levels. In other words, the processing progresses vertically between levels. In the example, the transformation from $a$ to $z$ has $x$ and $y$ as intermediate results.
The first rule matches $p_1$ and then recurses using the $all$ operator. This way the first pattern must not match the entire data sequence, but all others have to match the entire result of

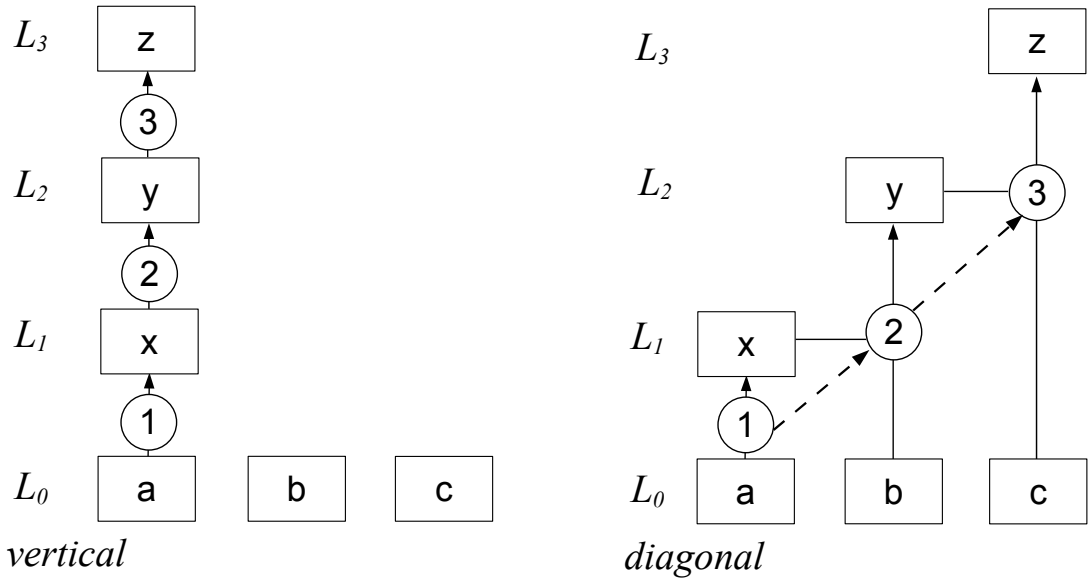


Figure 3.3: Illustration of Vertical and Diagonal Matching

their respective successor. In conjunction with transformations (see Section 3.4), vertical matching can be used to define multiple stages of processing. For instance, if $a$, $b$ and $c$ are atoms, matching pattern $\Lambda([a \Rightarrow b, b \Rightarrow c])$ with sequence $[a]$ yields the result $c$.

$$\frac{\langle{\sim}([p_1]), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_p, s_{out}, \sigma_p\rangle \quad \langle\Lambda(all(p_2)::P), r_p, \sigma_p\rangle \xrightarrow{m} \langle r_{out}, \epsilon, \sigma_{out}\rangle}{\langle\Lambda(p_1::p_2::P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_{out}, s_{out}, \sigma_{out}\rangle}\ \textsc{Vertical Success}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}{\langle\Lambda([p]), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \textsc{Vertical Single}$$

$$\frac{}{\langle\Lambda(\epsilon), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle\epsilon, s_{in}, \sigma_{in}\rangle}\ \textsc{Vertical Empty}$$

$$\frac{\langle{\sim}([p_1]), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle\Lambda(p_1::p_2::P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Vertical Match}\ \bot$$

$$\frac{\langle\sim([p_1]), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_p, s_p, \sigma_p\rangle \quad \langle\Lambda(all(p_2)::P), r_p, \sigma_p\rangle \xrightarrow{m} \bot}{\langle\Lambda(p_1::p_2::P), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Vertical Recurse}\ \bot$$

As defined in Section 3.1.2, the alternative notation $p_1 \rightarrow p_2$ can be used to express vertical combination of patterns $p_1$ and $p_2$.

**Vertical Repetition** Vertical repetition applies a pattern to the input sequence and then repeatedly to its result. The semantics can be expressed based on the vertical combination, choice and sequencing operators introduced above.

$$\frac{\langle((p \rightarrow p^{*_v}) \mid p \mid\sim(\epsilon)), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}{\langle p^{*_v}, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \textsc{Vertical Repeat}$$

The choice pattern defines a recursive case and two base cases. The recursive case expresses vertical repetition of a pattern $p$ as repeated vertical combination of $p$. Two base cases are necessary because the result of vertical combination is defined to be the result of the last pattern matched. In case $p$ can be matched at least once, the result must be that of $p$. Nevertheless, the empty sequence pattern expresses that $p^{*_v}$ succeeds even if $p$ does not match the input sequence.

**Diagonal Matching** The concept of diagonal matching can be understood as a combination of horizontal and vertical matching. The horizontal operators match a sequence of patterns successively to an input sequence. Each match removes zero or more items from the input sequence and produces a result. A pattern is never matched with the result of a previous match, only with the resulting output sequence. The vertical operator on the other hand matches patterns successively with the result of a previous match. A pattern is never matched with the output sequence of a previous match, only with the result. The diagonal operator allows the combination of patterns in such a way that they are matched successively to the input sequence *and* to the result of the previous match.

Diagonal matching is illustrated by the right-hand side of Figure 3.3. While data processing in horizontal matching is restricted to a single level and in vertical matching to transformations between adjacent levels $L_n$ and $L_{n+1}$, diagonal matching allows to process data from different levels because the result *and* the output sequence of a match are available to the next match. For example, the second parsing process uses both the result $x$ and the element $b$ to produce $y$.

Making result and output sequence of a match available to the next match can be achieved in two different ways.The output and result of a match are either combined to form a compound output sequence or to form a compound result. In the following, the former will be defined and it will be shown how the latter can be derived from this definition.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r, s_p, \sigma_{out}\rangle \quad combine(r, s_p) \mapsto s_{out}}{\langle \Delta p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{out}, \sigma_{out}\rangle}\ \textsc{Diagonal Success}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle \Delta p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Diagonal} \bot$$

The first variant of the operator yields $\epsilon$ as the matching result. Its output sequence is a combination of the result and the output sequence obtained from matching its operand. For instance, matching pattern $\Delta([a \Rightarrow x])$ with input sequence $[a, b, c]$ yields output sequence $[x, b, c]$ and result $\epsilon$. To express diagonal combination of multiple patterns, the horizontal operator $\sim$ can be used: matching $\sim([\Delta([a \Rightarrow x]), \Delta([x \Rightarrow y])])$ yields output sequence $[y, b, c]$ and result $\epsilon$.

The alternative operator $\Delta'$ can be expressed in terms of $\Delta$.

$$\frac{\langle \sim([\Delta p, \alpha^*]), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}{\langle \Delta' p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \textsc{Diagonal Variant}$$

The matching semantics of $\Delta' p$ is to first match $\Delta p$ which produces the compound result as the output sequence and then to match $\alpha^*$ with that output sequence. Because repetition is greedy and $\alpha$ matches any element, the overall result of the match is the content of the input sequence; the output sequence is empty. To express diagonal combination of multiple patterns, the vertical operator $\Lambda$ can be used: matching $\Lambda([\Delta'([a \Rightarrow x]), \Delta'([x \Rightarrow y])])$ yields result $[y, b, c]$ and output sequence $\epsilon$.

## 3.3 Instantiation Semantics

In this section, the instantiation semantics of patterns will be described. While matching is the operation of deconstructing data and possibly adding new bindings to the store, instantiation is the operation of constructing data by possibly using bindings from the store. Not all of the patterns that have a matching semantics also have meaningful instantiation semantics. Instantiation of such patterns fails.

### 3.3.1 Fundamental Pattern Expressions

**Atom** An atom pattern is instantiated by appending its literal value to the data sequence. Instantiating an atom never fails.

$$\frac{combine(s_{in}, a) \mapsto s_{out}}{\langle a, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Atom Success}$$

**Anything** The pattern $\alpha$ has an instantiation semantics in the context of variables, as will be described below. Instantiating $\alpha$ outside of the context of variables is ambiguous as it is unclear which value to instantiate for it and, therefore, instantiation fails.

$$\frac{}{\langle \alpha, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Any Instantiation} \bot$$

Ambiguity might be an interesting feature for some applications and, therefore, a different semantics is conceivable.

**Negation** The unary negation operator succeeds if its argument fails, and fails if its argument succeeds. In case of success, it has no effect on the data sequence.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}{\langle !p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{in}\rangle}\ \textsc{Negation Success}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}{\langle !p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Negation} \bot$$

**Variable** The variable pattern $n{:}p$ is instantiated by retrieving the value bound to $n$ from the store, matching $p$ with that value and instantiating the result with the input sequence and input store. The use of $combine$ in the second premise has the effect of wrapping lookup results into a sequence if they are not already a sequence so that the ensuing matching operation is guaranteed to work with a sequence.

$$\frac{\begin{array}{c}\sigma_{in}[n] \neq \bot \\ combine(\sigma_{in}[n], \epsilon) \mapsto s_c \\ \langle all(p), s_c, \varepsilon\rangle \xrightarrow{m} \langle r, \epsilon, \sigma_p\rangle \\ combine(s_{in}, r) \mapsto s_{out}\end{array}}{\langle n{:}p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Variable Success}$$

$$\frac{\sigma_{in}[n] = \bot}{\langle n{:}p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Variable Unbound} \bot$$

$$\frac{\begin{array}{c}\sigma_{in}[n] \neq \bot \\ combine(\sigma_{in}[n], \epsilon) \mapsto s_c \\ \langle all(p), s_c, \varepsilon\rangle \xrightarrow{m} \bot\end{array}}{\langle n{:}p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Variable Match} \bot$$

In the following example $a$, $b$, $c$ and $x$ are atoms: $\langle x{:}or([b, c]), [a], \{(x, b)\}\rangle \xrightarrow{i} \langle [a, b]\rangle$. The variable is instantiated by first looking up the name $x$ in the store. The lookup result is atom $b$. Next, the $combine$ function yields $[b]$ which is matched with $all(or([b, c]))$. The match is successful and produces the result value $b$. The input sequence $[b]$ is combined with this value, which yields the output sequence $[a, b]$.

**Ignore** Instantiating $ign(p)$ succeeds if $p$ can be instantiated, otherwise it fails. Even in case of success no changes are made to the data sequence.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_p\rangle}{\langle ign(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{in}\rangle}\ \textsc{Ignore Success}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}{\langle ign(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Ignore}\ \bot$$

**All** The operator $all$ is ignored during instantiation. The result of instantiating $all(p)$ is the result of instantiating $p$.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} result}{\langle all(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} result}\ \textsc{all success}$$

### 3.3.2 Hierarchical Instantiation

**Typed Sequence** A typed sequence pattern is instantiated by instantiating its type pattern and its content pattern. The type pattern must instantiate to a sequence containing an atom. This is, for example, the case if the type pattern is an atom.

$$\frac{\begin{array}{c}\langle p, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle [r_t]\rangle \\ atom?(r_t) \\ \langle {\sim}(P), \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_c\rangle \\ combine(s_{in}, \tau(s_c, r_t)) \mapsto s_{out}\end{array}}{\langle \tau(P, p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{TSeq Success}$$

$$\frac{\langle p, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \bot}{\langle \tau(P, p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{TSeq Inst}\ \bot$$

$$\frac{\langle p, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_p\rangle \qquad length(s_p) \neq 1}{\langle \tau(P, p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{TSeq Length}\ \bot$$

$$\frac{\langle p, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle [r_t]\rangle \qquad not\ atom?(r_t)}{\langle \tau(P, p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{TSeq Atom}\ \bot$$

$$\frac{\langle \sim(P), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \bot}{\langle \tau(P, p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \bot} \text{TSEQ CONTENT } \bot$$

### 3.3.3 Horizontal Instantiation

**Sequence** The sequence operator instantiates a sequence of patterns in order. The output data sequence of a pattern is the input data sequence of the following pattern. Therefore, it is not necessary to express the combination of results explicitly in the definition of the sequence operator. Sequential instantiation fails if any of the patterns matched in sequence fails.

$$\frac{\begin{array}{c}\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_p \rangle \\ \langle \sim(P), s_p, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle\end{array}}{\langle \sim(p::P), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{SEQUENCE SUCCESS}$$

$$\frac{}{\langle \sim(\epsilon), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{in} \rangle} \text{SEQUENCE EMPTY}$$

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{i} \bot}{\langle \sim(p::P), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \bot} \text{SEQUENCE } \bot$$

**Choice** Instantiation of a choice pattern attempts to instantiate a sequence of patterns in order until one of them succeeds. Instantiation fails for the empty sequence. The instantiation semantics of *or* are most useful in conjunction with alternatives that contain variables. The chosen alternative may depend on variables. This can be used for expressing conditions such as "if variable $v_1$ is bound, instantiate pattern $p_1$, else if variable $v_2$ is bound, instantiate pattern $p_2$". Because instantiation of variables entails matching their pattern component when instantiating, it is possible to make successful instantiation dependent on the type of the variable.

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle}{\langle or(p::P), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{CHOICE SUCCESS}$$

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}{\langle or(P), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle} \text{ CHOICE RECURSE}$$

$$\frac{}{\langle or(\epsilon), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot} \text{ CHOICE NONE } \bot$$

**Repetition** The pattern $p^*$ is ambiguous as it lacks information on how often $p$ should be instantiated. Therefore, its instantiation fails. In the next chapter an extended version of repetition will be presented in the form of the $^\circ$ operator. This operator has useful instantiation semantics. It is the basis for expressing complex restructuring operations.

### 3.3.4 Vertical and Diagonal Instantiation

The vertical and diagonal combinators do not have meaningful interpretations for instantiating. Their instantiation fails.

## 3.4 Transformations and General Purpose Patterns

This section extends the core pattern semantics defined in the previous sections with transformations and a number of general purpose pattern operators that can be described in terms of existing ones.

### 3.4.1 Unconditional Transformations

Matching patterns is a means to recognise and deconstruct data structures. Instantiating patterns is a means to create data structures. Transformations combine matching and instantiation to express restructuring of data. Variables are crucial in the transformation process as bindings are used to transfer data from the matching to the instantiation phase.

Unconditional transformations have the notation $p_l \Rightarrow p_r$ with $p_l$ and $p_r$ being patterns. Pattern $p_l$ is matched with the input sequence and pattern $p_r$ is instantiated with the resulting bindings. The result of matching $p_l$ is discarded. Transformations create a local name space: bindings created during the matching phase are only used for instantiation and are not part of the output store. A transformation fails if either matching the left-hand side or instantiating the right-hand side fails.

$$\frac{\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_m, s_{out}, \sigma_m\rangle \quad \langle p_r, \epsilon, \sigma_m\rangle \xrightarrow{i} \langle r_{out}\rangle}{\langle p_l \Rightarrow p_r, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_{out}, s_{out}, \sigma_{in}\rangle}\ \textsc{Transformation}$$

$$\frac{\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle p_l \Rightarrow p_r, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Transformation Match}\ \bot$$

$$\frac{\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_m, s_{out}, \sigma_m\rangle \quad \langle p_r, \epsilon, \sigma_m\rangle \xrightarrow{i} \bot}{\langle p_l \Rightarrow p_r, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Transformation Inst}\ \bot$$

For example, the transformation $\sim([x{:}\alpha, y{:}\alpha]) \Rightarrow \sim([y{:}\alpha, x{:}\alpha])$ matches two elements from the input sequence and swaps their position in the result. Matching this pattern with the input sequence $[a, b, c]$ produces result $[b, a]$ and output sequence $[c]$. The bindings for $x$ and $y$ are temporarily created during matching and not kept in the store.

Defining transformations as patterns and giving them a matching semantics has the advantage that they can be used as parts of compound patterns. For example, the pattern $(a \Rightarrow b)^*$ greedily matches a sequence of $a$s and in the result produces a $b$ for every $a$. In the rest of this work, patterns containing transformations are used extensively. For instance, such patterns are used to define pattern-based rewriting systems, see Section 4.2. A conditional version of transformations where matching a transformation depends on the results of other matches will be introduced in Section 4.3.

### 3.4.2 Definition of General Purpose Patterns

This section defines patterns that are useful for general purposes and that can be defined in terms of existing patterns. This is expressed by syntactic rewrite rules of the form $ptn_l \Longrightarrow ptn_r$ where $ptn_l$ and $ptn_r$ are meta-patterns. The symbol $p$ is a meta-variable that ranges over pattern expressions. In Section 3.7.1 a mechanism will be introduced to express rewriting of patterns in the pattern formalism using meta-transformations.

**Empty** Matching the pattern $empty$ succeeds with no effects on the store and data sequence if the data sequence is empty, otherwise it fails. Instantiating empty succeeds with

no effect on the data sequence.

$$empty \implies !\alpha$$

**Nothing** Matching and instantiating the pattern $nothing$ always succeeds without having an effect.

$$nothing \implies \sim(\epsilon)$$

**Maybe** Matching and instantiating the pattern $p?$ always succeeds. If $p$ succeeds it has the same effect as $p$, otherwise it has no effect.

$$p? \implies p \,|\, nothing$$

**Repetition** $\geq 1$ $p^+$ attempts to match $p$ with the input sequence as often as possible. It succeeds if $p$ can be matched at least once and returns the combined result of all matches.

$$p^+ \implies \sim([p, p^*])$$

**Unwrap** Matching the pattern $unwrap(p)$ succeeds only if matching $p$ has a sequence with one element as a result. In this case, the element is returned.

$$unwrap(p) \implies p \to \alpha$$

## 3.5 Path Polymorphic Matching

The operators in this section provide a flexible mechanism for defining matching of patterns along arbitrary paths in (nested) structures [89]. The basic idea is to define a pattern $p_f$ that matches data of interest, a pattern $p_a$ that defines data that might be interspersed between instances of $p_f$ and – for nested structures – a pattern $p_t$ that defines which typed sequences will be traversed. The resulting patterns can express searching, querying and collectively transforming sequential or hierarchical data.

### 3.5.1 Finding Instances of a Pattern in a Sequence

The pattern expression $find\ p_f\ among\ p_a$ looks for the first instance of a pattern $p_f$ in a sequence of elements that match the pattern $p_a$. It succeeds if $p_f$ matches an element and yields the match result otherwise it matches $p_a$ and recursively $find\ p_f\ among\ p_a$ in sequence. The pattern fails if the input sequence is empty. The semantics of $find$ is defined by the following rule:

$$find\, p_f\, among\, p_a \implies p_f \mid\sim([p_a, find\, p_f\, among\, p_a])$$

The result of matching a $find$ pattern is not only the result of successfully matching the second argument to an element, but also the result of all sequential matches of the first argument that lead up to this match. For example, matching $find\, c\, among\, \alpha$ with a sequence of atoms $[a, b, c, d]$ yields $[a, b, c]$ as the result and $[d]$ as the rest of the input sequence. A compound result without the results of matching $p_a$ is obtained by using the ignore operator: $find\ c\ among\ ign(\alpha)$ yields the result $[d]$.

### 3.5.2 Finding or Replacing All Instances

A successful match of the $find$ pattern yields as the result the first occurrence of a pattern in a sequence. The $findall$ pattern returns *all* instances of a pattern in a sequence.

$$findall\, p_f\, among\, p_a \implies empty \mid\sim([(p_f \mid p_a), findall\, p_f\, among\, p_a])$$

An empty result is returned if no pattern is found. Because $p_f$ is applied to all items and because it can be any pattern including a transformation, $findall$ can be used to replace all occurrences of a pattern in a sequence. This is expressed by the following introduction of the alias $replaceall$:

$$replaceall\, p_f\, among\, p_a \implies findall\, p_f\, among\, p_a$$

For example, matching the pattern $replaceall\ (x\ \Rightarrow\ b)\ in\ \alpha$ with $[a, x, a, x]$ yields the result $[a, b, a, b]$.

### 3.5.3 Traversing Hierarchical Structures

The previous two versions of $find$ operate on a sequence in a horizontal manner, attempting to match the items contained in the sequence sequentially. The following pattern finds

occurrences of patterns in nested structures. It expects a third argument $p_i$ which is a pattern that defines which typed sequences are searched.

$$
\begin{aligned}
&findall\, p_f\, among\, p_a\, in\, p_t \Longrightarrow\ empty \mid \\
&\sim([\tau([x{:}(findall\, p_f\, among\, p_a\, in\, p_t)], p_t) \Rightarrow x{:}\alpha^*, \\
&\quad findall\, p_f\, among\, p_a\, in\, p_t]) \mid \\
&\sim([(p_f \mid p_a), findall\, p_f\, among\, p_a\, in\, p_t])
\end{aligned}
$$

Because pattern matching progresses from left to right and the search descends immediately when a typed sequence of type $p_t$ is found, the above rule defines a depth-first recursive descent search in which the search path is controlled through the patterns $p_a$ and $p_t$.

## 3.6 Meta-Patterns

Meta-patterns are patterns that match or instantiate other patterns. Recognising, creating and manipulating patterns using patterns is a key concept underlying this work. It is the foundation of partial instantiation (Section 3.6.5), pattern abstraction (Section 3.7) and pattern grammars (Section 4.4).

### 3.6.1 Pattern Representation

In order to make the definition of the meta functionality in this section independent of a particular pattern representation, a set of functions for constructing, deconstructing and type testing patterns will be used. By abstractly viewing patterns as terms, patterns that have arguments, e.g., the sequence pattern $\sim$, can be distinguished from patterns that do not have arguments, e.g., $\alpha$. The predicate $zeroary?$ yields true for the latter. The abstract structure of a pattern with arguments can be divided into an operator part and an argument part. The accessors $operator$ and $args$ yield the respective parts of a pattern as suggested by their names. The pattern constructor $pattern$ is used to build a pattern given an operator and arguments.

Chapter 5 discusses a meta-circular implementation of pattern matching that uses typed sequences and atoms to represent patterns. For this representation, the implementation of the functions just discussed is straightforward. The $zeroary?$ function corresponds to $atom?$ and disassembling a pattern into its operator and argument part consists of separating the type and the content part of a typed sequence.

### 3.6.2 Patterns on the Data Level

Meta-patterns match and instantiate patterns with patterns. Because the data consists of patterns it is impossible to refer to a pattern on the data level without invoking its matching and instantiation semantics. A mechanism is required to distinguish between "passive" patterns as data and "active" patterns that match or instantiate data. This mechanism is based on operators that explicitly declare patterns as data. The simplest of these operators is $data$.

Matching pattern $data(p)$ succeeds if the first element of the input sequence is a syntactically equal pattern, otherwise matching fails.

$$\frac{p_1 = p_2}{\langle data(p_1), p_2{::}s_{out}, \sigma_{in}\rangle \xrightarrow{m} \langle p_1, s_{out}, \sigma_{out}\rangle}\ \textsc{Data Success}$$

$$\frac{p_1 \neq p_2}{\langle data(p_1), p_2{::}s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Data} \bot$$

$$\frac{}{\langle data(p_1), \epsilon, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Data Empty} \bot$$

Instantiating pattern $data(p)$ appends $data(p)$ to the input sequence.

$$\frac{combine(s_{in}, data(p)) \mapsto s_{out}}{\langle data(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Data Instantiate}$$

It shall be noted that the instantiation semantics entail that the pattern remains explicitly marked as data.

### 3.6.3 From Data to Pattern Level

Declaring a pattern as data enables pattern manipulation without invoking usual pattern semantics. The data operator explicitly declares a pattern to be on the data level. Moving patterns from the "passive" data level to the "active" pattern level corresponds to removing the data wrapper. The critical question is when this transformation is performed. The

$i\text{-}activate$ operator removes the data declaration of the pattern during instantiation in such a way that the pattern is part of the result. The next time the pattern is matched or instantiated it is "active". Instantiating $i\text{-}activate$ fails if its pattern argument is not marked as data.

$$\frac{combine(s_{in}, p) \mapsto s_{out}}{\langle i\text{-}activate(data(p)), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Activate Inst Success}$$

$$\frac{operator(p) \neq data}{\langle i\text{-}activate(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Activate Inst} \bot$$

The matching semantics of $i\text{-}activate$ is failure.

$$\frac{}{\langle i\text{-}activate(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Activate Match} \bot$$

Declaring a pattern as data means that it remains data until it is explicitly activated. As the rule ACTIVATE SUCCESS suggests, $i\text{-}activate$ and $data$ can be combined. This is expressed by the following definition of quote:

$$quote(p) \Longrightarrow i\text{-}activate(data(p))$$

The instantiation result of $quote(p)$ is $p$, which means that the pattern is treated as "passive" data by the instantiation process but is "active" in the result and will thus have its usual pattern semantics when matched or instantiated.

### 3.6.4 Quasiquotation

Patterns are based on the idea of interspersing literal data with expressions of the pattern language. The mechanism introduced in the previous sections makes manipulation of patterns as data possible. However, there is no way to intersperse "passive" patterns with "active" patterns. In this section a quasiquotation mechanism [98] is introduced that allows the declaration of patterns as data with "active" sub-patterns. The quasiquote operator is $qq$ and the unquote operator is $uq$. The following rules define matching and instantiation semantics that allow *arbitrary levels of meta-manipulation*, i.e., patterns that

manipulate meta-patterns. This is possible because quasiquotation can be used to create, recognise and deconstruct arbitrary patterns – including quasiquoted patterns.

**Instantiation Semantics** The difference between the quasiquote operator $qq$ and the $quote$ operator introduced in the previous section is that the instantiation semantics of $qq$ is not simply to append the pattern to the result sequence. Instead, a search is performed inside the pattern for "active" parts to be instantiated. For example, if the pattern $qq(\tau([uq(x{:}\alpha), x{:}\alpha], list))$ is instantiated with a store that contains the binding $(x, b)$, the result is $\tau([b, x{:}\alpha], list)$. That is, the pattern is instantiated literally with exception of the variable surrounded by $uq$. This variable is treated as "active" and thus the value is inserted. Nesting quasiquotes allows the treatment of quasiquoted patterns as data and also the expression of different levels of "activeness". Surrounding the pattern above with $qq$ and instantiating it yields the pattern without changes to it as a result, i.e., no part of the data would be treated as an "active" pattern. A part of a quasiquoted pattern is activated if for every quasiquote surrounding it, there is a matching unquote. For example, instantiating $qq(qq(uq(\tau([uq(x{:}\alpha), x{:}\alpha], list))))$ yields the result $qq(uq(\tau([b, x{:}\alpha], list)))$.

The formalisation of quasiquote instantiation is based on a binary version of $qq$ that carries an additional integer argument used for counting the amount of quasiquotes surrounding a pattern. The count is incremented for each quasiquote and decremented for each unquote. A pattern becomes "active" if it is surrounded by an unquote and the count is 1. In this case. the pattern has its usual instantiation semantics. If the count is greater than 0 and there is no unquote, two cases are distinguished. If the pattern is atomic, it is instantiated as data, i.e., appended literally to the result sequence. In case of a compound pattern, the overall pattern is considered to be data. However, the data may contain more unquotes that activate some of its parts. To take these unquotes into account, all parts have to be traversed and individually instantiated.

Rule QQI FIRST defines the behaviour for the outermost quasiquotation. It instantiates the binary version of $qq$ with a count of 1.

$$\frac{\langle qq(1, p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} result}{\langle qq(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} result}\ \text{QQI FIRST}$$

Instantiating a $qq$ or $uq$ inside a $qq$ leads to incrementation or decrementation respectively. Quasiquotes that occur unquoted inside quasiquotes are added to the result. This makes it possible to instantiate patterns containing quasiquotes.

$$\frac{\begin{array}{c} n_1 = n + 1 \\ \langle qq(n_1, p), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \langle [r] \rangle \\ combine(s_{in}, qq(r)) \mapsto s_{out} \end{array}}{\langle qq(n, qq(p)), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{ QQI INCREMENT}$$

$$\frac{\begin{array}{c} n > 1 \\ n_1 = n - 1 \\ \langle qq(n_1, p), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \langle [r] \rangle \\ combine(s_{in}, uq(r)) \mapsto s_{out} \end{array}}{\langle qq(n, uq(p)), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{ QQI DECREMENT}$$

Pattern activation is performed when the quotation count is 1 and the pattern is surrounded by an unquote. In this case, the instantiation result is the result of instantiating the unquoted pattern.

$$\frac{\langle p, s_{in}, \sigma_{in} \rangle \xrightarrow{i} result}{\langle qq(1, uq(p)), s_{in}, \sigma_{in} \rangle \xrightarrow{i} result} \text{ QQI ACTIVATE}$$

The traversal of patterns is described on their abstract structure because this does not require separate rules for all types of patterns. A quasiquoted pattern with arity 0 cannot be further traversed and is added to the result sequence. Quoted patterns with $arity > 0$ are disassembled, each part is instantiated separately and the results are reassembled to form the instantiation result. This is expressed using the constructor and destructor functions described above.

$$\frac{\begin{array}{c} n > 0 \\ zeroary(p) \\ combine(s_{in}, p) \mapsto s_{out} \end{array}}{\langle qq(n, p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{ QQI 0-ARY}$$

$$\frac{\begin{array}{c} n > 0 \\ operator(p) \mapsto o \\ args(p) \mapsto C \\ o \neq qq \qquad o \neq uq \\ \langle qq(n, o), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \langle [o_i] \rangle \\ qqinst(C, \epsilon, \sigma_{in}, n) \mapsto C_i \\ pattern(o_i, C_i) \mapsto p_i \\ combine(s_{in}, p_i) \mapsto s_{out} \end{array}}{\langle qq(n, p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{ QQI N-ARY}$$

The entire instantiation process fails if instantiating either the operator or the argument part of the quasiquoted pattern fails.

$$\frac{\begin{array}{c} n > 0 \\ operator(p) \mapsto o \\ o \neq qq \qquad o \neq uq \\ \langle qq(n, o), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \bot \end{array}}{\langle qq(n, p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \bot} \text{ QQI OPERATOR } \bot$$

$$\frac{\begin{array}{c} n > 0 \\ operator(p) \mapsto o \\ o \neq qq \qquad o \neq uq \\ \langle qq(n, o), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \langle s \rangle \\ length(s) \neq 1 \end{array}}{\langle qq(n, p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \bot} \text{ QQI OPERATOR NO TERM } \bot$$

$$\frac{\begin{array}{c} n > 0 \\ operator(p) \mapsto o \\ args(p) \mapsto C \\ o \neq qq \qquad o \neq uq \\ qqinst(C, \epsilon, \sigma_{in}, n) \mapsto \bot \end{array}}{\langle qq(n,p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \bot} \text{QQI CONTENT } \bot$$

The instantiation of the arguments using the correct quasiquotation nesting level is defined by the function $qqinst$.

$$\frac{\begin{array}{c} \langle qq(n,c), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \langle r \rangle \\ combine(s_{in}, r) \mapsto s_i \\ qqinst(C, s_i, \sigma_{in}, n) \mapsto s_{out} \end{array}}{qqinst(c{::}C, s_{in}, \sigma_{in}, n) \mapsto s_{out}} \text{QQINST RECURSE}$$

$$\frac{}{qqinst(\epsilon, s_{in}, \sigma_{in}, n) \mapsto s_{in}} \text{QQINST EMPTY}$$

$$\frac{\langle qq(n,c), \epsilon, \sigma_{in} \rangle \xrightarrow{i} \bot}{qqinst(c{::}C, s_{in}, \sigma_{in}, n) \mapsto \bot} \text{QQINST } \bot$$

The instantiation semantics of an unquote outside of a quasiquote is to instantiate the argument of the unquote and to surround the result with an unquote. This behaviour enables patterns to create patterns containing unquotes.

$$\frac{\begin{array}{c} \langle p, \epsilon, \sigma_{in} \rangle \xrightarrow{i} \langle [r] \rangle \\ combine(s_{in}, uq(r)) \mapsto s_{out} \end{array}}{\langle uq(p), s_{in}, \sigma_{in} \rangle \xrightarrow{i} \langle s_{out} \rangle} \text{UNQUOTE SUCCESS}$$

$$\frac{\langle p, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \bot}{\langle uq(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Unquote Pattern}\ \bot$$

$$\frac{\begin{array}{c}\langle p, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_i\rangle \\ length(s_i) \neq 1\end{array}}{\langle uq(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Unquote NoTerm}\ \bot$$

Interestingly, the quasiquote mechanism can also be utilised to define the internal structure of an unquote. For example, instantiating the pattern $uq(x{:}\alpha)$ with a store that contains a binding $(x, b)$ yields $[uq(b)]$ while instantiating $uq(qq(x{:}\alpha))$ yields $[uq(x{:}\alpha)]$.

**Matching Semantics** Using patterns to match patterns requires a mechanism to declare which parts of the pattern-level pattern should match the data-level pattern literally, i.e., should be treated as data, and which parts should have their usual matching semantics. This can be expressed using quasiquotation. For example, the pattern $qq(\sim([\alpha, uq(\alpha)]))$ matches a sequential pattern that literally contains $\alpha$ followed by a second element that can be anything.

A quasiquote that occurs inside another quasiquote requires that the first element of the input is a quasiquote. For example, the only data the pattern $qq(\sim([\alpha, qq(\alpha)]))$ matches is $\sim([\alpha, qq(\alpha)])$, i.e., a sequential pattern that literally contains $qq(\alpha)$ as the second element. On the other hand, the pattern $qq(\sim([\alpha, qq(uq(\alpha))]))$ allows an arbitrary pattern inside the quasiquote of the second element. For instance, it matches $qq(\sim([\alpha, qq(uq(x{:}\alpha))]))$. Based on this semantics, a counting of nesting levels is not necessary as comparisons with the input are performed directly.

A pattern inside a quasiquote becomes "active", i.e., has its usual matching semantics, if it is surrounded by an unquote.

$$\frac{\langle p, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}{\langle qq(uq(p)), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \text{QQM } \textsc{Activate}$$

Quasiquoted patterns are matched with the first element of the input sequence. If the pattern is 0-ary, this element must be syntactically equal to the pattern. In case of a compound pattern, the first element of the input sequence also has to be a compound pattern. The op-

erators have to match literally and the content of the pattern-level pattern must match the contents of the data-level pattern. The rule QQM N-ARY defines disassembling and reassembling of the patterns.

$$\frac{zeroary(p)}{\langle qq(p), p{::}s_{out}, \sigma_{in}\rangle \xrightarrow{m} \langle p, s_{out}, \sigma_{in}\rangle}\ \text{QQM 0-ARY}$$

$$\frac{\begin{array}{c} operator(p) \mapsto o \\ operator(p_d) \mapsto o_d \\ args(p) \mapsto C \\ args(p_d) \mapsto C_d \\ \langle qq(o), [o_d], \sigma_{in}\rangle \xrightarrow{m} \langle o_m, \epsilon, \sigma_o\rangle \\ qqmatch(C, C_d, \sigma_o) \mapsto (C_m, \epsilon, \sigma_{out}) \\ pattern(o_m, C_m) \mapsto p_i \end{array}}{\langle qq(p), p_d{::}s_{out}, \sigma_{in}\rangle \xrightarrow{m} \langle p_i, s_{out}, \sigma_{out}\rangle}\ \text{QQM N-ARY}$$

Matching a quasiquoted pattern with an empty input sequence fails. If the first element in the sequence is a compound pattern, matching fails in case the operator and argument parts of the pattern-level pattern cannot be matched with those of the data-level pattern.

$$\frac{}{\langle qq(p), \epsilon, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \text{QQM EMPTY } \bot$$

$$\frac{\begin{array}{c} operator(p) \mapsto o \\ operator(p_d) \mapsto o_d \\ \langle all(qq(o)), [o_d], \sigma_{in}\rangle \xrightarrow{m} \bot \end{array}}{\langle qq(p), p_d{::}s_{out}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \text{QQM OPERATOR } \bot$$

$$\frac{\begin{array}{c} operator(p) \mapsto o \\ operator(p_d) \mapsto o_d \\ args(p) \mapsto C \\ args(p_d) \mapsto C_d \\ \langle qq(o), [o_d], \sigma_{in} \rangle \xrightarrow{m} \langle o_m, \epsilon, \sigma_o \rangle \\ qqmatch(C, C_d, \sigma_o) \mapsto \bot \end{array}}{\langle qq(p), p_d{::}s_{out}, \sigma_{in} \rangle \xrightarrow{m} \bot} \text{ QQM N-ARY}$$

The helper function $qqmatch$ matches the content of the pattern-level pattern with the content of the data-level pattern. For example, matching $qq(\sim([\alpha, uq(ign(x{:}\alpha^*))]))$ with input $[\sim([\alpha, y, z])]$ results in a call to $qqmatch$ with $C = [\alpha, uq(x{:}\alpha^*)]$ and $C_d = [\alpha, y, z]$, as defined by rule QQM N-ARY. In $qqmatch$, the pattern $qq(\alpha)$ is matched with $[\alpha]$, which results in a literal comparison according to rule QQM 0-ARY. Next, $qq(uq(ign(x{:}\alpha^*)))$ is matched with $[y, z]$. The surrounding unquote activates the ignore-pattern as defined by rule QQM ACTIVATE. This leads to a match of $ign(x{:}\alpha^*)$ with $[y, z]$. The matching consumes $y$ and $z$, creates a binding $(x, [y, z])$. Because of the surrounding $ign$, an empty sequence is the result. The overall outcome of $qqmatch$ is an empty output sequence, a store updated with the binding and the result $[\alpha]$. From the result of $qqmatch$, the pattern $\sim([\alpha])$ is assembled as defined by rule QQM N-ARY. This pattern is the overall matching result.

$$\frac{\begin{array}{c} \langle qq(c), s_{in}, \sigma_{in} \rangle \xrightarrow{m} \langle r_c, s_c, \sigma_c \rangle \\ qqmatch(C, s_c, \sigma_c) \mapsto (r_C, s_{out}, \sigma_{out}) \\ combine(r_c, r_C) \mapsto r \end{array}}{qqmatch(c{::}C, s_{in}, \sigma_{in}) \mapsto (r, s_{out}, \sigma_{out})} \text{ QQMATCH RECURSE}$$

$$\frac{}{qqmatch(\epsilon, s_{in}, \sigma_{in}, n\rangle \mapsto \langle \epsilon, s_{in}, \sigma_{in} \rangle} \text{ QQMATCH EMPTY}$$

Matching a pattern $uq(p)$ succeeds if the first element of the input has the form $uq(p_d)$ and $p$ matches $[p_d]$. The result of this match is wrapped into an unquote. This behaviour makes it possible to use quasiquote and unquote to match patterns that contain quasiquotes and unquotes.

For example, in the pattern $qq(uq(b))$, the atom $b$ is surrounded by one quasiquote and one unquote. The pattern $qq(uq(x{:}\alpha))$ binds $x$ to $qq(uq(b))$ because the effect of

the quasiquote is neutralized by the unquote. However, pattern $qq(qq(uq(x{:}\alpha)))$ binds $x$ to $uq(b)$ because the inner $qq$ is matched with the $qq$ on the data level and the unquote activates the variable. To create a binding from $x$ to $b$, the pattern $qq(qq(uq(uq(x{:}\alpha))))$ adds another unquote. The outer unquote is matched with the unquote on the data level, as define by rule UQM SUCCESS. The inner unquote activates the variable.

$$\frac{\langle p, [p_d], \sigma_{in}\rangle \xrightarrow{m} \langle r, \epsilon, \sigma_{out}\rangle}{\langle uq(p), uq(p_d){::}s_{out}, \sigma_{in}\rangle \xrightarrow{m} \langle uq(r), s_{out}, \sigma_{out}\rangle}\ \text{UQM SUCCESS}$$

$$\frac{\langle all(p), [p_d], \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle uq(p), uq(p_d){::}s, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \text{UQM PATTERN } \bot$$

$$\frac{operator(e) \neq uq}{\langle uq(p), e{::}s, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \text{UQM OPERATOR } \bot$$

$$\frac{}{\langle uq(p), \epsilon, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \text{UQM EMPTY}\bot$$

Combined with the transformations introduced in Section 3.4, the quasiquotation mechanism allows the definition of arbitrary transformations on patterns. Because transformations are patterns as well and quasiquotation works on quasiquoted patterns, these meta-transformations can be transformed by meta-meta-transformations. An arbitrary number of meta-levels can be defined. Meta-transformations are used in Section 3.7 to implement an abstraction mechanism for patterns.

### 3.6.5 Partial Instantiation and Pattern Refinement

The instantiation semantics of variables introduced in Section 3.2 define failure in case that a variable is unbound in the store. Altering this semantics in such a way that the unbound variable itself is the result gives rise to the notion of partial instantiation of patterns. The modified semantics replace the rule VARIABLE UNBOUND $\bot$ with the following rule:

$$\frac{\sigma_{in}[n] = \bot \qquad combine(s_{in}, n{:}p) \mapsto s_{out}}{\langle n{:}p, s_{in}, \sigma_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Partial Instantiation}$$

Partial instantiation can be used to *refine* a pattern in a step-wise fashion by replacing variables with concrete values in each step. An example of partial instantiation that refines a pattern is the following specialisation of a general typed sequence pattern to one that matches only lists:

$$\langle \tau([c{:}\alpha^*], t{:}type), \epsilon, \{(t, list)\}\rangle \xrightarrow{i} \langle \tau([c{:}\alpha^*], list)\rangle$$

The result of instantiating a pattern with a store that does not include bindings for all variables results in a pattern on the data level. This means that partial instantiation is a meta-operation.

Partial instantiation can be used in combination with the path polymorphic patterns defined in Section 3.5 to query structures. The idea is to define general search patterns and to refine these patterns before a query using bindings that may be the result of previous matches. For example, the general query $find\ \tau([c{:}\alpha^*], t{:}type)\ among\ ign(\alpha)$ that finds all typed sequences in a sequence can be restricted to the more specific query $find\ \tau([c{:}\alpha^*], list)\ among\ ign(\alpha)$ that finds only lists. This can be achieved using partial instantiation as shown in the example above. Similarly, refinement of the "among" pattern and the "in" pattern (of the hierarchical $find$ operator) can be used to refine the search path of a query. These techniques are used extensively by the query and constraint language of XMF, see Chapter 6.

## 3.7 Pattern Abstraction

The operational semantics in this chapter formalise a set of core pattern abstractions. Pattern expressions can be created by combining these abstractions. So far, the only way for abstracting these pattern expressions, i.e., to hide their implementation and make them appear as built-in patterns, is through a meta-language that is different from the pattern formalism. This section extends the pattern formalism with a mechanism for pattern abstraction.

### 3.7.1 References

References provide a simple form of pattern abstraction: a name is associated with a pattern in such a way that the occurrence of the name is resolved to the pattern before matching or instantiation. An example is the pattern $empty$ which was defined in Section 3.4 as a reference to the pattern $!\alpha$.

There are several ways to implement references in the pattern formalism. One way is to define the mapping between a name and a pattern as part of the operational semantics: every reference is defined as an inference rule with a conclusion $\langle name, s, \sigma\rangle \xrightarrow{m/i} result$ and a single premise $\langle pattern, s, \sigma\rangle \xrightarrow{m/i} result$. This means that a name is replaced with a pattern by an operational rule of the system. Using this technique implies that the concept of reference is not part of the formalism. Adding new references means extending the formalism.

An alternative is the use of a meta-language to define references directly as transformations. Above, the meta-language operator "$\Longrightarrow$" was used for this purpose. The meta-language expression $name \Longrightarrow pattern$ means that an occurrence of $name$ can be rewritten to $pattern$. The extensibility mechanism defined in this section makes references explicit in the pattern formalism so that no external meta-language is needed – the pattern formalism *is* the meta-language. This means that the formalism can be extended from within. Reference resolution is a pattern-based meta-transformation with the left-hand side naming the pattern on the right-hand side.

The meta-functionality introduced in Section 3.6 enables transformations of patterns using patterns. Such meta-transformations are a suitable basis for implementing mappings from pattern references to patterns. For instance, the meta-transformation

$$empty \Rightarrow quote(!\alpha)$$

implements the above example using the quotation functionality described in the previous section. The left-hand side matches the atom $empty$ and the right-hand side produces the pattern $!\alpha$.

### 3.7.2 Statically Parameterised References

Formalising references as transformations provides not only the advantage that the reference resolution is defined by means of the pattern formalism but also that, in addition to simple name to pattern mappings, more powerful pattern abstractions can be defined. Parameterised references allow the passing of arguments during reference resolution. The arguments are used as values to instantiate variables in the meta-pattern on the right hand

side of the transformation. For example, the pattern operator $?$ which *tries* to match its argument can be defined as an unconditional transformation on a reference that consists of the operator name and a variable:

$$qq((uq(p{:}\alpha))?) \Rightarrow qq(or([uq(p{:}\alpha), {\sim}(\epsilon)]))$$

A reference to this pattern is $ref((b)?)$ which is transformed into $or([b, {\sim}(\epsilon)])$. The unquote on the right-hand side leads to the instantiation of the variable during the transformation. Otherwise, the variable itself would be part of the transformation result. The arguments passed to a parameterised reference are static in the sense that the argument values are contained literally in the pattern expression. The semantics of the $ref$ operator will be defined in the next subsection.

### 3.7.3 Dynamically Parameterised References

Dynamically parameterised references contain variables that are instantiated in the context of the current store before resolution. This enables values created from a previous match to be used as reference arguments. For example, matching the reference containing pattern ${\sim}([f{:}\alpha, ref(rest(uq(f{:}\alpha)))$ first yields a binding for the variable $f$ which is then passed to the referenced pattern $rest$. The definitions in this subsection not only formalise dynamically parameterised references but also parameterless and statically parametrised references as these are special cases of the former.

The $ref$ operator surrounds its operand with a quasiquote before instantiating it. Parts surrounded with an unquote, such as the variable in the example above, are "activated" which means that variables are replaced with values bound in the current store. This implements the dynamic parametrisation. The actual reference resolution is performed by a meta-pattern that defines meta-transformations from references to patterns. Such a pattern can be constructed from individual meta-transformations using $choice$. It has the form $or([m_1, ..., m_n])$ where $m_1, ..., m_n$ are meta-transformations such as those for $?$ and $empty$ defined above. Creating and manipulating this pattern, e.g., in order to add new references, can be expressed using meta-meta-transformations. The actual process of reference definition and manipulation is, however, outside the scope of this formalisation.

To be able to define reference resolution, let the pattern $p_{refs}$ refer to the meta-pattern that defines how to resolve references. This pattern is matched with the result of instantiating the reference; the matching result is the outcome of the resolution process. It is matched with the input sequence to produce the overall result of matching the reference.

$$\frac{\begin{array}{c}\langle qq(p), \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_i\rangle \\ \langle all(p_{refs}), s_i, \sigma_{in}\rangle \xrightarrow{m} \langle r_{ref}, \epsilon, \sigma_{ref}\rangle \\ \langle r_{ref}, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result\end{array}}{\langle ref(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \textsc{Reference Success}$$

$$\frac{\langle qq(p), \epsilon, \sigma\rangle \xrightarrow{i} \bot}{\langle ref(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Reference Dynamic} \bot$$

$$\frac{\begin{array}{c}\langle qq(p), \epsilon, \sigma\rangle \xrightarrow{i} \langle s_i\rangle \\ \langle all(p_{refs}), s_i, \sigma_{in}\rangle \xrightarrow{m} \bot\end{array}}{\langle ref(p), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Reference Resolve} \bot$$

The instantiation semantics of references are very similar to the matching semantics. The actual reference resolution follows the same steps as in the matching case, the only difference is that the result of resolving the reference is instantiated rather than matched.

$$\frac{\begin{array}{c}\langle qq(p), \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_i\rangle \\ \langle all(p_{refs}), s_i, \sigma_{in}\rangle \xrightarrow{m} \langle r_{ref}, \epsilon, \sigma_{ref}\rangle \\ \langle r_{ref}, s_{in}, \sigma_{in}\rangle \xrightarrow{i} result\end{array}}{\langle ref(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} result}\ \textsc{Reference Inst}$$

$$\frac{\langle qq(p), \epsilon, \sigma\rangle \xrightarrow{i} \bot}{\langle ref(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Reference Inst Dynamic} \bot$$

$$\frac{\begin{array}{c}\langle qq(p), \epsilon, \sigma\rangle \xrightarrow{i} \langle s_i\rangle \\ \langle all(p_{refs}), s_i, \sigma_{in}\rangle \xrightarrow{m} \bot\end{array}}{\langle ref(p), s_{in}, \sigma_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Reference Inst Resolve} \bot$$

In the next chapter references will be used extensively to define productions of grammars. Chapter 5 provides examples of using dynamically parameterised references to avoid left-recursive definitions.

## 3.8 Summary and Conclusions

This chapter formalised recognising, creating, transforming and searching structures. The formalisation is based on elementary patterns and means to horizontally, vertically and diagonally combine them. In conjunction with transformations, diagonal combination is an important step in the direction of pattern-based computing. It gives rise to a staged processing of data, where each stage is defined by a pattern. Typed sequence patterns allow manipulation of arbitrary tree structures. Path polymorphic operators give fine-grained control for traversing structures. Meta-patterns apply the pattern functionality self-referentially. This is facilitated by a quotation and quasiquotation mechanism that separates between "active" and "passive" patterns.

Interestingly, the quasiquotation mechanism can be used to match and instantiate data-level patterns that contain quasiquotes and unquotes. This is the key to arbitrary levels of meta-functionality. One application of meta-patterns is pattern abstraction. Meta-transformations are used for resolving references. Dynamic references define a kind of "function call" mechanism for patterns where parameter passing and variable substitution in the body are defined entirely through matching and instantiation.

The functionality defined so far provides a versatile and extensible framework for structural manipulation. With regards to the hypothesis, the definition of an operational semantics ensures that the pattern approach provides a solid formal foundation. The aim of the next chapter is to utilise this foundation to show that "the systematic creation and layering of languages can be reduced to the elementary operations of pattern matching and instantiation".

# Chapter 4

# Towards Pattern-based (Meta-)Programming

This chapter provides a formal foundation for programming and meta-programming based on the pattern core introduced in the previous chapter. The first step in this direction is the definition of rewriting systems. A rewriting system consists of transformations and a strategy for applying these transformations – both are expressed as patterns. The path polymorphic $find$ operator plays a key role in the formalisation of subterm rewriting. Different kinds of rewriting systems can be distinguished based on strategies and the way they constrain transformations. Purely concatenative rewriting systems restrict transformations to mappings from names to programs with the goal of giving transformations a purely functional semantics. Less restrictive pattern-based definitions combine the benefits of functional semantics with powerful means for defining functions (Section 4.2). Based on the foundation of rewriting systems, the notion of pattern-based computing is formalised in such a way that the pattern system has the ability to invoke itself. Self-invocation is utilised by conditional rules whose results depend on the execution of dynamically created programs. Temporal views define a mechanism to hide certain computational steps of a rewriting system (Section 4.3). Pattern-based (un-)parsing is based on a unified representation of character strings and syntax trees as typed sequences. Grammars are defined by a set of meta-transformations (Section 4.4). Individual stages of parsing, computation and unparsing define staged processing of programs. Staged processing is formalised using vertical combination. Structural views allow the introduction of arbitrary layers of program representation beyond the syntactic limitations of typed sequences (Section 4.5). Together, structural and temporal views are the formal foundation for "illusionising" execution models. Finally, elliptical patterns are a practical extension for programs that transform repetitive structures (Section 4.6).

## 4.1 Motivation

The previous chapter formalised fundamental operations on structures in terms of pattern matching and instantiation. It also defined a meta-functionality and an abstraction mechanism for patterns. How can these concepts be used to formally define a programming and meta-programming system? Building a programming language based on the pattern formalism poses more detailed questions. How does the character-based syntax of a language relate to the typed sequence data representation? What is the execution model of the language and how is this execution model implemented through pattern matching and instantiation? What infrastructure does the language provide and how are programs written? A meta-programming system demands that these questions do not have fixed answers but that syntax and execution mechanisms are adjustable. The following sections provide the necessary operators, techniques and infrastructure to define languages using patterns. The result is the formal foundation of the meta-programming system Concat that will be introduced in the next chapter.

## 4.2 Pattern-based Rewriting Systems

Rewriting systems are generic tools for describing the stepwise, formal manipulation of structures. A rewriting system consists of a set of formal rules that define how (part of) a given structure is reduced to a new structure. The application of rules is defined by strategies. The goal is to derive a normal form, i.e., a structure that cannot be reduced any further. A rewriting system can be used to formalise the operation of a programming system: the structure manipulated by the rewriting process is the program; the rewriting rules and their application strategy define the interpreter for the program [40]. The aim of this section is to show how such interpreters can be defined with the patterns already introduced and how different strategies and rule restrictions lead to different programming models.

Based on the types of structures they process, rewriting systems can be classified into string, graph or term rewriting systems [5, 12]. The rewriting systems that will be defined in this section rewrite typed sequences. Section 4.4 will show how strings can be represented as typed sequences in order to provide a uniform notation enabling rewrite rules to be defined on strings. However, this is just a special case and in general the hierarchical semantics of typed sequence patterns allows rewriting of structured data. Typed sequences can be viewed as terms constructed by the binary operator $\tau$ and the operator :: over the set of atoms and $\epsilon$. In this sense, the rewriting systems that will be introduced below can be seen as term rewriting systems.

### 4.2.1 From Transformations to Rewriting Systems

The pattern-based transformations defined in the previous chapter are a powerful means for defining a single step of restructuring data. A key design decision is that unconditional transformations are not only based on pattern expressions but are themselves pattern expressions. The closure property of pattern composition entails that arbitrary compound patterns can be built from transformations.

For instance, the pattern $p_{l_1} \Rightarrow p_{r_1} \mid ... \mid p_{l_n} \Rightarrow p_{r_n}$ expresses that there are several alternative transformations. The left-to-right semantics of the $choice$ operator defines a clear order in which the transformations are applied. A pattern of the form $(p_l \Rightarrow p_r)^{*v}$ uses the vertical repetition operator to express repeated application of transformations. Combining both choice and repetition and assuring that the entire sequence is rewritten yields patterns that have the following schema:

$$all((p_{l_1} \Rightarrow p_{r_1} \mid ... \mid p_{l_n} \Rightarrow p_{r_n})^{*v})$$

Such patterns define rewriting system: rules are transformations and the strategy is defined by the composition of prioritised choice, vertical repetition and the $all$ operator. The strategy consists of trying the rules in the order in which they appear, applying the first rule that matches. After successful application this process is repeated on the results. Repetition terminates if no more rules apply.

An example for the definition of a rewriting system is the pattern $all((a \Rightarrow b \mid b \Rightarrow c)^{*v})$ in which $a$, $b$ and $c$ are atoms. When matching the pattern with input $[a]$, the first transformation is applied and rewrites the input to the result $[b]$. Next, the vertical repetition operator matches this result with the choice operator, which for this input applies the second transformation that rewrites $[b]$ to $[c]$. After that, rewriting terminates as none of the rules matches $[c]$.

The algorithm in Listing 1 illustrates the basic rewriting strategy expressed by the above patterns. The function `rewrite` transforms a term by repeatedly applying rules until no more rule can be applied. The rules are applied in the order in which they appear in the list `rules`. The repeated application of rules is expressed by the outer do-while-loop starting in line 3; attempting the rules one by one is expressed by the inner for-loop starting in line 5. The outcome of a rule application (represented by variable `aout`) consists of a result part and a rest part. A successful rewriting step was performed if the result part of `aout` is not `failed` and if the rest part is empty. This is expressed by the conditional part of the if-statement starting in line 7.

**Listing 1** Basic Rewriting System Algorithm

```
1  rewrite(rules, term_in) -> term_out
2    term = term_in
3    do
4      success = false
5      for(i=0; i<rules.count; i++)
6        aout = apply(rules[i], term)
7        if(aout != failed && aout.rest == empty)
8          term = aout.result
9          success=true
10         break
11   while (success)
12   return term
```

### 4.2.2 Subterm Rewriting Strategies

The rewriting semantics just defined require that transformations rewrite the entire input sequence. Subterm rewriting allows transformations that define rewriting on just a part of the sequence [5]. If $lxr$ is a sequence consisting of subsequences $l$, $x$ and $r$ and there is a transformation $x \rightarrow y$, the transformation $lxr \rightarrow lyr$ is applied. In other words, the *context* of the subsequence affected by the transformation is automatically added to the result.

Applying a transformation to a subsequence requires finding a reducible subsequence, applying the transformation and combining the result with the subsequences to the left and to the right of the result. Part of this can be achieved using the diagonal operator:

$$(\Delta(p_{l_1} \Rightarrow p_{r_1} \mid ... \mid p_{l_n} \Rightarrow p_{r_n}))^*$$

Systems that follow this schema are a step in the right direction. Rewriting subsequences is supported as the diagonal operator combines the remaining sequence, i.e., the subsequence to the right of the replaced sequence, with the result. Because the compound result is returned as the output sequence, repeated application is expressed using the horizontal repetition operator. For example, the system $(\Delta(a \Rightarrow b \mid b \Rightarrow c))^*$ rewrites the input $[a, b, c]$ to the result $[c, b, c]$. However, this strategy is restrictive as it demands that subsequences matched by the left-hand side of a transformation start at the beginning of the input sequence. Therefore, only data of the form $xr$ with context on the right can be rewritten using rule $x \rightarrow y$. The general case $lxr$, with context to the left and to the right, can be defined by applying the $find$ operator of Section 3.5.1:

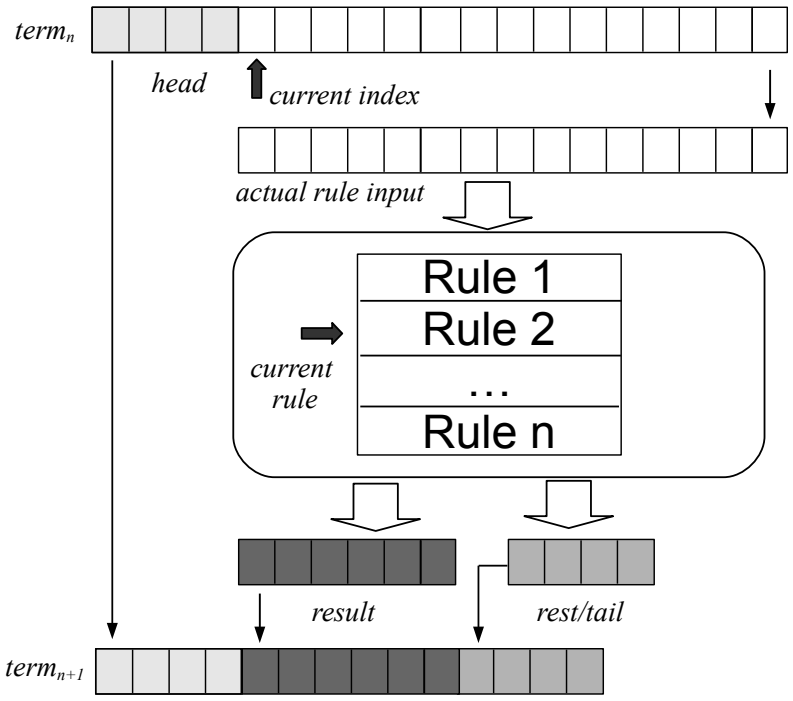


Figure 4.1: Illustration of the Abstract Machine's Rule Core

$$(\Delta find\,(p_{l_1} \Rightarrow p_{r_1} \mid ... \mid p_{l_n} \Rightarrow p_{r_n})\ among\ \alpha)^*$$

The strategy expressed through this pattern is to start from the very left in the input sequence and to try applying the transformations in the order in which they appear in the choice pattern. If none of the transformation can be applied, the same is tried on the input sequence with the first element removed. In case of a successful transformation, $find$ combines the result with the subsequence to the left of the reducible sequence, i.e., the part that was skipped when no transformation matched. This result is then combined *diagonally* with the rest of the input sequence unaffected by the transformation.

For instance, matching $(\Delta find(a \Rightarrow b \mid b \Rightarrow c)\ among\ \alpha)^*$ with $[c, a, b]$ attempts matching the first transformation with $[c, a, b]$, which fails and causes $find$ to move on to $[a, b]$. Next, transformation $a \Rightarrow b$ rewrites subsequence $[a]$ to $[b]$. The context to the left, $[c]$, is added to the result by $find$ which returns $[c, b]$. This result is combined with the right context $[b]$ to form the output sequence $[c, b, b]$. Because a successful rewriting step was performed, the process is repeated for the updated sequence. The next step yields $[c, c, b]$ and the final step results in $[c, c, c]$.

Figure 4.1 illustrates the concept of subterm rewriting as the operation of an abstract term rewriting machine. It shows a single rewriting step where $term_n$ is rewritten

to $term_{n+1}$. The *current index* points to the subterm on which the rules are attempted; *current rule* points to the particular rule that is attempted. The *current rule* pointer is incremented when the application of a rule fails. The *current index* pointer is incremented when the application of all rules fails. Rewriting terminates when the *current index* runs out of the bounds defined by the input term. The overall result of the rule application is constructed from the heading subterm, the result of applying a rule on the subterm following it and the part of that subterm which was not affected by the rule application.

The algorithm in Listing 2 extends that in Listing 1 and details the operation of the abstract machine. Fundamental to the algorithm is the division of the input term into three parts: the subterm to which the rule is applied as well as the subterms leading and trailing it. An index variable is used to keep track of the leading subterm; the function `range` is used to get a subterm based on start and end indices. The trailing subterm is the rest part of the outcome of apply (represented by `aout.rest`). The if-statement starting in line 15 increases the index variable in case the index is smaller than the length of the term. To produce the result of a rewriting step the three subterms are appended, as expressed in lines 10 and 11.

**Listing 2** Subterm Rewriting System Algorithm

```
1  rewrite(rules,term_in) -> term_out
2    term = term_in
3    index = 0
4    do
5      success = false
6      for(i=0; i<rules.count; i++)
7        aout = apply(rules[i],
8                     term.range(index,term.length))
9        if(aout!=failed)
10         lterm = append(term.range(0,index), aout.result)
11         term = append(lterm, aout.rest)
12         index=0
13         success=true
14         break
15     if(aout == failed && index < term.length)
16       index++
17       success = true
18   while (success)
19   return term
```

### 4.2.3 Purely Concatenative Rewriting Systems

The rewriting systems defined in the previous section allows arbitrary data sequences to be rewritten by arbitrary transformations. This section explains how restrictions on data and transformations can be used to implement a concatenative programming system. As discussed in Chapter 1, the execution state of a concatenative program can be represented as a program so that the entire program execution can be described by a sequence of programs. This makes concatenative programs particularly well suited for rewriting.

The basic distinction of program elements is between items (literals and quotations) and operators. A definition of the execution semantics using the pattern formalism requires that items and operators are structurally distinguishable. This requirement can be fulfilled by representing both operators and items as typed sequences. Operators have the *operator* type and all typed sequences with different types are items as expressed by the following meta-transformations:

$$op \Rightarrow qq(\tau([ref(name)], operator))$$

$$item \Rightarrow qq(\tau([\alpha^*], !operator))$$

The reference $name$ resolves to a pattern that recognises operator names. An example of an operator is $\tau([swap], operator)$. The above rules define the set of possible operators and items. A concrete concatenative rewriting system can restrict this set to exactly those operators and items available.

Programs are sequences of items and operators. This is defined by the following pattern:

$$program \Rightarrow (ref(item)|ref(operator))^*$$

An item that is of particular importance in concatenative programming is the quotation that serves as a list structure and program representation at the same time:

$$quot \Rightarrow qq(\tau([ref(program)], quotation))$$

To avoid cluttering the presentation, the internal structure of operators and items will be hidden wherever it is irrelevant to the discussion and where it is clear that an element is an item or an operator. For instance, instead of $\tau([swap], operator)$ and $\tau([2, 3], number)$ the representation $swap$ and *23* will be used. In Section 4.5.3, a mechanism to define such structural hiding will be introduced.

Program execution of a concatenative program is performed from left to right. Items remain in place between rewriting steps unless an operator is applied to them. Hence,

executing a program such as `1 2 3 swap dup` from left to right involves scanning over the numbers and then applying the first operation – in this case `swap`. Scanning over items can be expressed with a modification of the more general pattern for subterm rewriting defined in the previous section.

$$(\Delta find\,(p_{l_1} \Rightarrow p_{r_1}\ |...|\ p_{l_n} \Rightarrow p_{r_n})\ among\ ref(item))^*$$

The search pattern $find$ is configured with $ref(item)$ instead of $\alpha$ which has the effect that only items are scanned over during matching. A pure concatenative programming system is defined by transformations that have an operator on the left and a program on the right. Such transformations can be defined by a meta-pattern:

$$qq(uq(ref(operator)) \Rightarrow uq(ref(program)))$$

The definition implies that both the left- and right-hand sides of the transformation are concatenative programs without variables; transformations are context-free substitutions of a single element program with another program. Primitive programs, i.e., built-in operators such as $dup$ and $swap$, perform elementary data restructuring. Each transformation defines a new operator by mapping it to a program. Program execution consists of replacing non-primitives with definitions until a primitive is reached. For example, the program $[2, square]$ is rewritten into $[2, dup, *]$ by application of the rule $square \Rightarrow\ \sim([dup, *])$.

### 4.2.4 Concatenative Rewriting with Patterns

In a system restricting transformations less severely than the purely concatenative system above, primitive concatenative words can be defined as transformations. For example, the definition of the $dup$ operator is:

$$\sim([x{:}ref(item), dup]) \Rightarrow\ \sim([x{:}ref(item), x{:}ref(item)])$$

Although this transformation does not follow the strict rule that the only element on the left-hand side of a transformation has to be an operation, it still defines a functional semantics for $dup$. The transformation only affects items appearing to the left of $dup$ in the program.

In general, transformations that allow patterns different from operators and items and at the same time preserve the functional property are restricted in the following way: the left-hand side of the transformation contains only a single operator that needs to be the rightmost element. The remaining patterns on the left-hand side must only match items appearing before that operator in the program. The right-hand side is unrestricted. Such

transformations have the form

$$p_l \, operator \Rightarrow p_r$$

where $p_r$ is any pattern and $p_l$ is a pattern for which the following condition holds for all sequences $s$:

$$\langle p_l, s, \sigma_1 \rangle \xrightarrow{m} \langle r, \epsilon, \sigma_2 \rangle \vdash \langle find \, operator \, among \, \alpha, s, \sigma_1 \rangle \xrightarrow{m} \bot$$

This condition expresses that $p_l$ does not match any sequences containing operators. Allowing operator definitions of this form not only for primitive operators but also for user-defined operators yields a paradigm combining the execution properties of concatenative systems with the expressive power of pattern matching.

The transformation $\sim([dup, drop]) \Rightarrow \sim(\epsilon)$ violates the restrictions by having two operators on the left-hand side. This implies that either the result of $drop$ depends on the operator to its left or that the result of $dup$ depends on the operator on its right. Both interpretations are non-functional. While illegal on the "regular" programming level, such transformations are useful for expressing program transformations explicit. In Concat, they are defined as macros.

Pattern-based manipulation of operators is possible if the operators are contained inside quotations. Quotations provide means to manipulate programs as data. A call mechanism allows execution of quoted programs. For example, the concatenative program $[2, 3, \tau([swap, dup], quotation), rest, call]$ is executed by first applying $rest$ which removes the first element of the quotation. This code manipulation results in program $[2, 3, \tau([dup], quotation), call]$. Next, $call$ executes the content of the quotation, which yields the program $[2, 3, dup]$. The final result after applying $dup$ is $[2, 3, 3]$.

The concatenative operators $rest$ and $call$ can be implemented by transformations which manipulate quotations [170]. The transformation

$$\sim([\tau([\alpha, r{:}\alpha^*], quotation), rest]) \Rightarrow \tau([r{:}\alpha^*], quotation)$$

binds $r$ to a sequence containing all but the first element of the matched quotation; it instantiates $r$ inside a quotation which yields a new quotation without the first element. The transformation

$$\sim([\tau([p{:}\alpha^*], quotation), call]) \Rightarrow p{:}\alpha^*$$

binds $p$ to all elements in a quotation and instantiates $p$ outside of a quotation context. This "unquoting" leads to the execution of the program that was inside the quotation.

Like all concatenative operations, $call$ has a functional semantics: it maps data con-

taining a quotation to data that is the result of executing the program contained in the quotation. However, based on the above definition, the fine-grained steps of the rewriting system make it visible that $call$ is actually implemented through program transformation. This violates the functional principle. To maintain the functional semantics of $call$, its implementation details have to be hidden. Section 4.5.6 will define a mechanism to achieve this: temporal views allow "black-boxing" certain operations by hiding steps in the course of program execution.

## 4.3 Computing with Patterns

The rewriting systems introduced in the previous section give rise to a notion of pattern-based computing where the data sequence is a program and the patterns define an interpreter for that program. The functionality of the interpreter depends on transformations defining single computation steps and patterns specifying a strategy for performing these steps. Constraints on transformations and the application strategy are the basis of different computational models, as was demonstrated by two variants of concatenative systems. In this section, pattern-based computing will be formalised and based on this foundation conditional transformations will be introduced.

### 4.3.1 Formalising Pattern-based Computing

A rewriting system is defined by a pattern $p_{rws}$ that when matched performs manipulation of an input sequence. The operator $\phi$ defines matching of $p_{rws}$.

$$\frac{\langle p_{rws}, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}{\langle \phi, s_{in}, \sigma_{in}\rangle \xrightarrow{m} result}\ \textsc{Compute}$$

The instantiation semantics of $\phi$ is failure. A rewriting system can create an instance of itself if $\phi$ appears in a transformation. Computations may invoke other computations by creating programs, computing the results of these programs and matching patterns with the result. Conditional transformations are based on these techniques.

### 4.3.2 Conditional Transformations

Conditional transformations utilise the computational mechanism described in the previous subsection. They resemble a style of definition that is reminiscent of inference rules such as those used in this and the previous chapter to define the operational semantics of

matching and instantiation. When used in rewriting systems, conditional transformations provide a foundation for conditional rewriting. The basic syntactic structure of conditional transformations is $p_l \Rightarrow p_r \Longleftarrow c_1 \ldots c_n$. This structure corresponds to inference rules of the form:

$$\frac{c_1 \ldots c_n}{p_l \rightarrow p_r}$$

The result of a conditional transformation depends not only on the result of matching and instantiating two patterns, but also on the result of computing programs constructed through pattern instantiation. The execution semantics of conditionals is based on the repeated creation and computation of programs and on the matching of program results.

For example, the following conditional rule increments a number contained in a list: $(\sim([\tau([x{:}\alpha], list), linc]) \Rightarrow \tau([y{:}\alpha], list)) \Longleftarrow (\sim([x{:}\alpha, inc]) \Rightarrow y{:}\alpha)$. Assuming that there is a transformation rewriting $[2, inc]$ to $[3]$, matching the conditional transformation just defined with $\tau([2], list)$ yields $\tau([3], list)$. This result is produced by first matching the left-hand side $\tau([y{:}\alpha], list)$ of the transformation with the input sequence. This creates the binding $(x, 2)$. In the context of this binding, the left-hand side $\sim([x{:}\alpha, inc])$ of the first and only conditional is instantiated producing the result $[2, inc]$. The rewriting system is invoked with this result as input and produces $[3]$. Matching the right-hand side of the conditional $y{:}\alpha$ with this computation result produces the binding $(y, 3)$. In the context of this binding the right-hand side $\tau([y{:}\alpha], list)$ of the transformation is instantiated and produces the final result $\tau([3], list)$.

The transformation in the example contains a single conditional. In general, a transformation may contain an arbitrary number of conditionals. Given input sequence $s_0$ and state $\sigma_0$, conditional transformations of the form

$$h_l \Rightarrow h_r \Longleftarrow c_{l_1} \Rightarrow c_{r_1}, c_{l_2} \Rightarrow c_{r_2}, ..., c_{l_n} \Rightarrow c_{r_n}$$

produce result $r$, output sequence $s_1$ and state $\sigma_0$ following the scheme shown in Table 4.1.

$$\begin{array}{l}
\langle h_l, s_0, \sigma_0\rangle \xrightarrow{m} \langle r_0, s_1, \sigma_1\rangle, \\
\langle c_{l_1}, \epsilon, \sigma_1\rangle \xrightarrow{i} \langle p_1\rangle, \langle \phi, p_1, \sigma_1\rangle \xrightarrow{m} \langle \epsilon, e_1, \sigma_{c_1}\rangle, \langle c_{r_1}, e_1, \sigma_1\rangle \xrightarrow{m} \langle r_1, \epsilon, \sigma_2\rangle, \\
\langle c_{l_2}, \epsilon, \sigma_2\rangle \xrightarrow{i} \langle p_2\rangle, \langle \phi, p_2, \sigma_2\rangle \xrightarrow{m} \langle \epsilon, e_2, \sigma_{c_2}\rangle, \langle c_{r_2}, e_2, \sigma_2\rangle \xrightarrow{m} \langle r_2, \epsilon, \sigma_3\rangle, \\
\ldots \\
\langle c_{l_n}, \epsilon, \sigma_n\rangle \xrightarrow{i} \langle p_n\rangle, \langle \phi, p_n, \sigma_n\rangle \xrightarrow{m} \langle \epsilon, e_n, \sigma_{c_n}\rangle, \langle c_{r_n}, e_n, \sigma_n\rangle \xrightarrow{m} \langle r_n, \epsilon, \sigma_{n+1}\rangle, \\
\langle h_r, \epsilon, \sigma_{n+1}\rangle \xrightarrow{i} \langle r\rangle
\end{array}$$

Table 4.1: Execution Scheme for Conditional Transformations

The left-hand side of the head $h_l \Rightarrow h_r$ is instantiated; then the conditionals are processed by instantiating their left-hand side, executing the instance and matching the right-hand side with the result. This is repeated for all conditionals. With the bindings created by the last conditional, the right-hand side of the head is instantiated as expressed by the following rule in which $C$ is a sequence containing the conditionals of the transformation:

$$\frac{\begin{array}{c}\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_m, s_{out}, \sigma_m\rangle \\ \langle cexec(C), \epsilon, \sigma_m\rangle \xrightarrow{m} \langle \epsilon, \epsilon, \sigma_C\rangle \\ \langle p_r, \epsilon, \sigma_C\rangle \xrightarrow{i} \langle r_{out}\rangle\end{array}}{\langle p_l \Rightarrow p_r \Longleftarrow C, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_{out}, s_{out}, \sigma_{in}\rangle}\ \textsc{CTrans}$$

A conditional transformation fails if matching the left-hand side of the head, executing the conditionals or instantiating the right-hand side of the head fails.

$$\frac{\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}{\langle p_l \Rightarrow p_r \Longleftarrow C, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{CTrans Match}\ \bot$$

$$\frac{\begin{array}{c}\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_m, s_m, \sigma_m\rangle \\ \langle cexec(C), \epsilon, \sigma_m\rangle \xrightarrow{m} \bot\end{array}}{\langle p_l \Rightarrow p_r \Longleftarrow C, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{CTrans Compute}\ \bot$$

$$\frac{\begin{array}{c}\langle p_l, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_m, s_m, \sigma_m\rangle \\ \langle cexec(C), \epsilon, \sigma_m\rangle \xrightarrow{m} \langle \epsilon, \epsilon, \sigma_C\rangle \\ \langle p_r, \epsilon, \sigma_C\rangle \xrightarrow{i} \bot\end{array}}{\langle p_l \Rightarrow p_r \Longleftarrow C, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{CTrans Inst}\ \bot$$

The matching semantics of the operator $cexec$ (for conditional execution) formalises the instantiate-compute-match loop. It adds bindings created from matching the results of programs to the store. The base case of $cexec$ is an empty sequence of conditions. In this case the store remains unmodified. This semantics implements the intuitive notion that a conditional transformation without conditionals corresponds to an unconditional transformation.

$$\frac{\begin{array}{c}\langle p_l, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_{inst}\rangle \\ \langle \phi, s_{inst}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{res}, \sigma_{res}\rangle \\ \langle all(p_r), s_{res}, \sigma_{res}\rangle \xrightarrow{m} \langle r_m, \epsilon, \sigma_m\rangle \\ \langle cexec(T), \epsilon, \sigma_m\rangle \xrightarrow{m} \langle r_e, s_e, \sigma_{out}\rangle\end{array}}{\langle cexec(p_l \Rightarrow p_r{::}T), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{in}, \sigma_{out}\rangle}\ \textsc{cexec Success}$$

$$\frac{}{\langle cexec(\epsilon), s, \sigma\rangle \xrightarrow{m} \langle \epsilon, s, \sigma\rangle}\ \textsc{cexec Empty}$$

The helper pattern $cexec$ fails if any of the conditionals fails to instantiate, compute or match the result pattern.

$$\frac{\langle p_l, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \bot}{\langle cexec(p_l \Rightarrow p_r{::}T), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{cexec Inst}\ \bot$$

$$\frac{\begin{array}{c}\langle p_l, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_{inst}\rangle \\ \langle \phi, s_{inst}, \sigma_{in}\rangle \xrightarrow{m} \bot\end{array}}{\langle cexec(p_l \Rightarrow p_r{::}T), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{cexec Compute}\ \bot$$

$$\frac{\begin{array}{c}\langle p_l, \epsilon, \sigma_{in}\rangle \xrightarrow{i} \langle s_{inst}\rangle \\ \langle \phi, s_{inst}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{res}, \sigma_{res}\rangle \\ \langle all(p_r), s_{res}, \sigma_{res}\rangle \xrightarrow{m} \bot\end{array}}{\langle cexec(p_l \Rightarrow p_r{::}T), s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{cexec Match}\ \bot$$

The behaviour in case of failure allows a form of localised backtracking to be expressed when using conditional rules in a rewriting system. If the conditionals express steps of a computation towards a final result, steps already performed are undone if a condition that is subsequently executed fails. Conditional transformations are the primary computational mechanism of Concat (see Chapter 5).

## 4.4 Parsing and Unparsing with Patterns

Parsing is the process of recognising the structure encoded in a linear representation and generating an internal representation encoding this structure explicitly [64]. The basic idea is that structured representations are more suitable for most programmatic manipulation tasks. Unparsing is the opposite process of transforming a structured representation into a linear representation. From the viewpoint of a system that operates on a parse result, the linear representation is $external$, while the structured representation is $internal$. In the context of programming, the external representation is typically a character string, while the internal representation is some system-specific data structure.

### 4.4.1 Unifying External and Internal Representation

Typed sequences can be arbitrarily nested and thus are a suitable basis for an internal representation. However, they can also be used to represent strings of characters. This way, both the external and internal representation can be unified. This means that parsing can naturally be described as a pattern-based transformation process on typed sequences.

In the following, a character representation is used that encodes characters as atoms and wraps these atoms in a sequence of type char to make the interpretation explicit. For instance, the characters 'a' and ' ' (space) are represented as sequences $\tau([a], char)$ and $\tau([space], char)$. Strings are sequences of characters of type string. For example, the string "abc" is represented as $\tau([\tau([a], char), \tau([b], char), \tau([c], char)], string)$.

### 4.4.2 Grammar as Meta-Patterns

According to Chomsky [28], a formal, context-free grammar is a tuple $(N, \Sigma, P, S)$ where $N$ is a set of nonterminal symbols, $\Sigma$ is a set of terminal symbols, $S$ is a start symbol from $N$ and $P$ is a set of production rules of the form $n \rightarrow (\Sigma \cup N)^*$ where $n \in N$ and $^*$ denotes the Kleene closure. The language defined by such a grammar is the set of sequences $L \subseteq \Sigma^*$ that can be derived by starting with $S$ and replacing nonterminals according to the rules.

This style of defining a grammar can be expressed in terms of the pattern formalism as follows: nonterminals are references, the start symbol is a "start reference", terminals are typed sequences or atoms and productions are meta-transformations. The difference is that productions are not restricted to symbols from $\Sigma \cup N$ but can be defined using pattern operators. This is akin to grammar formalisms such as Extended Backus Naur Form [86]. However, the productions are not used to generate sentences, but to recognise sentences and generate results [64]. Parsing starts by applying the "start reference" to an input se-

quence. Application of a production, i.e., substituting a nonterminal with its definition, is implemented by the reference resolution mechanism introduced in Section 3.7. A grammar with productions $p_1, p_2, ..., p_n$ corresponds to a pattern $p_1|p_2|...|p_n$. The sequential semantics of the choice operator entail that the order of productions is significant.

The following is an example of a grammar fragment that recognises a natural number consisting of a sequence of digits. The surrounding choice pattern is omitted. Using the prefix $rec$ is a convention for indicating a recognition process that does not perform any transformation on the input data.

$$rec\text{-}nat \Rightarrow qq(ref(rec\text{-}digit)^*)$$
$$rec\text{-}digit \Rightarrow qq(\tau([0|...|9], char))$$

When matching $\tau([ref(rec\text{-}nat)], string)$ with $\tau([\tau([2], char), \tau([3], char)], string)$ the $rec\text{-}nat$ reference is resolved to $ref(rec\text{-}digit)^*$. The repetition operator matches the pattern $ref(rec\text{-}digit)$ greedily with the input. For every successive match, the reference is resolved to the pattern $\tau([0|...|9], char)$. The third attempt of matching the pattern fails because the input is empty. The result is a sequence consisting of the recognised characters.

The following is a grammar that transforms the string representation of a number into a typed sequence with the digits as symbols. Using the prefix $int$ is a convention for indicating that the pattern performs internalisation.

$$int\text{-}nat \Rightarrow qq(ds{:}ref(digit)^* \Rightarrow \tau([ds{:}\alpha^*], nat))$$
$$int\text{-}digit \Rightarrow qq(\tau([d{:}(0|...|9)], char) \Rightarrow \tau([d{:}\alpha)], symbol))$$

The process of unparsing a natural number, i.e., transforming its typed sequence representation into a character representation, can be described as follows. Using the prefix $ext$ is a convention that indicates the pattern performs externalisation.

$$ext\text{-}nat \Rightarrow qq(\tau([ds{:}ext\text{-}digit^*], nat) \Rightarrow ds{:}\alpha^*)$$
$$ext\text{-}digit \Rightarrow qq(\tau([d{:}\alpha], symbol) \Rightarrow \tau([d{:}\alpha], char)$$

Grammars defining transformations between internal and external representations are a foundation for syntactic extensibility as will be demonstrated in Section 4.5.4.

## 4.5 Staged Processing and Views

Computation and parsing are key components of a pattern-based programming system. This section describes how these components can be combined to formalise the execution of programs as staged processing [155].

### 4.5.1 Internalisation, Computation and Externalisation

The execution process of a program in external representation can be separated into three fundamental stages: internalising, computing and externalising. This separation is expressed by the following function definition where the argument of $Prog_e$ is a program in external representation and where $int$ (for internalise) maps a program from external to internal representation, $cmp$ (for compute) maps a program in internal representation to a modified program in internal representation, and $ext$ (for externalise) maps a program in internal representation to an external representation:

$$execute(Prog_e) = ext(cmp(int(Prog_e)))$$

This function definition captures the essence of staged processing of programs but not the configurability of the stages. The three stages can be configured independently through patterns. Let $PtnInt_i$, $PtnCmp_i$ and $PtnExt_i$ be patterns in internal representation describing internalisation, computation and externalisation respectively. Accordingly, the 3-tuple $\langle PtnInt_i, PtnCmp_i, PtnExt_i \rangle$ is a configuration. Each stage of $execute$ is defined by one part of the configuration:

$$execute(\langle PtnInt_i, PtnCmp_i, PtnExt_i \rangle, Prog_e) =$$
$$ext(PtnExt_i, cmp(PtnCmp, int(PtnInt_i, Prog_e)))$$

This definition captures both the configurability and the sequential execution of the three stages.

### 4.5.2 Staging as Vertical Combination

The configuration of each stage is defined by a combination of patterns. The functionality of each stage is to match the pattern with a program. This unified view is possible when the external representation of a program (character strings) and its internal representation are typed sequences, as discussed in Section 4.4.1. Accordingly, the parameterised versions of $int$, $cmp$ and $ext$ are merely aliases for a function $match$ that applies a pattern to a sequence and returns the result. An alternative definition of $execute$ stressing this fact is

the following:

$$execute(\langle PtnInt_i, PtnCmp_i, PtnExt_i\rangle, Prog_e) =$$
$$match(PtnExt_i, match(PtnCmp, match(PtnInt_i, Prog_e)))$$

The body of the $execute$ function defines execution of three stages as three matches where the result of one match is the input of another match. This exact semantics is captured by the vertical combination operator introduced in Section 3.2.5. Thus, the nested application of match can be expressed by the pattern:

$$PtnInt_i \rightarrow PtnExec_i \rightarrow PtnExt_i$$

This means that the notions of both staging and configuration of stages are expressed by a single vertically combined pattern. This technique will be used in Section 4.5.3 to define the view mechanism.

Each stage is defined by patterns that are matched against the program in internal or external representation. Each pattern describes the complete processing of each stage. For example, if the execution stage consists of a sequence of rewriting rules that are repeatedly applied to the program, the pattern for the execution stage must express both the repeated application of rules and the rules themselves. The separation of the execution model and the objects being executed can be expressed in the external representation; the combination is then performed during internalisation.

### 4.5.3 Structural View Abstraction

Underlying the separation of program execution into internalisation, computation and externalisation is the more general concept of transforming data from one representation schema to another, manipulating it and mapping the result back so that changes are visible in the original schema. This corresponds to the notion of querying and updating views in databases [39]. The previous section demonstrated that staged execution can be expressed through vertical combination. The following meta-transformation uses vertical combination to abstract computation on a view:

$$qq(viewcomp([uq(int{:}\alpha), uq(ext{:}\alpha), uq(comp{:}\alpha)])) \Rightarrow$$
$$qq(uq(int{:}\alpha) \rightarrow uq(comp{:}\alpha) \rightarrow uq(ext{:}\alpha))$$

The meta-transformation resolves a reference to $viewcomp$ parameterised with patterns for internalisation, externalisation and the computation to be performed on the view. The result is a vertical combination of the three argument patterns.

To abstract the actual view concept the definition of the internalisation and externalisation has to be separated from the definition of the computation performed on the internalised data. This way a view can be defined independent of a computation. The following meta-meta transformation implements the view abstraction:

$$\begin{array}{l} qq(view([uq(int{:}\alpha), uq(ext{:}\alpha), uq(name{:}\alpha)])) \Rightarrow \\ \quad qq(qq(uq(uq(name{:}\alpha))(uq(comp{:}\alpha))) \Rightarrow \\ \quad\quad qq(uq(uq(int{:}\alpha)) \rightarrow uq(comp{:}\alpha) \rightarrow uq(uq(ext{:}\alpha)))) \end{array}$$

The result of matching a reference to the meta-meta-transformation with arguments for internalisation, externalisation and a view name is a meta-transformation. This meta transformation resolves a reference consisting of the view name and the computation pattern. The target of the reference is the pattern that combines the view and the computation.

The following example discusses a view that maps between a comma-separated character representation and a structured representation of a list of natural numbers. The meta-transformations in Table 4.2 define the patterns involved. The patterns referred to by $ref(\textit{int-list})$ and $ref(\textit{ext-list})$ are the two components of the view. The pattern referred to by $ref(rest)$ removes the first element of the list in internal representation. The definition of pattern $\textit{int-nat}$ that is used by $\textit{int-list}$ to internalise natural numbers can be found in Section 4.4.2.

$$\begin{array}{l} \textit{int-list} \Rightarrow qq(\sim([x{:}\sim([ref(\textit{int-nat}), \\ \qquad\qquad \sim([ign(\tau([comma], char)), ref(\textit{int-nat})])^*])]) \\ \qquad\qquad \Rightarrow \tau([x{:}\alpha^*], list)) \\ \textit{ext-list} \Rightarrow qq(\tau([x{:}\alpha^*], list) \\ \qquad\qquad \Rightarrow x{:}\sim([ref(\textit{ext-nat}), \\ \qquad\qquad\qquad \sim([\sim(\epsilon) \Rightarrow \tau([comma], char), ref(\textit{ext-nat})])^*])) \\ rest \Rightarrow qq(\tau([\alpha, r{:}\alpha^*], list) \Rightarrow \tau([r{:}\alpha^*], list)) \end{array}$$

Table 4.2: Pattern Definitions for the $natListView$ Example

The view with the name $natListView$ is created through the following reference:

$$ref(view(ref(\textit{int-list}), ref(\textit{ext-list}), natListView))$$

This reference is resolved to the meta-transformation

$$\begin{array}{l} natListView(comp{:}\alpha) \Rightarrow \\ \quad qq(ref(\textit{int-list}) \rightarrow uq(comp{:}\alpha) \rightarrow ref(\textit{ext-list})) \end{array}$$

which expects a pattern that defines an operation on a list of numbers in internal representation. By applying this meta-transformation, the reference

$$ref(natListView(ref(rest)))$$

is resolved to the actual execution pattern:

$$qq(ref(int\text{-}list) \rightarrow uq(ref(rest)) \rightarrow ref(ext\text{-}list))$$

If this pattern is matched with the input sequence

$$[\tau([1], char), \tau([0], char), \tau([comma], char), \tau([2], char)]$$

matching $int\text{-}list$ first transforms the character representation into a structured list representation where the digits become symbols:

$$\tau([\tau([\tau([1], symbol), \tau([0], symbol)], nat), \tau([\tau([2], symbol)], nat)], list)$$

Matching $ref(rest)$ with this list representation yields the computation result in internal representation:

$$\tau([\tau([\tau([2], symbol)], nat)], list)$$

Matching the pattern $ext\text{-}list$ transforms the list back into the character representation:

$$[\tau([2], char)]$$

Because both the internal and external representation are typed sequences, views can be defined on views to form an arbitrary number of stacked view layers. This concept is further explored in the context of Concat's view implementation, see Section 5.4.

### 4.5.4 Extensible Syntax for Programs and Data

In this and the previous chapter, a term notation using the constructor $\tau$ is used for typed sequences. Concat, which will be introduced in the next chapter, uses a text-based notation which is more suitable for actual programming: typed sequences start with a square bracket, followed by a colon and a type identifier. After a mandatory whitespace, the content of the sequence follows, terminated by a closing bracket. The content consists of atoms or typed sequences separated by whitespaces. Accordingly, the list

$$\tau([\tau([\tau([1], symbol), \tau([0], symbol)], nat), \tau([\tau([2], symbol)], nat)], list)$$

can be written as:

```
[:list [:nat [:symbol 1] [:symbol 0] ]
       [:nat [:symbol 2] ]]
```

The internalisation of this notation can be broken down into two parts: (1) internalisation of the surrounding brackets and type and (2) internalisation of the content of the sequence. Both parts are expressed by the two productions in Table 4.3.

$$
\begin{array}{l}
\textit{int-tseq} \Rightarrow qq(\sim([\tau([\textit{obracket}], \textit{char}), \tau([\textit{colon}], \textit{char}), \\
\qquad\qquad ref(\textit{type}), ref(\textit{int-seq}), \tau([\textit{cbracket}], \textit{char})])) \\
\textit{int-seq} \Rightarrow qq(\sim(\epsilon) \mid \sim([ref(\textit{int-elem}), \\
\qquad\qquad \sim([ign(ref(\textit{white})), ref(\textit{int-elem})])^*]))
\end{array}
$$

Table 4.3: Internalising an Alternative Syntax for Typed Sequences

The production $int\text{-}seq$ states that a sequence is either empty or consists of elements separated by whitespaces. The interesting part is the definition of $int\text{-}elem$. The following production defines a system where arbitrarily nested typed sequences are the only elements:

$$\textit{int-elem} \Rightarrow qq(ref(\textit{int-tseq}))$$

Both typed sequence notations encode the type and structure of data explicitly. The verbosity inherent in this encoding renders it impractical for expressing all parts of a program. Section 4.4.2 discussed parsing of character-based notations and introduced the parser $int\text{-}nat$ which transforms a sequence of digits into an internal natural number representation. Using this parser, the string 10 is internalised into the same representation as the string `[:nat [:symbol 1] [:symbol 0] ]`.

The following modified version of production $int\text{-}elem$ adds $int\text{-}nat$ as an additional choice.

$$\textit{int-elem} \Rightarrow qq(ref(\textit{int-nat}) \mid ref(\textit{int-tseq}))$$

Based on this definition, the parser is able to parse `[:list 10 2]`, a notation which mixes the standard notation for typed sequences with the special literal syntax introduced by a parser. The parsing result is the same as for the more verbose definition above that makes the internal structure of a natural number explicit. The same principle just described for parsing also applies for unparsing.

The basic idea is to provide the pattern formalism with means of extending its own syntax by defining alternative notations for typed sequences. These notations need to encode the type and content part in such a way that the resulting notation can be recognised

by the parser and generated by the unparser. By using a *default notation* for typed sequences, new syntax can be defined as a transformation on typed sequences, one side of the transformation being concerned with character strings and the other side with direct encodings of structure using typed sequences.

Several views may be available on different nesting levels. For example, given a notation in which symbols start with the character `'`, the list above can be written as `[:list [:nat '1 '0] [:nat '2]]` which allows symbolic manipulation of the structure of natural numbers without the need to know the internal structure of symbols. Similarly, there might be a notation for lists in round brackets which allows writing `(10 2)`. All these syntaxes lead to the same internal representation, i.e., they are just alternatives to the standard notation for typed sequences. Enabling, disabling or enforcing alternative representation can be expressed by manipulating the production $int\text{-}elem$.

The prerequisite for this kind of syntactic extension is an interface defining interaction between the pattern system and external editing and display processes. The communication is based on sequences of characters that are encoded as typed sequences of type $char$. For example, from the viewpoint of the pattern system, the first four items of the external representation of `[:list 10 2]` are `[:char obracket] [:char colon] [:char l] [:char i]`. The display process, however, renders all typed sequences of type $char$ as characters on the screen. The distinction between displaying a character and displaying a typed sequence of type character does not pose a problem. To display `x` on the screen, the representation is `[:char x]`; to display `[:char x]`, it must be a character sequence with the first four characters being `[:char obracket] [:char colon] [:char c] [:char h]`. The role of the I/O interface is to map between the character encoding of an external editor and that of the pattern system.

### 4.5.5 Extensible Syntax for Patterns

The previous section defined a parser for typed sequences and showed how extensions to the parser allow the definition of alternative syntaxes for typed sequences. Such definitions are possible because there is a standard notation for typed sequences that can be used to express internal structures. The same principle can be applied at the pattern level. Given a standard notation for patterns, meta-patterns can be used to define transformations from the character representation of the pattern language to the internal representation. The following production defines a notation in which references are written with surrounding

inequality signs:

$$\begin{aligned} \textit{int-ref} \Rightarrow qq(\ ([\tau([<], char), p{:}ref(\textit{int-pexp}), \tau([>], char)]) \\ \Rightarrow qq(ref(uq(p{:}\alpha)))) \end{aligned}$$

The production assumes the existence of a pattern $int\text{-}pexp$ that internalises pattern expressions. The newly introduced syntax allows writing `<nothing>`, which is equivalent to $ref(nothing)$. Concat uses this syntactic extension mechanism extensively to define syntaxes of programs and meta-programs.

### 4.5.6 Temporal Views on Computations

Two of the main advantages of defining program execution in terms of a rewriting system are complete access to the program state and fine-grained control over the execution process. While rules can be restricted to access only certain parts of a program, as discussed in Sections 4.2.3 and 4.2.4, it is nevertheless possible to define less restricted access to the program at a meta-level. This can, for example, be used to implement optimisations or code injection as in Aspect-Oriented Programming (AOP) [102].

While a detailed view on the structure and behaviour of the program provides an ideal foundation for pattern operations, this level of detail might be inappropriate for purposes of programming or analysing a system. For example, users of a language are usually not interested in the details of internal representation. Viewing the effects of a function application does not require a view on the internals of the functions. Similarly, an aspect definition is based on join points – events in an execution model, not in the actual system implementation – and, therefore, the system must render the execution in terms of the user model.

The problem of viewing the system execution in terms of a user model can be broken down into a structural aspect ("how is an execution step visualised?") and a temporal aspect ("which execution steps are visible?"). Section 4.5.3 defined the structural part of the solution by defining a technique for hiding the internal representation of a program behind a syntactic interface: views. This section discusses how certain steps in the execution can be hidden.

The basic idea is to determine visibility by a pattern that is matched after every execution step. That pattern pattern is connected with the external display process. If matching succeeds the system state is shown, otherwise it is hidden. The use of temporal views will be illustrated by "black-boxing" operations in the concatenative rewriting system. As defined in Section 4.2, concatenative semantics can be enforced on a generic rewriting system by restricting its transformations through meta-patterns and by defining specific

application strategies for transformations.

In the concatenative system that allows patterns (see Section 4.2.4), transformations that only manipulate items and operators inside quotations define a functional semantics even when all execution steps are visible. An operation such as $call$, on the other hand, instantiates operators outside of the quotation context. This seems to violate the concept that all operations must be functions mapping between data, i.e., between items. However, if an application of $call$ is viewed from a "black-box" perspective, it maps data containing a quotation to data that is the result of executing the program contained in the quotation, which is functional.

For example, the program $[2, 3, \tau([swap], quotation), call]$ yields $[3, 2]$. Looking only at these two execution states, $call$ appears to be mapping from data to data in a single step. The execution state $[2, 3, swap]$ in between makes an implementation detail of $call$ visible. The execution of $swap$ again has functional semantics. The crucial point is the shift of the abstraction levels that occurs when an operator is rewritten to its implementation – in case of $call$ a program containing operators. Seeing this implementation detail of $call$ destroys the illusion of functional semantics.

By default, all steps of the rewriting process are visible and there is no explicit distinction between "black-box" and "white-box" views on the program execution. Hiding the "internal" execution steps of an operation can be achieved by explicitly marking program state. For the concatenative system, there also needs to be a way to determine when execution of the internal steps of an operation have finished and execution resumes with the next operation on the same abstraction level. Both problems can be solved by explicitly or implicitly inserting an operation $hide$ as the right-most element of every transformation that inserts operations into the program. Operation $hide$ has the sole purpose of controlling "black-boxing". The definition of $hide$ is:

$$hide \Rightarrow \sim(\epsilon)$$

For instance, the program $[2, 3, \tau([swap], quotation), call, dup]$ is executed from left to right. The first operation that will be executed is $call$. The next operation after $call$, $dup$, will be executed only after all operations that $call$ adds to the program have finished. Placing $hide$ in front of $dup$ makes the point of continuation (the "return address") explicit. The first step of $call$'s execution produces $[2, 3, swap, hide, dup]$. Next, $swap$ is executed and yields $[3, 2, hide, dup]$. Because there are only items before $hide$, this operation is executed next. The semantics of $hide$ is to do nothing. The result is $[3, 2, dup]$. Finally, $dup$ is executed and yields $[3, 2, 2]$.

This shows how internal execution states of an operation can be explicitly tracked. As

discussed above, to actually hide the states that contain the operator $hide$, a pattern needs to be defined. This pattern is $!find\ hide\ among\ \alpha$. The pattern fails to match if there is a $hide$ operator anywhere inside the program. By applying this technique, the step from $[2, 3, \tau([swap], quotation), call, dup]$ to $[2, 3, swap, dup]$ is hidden and, for this reason, the system provides a functional "black-box" view on the execution of $call$. Together, the restriction of transformations, the definition of an application strategy, the structural views that hide the internals of operators and items and the temporal view mechanism turn the generic rewriting system into a concatenative programming system. In other words, they create the illusion of a concatenative system based on a rewriting system.

## 4.6 Elliptical Patterns: A Practical Extension

Structures that contain several instances of the same pattern are abundant. For example, a dictionary can be represented as a sequence of key-value bindings like the following:

$$\tau([\tau([x, a], bnd), \tau([y, b], bnd), \tau([z, c], bnd)], dict)$$

An alternative representation of a dictionary consists of separate sequences of keys and values. Accordingly, the example dictionary is represented as follows:

$$\tau([\tau([x, y, z], keys), \tau([a, b, c], vals)], dict)$$

The two representations contain the same information, but they structure this information differently. The repetition operator is suitable for expressing the repetition in both structures. The following pattern matches the first representation:

$$\tau([\tau([\alpha, \alpha], bnd)^*], dict)$$

The wildcard $\alpha$ is used in place of actual keys and values to match arbitrary key-value pairs. The following pattern matches the second representation:

$$\tau([\tau([\alpha^*], keys), \tau([\alpha^*], vals)], dict)$$

While both patterns capture the repetitive structures, they cannot be used to transform between these structures.There reasons for this are twofold. Firstly, based on the standard variable binding semantics of the pattern formalism, the use of variables instead of $\alpha$s in the first pattern would express equality rather than similarity. Secondly, the instantiation semantics of the repetition operator is failure. This section presents an extension for

conveniently transforming repetitive structures. The extension is implemented by the new repetition operator $^\circ$ that circumvents the aforementioned restrictions. With this operator, the transformation between the two dictionary representation is defined as follows:

$$\tau([\tau([k{:}\alpha, v{:}\alpha], bnd)^\circ], dict) \Rightarrow \tau([\tau([(k{:}\alpha)^\circ], keys), \tau([(v{:}\alpha)^\circ], vals)], dict)$$

The atoms $k$ and $v$ are names of variables binding keys and values during matching.

### 4.6.1 Matching Semantics

Variable matching is based on a strict binding strategy, as defined by the function $cbind$ in Section 3.2.1: the attempt to rebind a variable to a different value fails. When a pattern containing a variable is matched repeatedly with the input, the values matched for the variable pattern must be the same for each repetition. In terms of expressive power, repetition can abstractly express syntactic equality in a sequence of values of arbitrary length. For example, the pattern $(x{:}ref(item))^*$ defines a sequence of syntactically equal values. Transforming requires binding *similar* values to a variable repeatedly during matching, as this is the only way to carry information over to the instantiation phase. This is where the semantics of structural *equality* defined by the pattern above stands in the way. For example, an attempt to match the first version of the example dictionary with the pattern

$$\tau([\tau([k{:}\alpha, v{:}\alpha], bnd)^*], dict)$$

fails in the second repetition because binding $k$ to $y$ fails: $k$ is already bound to $x$. The first step for realising transformations of repetitive structures is to allow variables inside a repetition pattern to be bound to multiple values. Based on the new semantics, instead of failing, repetition creates bindings $(k, [x, y, z])$ and $(v, [a, b, c])$.

The goal is to implement this special variable matching semantics of the $^\circ$ operator without having to change the general variable matching semantics. The basic idea to achieve this when matching a pattern $p^\circ$ is to initially match $p$ with the empty store $\epsilon$ in every repetition. This entails that variables from previous repetition steps are invisible and every repetition step produces a store with bindings of variables in $p$. After the last repetition step, the individual stores are combined into a single store in which the variables in $p$ are bound to sequences of values. The resulting store contains the bindings created during matching $p^\circ$. To produce the final store, this store is merged with the store that existed before the elliptical pattern was matched. This merge is performed in such a way that bindings created by $p^\circ$ and previously existing bindings have to be consistent.

The combination of the individual stores of each repetition into a single store is per-

formed by the helper function $mbind\text{-}all$ (for multiply bind all). Function $cbind\text{-}all$ (for consistently bind all) performs the merge with the existing store.

$$\frac{\begin{array}{c}\langle p, s_{in}, \varepsilon\rangle \xrightarrow{m} \langle r, s_p, \sigma_p\rangle \\ \langle p^\circ, s_p, \varepsilon\rangle \xrightarrow{m} \langle r_{p^\circ}, s_{out}, \sigma_{p^\circ}\rangle \\ mbind\text{-}all(\sigma_p, \sigma_{p^\circ}) \mapsto \sigma_{mba} \\ cbind\text{-}all(\sigma_{mba}, \sigma_{in}) \mapsto \sigma_{out} \\ combine(r_p, r_{p^\circ}) \mapsto r_{out}\end{array}}{\langle p^\circ, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle r_{out}, s_{out}, \sigma_{out}\rangle}\ \textsc{Elliptical Match}$$

$$\frac{\langle p, s_{in}, \varepsilon\rangle \xrightarrow{m} \bot}{\langle p^\circ, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \langle \epsilon, s_{in}, \sigma_{in}\rangle}\ \textsc{Elliptical Match Zero}$$

Elliptical matching fails if a pattern matches the input sequence, but the store created during the matching cannot be merged with the existing store. This occurs when there is a previous binding with the same name and different value already in the store. By definition, $mbind\text{-}all$ and $combine$ never fail.

$$\frac{\begin{array}{c}\langle p, s_{in}, \varepsilon\rangle \xrightarrow{m} \langle r, s_p, \sigma_p\rangle \\ \langle p^\circ, s_p, \varepsilon\rangle \xrightarrow{m} \langle r_{p^\circ}, s_{out}, \sigma_{p^\circ}\rangle \\ mbind\text{-}all(\sigma_p, \sigma_{p^\circ}) \mapsto \sigma_{mba} \\ cbind\text{-}all(\sigma_{mba}, \sigma_{in}) \mapsto \bot\end{array}}{\langle p^\circ, s_{in}, \sigma_{in}\rangle \xrightarrow{m} \bot}\ \textsc{Elliptical Match} \bot$$

Function $mbind\text{-}all$ combines two stores. The bindings from the first store are added to the bindings of the second store by applying the function $mbind$ to all bindings in the first store. Function $mbind$ binds a name to a value wrapped in a sequence if no previous binding exists. However, if a binding exists, $mbind$ prepends the new value to the existing one. The definition of $mbind$ entails that variables occurring in nested elliptical patterns are nested in sequences. The amount of sequences wrapped around a value corresponds to the number of elliptical operators around the variable pattern that created it.

$$\frac{\begin{array}{c}\sigma[n] = \bot \\ combine(v, \epsilon) \mapsto s\end{array}}{mbind(n, v, \sigma) \mapsto \sigma[n := s]}\ \text{MBIND FRESH}$$

$$\frac{\begin{array}{c}\sigma[n] \neq \bot \\ combine(v, \sigma[n]) \mapsto s\end{array}}{mbind(n, v, \sigma) \mapsto \sigma[n := s]}\ \text{MBIND EXISTING}$$

The incremental application of function $mbind$ that combines all bindings of a store with bindings in another store is defined recursively. Let $binding$ be a function mapping a store with at least one binding to a sequence. The sequence contains an arbitrary binding from the store and the store with that binding removed.

$$\frac{\begin{array}{c}\sigma_i \neq \varepsilon \\ binding(\sigma_i) \mapsto [(n, v), \sigma_r] \\ mbind(n, v, \sigma_c) \mapsto \sigma_u \\ mbind\text{-}all(\sigma_r, \sigma_u) \mapsto \sigma_{out}\end{array}}{mbind\text{-}all(\sigma_i, \sigma_c) \mapsto \sigma_{out}}\ \text{MBIND-ALL RECURSE}$$

$$\frac{}{mbind\text{-}all(\varepsilon, \sigma_c) \mapsto \sigma_c}\ \text{MBIND-ALL EMPTY}$$

Similar to $mbind\text{-}all$, the function $cbind\text{-}all$ combines two sets of bindings by applying $cbind$, which is defined in Section 3.2.1.

$$\frac{\begin{array}{c}\sigma_i \neq \varepsilon \\ binding(\sigma_i) \mapsto [(n, v), \sigma_r] \\ cbind(n, v, \sigma_c) \mapsto \sigma_u \\ cbind\text{-}all(\sigma_r, \sigma_u) \mapsto \sigma_{out}\end{array}}{cbind\text{-}all(\sigma_i, \sigma_c) \mapsto \sigma_{out}}\ \text{CBIND-ALL RECURSE}$$

$$\frac{}{cbind\text{-}all(\varepsilon, \sigma_c) \mapsto \sigma_c}\ \text{CBIND-ALL EMPTY}$$

Failure of $cbind$ for any of the bindings in the store results in failure of $cbind\text{-}all$.

$$\frac{\begin{array}{c}\sigma_i \neq \varepsilon \\ binding(\sigma_i) \mapsto [(n, v), \sigma_r] \\ cbind(n, v, \sigma_c) \mapsto \bot\end{array}}{cbind\text{-}all(\sigma_i, \sigma_c) \mapsto \bot}\ \text{CBIND-ALL } \bot$$

### 4.6.2 Instantiation Semantics

The instantiation semantics of repetition defined in Section 3.3.3 is failure because there is no way to determine how often to instantiate its argument. With the matching semantics defined in the previous section multiple bindings are created for a single variable name. For patterns depending on the successful instantiation of variables for their own success, elliptical instantiation can derive the number of instantiation steps from the variable bindings: the number of bindings to the same variable name determines how often a pattern is to be instantiated.

A record must be kept of how many times a pattern was instantiated in order to determine which binding to use at which repetition step and to find out when there are no more bindings. This record must be multidimensional because elliptical operators may be nested. It is helpful in this context to think of variable lookup as access to a multidimensional array. Each elliptical operator corresponds to a loop incrementing an index. During instantiation, the loop terminates if its index exceeds the number of entries in array dimensions. For example, to access $a$, $b$, $c$, and $d$ in the binding $(x, [[a, b], [c, d]])$ the correct indices are $[0, 0]$, $[0, 1]$, $[1, 0]$ and $[1, 1]$ respectively. A lookup fails for indices greater than 1. The index $[1, 0]$ indicates that there are two elliptical operators around the pattern currently being instantiated. The outer operator is in the second repetition while the inner operator is still in the first repetition.

The sequence of indices is represented by the state $\iota$. The instantiation semantics of elliptical matching requires two extensions to the pattern formalism. Firstly, the state of elliptical instantiation $\iota$ has to be passed on by all instantiation rules to make it accessible to variable instantiation. This passing is expressed by adding the state explicitly to all rules below. Secondly, the variable instantiation has to be modified to take the current state of elliptical instantiation into account.

When $^\circ$ is instantiated, a new index is added to the sequence of indices and set to 0. Then the helper pattern $rep$ (for repeat) is instantiated. If $rep$ fails, the input sequence is the result of elliptical repetition.

$$\frac{append(\iota_{in}, [0]) \mapsto \iota_u \quad \langle rep(p), s_{in}, \sigma_{in}, \iota_u\rangle \xrightarrow{i} \langle s_{out}\rangle}{\langle p^\circ, s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Elliptical Instance}$$

$$\frac{append(\iota_{in}, [0]) \mapsto \iota_u \quad \langle rep(p), s_{in}, \sigma_{in}, \iota_u\rangle \xrightarrow{i} \bot}{\langle p^\circ, s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \langle s_{in}\rangle}\ \textsc{Elliptical Inst Zero}$$

The helper $rep$ instantiates its argument and recurses with the current index incremented. $repeat$ must succeed at least once, otherwise it fails. Let $inc$ be a function that increases the last index in an index sequence. For example, $inc([2, 1, 3])$ yields $[2, 1, 4]$. Incrementing the last index reflects that a new repetition caused by the innermost elliptical operator begins.

$$\frac{\langle p, s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \langle s_{inst}\rangle \quad inc(\iota_{in}) \mapsto \iota_u \quad \langle rep(p), s_{inst}, \sigma_{in}, \iota_u\rangle \xrightarrow{i} \langle s_{out}\rangle}{\langle rep(p), s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \langle s_{out}\rangle}\ \textsc{Elliptical Greedy}$$

$$\frac{\langle p, s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \langle s_{inst}\rangle \quad inc(\iota_{in}) \mapsto \iota_u \quad \langle rep(p), s_{inst}, \sigma_{in}, \iota_u\rangle \xrightarrow{i} \bot}{\langle rep(p), s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \langle s_{inst}\rangle}\ \textsc{Elliptical Once}$$

$$\frac{\langle p, s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \bot}{\langle rep(p), s_{in}, \sigma_{in}, \iota_{in}\rangle \xrightarrow{i} \bot}\ \textsc{Elliptical} \bot$$

Looking up variables by indices is defined by the $lookup$ function. It retrieves the nested structure with the values and applies $get\text{-}elem$ to extract the indexed value.

$$\frac{x = \sigma[n] \qquad get\text{-}elem(\iota, x) \mapsto result}{lookup(\sigma, n, \iota) \mapsto result}\ \text{LOOKUP}$$

Function $get\text{-}elem$ is the function that actually performs the access by index. Let $nth$ be a function that maps a sequence $s$ and an index $n$ to the n-th element of the sequence or to $\perp$ in case $n$ is too large for $s$.

$$\frac{nth(i, s_{in}) \mapsto s_u \qquad get\text{-}elem(\iota, s_u) \mapsto result}{get\text{-}elem(i{::}\iota, s_{in}) \mapsto result}\ \text{GET-ELEM RECURSE}$$

$$\frac{}{get\text{-}elem(\epsilon, s_{in}) \mapsto s_{in}}\ \text{GET-ELEM DIRECT}$$

$$\frac{nth(i, s_{in}) \mapsto \perp}{get\text{-}elem(i{::}\iota, s_{in}) \mapsto \perp}\ \text{GET-ELEM BOUNDS } \perp$$

The function $get\text{-}elem$ retrieves the $n$th value from every sequence. The value retrieved with the last index is the value that is instantiated.

Elliptical matching and instantiation with arbitrary nesting levels is utilised in XMF (see Chapter 6) for defining inter-model transformations and transformations between models and views.

## 4.7 Summary and Conclusions

Based on the core functionality defined in Chapter 3, this chapter formalised key components of a pattern-based programming and meta-programming system. The fact that fundamental notions of parsing, computing and staged processing of programs can naturally be expressed through pattern operations highlights the expressiveness of the pattern

formalism. As the aforementioned concepts are fundamental for creating languages, this chapter strongly supports the first part of the hypothesis, which states that "the systematic creation and layering of languages can be reduced to the elementary operations of pattern matching and instantiation". The layering aspect of the hypothesis is supported by the pattern-based view mechanisms, which serves as an implementation tool for layering. In addition to that, a systematic way of building a language on top of a pattern-based rewriting system is introduced. The steps involved are (1) restricting transformation through meta-patterns, (2) defining an application strategy for transformations using pattern operations, (3) using structural views for hiding the internal representation of program state behind a syntactic interface and (4) using temporal views to select which execution steps are visible.

Supporting the second part of the hypothesis, which states that the pattern approach "provides a formal and practical foundation for language-driven modelling, programming and analysis" is the task of the following chapters. The application of the pattern approach for language-driven programming is the topic of the next chapter, where the techniques introduced in this chapter will be utilised to define a highly configurable, self parsing (meta-)programming system.

# Chapter 5

# Language Engineering with Concat

The previous two chapters formally introduced a core of pattern functionality and extensions for pattern-based parsing, computing, layering and syntactic abstraction. This chapter discusses a practical application of this theoretical foundation in the form of Concat, a framework for creating and relating software languages. At its core, Concat is a pattern-based rewriting system that manipulates programs. A meta-language defines the rules of the rewriting system and the syntax of the programs being manipulated. This meta-language is highly extensible. New language abstractions and execution models can be defined by applying the meta-language to itself.

Concat follows a layered approach to syntax and supports different degrees of syntactic freedom for programs and meta-programs in the form of typed sequences, literals and special forms (Section 5.1). The default implementation of the meta-language is called *Core Concat*. The primary concepts of Core Concat are *operations*, *views*, *productions* and *macros*. Operations provide a special syntax for defining pattern-based concatenative rules. Views are bidirectional transformations that define alternative notations for typed sequences. Macros provide means to break out of the constraints of operations and views by allowing non-functional rewriting rules and unidirectional syntactic transformations (Section 5.2).

The implementation of combinatory logic is a case study that showcases language creation, especially the use of views to define custom syntaxes on the program and meta-program levels (Section 5.3). The application of views can be generalised to a layered approach to computation (Section 5.4). A metacircular implementation of Concat provides means to change the syntax and semantics of the meta-language from within (Section 5.5).

⟨*program*⟩ ::= ⟨*data*⟩

⟨*data*⟩ ::= ⟨*elem*⟩ (⟨*sep*⟩$^+$ ⟨*elem*⟩)$^*$ | $\epsilon$

⟨*elem*⟩ ::= ⟨*operator*⟩ | ⟨*literal*⟩ | ⟨*sform*⟩

⟨*literal*⟩ ::= ⟨*char-not-sep*⟩$^+$

⟨*operator*⟩ ::= ⟨*char-not-sep*⟩$^+$

⟨*sform*⟩ ::= ⟨*typed-seq*⟩ | ⟨*start*⟩ ⟨*char*⟩$^*$ ⟨*end*⟩?

⟨*typed-seq*⟩ ::= '`[:`' ⟨*typeid*⟩ (⟨*sep*⟩$^+$ ⟨*data*⟩)? ⟨*sep*⟩$^*$ '`]`'

⟨*typeid*⟩ ::= ⟨*literal*⟩

⟨*sep*⟩ ::= ⟨*space*⟩ | ⟨*tab*⟩ | ⟨*newline*⟩

⟨*start*⟩ ::= ⟨*char*⟩$^+$

⟨*end*⟩ ::= ⟨*char*⟩$^+$

⟨*char*⟩ ::= *any character*

⟨*char-not-sep*⟩ ::= *any character except* ⟨*sep*⟩

Figure 5.1: Syntactic Framework for Programs

## 5.1 Syntactic Framework

Concat provides a great degree of syntactic freedom for creating languages and for extending its own meta-language. By implementing a layered approach to syntax, Concat combines the advantages of a uniform syntactic framework that serves as a basis for pattern-based program manipulation with unrestricted syntactic freedom and fine-grained control on the character-level.

### 5.1.1 Typed Sequence Notation

In Concat, data is stored internally either as an atomic value or a typed sequence $\tau(S, T)$ where $T$ is an atomic type identifier and $S$ is a sequence. Concat exposes its internal representation through a uniform notation for typed sequences. The grammar in Figure 5.1 introduces the core syntactic concepts. Its purpose is instructional rather than definitorial as it is ambiguous and some of its productions have the sole purpose of introducing syntactic concepts. The grammar describes the syntactic boundaries of languages in Concat and not the syntax of a concrete language, as will be explained below.

The notation for programs and data is a sequence of elements with separators in be-

tween. An element is either an operator, a literal or a special form. A separator is either a tab, a space or a newline character. Both operators and literals are sequences of characters that must not contain a separator. The distinction between operators and literals is part of concrete language grammars that further refine the grammar in Figure 5.1. Defining literal syntax is a basic mechanism for syntactic abstraction in Concat. As separators are not allowed in literals, the mechanism is, however, restricted to rather basic syntactic structures. Therefore, literals are primarily used for syntactically encoding elementary types, e.g., numbers, characters, symbols. Examples of literals are `23`, `'symbol` and `1/3`.

Special syntactic forms provide more freedom for syntactic extension than literals. A special syntactic form is an arbitrary sequence of characters including separators. To avoid ambiguities, a unique start-tag and, in case of variable length content, an end-tag is required. By convention, the start-tag begins with the character `` ` `` and the end-tag is the character `;`. Exceptions are possible, for example in the case of strings where both the start and end-tag is a quotation mark. Examples of special syntactic forms are `` `infix 2 + 3; `` and `` `select * from cars;``.

Typed sequences are special forms with the start-tag `[:` and the end-tag `]`. Between the two, there is a type identifier followed by data. The definition is indirectly recursive because typed sequences may contain data, data may contain elements, special forms are elements and a typed sequence is a special form. The recursive definition allows nesting of typed sequences up to an arbitrary depth. Furthermore, because elements may be part of the data, concrete syntax can be mixed with the generic typed sequence syntax; examples are `[:rational 20 3]` and `[:assoc [:key 'a] [:val 10]]`.

Typed sequences provide a notation for the *internal representations of programs*. Languages using the same representation for programs and data are called homoiconic [95, 116].

### 5.1.2 Syntactic Layering

The grammar presented in the previous section does not define the actual literals, special forms and typed sequences a concrete user-defined language supports. Instead, it defines on a more abstract level the syntactic boundaries, i.e., the set of possible operators, literals and special forms shared by all languages. Language definitions *refine* the above grammar by restricting these elements to a subset. For instance, a language supporting only numbers and boolean as data types restricts literals accordingly. In effect, it replaces the abstract production

$\langle literal \rangle ::= \langle char\text{-}not\text{-}sep \rangle^*$

with the more concrete production

⟨*literal*⟩ ::= ⟨*number*⟩ | ⟨*bool*⟩

Because the example language contains only numbers and booleans, it also restricts special forms to the empty set – with the effect of disallowing the use of any typed sequences in the language.

Table 5.1 shows the five levels of syntax in Concat. Every level defines a set of possible representations. Every level $R_n$ for $n > 0$ defines a subset of the level $R_{n-1}$. The choice of a character set on Level $R_1$ is the first restriction on the representational capabilities of the machine to character strings according to an encoding scheme. Representations on level $R_2$ are sequences of elements. On this level, typed sequences do not exist. The grammar in Figure 5.1 defines Level $R_3$ through the production $typed\text{-}seq$. On $R_3$, typed sequences are built-in special forms. Level $R_4$ refines $R_3$ by specifying only those literals, special forms and typed sequences available in the language. The mechanisms that allow this in Core Concat are views, operations and macros.

| Level | Description | Restriction |
|---|---|---|
| $R_0$ | arbitrary representation | limited by machine |
| $R_1$ | string | character code |
| $R_2$ | elements | Concat fixed grammar |
| $R_3$ | elements + typed sequences | Concat framework grammar |
| $R_4$ | concrete elements | language-specific grammar |

Table 5.1: Layers of Syntax in Concat

The stepwise refinement of language syntax just described is a common technique in language frameworks based on a uniform syntax, e.g., XML. Instead of defining the grammar of a language directly as a restriction of the Kleene closure on a character set ($R_1$), an intermediary syntactic layer is introduced on which all languages are based. Concat uses a variation of this technique by defining two separate layers $R2$ and $R3$.

The main advantage of this approach is that layer $R_3$ allows a uniform way of structured access that is independent of the particular grammar of a language on $R_4$. All languages on $R_4$ can be processed with tools built for $R_3$, e.g., with pattern operations that work on typed sequences. The disadvantage is that the syntactic freedom of languages is restricted by the need to conform to the syntactic framework imposed on $R_3$.

Concat utilises the benefits of a uniform syntax while, at the same time, providing a mechanism to break out of the uniformity. Literals and special syntactic forms defined on $R2$ are also part of $R3$. The syntactic freedom they offer creates, in effect, an escape mechanism from $R_4$ to $R_1$. Custom grammar based on strings can be combined with uniform syntax of typed sequences. The fact that in Concat strings and characters are just

certain types of typed sequences allows their manipulation with the same pattern-based mechanisms defined for typed sequences on $R_3$, as will be shown in the following.

### 5.1.3 Unified Program Representations

Concat is based on the uniform representation of character strings and tree structures introduced in Section 4.4. It represents characters and strings as typed sequences. Characters are typed sequences containing an atom that encodes the character. In Concat, atoms are literals that start with `/a` followed by the atomic value in parentheses. Strings are typed sequences with characters as content. For example, string ”abc” is represented by the following typed sequence:

```
[:string [:char /a(a)] [:char /a(b)] [:char /a(c)]]
```

Concrete syntaxes for characters and strings are views on the internal representation. The example string can be written as `"abc"` and character `[:char /a(c)]` can be written as `^c`. The fact that characters and strings are just another form of typed sequences is an important design principle in Concat. It allows a unified treatment of all stages of program processing through pattern matching.

The distinction between scanning and parsing a character representation is not required as Concat’s uniform representation naturally supports scannerless parsing [164]. Nevertheless, using the vertical combinator it is possible to introduce these phases as a pattern of the form $(Ptn_{scan} \rightarrow Ptn_{parse})$ which first matches $Ptn_{scan}$ with a string and then $Ptn_{parse}$ with a sequence of tokens.

Concat is designed to recognise, manipulate and create its own character representation and is thus largely self-contained. What it requires, however, is an interface to the outside world to display the encoded characters. For example, the character representation of the number 23 is `[:char /a(2)] [:char /a(3)]`. In cases where this representation is not explicitly desired, it must be displayed as `23`.

### 5.1.4 Standard Notation for Patterns

Concat’s meta-language provides a concrete syntax for the pattern expressions defined in the previous two chapters. Table 5.2 introduces this syntax by example. Sequencing, choice, and vertical and diagonal combination are defined by left-associative infix operators. Precedence can be expressed by grouping patterns inside parentheses. The variable notations `$x`, `#x` and `@x` are shorthand for `x:item`, `x:any` and `x:any*` respectively. These shorthands are implemented using the meta-level view mechanism, see Section 5.5.2. Angle brackets may be omitted for unparameterised references used as the

| Category | Pattern | Examples |
|---|---|---|
| Basic | atom | `/a(b)` |
| | any | `<any>` |
| | variable | `x:any`, `$x`, `#x`, `@x` |
| | param. reference | `<<space> <nat> sep_list>` |
| Types | character | `^a ^space` |
| | string | `"concat string"` |
| | symbol | `'concat_symbol` |
| Hierarchical | typed sequence | `[:rational 1 2]` |
| | untyped sequence | `[1 2 3]` |
| Modifiers | negation | `!<a>` |
| | maybe | `<a>?` |
| | ignore | `&i(<a>)` |
| | escape | `/p(a:1)` |
| | group | `(<b> \| <c>)` |
| Horizontal | sequencing | `<a> <b> <c>` |
| | choice | `<a> \| <b> \| <c>` |
| | repetition | `<a>*` |
| | repetition $\geq 0$ | `<a>+` |
| | characters | `/s(abc)` |
| Two dimensional | vertical | `<a> -> <b> -> <c>` |
| | diagonal | `<a> /> <b> /> <c>` |
| Transformative | uncond. transformation | `(a => b)` |
| | cond. transformation | `do (a => b)` |
| | | `if c => d && e => f.` |
| Quoting | literal quote | `&l(a b c)` |
| | quasiquote | `&q(a b c)` |
| | unquote | `&q(a ,b c)` |

Table 5.2: Concrete Syntax for Pattern Expressions

pattern part of a variable. The pattern `<item>` matches all elements of a language except operators. The characters `&` and `/` followed by a letter control the recognition process. Atoms start with `/a` and `/s(abc)` is a shorthand for `(^a ^b ^c)`, i.e., a sequential pattern that matches three characters. The pattern expression `&i(<p>)` ignores the result of the pattern reference `<p>`, `&l` and `&q` are used for quoting. Like regular references, parameterised references are surrounded by angle brackets and use postfix notation.

Concat allows the use of program-level literals in patterns. For instance, the choice pattern `1|2` contains number literals. The parser of the meta-language first attempts to recognise parts of a pattern as meta-elements of the pattern language. If that fails, the parser attempts to recognise a program-level literal. This may lead to ambiguities between program and pattern syntax. For instance, literal syntax that represents a key-value pair with key `a` and value `1` as `a:1` is in conflict with the variable syntax on the meta-level.

To avoid that the occurrence of such a literal in a pattern is parsed as a variable, the literal has to be escaped. The pattern `/p(a:1)` explicitly states that the content between the parentheses is a program-level literal.

Section 5.5.2 shows how the standard pattern syntax of Concat can be defined in a meta-circular way, including the escape mechanism just discussed. Section 5.3.4 defines a variable notation for terms in combinatory logic that is an example for how alternative syntaxes for patterns can be introduced. The meta-language of Core Concat defines a set of computational concepts that are based on the pattern syntax just described.

## 5.2 Concepts of Core Concat

Core Concat is based on a set of computational concepts and associated syntax and semantics that have proven useful in implementing several languages including Concat itself. The concepts of Core Concat are operations, macros, productions and views. *Views* introduce new syntax by configuring the stages of internalisation and externalisation. *Operations* and the associated execution model define the principal means of computation by configuring the computation stage. *Macros* add pre-processing to the internalisation, computation and externalisation stages in the form of unrestricted program transformations. *Productions* provide an abstraction mechanism for patterns that is utilised by the other core concepts. Internally, all these different concepts perform pattern operations. Nevertheless, the combination of introducing custom syntax (based on the pattern syntax of the previous section) and hiding the actual pattern operations gives the concepts direct semantics [103]. Transformations, in conditional and unconditional form, play an important role in Core Concat as they are the basis of operations, macros, productions and views.

### 5.2.1 Abstracting Patterns with Productions

Productions define pattern abstractions based on the parameterised references introduced in Section 3.7. Productions are meta-transformations, i.e., transformations that match patterns with patterns and instantiate patterns with bindings that have patterns as values. To distinguish between patterns in their role as active code and passive data, the quasiquotation mechanism introduced in Section 3.6.4 is used. The right-hand side of a production is automatically quasiquoted during internalisation. The unquote operator (written as a comma) is used to activate parts of passive patterns. Productions are capable of expressing more than just references to patterns. Parameterised productions may have an

arbitrary number of arguments that can be used to construct the pattern on the right-hand side. Listing 3 shows three examples of productions.

**Listing 3** Examples of Productions

```
1 keyword ⇒ 'if | 'do | 'while.
2
3 $x ? ⇒ ,$x | <nothing>.
4
5 $sep $elem sep_list ⇒
6     (elems:(,$elem (&i(,$sep) ,$elem)*) ⇒ [@elems]).
```

The first production is unparameterised. It associates the name `keyword` with the pattern `'if | 'do | 'while`. Each occurrence of `<keyword>` is dynamically resolved to this pattern during matching or instantiation .

The second production defines the pattern operator `?` that expresses that a matching or instantiation is optional. The production is parameterised and expects a single pattern as argument. The comma preceding the variable `$x` on the right-hand side of the rule is concrete syntax for the unquote operator. It has the effect of activating the variable and turning it into a meta-variable that ranges over patterns. When a parameterised reference to the `?`-pattern is resolved, the variable is instantiated with the pattern passed as an argument. For example, the pattern `<<item> ?>` is resolved to the pattern `<item>|<nothing>`. The semantics of the choice operator defines that `<item>` is tried first and only if `<item>` fails `<nothing>` is tried second – and always succeeds.

The third production abstracts a parser for string encodings of value sequences, e.g., comma-separated numbers or white-space separated items. The production expects two arguments: a pattern that matches the separators and a pattern that matches the elements being separated. For instance, assuming that the pattern `<i_nat>` transforms string encodings of natural numbers into an internal representation, the following pattern transforms a string of number encodings separated by a comma or semicolon into a list of natural numbers in internal representation: `<(^,|^;) <i_nat> sep_list>`. As defined in Section 5.1.4, character literals are prefixed with the symbol `^`. The pattern is resolved to the following transformation:

```
(elems:(<i_nat> (&i(^,|^;) <i_nat>)*) ⇒ [@elems])
```

The left-hand side of the transformation tries to match the first number by matching `<i_nat>` with the string. It then repeatedly tries to match one of the separators followed by `<i_nat>`. The compound matching result is bound to the variable `elems`

which is instantiated inside an untyped sequence on the right-hand side to yield the list of internalised values.

### 5.2.2 Creating Syntactic Interfaces with Views

The view mechanism of Concat provides means to hide the actual representation of data types behind a user-defined syntactic interface. This is necessary because of Concat's homoiconic nature. Hiding is achieved through a bidirectional mapping between external and internal representation. For example, the rational number $\frac{1}{10}$ can be represented internally as a typed sequence $\tau(\langle 1, 10\rangle, rational)$ which can be written in Concat as `[:rational 1 10]`. A view on rational numbers allows programmers to use the literal syntax `1/10` instead of the generic typed sequence notation. Both the generic typed sequence notation and the literal notation have the same internal representation. Technically, the new syntax introduced by the view is just an alternative. Conceptually, it can be used to hide the actual representation of rationals by making it illegal to write the typed sequence in a program that uses rational numbers, but legal in a context where basic functions such as addition or multiplication for rationals are implemented.

A view consists of two transformations that map between internal and external representation of data. The transformations are defined as a set of productions. A view definition starts with the keyword `view` followed by the identifier of the type for which the view is defined. The two blocks that define the transformation for internalisation and externalisation begin with keywords `:int` and `:ext` respectively. Each block consists of a non-empty set of productions. The first production in each block must be the starting point of the transformation. The following example is a view definition for positive rational numbers.

**Listing 4** View Definition for Rational Numbers

```
1  view rational:
2    :int
3       i_rational ⇒
4          (n:i_nat ^/ d:i_nat ⇒ [:rational $n $d]).
5       i_nat ⇒
6          (d:i_digit* ⇒ [:nat @d]).
7    :ext
8       e_rational ⇒
9           ([:rational n:e_nat d:e_nat] ⇒ @n ^/  @d).
10      e_nat ⇒
11          ([:nat d:e_digit*] => @d).
```

In the Listing 4, `i_rational` internalises rational numbers with a reference to the production `i_nat` and `e_rational` externalises rational numbers with a reference to the production `e_nat`. The prefixes `i_` and `e_` are a convention to distinguish between productions for internalising and externalising.

By default, syntax defined for the program level is also available at the meta-level. For example, the concrete syntax for rational numbers, i.e., its external representation, can be used in a pattern such as `(1/3|1/2)`. In cases where this leads to ambiguities, the program-level syntax can be explicitly escaped using the program escape operator `/p`. With explicit escape, the example above becomes: `(/p(1/3)|/p(1/2))`

### 5.2.3 Defining Semantics with Operations

*Operations* are the main computational mechanism in Core Concat. Operation definitions consist of a number of cases that define a mapping between arguments and a result in a functional way. Each case is defined by an unconditional or conditional transformation. Executing the operation corresponds to trying the cases in order on the input until one succeeds and produces the result of the operation.

Operation definitions start with the keyword `opdef` followed by the operator symbol, a colon and a sequence of transformations terminated with a full stop. In case there is only a single transformation with an empty left-hand side, the colon in the operation definition may be omitted. Listing 5 shows examples of simple operations on sequences.

**Listing 5** Basic Sequence Operations

```
1 opdef first:
2   [$x @y]  ⇒ $x.
3   []  ⇒ 'error.
4
5 opdef rest:
6   [$x @y]  ⇒ [@y].
7   []  ⇒ 'error.
8
9 opdef second ⇒ rest first.
```

The operation `first` returns the first item in a sequence and the operation `rest` returns the sequence with the first item removed. Both return the symbol `'error` if the argument sequence is empty.

Operations are purely functional in the sense that their results depend solely on argument values that need to be items. However, the application of operations in Core Concat is not based on function calls with a fixed number of individual arguments, but on a single argument that is a sequence. Programs are executed strictly from left to right with each operation working on the results of previous operations. Data that is unaffected by an operation is passed on to the next operation together with the result. Syntactically, this means that programs are written in postfix notation. Macros are built-in means to break out of this syntactic and semantic scheme. Operators may occur on the right-hand side of transformations within operation definitions. This corresponds to "calling" other operations. The definition in line 9 of Listing 5 abstracts the program `rest first` by introducing the operator `second`. The following lines show the stepwise execution of a program where `rest` and `first` are applied to a sequence:

```
[1 2 3] second
[1 2 3] rest first
[2 3] first
2
```

Internally, operations are rewritten into rules of a pattern-based concatenative rewriting system as defined in Section 4.2.4. An operation definition of the form

```
opdef name:
  (lhs1 ⇒ rhs1).
  ....
  (lhsN ⇒ rhsN).
```

is transformed into a sequence of choices. Each choice is a transformation:

```
lhs1 name ⇒ rhs1 | ... | lhsN name ⇒ rhsN
```

The left-hand side of the transformations have the operator name inserted as the last element on the left-hand side. In effect, the operation definition ensures that each rule has an operator name that serves as a dispatch. By checking that the left-hand side does not contain operators during parsing, the concatenative programming style can be enforced. Listing 20 in Section 5.5.2 defines the transformation of an operator definition into a choice pattern.

Listing 6 shows the implementation of `map`, an operation that applies a program to each element in a sequence and produces a sequence that contains the results.

**Listing 6** Mapping over Sequences in Concat

```
1 opdef map:
2     do ([ $f @r ] [ @p ] ⇒ [ $f1 @r1 ])
3     if   $f @p ⇒ $f1
4     &&  [ @r ] [ @p ] map ⇒ [ @r1 ].
5     [ ] [ @p ] map ⇒ [ ].
```

The stepwise execution of the `map` operation will be discussed by example based on the program `[[1][2][3]] [first] map`. This program has the result `[1 2 3]`. The first transformation starting in line 2 implements the recursive case of the operation. It has two conditionals that are defined in line 3 and 4. The application of the operation to the argument `[[1][2][3]] [first]` starts by attempting to match the data with the left-hand side of the operation, which is the pattern `[$f @r] [@p]`. The result of this match are the following variable bindings that are added to the store: $(\texttt{f}, \texttt{[1]})$, $(\texttt{r}, \langle \texttt{[2][3]} \rangle)$ and $(p, \langle \texttt{first} \rangle)$. In the context of these bindings, the left-hand side of the first conditional `$f @p` is instantiated. The result is the program `[1] first`. This program is executed and yields the result `1` which is matched by pattern `$f1`. The resulting binding $(\texttt{f1}, 1)$ is added to the store. With the updated store, the left-hand side of the second conditional `[@r][@p] map` is instantiated. The result is `[[2][3]] [first] map`.

The execution of this resulting program follows the execution pattern just described and leads to more recursive calls "waiting" for the result, each with its own instance of a store. The two final recursive calls of the execution are `[[3]] [first] map` and `[] [first] map`. During the execution of the latter, the first transformation fails to match and, therefore, the second transformation – the base case – is tried. The base case yields the result `[]` which, when matched with `[@r1]`, yields a binding $(\texttt{r1}, \epsilon)$. After

this, the instantiation of the pattern `[$f1 @r1]` results in `[3]`. This process of returning the result and instantiating the right-hand side of the transformation header is repeated for all waiting "calls" which leads to the stepwise construction of the sequence `[1 2 3]`.

The operation `map` is *higher-order* in the sense that it takes a program as an argument and executes it. The execution is expressed by the left-hand side of the conditional in line 3. The variable `p` is bound to the content to of the second sequence which, for this example, is the program `first`.

The operations examined so far operate on untyped sequences and use generic types for variables. Listing 7 shows an example of how patterns can be used to ensure correct types are passed to the operation.

**Listing 7** Example for using Typed Sequences in Operations

```
1 opdef valid_char:
2   do ([:string c:char r:char* ] ⇒ $c [:string @r])
3   if   $c valid? ⇒ true.
```

The operation `valid_char` removes the first character from a string if the operation `valid?` yields `true` for this character.

### 5.2.4 Program Transformation with Macros

Although operations and views are based on program transformations they restrict the kinds of transformations that can be defined in order to enforce certain execution semantics, as demonstrated in Section 4.2. Operations associate operator symbols with functional semantics. This implies that the transformations must not "touch" operators unless these operators are inside a sequence.

Views define alternative representations for a typed sequence. The target of the internalisation transformation and the source of the externalisation transformation must be typed sequences. Some useful computations, for example code optimisations or syntax definitions with programs rather than typed sequences as targets, cannot be expressed directly with operations or views because of these restrictions.

Macros work around these restrictions by providing a mechanism for defining unrestricted transformations. There are two types of macros. Computation macros operate on programs in internal representation and only work on the outermost nesting level of programs, i.e., they are not applied to the content of sequences. Internalisation macros transform parts of an external program representation into an internal representation. They are applied to data at all nesting levels, i.e., also on data contained in sequences.

Internalisation and externalisation macros are defined using the keyword `imacro` and `emacro` respectively followed by a pair of parentheses containing a start-tag and, optionally, an end-tag. The ensuing sequence of transformations defines different cases of program transformations. After the keyword `with`, a set of productions may follow. In Listing 8, a macro allowing simple infix operations is defined.

**Listing 8** Example of an Internalisation Macro

```
1 imacro (% ;):
2     <ws>* x:i_nat <ws>* o:i_op <ws>* y:i_nat <ws>*
3     ⇒ $x $y #o.
4   :with
5     i_op ⇒ ((^+  | ^- ) -> <to_op>).
```

The start-tag of the infix operation is the character `%`, the end-tag is the character `;`. The macro recognises two numbers with either an operator + or an operator − between them and transforms both the numbers and the operator into an internalised postfix representation. Whitespaces are recognised by `<ws>` and may be interspersed arbitrarily. The production `i_op` recognises the operators on the character levels and using vertical combination applies the built-in `<to_op>` pattern to yield an internal operator representation. Because the resulting value is not an item, a `#`-variable is used on the right-hand side of the transformation to instantiate the operator. An example of a macro application is the transformation of the program `1 % 2 + 3 ; 4` into the program `1 2 3 + 4`. In effect, the macro takes complete control of the parsing process between `%` and `;`.

Externalisation macros allow an unrestricted mapping for programs from internal to external representation. The difference compared to the externalisation part of views is that the scope of the macro transformation is not limited to a single typed sequence.

Concat transforms internalisation and externalisation macros into rewrite rules. This is achieved by a meta-transformation that prepends and appends the start-tag and the end-tag respectively to the left-hand side of the transformation. For example, the internalisation macro in Listing 8 is transformed into the rule:

```
/s(&) <ws>* x:i_nat <ws>* o:op <ws>* y:i_nat <ws>* /s(;)
⇒ $x $y #op.
```

While internalisation and externalisation macros have external program representations as source and target respectively, computation macros operate entierely on the internal program representation. Syntactically, computation macros start with a keyword `cmacro`,

followed by the macro name and a colon. A set of transformations defines the semantics of the macro. The transformations are unrestricted and can arbitrarily manipulate the program. The computation macro in Listing 9 defines optimisations for shuffle words.

**Listing 9** Example of a Computation Macro

```
1 cmacro optimisations:
2     dup drop ⇒ .
3     swap swap ⇒ .
```

The macro eliminates pairs of operations that have no effect on the result and thus need not be executed. The optimisations only apply to operations that are not contained in sequences. The reason for this is that operations might be applied to code in sequences before that code is executed. Optimisations inside sequences could lead to undesired results. For example, the program `[dup drop] length` is expected to yield the result `2` but will yield `0` if the optimisation is performed before execution.

### 5.2.5 Pattern Matching with Concrete Syntax

A view definition introduces an alternative notation for a typed sequence. The notation can be used both at the program and at the meta-level. For example, the literal syntax of strings in Core Concat defines a notation that allows writing `"abc"` instead of `[:string ^a ^b ^c]`. When this notation is used in a pattern, it means that the string is matched or instantiated literally.

Patterns can be defined based on a string's typed sequence representation. For example, the pattern `[:string ^a ^b @r]` matches a string that starts with the characters `^a` and `^b` followed by an arbitrary number of characters `@r`.

Core Concat provides a mechanism for defining patterns based on literal syntax. Since the internalisation at the program level and at the meta-level can be controlled separately, there is no need to use the same view definition for both levels. Thus, the view defined for the pattern-level may be based on typed sequence patterns such as the one just described. For example, instead of defining the internalisation of strings as

```
^" s:<str-char>* ^" ⇒ [:string @s]
```

it can be defined to include patterns. The default mechanism to do this is with the predefined pattern `<ptn-esc>` (for pattern escape) that matches any pattern between two backslashes. Changing the above definition to

```
^" s:(<ptn-esc>|<str-char>)* ^" ⇒ [:string @s]
```

allows to write a pattern in place of a character. Using this definition, the pattern `[:string ^a ^b @r]` can be expressed as `"ab\@r\"`. This allows definitions such as the first version of `ab-prepend` in Listing 10:

**Listing 10** Concrete Syntax Manipulation of Strings

```
1 opdef ab-prepend:
2   "\@s\" => "ab\@s\".
3
4 opdef ab-prepend:
5   [:string @s] => [:string ^a ^b @s]
```

The internalisation stage of the pattern language actually transforms the operation definition into the same internal representation as the second version of `ab-prepend`.

## 5.3 Case Study: Implementing Combinatory Logic

This section presents a comprehensive case study of implementing a language in Concat: combinatory logic [33]. Combinatory logic was introduced by Haskell Curry and Moses Schönfinkel in the 1920s. Originally meant for investigating the foundations of mathematics, combinatory logic is now an important tool in research on programming languages [75]. It consists of several systems of combinators that, similar to lambda calculi, provide a formal foundation for computing. The difference to lambda calculi is that combinatory logic is not based on bound variables and thus avoids substitution and $\alpha$-conversion altogether [75].

Despite its relatively simple syntax and semantics, combinatory logic is well suited to demonstrate a wide range of Concat's features: typed sequences as term representation, operations that work on the term representation, program-level views that define concrete notations for terms and transform bidirectionally between concrete notation and internal term representation, meta-level views that allow variables in SKI terms and provide the basis for defining operations using concrete syntax as well as dynamically parameterised productions for recognising left-associative combination of terms.

### 5.3.1 Definitions

This section defines a combinatory logic based on the combinators $S$, $K$ and $I$. The system is also called SKI calculus or SKI combinator calculus. The alphabet of the pure

SKI calculus is:

$$\sum = \{S, K, I, (, )\}$$

SKI terms are defined recursively as follows:

- $S$, $K$ and $I$ are SKI terms
- if $x$ and $y$ are SKI terms, then so is $(xy)$

Examples of SKI terms are $((SK)I))$ and $((S(KS))K)$. Let the concatenation of two terms $x$ and $y$ to a term $xy$ with no surrounding parentheses be equivalent to $(xy)$ and let the concatenation operation be left-associative. Accordingly, the two examples can be rewritten as follows:

$$SKI \equiv ((SK)I))$$

$$S(KS)K \equiv ((S(KS))K)$$

Extending the alphabet by constants that explicitly represent data simplifies reasoning about the SKI-Calculus. In the following, the symbols $a$, $b$ and $c$ will be used to refer to data items. The reduction of an SKI term is defined by the rules in Table 5.3. The symbols $x$, $y$ and $z$ stand for arbitrary terms.

| | | |
|---|---|---|
| $Sxyz$ | $\rightarrow$ | $xz(yz)$ |
| $Kxy$ | $\rightarrow$ | $x$ |
| $Ix$ | $\rightarrow$ | $x$ |

Table 5.3: Rules of the SKI Calculus

The rules define replacement on subterms within an SKI term. For example, the term $aIbc$ reduces to $abc$ because the subterm $Ib$ is reducible by an application of the third rule. The following is an example of a stepwise derivation of a term that has the effect of swapping the positions of $a$ and $b$.

$$(K(SI))Kab \rightarrow K(SI)a(Ka)b \rightarrow$$

$$SI(Ka)b \rightarrow Ib(Kab) \rightarrow$$

$$b(Kab) \rightarrow ba$$

There is an alternative derivation of this term because in the term $Ib(Kab)$ both $Ib$ and $Kab$ are reducible expressions. Reducing $Kab$ first, the stepwise derivation is:

$$Ib(Kab) \rightarrow Iba \rightarrow ba$$

### 5.3.2 Basic Implementation

In the following, the implementation of the SKI combinator calculus in Concat will be discussed. A representation of SKI terms has to fulfil three requirements:

- encode the atomic terms $S$, $K$ and $I$ of the pure calculus and all atomic constants added, e.g., $a$, $b$ and $c$
- represent the nested structure of compound SKI terms
- be uniquely identifiable as an SKI term

For the representation of the atomic terms of the calculus, the generic symbol data type available in Core Concat is used. Symbols start with an apostrophe and may consist of any characters except separators. The SKI term $S$ is represented by the symbol `'S`. Compound terms are represented by untyped sequences. For example, the content of the term $SKI$ is represented by the structure `[['S 'K] 'I]`. In order to make SKI terms uniquely identifiable, a sequence of type *ski* is used to wrap the actual content of the term. The complete representation of the term $SKI$ is `[:ski [['S 'K] 'I]]`. The rules of the ski calculus are implemented as a reduction operation on this representation as shown in Listing 11.

**Listing 11** Reduction in the SKI Combinator Calculus

```
1  opdef  reduce:
2   [:ski ['I $x]] ⇒ [:ski $x].
3   [:ski [['K $x] $y]] ⇒ [:ski $x].
4   [:ski [['S $x] $y] $z]] ⇒ [:ski [[$x $z] [$y $z]]].
5
6   do ([:ski [ $x $y ]] ⇒ [:ski [ $z $y ]])
7   if  [:ski $x] reduce ⇒ [:ski $z].
8
9   do ([:ski [ $x $y ]] ⇒ [:ski [ $x $z ]])
10  if [:ski $y] reduce ⇒ [:ski $z].
```

The operation `reduce` consists of three unconditional transformations (lines 2-4) and two conditional transformations (lines 6-7). The unconditional transformations implement the reductions for $S$, $K$ and $I$ as defined in Table 5.3. As these reductions are defined on subterms, the implementation must provide a mechanism to descend into the nested structure of the term representation to find reducible subterms. This is implemented with

two conditional rules. If $x$, $y$ and $z$ are SKI terms, the rule starting in line 6 can be read as: If term $x$ can be reduced to term $z$, then term $xy$ can be reduced to term $zy$. Alternatively: The term $xy$ can be reduced to term $zy$, if term $x$ can be reduced to term $z$. The transformation starting in line 9 has similar semantics. The difference is that $y$ is the reducible subterm. As the definitions are recursive, reducible subterms are found at an arbitrary depth. The reduction of the term $b(Kab)$ to $ba$ will be discussed as an example. The program that expresses the reduction of the term is:

```
[:ski ['b [['K 'a] 'b]]] reduce
```

When the program is executed, the first three unconditional transformations in reduce fail since the term does not begin with $S$, $K$ or $I$. The first conditional transformation also fails because instantiating the left-hand side of the conditional with the binding $(x,$`'b`$)$ yields `[:ski 'b] reduce` which fails. The last transformation succeeds because the variable `$y` is bound to `[['K 'a] 'b]`. This leads to the left-hand side of the conditional being instantiated to `[:ski [['K 'a] 'b]] reduce`. Executing this program yields `'a`, which is bound to `$z`. The resulting term after instantiating the right-hand side of the transformation header is `[:ski ['b 'a]]`.

The operation `reduce` defines a single derivation step for SKI terms. A complete derivation may consist of several steps until a canonical form is reached, which means that there are no more reducible subterms. This is implemented by the operation `derive` as shown in Listing 12.

**Listing 12** Derivation in the SKI Combinator Calculus

```
1 opdef derive:
2   do ([:ski $x ] ⇒ [:ski $y ])
3   if  [:ski $x ] reduce derive ⇒ [:ski $y ].
4   [:ski $x] ⇒ [:ski $x].
```

The recursive case of derive succeeds if sequential application of `reduce` and `derive` on a term succeeds. In that case, it yields the result of this application. The base case always succeeds and does not change the term. The derivation of the term $Ib(Kab)$ to $b(Kab)$ and finally to $ba$ will be discussed as an example. When the corresponding program

```
[:ski ['I ['b [['K 'a] 'b]]]] derive
```

is executed, the first transformation of `derive` is attempted. This leads to the execution of

```
[:ski ['I ['b [['K 'a] 'b]]]] reduce derive
```

In `reduce` the first transformation matches and yields:

```
[:ski ['b [['K 'a] 'b]]]
```

This is the term of the previous example. The operation `derive` is applied to this term:

```
[:ski ['b [['K 'a] 'b]]] derive
```

Again, the recursive case is attempted:

```
[:ski ['b [['K 'a] 'b]]] reduce derive
```

This time the last transformation of `reduce` succeeds and leads to the application of the transformation for $K$. The result is `[:ski ['b 'a]]`. The operation `derive` is applied to this result and again the recursive case yields:

```
[:ski ['b 'a]] reduce derive
```

This time `reduce` fails and thus the conditional of the recursive case of `derive` also fails. Therefore, the base case of `derive` is tried which terminates the recursion and yields the final result `[:ski ['b 'a]]`.

### 5.3.3 Concrete Syntax for SKI Terms

The implementation discussed in the previous section can be used for experimenting with reductions of the SKI calculus. By introducing new rules, the calculus can be extended with new combinators. However, the verbosity of the representation of SKI terms makes this inconvenient. Unlike in the mathematical notation, all parentheses need to be written, the symbols have to start with an apostrophe and the terms must be surrounded by a typed sequence. In the following, the introduction of a concrete syntax based on the mathematical notation will be discussed. The syntax is implemented as a view on the typed sequence representation. Transformations need to be defined that map bidirectionally between the mathematical notation and the internal representation used so far.

As discussed in Section 5.3.1, the mathematical notation of SKI terms can be written with or without parentheses. If there are no parentheses the concatenation of terms is left associative. Parsers for left associative operators are typically implemented using left recursive grammar rules [133]. However, as the execution semantics of Core Concat imply that patterns are matched strictly from left to right, left recursion leads to infinite

regress. This is a well known problem for top down parsers [3]. In Core Concat, this problem is solved by providing a mechanism to avoid left-recursion altogether: dynamically parameterised productions.

The following code defines the internalisation part of the view definition for SKI terms. It can be understood as a grammar that defines how a term is parsed into an internal representation.

**Listing 13** Grammar for Internalising SKI Terms

```
1  ski_start ⇒ ^`.
2  ski_char ⇒ ^S | ^K | ^I | ^a | ^b | ^c.
3  ski_sym ⇒ (c:<ski_char> ⇒ [:symbol $c]).
4  ski_group ⇒ (^( t:<ski_term> ^) ⇒ $t).
5  ski_item ⇒ <ski_sym> | <ski_group>.
6  ski_term ⇒ (&i(s:<ski_item>) <$s ski_rest>)
7              | <nothing>.
8  ski_literal ⇒ (<ski_start> t:<ski_term> ⇒ [:ski $t]).
9  $v ski_rest ⇒ ((&i(s:<ski_item>) <[,$v $s] ski_rest>)
10                | (<nothing> ⇒ ,$v)).
```

The only difference between the original mathematical notation defined in Section 5.3.1 and the one introduced in Concat is that every SKI term starts with an apostrophe. This is defined by `ski_literal` and `ski_start`. Production `ski_term` defines that an SKI term consists of an `ski_item` and an `ski_rest` or is empty. An `ski_item` is an `ski_symbol` or an `ski_group`. SKI symbols (`ski_sym`) are represented by characters `S`, `K`, `I`, `a`, `b` and `c`. An `ski_group` is an `ski_term` in parentheses. The parameterised production `ski_rest` expects the current result of the matching process. It attempts to match an `ski_item` and, if successful, calls itself recursively with a sequence that consists of its previous argument and the result of recognising `ski_item` – both combined into a sequence. If there is no `ski_item` to match, i.e., the term is completely recognised, the argument is returned.

Figure 5.2 depicts the parse tree for the term `` `S(KS)K ``. To improve the presentation, the `ski_` prefix is omitted from all tree nodes. The parse result is the structure `[:ski [['S ['K 'S]] 'K]]`. The parsing process corresponds to a left-to-right, depth first traversal of this tree. Without parameterised productions, the data flow for constructing the result of a parse is strictly from the leafs *upwards* to the root, with each parent node combining or discarding results from its children.

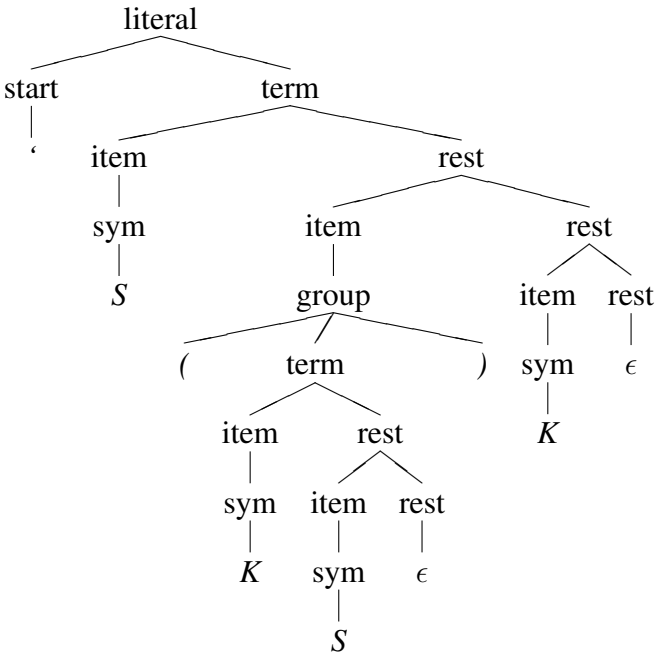


Figure 5.2: Parse Tree for SKI Terms

Because the grammar in Listing 13 avoids left recursion and thereby automatic association to the left, the results need to be combined differently in order to reflect the left-associativity of concatenating SKI terms. Passing parameters to `ski_rest` corresponds to sending results *downwards* in the tree. It is helpful to think about the parsing process in terms of function calls. For example, the node $term$ that is under the node $group$ in Figure 5.2 has two children: $item$ and $rest$. First, $term$ calls $item$, $item$ calls $sym$ and finally $sym$ recognizes `K`. The result `K` is returned to $term$, which uses it as a parameter when calling $rest$. This is the point where the result is passed downwards. Next, $rest$ calls $item$ which returns the result `S`. Results `K` and `S` are combined into `[K S]` and $rest$ is called again with this result as an argument. Because there is nothing left to parse, the result `[K S]` is returned.

With the first part of the view just described, SKI terms in mathematical notation can be parsed into the typed sequence structure on which the reduction and derivation operations are defined. To display results of these operations in the mathematical notation, the reverse operation is required: the internal typed sequence representation has to be transformed into concrete syntax. This is defined by the second part of the view that defines externalisation, as shown in Listing 14.

**Listing 14** Grammar for Externalising SKI Terms

```
1 e_ski_sym ⇒ ([:symbol c:char] ⇒ $c).
2 e_ski_group ⇒ (t:<e_ski_term> ⇒ ^( $t ^)).
3 e_ski_term ⇒ ([l:(<e_ski_sym> | <e_ski_term>)
4     r:(<e_ski_sym> | <e_ski_group>)] ⇒ [$l $r])
5     | <e_ski_sym> | <nothing>.
6 e_ski_literal ⇒ ([:ski x:<e_ski_term>] ⇒ ^` @x).
```

The conversion between internal and external representation can be broken down into three parts: recognising the typed sequence that wraps the content, converting symbols to characters and converting the tree structure expressed by nested sequences into concatenated terms or groups. Production `e_ski_literal` generates the start character followed by the result of `e_ski_term`. Production `e_ski_sym` unwraps characters by removing the surrounding symbol type. Production `e_ski_term` distinguishes several cases: the basic distinction is that a term is either a sequence with two elements, a symbol or nothing. In case of a sequence, the left element is either a symbol or again a term. The right element is either a symbol or a group. By being able to recognise a group when the right-hand element is a sequence, the externalisation manages to generate only those parentheses that are actually needed.

### 5.3.4 Concrete Syntax for Operations on SKI Terms

The previous two sections discussed an implementation of the SKI calculus based on a structured representation and a view definition that hides this representation behind a mathematical notation for terms. This improves the notation when experimenting with the derivation of SKI terms. However, the definition of new reductions is still based on the internal representation of Listings 11 and 12.

A more convenient way is to use the mathematical notation of Table 5.3.Concat supports pattern matching using concrete syntax by distinguishing between concrete syntax on program level and concrete syntax on meta-program level. The standard behaviour is to use the same syntactic definitions for both levels. This means that the view defined in algorithms 13 and 14 is used to recognise occurrences of SKI literals in programs and patterns.

By defining a separate transformation for the pattern level, grammar elements can be mixed into the literal syntax. This results in their insertion into the underlying typed sequence representation. The standard syntax to do this in Concat is inserting patterns between backslashes as shown in Section 5.2.5. For the derivation rule $Kxy \rightarrow x$ this would mean writing concrete syntax `'K\$x\\$y\  => '\$x\`. This syntax is verbose compared to the mathematical notation.

A more flexible approach is to define new syntax for pattern expressions so that they fit the syntactical context. The following production transforms the characters `x`, `y` and `z` into variables, as shown in Listing 15.

**Listing 15** Defining the Meta-Language Syntax for SKI Variables

```
1 ski_var ⇒ (n:((^x | ^y | ^z) -> <to_atom>) ⇒
2              [:var $n [:any]]).
```

First, the characters are converted into symbols by matching a reference to the built-in pattern `<to_atom>`, then the result is inserted into the internal representation of variables (see Section 5.5). Combining this definition with those in Listing 13 and changing the `ski_item` production in line 5 to

```
ski_item ⇒ <ski_var> | <ski_sym> | <ski_group>.
```

makes parsing SKI patterns with variables `x`, `y` and `z` possible. This means that the rules for SKI can be defined using the concrete syntax of terms as shown in Listing 16.

**Listing 16** Reduction and Derivation in Concrete Syntax

```
1  opdef reduce:
2    'Ix   ⇒ 'x.
3    'Kxy  ⇒ 'x.
4    'Sxyz ⇒ 'xz(yz).
5
6    do ('xy ⇒ 'zy) if 'x reduce ⇒ 'z.
7    do ('xy ⇒ 'xz) if 'y reduce ⇒ 'z.
8
9  opdef derive:
10     do ('x ⇒ 'y)
11     if 'x reduce derive ⇒ 'y.
12
13    'x ⇒ 'x.
```

The reduce and derive operations are not only equivalent to those in Listings 11 and 12 in the sense that both produce the same results. Moreover, the view mechanism hides the fact that the very same structures are created internally.

## 5.4 Language Layering in Concat

The previous section described combinatory logic as a case study for creating a language in Concat. From an outer perspective, Concat appears to compute, for example, the term `'a` as a result of the input `'Ia`. That is the illusion the system creates for the user. The user perceives the computation as a "black box" consuming a sequence as input and returning a sequence as output. The situation is depicted in Figure 5.3 with the computation being referred to as *cmp*.

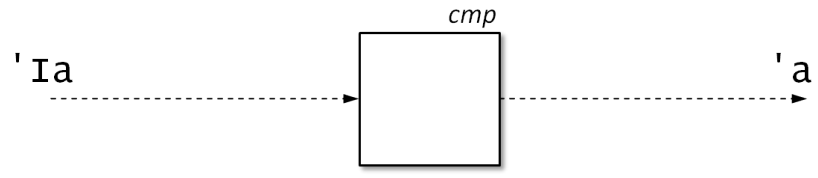


Figure 5.3: Black-Box View on Reduction of an SKI Term

As a matter of fact, the computation *cmp* is performed by another computational process $cmp'$. Semantically, $cmp'$ performs the same computation as *cmp* does but relies on a

different encoding of the input; $cmp'$ also uses a different encoding for the output. As Figure 5.4 shows, $cmp'$ uses a nested structure to represent ski terms.

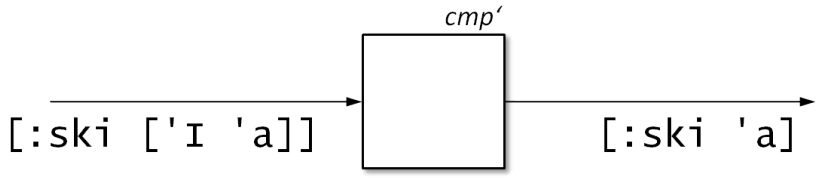


Figure 5.4: Internal Reduction of an SKI Term

As indicated, both computations are related to each other. The “upper” layer pictured by Figure 5.3 can be substituted by the “lower” layer shown in Figure 5.4 if two transformations are given: one transformation converts the input format of $cmp$ to the input format of $cmp'$ and the other transformation adapts the output format of $cmp'$ to the output format of $cmp$. The input conversion is an act of *internalising* an input sequence to the expectations of the “lower” layer, whereas the output conversion is an act of *externalising* the output of the “lower” layer towards the demands of the “upper” layer. The arrangement is shown in Figure 5.5 with $int$ being the internalisation transformation and $ext$ being the externalisation transformation. These transformations define functional mappings between representations.

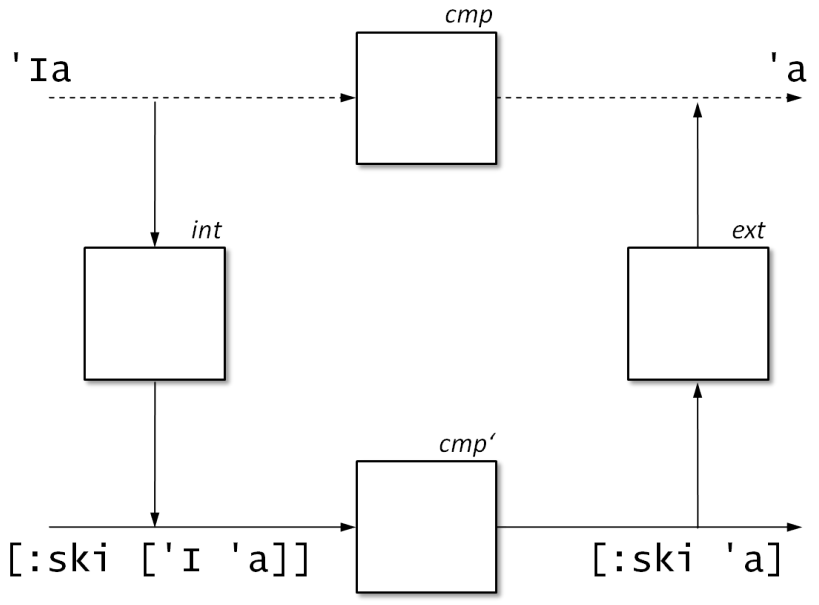


Figure 5.5: Layered Computation of an SKI Term

Formally, both computations are related by

$$cmp \Leftrightarrow int \circ cmp' \circ ext \tag{5.1}$$

Since $int$ and $ext$ only relate encodings to each other, they fulfil an important requirement:

$$int \circ ext = Id \tag{5.2}$$

This is the "identity requirement", which was introduced in Chapter 1 and reflects the essence of the notion of layering. The kind of bidirectional transformations introduced as views in Section 5.2.2 always need to fulfil the identity requirement. Views are used to layer a system in levels of computation. Depending on the perspective, the "upper" layer computational process is an *abstraction* of some "lower" layer process, or the "lower" layer is a *realisation* of some "upper" layer. The attributes "upper" and "lower" do not qualify a layer as "superior" or "inferior"; they just refer to relative arrangements in visualisations like Figure 5.5.

The behaviour of $int$ and $ext$ is defined by description $Int$ and $Ext$. For the SKI calculus, $Int$ is defined by Listing 13 and $Ext$ is defined by Listing 14. As a consequence of the relations in Figure 5.5 the *descriptions* of computational processes can be related by internalisation and externalisation as well. The computational process $cmp$ is described by the transformation `` `Ix=>`x ``. This is the "agreed upon" encoding of computational processes on this layer. In other words, this is the language (the encoding) a programmer uses to configure the computation $cmp$. If $cmp$ is resolved by the composition of $int$, $cmp'$ and $ext$, the description of $cmp$ can be translated into the language used to configure the computational process $cmp'$. Accordingly, the above computation description is translated into `[:ski ['I $x]] => [:ski $x]`. As described in Chapter 1, in general, the mapping between computation descriptions is defined by processes $map$ and $map^{-1}$. Given that $Cmp$ and $Cmp'$ denote the descriptions of $cmp$ and $cmp'$, respectively, the following holds:

$$Cmp \circ map = Cmp'$$

$$Cmp' \circ map^{-1} = Cmp$$

This relation is included in Figure 5.6, which is a variant of Figure 5.5. Transformations $map$ and $map^{-1}$ need to obey the law of "descriptional identity", which resolves to the identity requirement.

$$Cmp \circ map \circ map^{-1} = Cmp \Leftrightarrow map \circ map^{-1} = Id$$

In case of implementing $cmp$ via $cmp'$, the transformation $map$ is given and $map^{-1}$ is

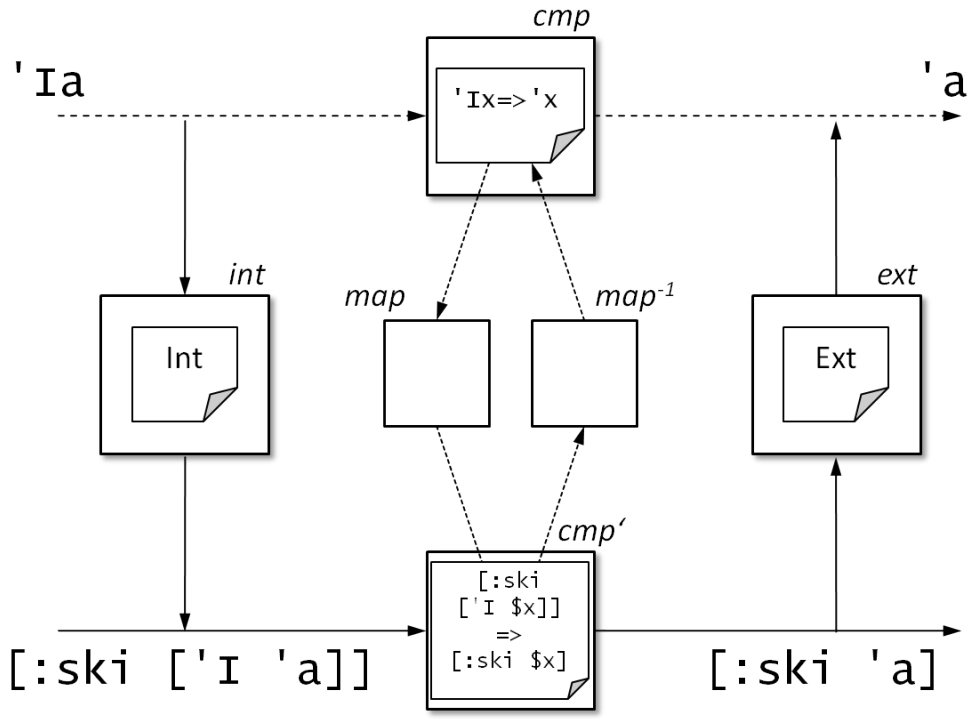


Figure 5.6: Extending the Meta-Language with SKI Syntax

not explicitly formulated. A definition of $map^{-1}$ is required for looking at stages of meta-system execution in terms of $Cmp$ rather than $Cmp'$.

To conclude, any computation can be resolved into another layer of computation in two ways: functionally and description wise. The transformations required for internalisation and externalisation need to fulfil the identity requirement. This is the prerequisite for layering, computationally and language-oriented. Concat is a demonstration of how layering can be systematically utilised in the design of a programming system. In Concat, the same language is used for describing all computational processes, including $map$ and $map^{-1}$. As is described in Chapter 1, any computation or computation description, i.e., any box in Figure 5.6 can be subject of further layering until an elementary level of computation is reached – or the language describes itself.

## 5.5 Metacircular Implementation

This section discusses an implementation of Concat's pattern meta-language in itself. The two metacircular interpreters [1] for matching and instantiating as well as the parsing processes are implemented using the same language engineering tools and techniques discussed in the previous sections. Pattern expressions are represented by typed sequences.

Views define concrete syntax for these typed sequences and operations implement matching and instantiation. User access to the metacircular implementation allows to change the semantics of existing pattern operators and to implement new features that cannot be expressed using pattern abstraction alone. Control over the internalisation process of the meta-language allows to introduce new language features and to express them using pattern expressions.

### 5.5.1 Implementation Alternatives

There are two fundamental approaches to implementing languages in Concat that can be classified as compilational or interpretational. Below, both approaches will be examined as candidates for a meta-circular implementation of Concat's meta language.

**Compilational Approach** The compilational approach uses the internalisation stage to transform every concrete syntax expression of a source language directly into an executable program of a target language that implements the expression semantics. The target language is defined by a set of operations. For example, the matching semantics of sequences can be implemented by the operation `match-seq` which is shown in Listing 17.

**Listing 17** Sequential Matching (Compilational Approach)

```
1 opdef match-seq:
2   do ($sin $bin [[@p1] @pr] ⇒ $res $sout $bout)
3     if ($sin $bin @p1 ⇒ $r1 $s1 $b1)
4     && ($s1 $b1 [@pr] match-seq ⇒ $r2 $sout $bout)
5     && ($r1 $r2 combine ⇒ $res).
6   ($sin $bin [] ⇒ [:res] $sin $bin).
```

The sequential operator matches a sequence of patterns with input data and a store. The pattern arguments of `match-seq` are quoted programs. The quoted programs are executed in the conditional part of the operation. A sequential pattern `any any` is, for example, transformed into the program `[[[match-any] [match-any]] match-seq]` which is called on a sequence and a store.

Because patterns have both matching and instantiation semantics, there are actually two sets of operations. The advantage of implementing the pattern language with the compilational approach is the modularity of the code. Each pattern expression is implemented separately and thus changing the implementation is achieved by overwriting sin-

gle definitions. The disadvantage is that because there are two interpretations, there need to be unique names for operators which means patterns for matching have a different representation than patterns for instantiation and the internalisation has to know which representation to produce depending on the context. Also, the internal representation of pattern expressions as programs does not directly allow for the view mechanism to translate between concrete and abstract syntax.

**Interpretational Approach** The metacircular implementation of Concat is based on an interpretational approach. This approach defines an internal representation for programs and an interpreter to manipulate this representation. In principle, the same technique was used in Section 5.3 to implement combinatory logic. For the pattern language, however, two interpreters are needed for matching and instantiation. Patterns are internally represented by typed sequences where the operator name corresponds to the sequence type. For example, the transformation

```
x:any wrap ⇒ [:quot x:any]
```

has the internal representation

```
[:utrans [:seq [:var 'x [:any]] [:op 'wrap]]
           [:tseq 'quot [[:var 'x [:any]]]]]
```

The interpreters match and instantiate provide a set of rules that dispatch on these types. This is shown exemplarily in Listing 18.

**Listing 18** Schema for Match and Instantiate Implementation

```
1 opdef match:
2  do $sin $bin [:seq $f @r] ⇒ ...
3  do $sin $bin [:var $n $p] ⇒ ...
4  ...
5
6 op instantiate:
7  do $sin $bin [:seq $f @r] ⇒ ...
8  do $sin $bin [:var $n $p] ⇒ ...
9  ...
```

The internalisation process of Concat transforms operation definitions into transformation rules of a rewriting system by adding the name after the `opdef` keyword as the rightmost element on the left-hand side of the rule. This implies that the rule for sequential matching in line 2 is transformed into

```
do $sin $bin [:seq $f @r] match => ...
```

and the rule for sequential instantiation in line 7 into:

```
do $sin $bin [:seq $f @r] instantiate => ...
```

This makes it clear that the implementation is basically a rewriting system that dispatches on four contextual properties: interpretative context (match or instantiate), operator type (`seq`, `var`,...), arguments to the operator and the state of the data sequence and the store.

### 5.5.2 Internalising the Pattern Language

The internalisation of patterns is defined as a set of views on the typed sequence representation. Listing 19 is an excerpt of the view definition. The structure of pattern expressions resembles that of programs defined in Section 5.1.1: a pattern is a sequence of grammar items. The meta-level grammar actually defines a superset of the program-level grammar which means that programs are valid patterns. This is because program items (`pitem`) are instances of grammar items (`gitem`) and because typed sequence patterns with atoms as types and program items as content have the same syntax as regular typed sequences. To avoid conflicts between program syntax and pattern syntax, parts of a pattern can explicitly be declared as program literals by surrounding them with `/p(...)`.

A pattern expression (`gprog`) consists of a sequence of grammar items that are possibly annotated with a postfix operator (`gpost`) and are separated by infix operators. Examples of postfix operators are repetition (`*`) or elliptical matching (°). Examples of infix operators are sequencing (white space), vertical combination (`->`) and choice (`|`). The definition of `pattern` uses a dynamically parameterised reference to the recursive production `rest`. The production `rest` constructs binary pattern operators by recognising infix operators. The resulting parsing process prioritises postfix operators and associates infix operators to the left. Grouping (`group`) defines priorities explicitly by surrounding patterns with round brackets. Typed sequence patterns (`gtseq`) have the same syntax as regular typed sequences but may contain grammar items as content and a group pattern in place of the type identifier. Variables (`var`) consist of a name, followed by a colon and a grammar item possibly annotated with a postfix operator. The production `dvar` defines an alternative variable syntax for writing `$x` as a shorthand for `x:item`. The left-hand and right-hand sides of an unconditional transformation (`utrans`) are either empty or consist of a `gprog`.

**Listing 19** Pattern Language Grammar in Concat (excerpt)

```
gprog ⇒ (i:gpost <$i p:grest> ⇒ @p) | <nothing>.

gpost ⇒ (i:gitem (t:(^* ⇒ /a(rep)) |
  t:(^° ⇒ /a(ell)) | ... | item) ⇒
  ([:($t) $i] | $i)).

gitem ⇒ (<pexplicit> | <group> | <var> |
          <gtseq> | <utrans> | ... | <pitem>).

pexplicit ⇒ /s(/p() <pitem> /s()).

group ⇒ ( ^( <ws> g:gprog <ws> ^) ⇒ $g).

$e1 grest ⇒
 (<ws> /s(|) <ws> e2:gpost
  res:<[:or ,$e1 $e2] grest> ⇒ $res) |
 (<ws> /s(->) <ws> e2:gpost
  res:<[:vert ,$e1 $e2] grest> ⇒ $res) |
 ...
 (<ws> sep <ws> e2:gpost
  res:<[:seq ,$e1 $e2] grest> ⇒ $res) |
 (<nothing> ⇒ $e1).

utrans ⇒ (^( lhs:(<gprog> | <nothing>) <ws> /s(=>)
          <ws> rhs:(<gprog> | <nothing>) ^) ⇒
          [:utrans $lhs $rhs].

var ⇒ (n:name /s(:) p:gpost ⇒ [:var $n $p]).

dvar ⇒ (^$ n:name ⇒ [:var $n 'item]).

gtseq ⇒ (/s([:) t:(<grouping>|<atom>) <sep> <ws>
         c:<<white> <gitem> sep_list>? ws /s(]) ⇒
         [:tquot $t $c]).
```

The fact that the internalisation process of the meta-language is controllable in Concat provides means to introduce language abstractions that hide the underlying pattern-based rewriting. The primary abstraction for computing in Core Concat are operations. Operations are based on rewriting but enforce a functional semantics. The transformations in Listing 20 implement the internalisation process of operation definitions. An operation definition consists of a keyword `opdef`, a symbolic name for the operator, and a sequence of transformations.

**Listing 20** Internalising Definitions of Operations

```
1 operation ⇒ /s(opdef) <ws> n:name <ws> ^: <ws>
2  x:((<utrans> -> <$n add-name>) |
3     (<ctrans> -> <$n add-name>))+ ⇒ [:or @x]
4
5 $n add-name ⇒
6  ([:utrans [@l] $r] ⇒ [:utrans [@l ,$n] $r]) |
7  ([:ctrans [@l] $r $c] ⇒ [:ctrans [@l ,$n] $r $c]).
```

The internalisation combines the sequences into a choice operator and appends the operator name to the left-hand side of each transformation. The appending is performed by the parameterised production `add-name`. This production is referenced by a vertical combinator in order to define a second stage of processing after an unconditional (`utrans`) or conditional (`ctrans`) transformation was parsed. The parameter to `add-name` is the operator name.

### 5.5.3 Implementing Pattern Operators

Conditional rules can be used to implement matching and instantiation in such a way that the rules closely correspond to the derivation rules in Chapter 3. Therefore, only an excerpt of the implementation will be presented. In Listing 21 is an excerpt of the implementation of `instantiate` that shows the code for sequential instantiation.

**Listing 21** Metacircular Implementation of Sequential Instantiation

```
1 opdef instantiate:
2   ...
3   do ($sin $bin [:seq $p1 @pr] ⇒ $sout)
4   if  $sin $bin $p1 instantiate ⇒ $s1
5   && ($s1 $bin [:seq @pr] instantiate ⇒ $sout).
6   $sin $bin [:seq ] ⇒ $sin.
7   ...
```

The header of the conditional rule binds `p1` to the first element of the pattern sequence and `pr` to the rest of the pattern sequence. The first conditional instantiates the pattern with the initial input and binds `s1` to the result. The second conditional calls `instantiate` on `s1`, the original bindings and the sequence pattern with the first element removed. By doing so, the rule manipulates the internal program representation and creates a new pattern. The result of instantiating this pattern is the end result of the instantiation. The unconditional rule in line 6 implements the base case of the recursion: in case the sequence pattern is empty the input sequence `sin` remains unchanged.

Listing 22 shows the implementation of matching an unconditional transformation.

**Listing 22** Metacircular Implementation of Unconditional Transformations

```
1 opdef match:
2  ...
3  do ($sin $bin [:utrans $lhs $rhs] ⇒ $res $sout $bout)
4  if ($sin $bin $lhs match ⇒ $res $sout $b1)
5  && ([] $b1 $rhs instantiate ⇒ $res).
6  ...
```

The header binds `lhs` to the pattern that makes up the left-hand side of the transformation and `rhs` to the pattern that makes up the right-hand side. The first conditional matches the left-hand side with the initial input and the second conditional instantiates the right-hand side with an empty sequence and the bindings resulting from the match. The transformation yields the result produced by instantiation and the output sequence and store produced by matching.

## 5.6 Summary and Conclusions

Concat demonstrates the applicability of the pattern core and its programming related extensions to language engineering. While the previous chapters supported the hypothesis that pattern matching and instantiation provides a *formal* foundation for LDSE, Concat supports the hypothesis that this foundation is also a *practical* for language-driven programming. Everything in Concat is based on patterns. By applying its language engineering tools to its own meta-language, Concat can hide its pattern-based foundation and change itself from within. The degrees of change supported are the following:

- New concrete syntax for literals on the meta-level, e.g., a pattern that uses concrete syntax for rational numbers `(1/3|1/2)` (views)
- New meta-language syntax. For example, a new syntax for variable bindings such as `$x` (meta-views)
- New pattern expressions such as `<rational>` or `<seplist>` (productions)
- Modification of pattern semantics and new features that cannot be implemented by combining expressions, but require access on a lower level, e.g., memoization or backtracking (metacircular implementation)

The changes supported range from smaller syntactic adjustments to complete replacement of the language syntax and semantics. In this sense, Concat is not only an engine for creating languages but also an engine for its own evolution. In other words, Concat can sustain itself [56].

The general philosophy of Concat is to strike a balance between the unambiguity and ease of structural manipulation a uniform framework provides and the freedom and flexibility needed for implementing a wide range of languages. This philosophy is embodied in all aspects of the implementation in the form of mutually complementary concepts. Typed sequences define a uniform syntax and views provide means to break out of the restrictions this entails. Operations enforce functional semantics and macros provide means to circumvent these semantics. Core Concat provides a set of operational concepts and at the same time the means to replace these concepts with new ones.

Concat is a multi-level framework in several regards. Program syntax can be described in five layers with refinement relationships between each. A three-level distinction is made between program, meta-program and meta-meta-program. Views provide a mechanism to implement a programming system in an arbitrary number of layers where each layer provides a different perspective on programs and computations.

An interesting aspect of Core Concat is the way operations and different types of macros are implemented using the rewriting system. An operation definition associates an operator with a functional semantics. Internally, the operator is appended to the left-hand side of each rule in the body of the operation definition. In effect, the operator defines a context for the rules in the body. Accordingly, the operation definition can be interpreted as a statement that the rules in the body can be applied in the context of the operator symbol. For internalisation macros, a start-tag serves as a left-hand context to the definition to the body. By using start- and end-tags, there is a context to the left and to the right. The underlying principle is that different kinds of computational entities can be defined by separating transformations and context.

# Chapter 6

# XMF: A Pattern-based (Meta-)Modelling Framework

XMF (XML Modeling Framework)[1] is a modelling and meta-modelling tool based on the pattern core introduced in Chapter 3. XMF was initially developed for teaching modelling and language-driven engineering and has been successfully used in university teaching [143]. Its main goals are to provide flexible means to create and relate modelling languages and to make the modelling process interactive. XMF consists of a browser-based, integrated development environment, a pattern-based transformation language and a JavaScript API that exposes pattern functionality. XMF is based on a three-level meta-architecture in which inter-level relationships are formally defined by pattern instantiation and different kinds of relationships between models are defined by a relationship language (Section 6.1). XMF implements this meta-architecture and in addition to that defines a view mechanism for graphical model syntax (Section 6.2). The abstract and concrete syntax of models are based on XML. Models and views are defined using the *XML Pattern Language for Transformations* (XPLT) that implements matching and instantiation on Document Object Model (DOM) trees. Technically, XPLT is a schema language. However, schemas are defined in such a way that they form the basis for transformation, refinement and querying. Based on XPLT patterns, the view mechanism derives bidirectional transformation between XML model representations and graphical views in XHTML (Section 6.3). XMF contains two built-in modelling languages for class and object modelling (Section 6.4). These are implemented in XPLT (Section 6.5). A JavaScript API serves as a constraint language for formalising the relationship between models and modelling languages and for defining semantic model constraints. The API functionality is based on referencing and refining patterns and querying with patterns (Section 6.6).

[1]The system discussed in this chapter was created by the author. It is not related to the eXecutable Metamodeling Facility (XMF) [159].

## 6.1 Meta-Architecture

In recent years, Domain Specific Modelling Languages (DSMLs) have seen growing interest in research and practice. DSMLs promise the same advantages for modelling that domain specific programming languages promise for programming: higher expressibility that leads to more readable and maintainable models. The systematic use of specialised languages for modelling different aspects of a system requires the ability to create and relate languages in a flexible manner. For many years, the Unified Modeling Language (UML) [130, 131] has been the most prominent modelling language. Although it is a general purpose object-oriented language, it provides a meta-architecture that describes the relationships between models, languages and meta-languages. In the following, the UML meta-architecture serves as a starting point for developing a uniform meta-architecture that supports a pattern-based approach to creating and relating modelling languages.

### 6.1.1 The UML Meta-Architecture

In the literature, the UML meta-architecture is commonly depicted in the spirit of Figure 6.1a) [90]. There are four layers, usually labelled M0 to M3 from bottom to top, each lower layer being an instance of its direct upper layer.

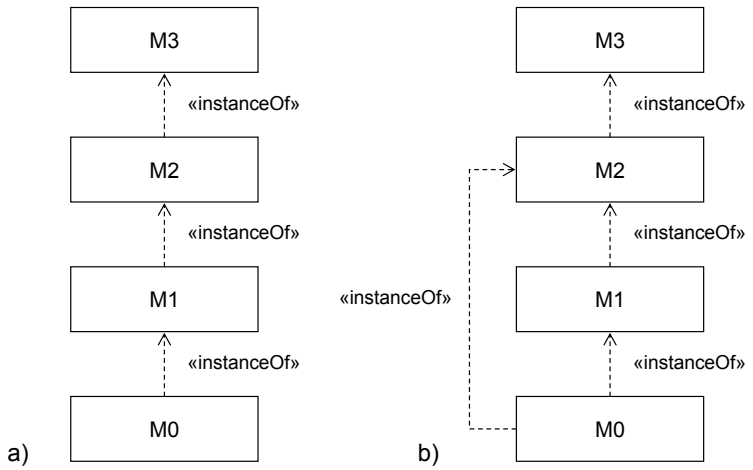


Figure 6.1: Two Competing Presentations of the 4-Level Meta-Architecture

A competing representation of the architecture includes an additional “instanceOf”-arrow pointing from M0 to M2 as shown in Figure 6.1b) [73]. While levels M1 to M3 are defined precisely in the literature, M0 and its relationship to M2 remain unclear. In some publications, level M0 is even defined to be outside of the modelling system and referring to the actual “real world” objects being modelled [90]. To clarify the relationships be-

tween the levels, a more detailed version of the architecture is presented in Figure 6.2. The abbreviation CM stands for class model. A class model consists of classes, associations, inheritance and other class-specific concepts. These concepts in CM are defined by CL, the class modelling language. CL is a specification of concepts, of which a concrete CM is an instance. This is the reason of existence for the arrow labelled "instanceOf" between CM and CL. The class meta-language (CML) is used to specify the language constructs available for use on CL. In other words, CL is a concrete specification of a modelling language in terms of the language CML. In that sense, CL is an instance of CML. All this is not in conflict with the common understanding of meta-level architectures.

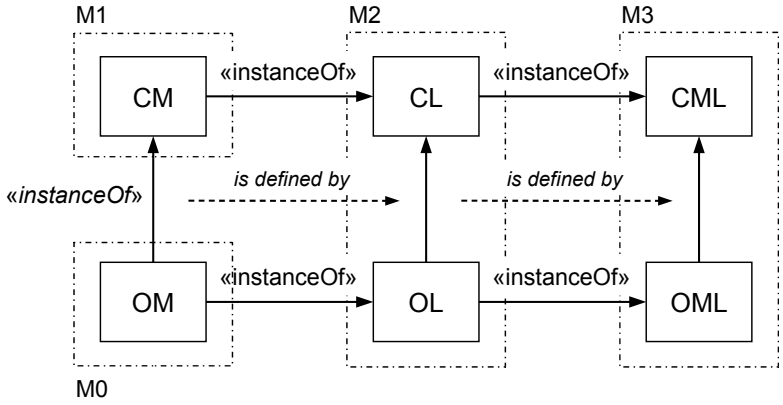


Figure 6.2: Alternative Presentation of the UML Meta-Architecture

However, and this fact seems to be overlooked at least in some presentations of the meta-architecture, the same reasoning holds for the lower half of Figure 6.2. An Object Model (OM) consists of objects, values, references and possibly other object-specific concepts. These concepts are defined by the Object Language (OL). The Object Meta-Language (OML) specifies the language constructs available for use on OL. The interesting part is that the world of classes and the world of objects are interconnected. This interconnection is defined through a relationship between the class language (CL) and the Object Language (OL) that determines how OM and CM are related.

For example, the relationship may be defined by a constraint which states that (1) every object needs a class, (2) attribute values of objects must be defined in the corresponding class and (3) actual associations between objects are constrained by the corresponding multiplicities defined in the class models [88]. The concrete relationship between OM and CM is defined by the arrow between OL and CL which is in turn defined by the vertical arrow between OML and CML. The OML-CML arrow defines that there are means to relate instances of OML and CML. Without this arrow, the upper and lower half of the architecture would not be connected and only self-contained models of OM and CM could

be defined.

Object model OM is at meta-level zero (M0) and class model CM at meta-level one (M1). CL and OL together constitute meta-level two (M2), CML and OML constitute meta-level three (M3). This completes a coherent description of the meta-level architecture. Figure 6.1b) can be seen as a condensed view of Figure 6.2: The arrow pointing from M2 to M3 in Figure 6.1b) actually comprises two arrows. Figure 6.1a), however, is obviously incomplete. The "instanceOf" relationship between OM and CM is of a different kind than the other "instanceOf" relationships. It is the only arrow whose semantics can be arbitrarily defined in the meta-level architecture by specifying the relationship between OL and CL on M2. This means that the arrow must not necessarily define an "instanceOf" relationship but may define any relationship between models. The only reason it is an "instanceOf" relationship in the UML architecture is that the related models are class and object models.

The reasoning so far suggests that it seems necessary to add ever higher levels to the architecture infinitely because the meta-languages on M3 need to be defined by meta-meta-languages which again need to be defined by meta-meta-meta-languages and so on. A technique to stop the infinite regress is to either predefine M3, i.e., there is no account of the languages on M3 within the system, or to define the languages M3 in a metacircular way, i.e., with means of M3. Nevertheless, it is possible to introduce a fixed but arbitrary number of higher levels if necessary. It shall be noted that the languages CML and OML on M3 and CL and OL on M2 must not necessarily be separate languages. For example, if CML = OML, the language for defining CL and ML is the same. This implies that the relationship language RL relates elements of the same language.

### 6.1.2 A Pattern-based Meta-Architecture

The discussion of the previous section is abstract in the sense that it does not state how a language is actually defined. The horizontal arrows between M1 and M2 and between M2 and M3 define "instanceOf" relationships. So far, the notion of a model being an instance of a modelling language is vague. In contrast, the notion of instantiating a pattern developed in the previous chapters defines a clear semantics: data is an instance of a pattern if there is a set of bindings for which the instantiation of the pattern yields the data. This *syntactic* "instanceOf" relationship is adequate to describe the relationships between M1 and M2 and M0 and M2. Accordingly, if modelling languages CL and OL are defined by a pattern, the respective models CM and OM are instances.

For example, if the pattern fragment `[:class [:name x:string] ... ]` defines that a class consists of a name, all typed sequences that are instances of this pattern

are valid classes, e.g., `[:class [:name "car"] ... ]`. Section 5.5 demonstrated that a pattern can itself be represented as a typed sequence which is an instance of a pattern language. If M3 contains one or more pattern languages, the relationship between M2 and M3 can also be defined as pattern instantiation. Just as the concrete "car" class in the example above is an instance of the "class" pattern, this pattern is an instance of the pattern language. In that way, a three-level system M3-M2-M1 is described in terms of two two-level systems M3-M2 and M2-M1.

By defining – in a metacircular way – that the languages on M3 are patterns that define a pattern language, the architecture can be described entirely in terms of patterns and typed sequences. Firstly, the artefacts on all levels are typed sequences (or any structured representation underlying the pattern system). Secondly, the "instanceOf" relationships between levels M1 and M2 and M2 and M3 are defined by pattern instantiation. Thirdly, the relationships between M0 and M1 and the intra-level relationships on M2 and M3 are structural relationships between typed sequences. The last point requires a more detailed explanation.

It was stated above that the relationship between models is described by relating elements on level M2. For the uniform interpretation of the architecture just introduced, this means that relationships are defined on parts of patterns. For example, the relationship between an object and a class on the model level may be encoded by adding a class reference $class$ to each object: `[:object [:class x:string] ... ]`. The knowledge about this encoding can then be defined syntactically on M2 by stating that an object is an instance of a class if the name of the class and the class reference are the same. The relationship may be refined by also requiring that an object must contain values for all attributes defined by the referenced class – and its parent classes in case of inheritance. The pattern functionality introduced in Chapter 3 defines means to transform, refine and query data that can all be expressed by matching and instantiating patterns. By using this functionality to describe the relationship between M0 and M1, the entire meta-architecture can be formalised using the pattern approach.

The insights just presented lead to the pattern-based meta-architecture in Figure 6.3. The architecture consists of three levels: model, language and meta-language level. There is a single pattern language PL on the meta-language level. Patterns P1 and P2 on the language level define modelling languages of which the Data D1 and D2 on the model level are instances. The architecture generalises the class and object-models of the architecture in Figure 6.2 to arbitrary models D1 and D2 and instead of the specific "instanceOf" relationship between classes and objects defines an arbitrary relationship RI (relationship instance) between these models. RI is defined by a relationship R between the patterns P1 and P2.

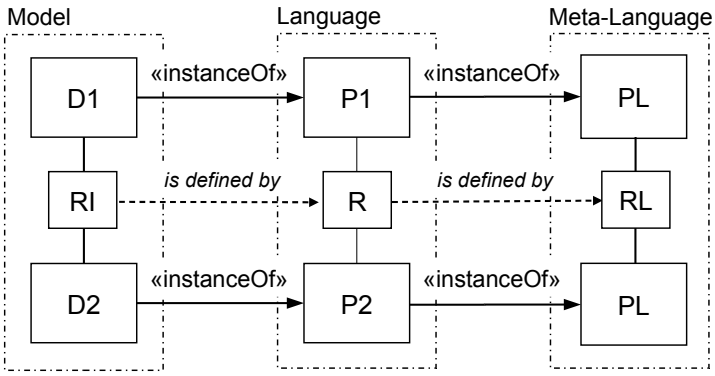


Figure 6.3: A Pattern-based Meta-Architecture

The relationship R is defined using a relationship language RL that relates elements of the pattern language PL on the meta-language level. Based on the architecture different kinds of model relationships can be distinguished. R defines an inter-language relationship if P1≠P2 and else an intra-language relationship between models that are instances of the same language. Intra-language relationships can be further sub-divided into intra-model relationships between elements of a single model (D1=D2) and inter-model relationships between elements in different models (D1≠D2).

## 6.2 XMF Overview

Starting from the UML meta-model the previous section developed a meta-architecture describing the relationships between models, modelling languages and meta-languages on three layers. The key idea is to represent all models in a uniform way and to define all relationships as pattern operations. The discussion so far was based on typed sequences and ignored the aspect of concrete model syntax entirely. This section will describe how the XMF framework implements the pattern-based meta-architecture and provides a view mechanism allowing graphical syntaxes for models. The main components of XMF are a browser-based model and meta-model editor, the XPLT pattern language and a JavaScript API exposing functionality for matching, instantiating, transforming, refining and querying patterns. This API is the basis of the relationship language RL.

### 6.2.1 Defining Models, Meta-Models and Views

The XMF framework uses XML to store models and XHTML for graphical representations of models. The XPLT pattern language implements matching and instantiation of

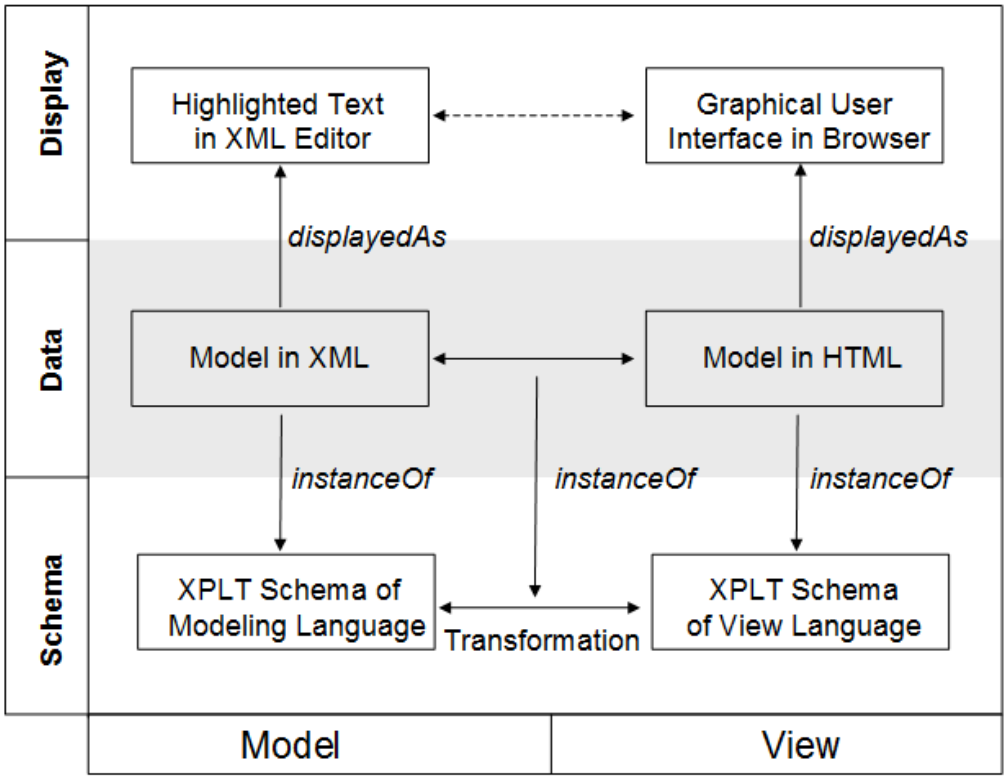


Figure 6.4: Models, Views and the User Interface

XML documents as operations on Document Object Model (DOM) trees [171]. There is a close relationship between typed sequences and XML which makes it possible to apply the pattern concepts introduced so far directly to XML. Both XML and typed sequences are a notation for tree structures with explicit type information for each node. There is a straightforward mapping between typed sequences and a subset of XML without attributes, namespaces, mixed content and other more advanced features. For instance, a typed sequence `[:a [:b 1]]` corresponds to an XML element `<a><b>1</b></a>`. In Section 5.5 a typed sequence representation for patterns was introduced where each expression of the pattern language is represented as a typed sequence of a specific type. Accordingly, a sequence pattern is represented as `[:seq ...]`. A similar representation is used to express patterns in XMF: `<sequence>...</sequence>`.

A modelling language in XMF consists of a set of XPLT patterns for defining XML-representations of models, an optional set of XPLT patterns defining an HTML-based interface language for displaying and/or editing models in a browser and an optional set of intra- and inter-model constraints for refining modelling language patterns and relating modelling languages. XMF attempts to derive bidirectional transformations [36] between models in XML and views in HTML automatically through the variable names in both patterns. The constraint language (RL in Figure 6.3) is built on the pattern-based

search mechanism described in Section 3.5. Figure 6.4 illustrates the relationship between a model in XML and a view in HTML. The patterns for the XML and HTML representations define a *schema* for valid models. A valid model is an instance of the model schema and a valid view on a model is an instance of the view schema. A display process interprets the XML representation and displays the XML text in a code editor with syntax highlighting. A different display process renders the HTML in the browser using widgets. The transformation on the display level between XML text and graphical user interface elements is performed indirectly on the data level. Changes made to the view representation are mapped back to the model representation. The storage format for models is XML text while the browser-internal representation on all levels is a DOM tree.

The technique used for introducing graphical syntax in XMF is based on the same principle as the technique used for introducing textual syntax in Concat. In Concat, everything is either an atom or a typed sequence and the way to break out of the restrictions of this syntactic framework is to define a view representation and to "hand over" the view data to a display process. For Concat, this is any text editor that can, for example, render a character-based encoding `[:char /a(2)] [:char /a(3)]` as 23, a syntactic representation being neither a typed sequence nor an atom. XMF extends this principle to graphical representations by transforming data into an HTML representation that will be rendered by the display process as something outside of the representational possibilities of XML. The character representation in Concat and the HTML representation in Concat define the shared space of an interface between a process describing symbolic interpretation and a process interpreting symbols as rendering information.

### 6.2.2 User Interface

XMF is a browser-based application. The screenshot in Figure 6.5 depicts the user interface of XMF which is divided into four areas. The upper half of the user interface consists of two editors for models. There are several reasons for providing two editors. Firstly, by opening the same model in both editors, XMF provides multiple simultaneous views on a model. Secondly, editing related models and formulating inter-model constraints is more convenient this way. The editing of models can be performed in XML representation or through HTML widgets in the browser. The changes made are propagated to the DOM tree. The built-in XML editors support syntax highlighting and automatic alignment of source code. The tabs above each editor allow switching between text-based representation and graphical view and between the model (data) and the language (pattern) levels. The lower half of the user interface consists of an editor for model constraints and a logging window. Buttons control constraint checking and allow to clear the log.

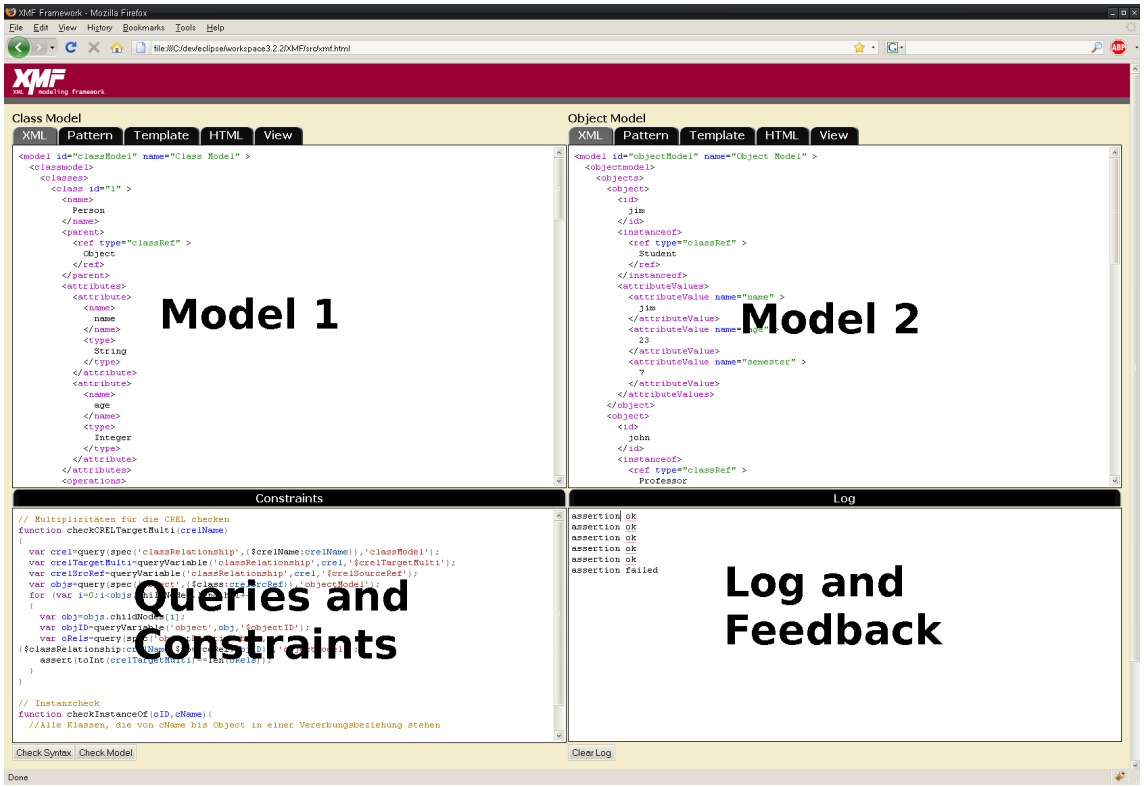


Figure 6.5: Screenshot of XMF

## 6.3 The XPLT Language

The central component of XMF is the *XML Pattern Language for Transformations* (XPLT) and its underlying transformation engine. XPLT was developed specifically for XMF but it has since evolved into a stand-alone schema and transformation language for XML.

### 6.3.1 XPLT Patterns

XPLT implements matching and instantiation of patterns on DOM trees representing XML or XHTML documents. The XPLT pattern languages is comparable to XML schema languages such as W3C XML-Schema or Relax NG but provides a more general pattern-based approach comprising schema definition, document transformation, query and search. XPLT patterns are based on the matching and instantiation semantics of a subset of operators defined in Chapter 3. For example, it implements sequencing, alternation and repetition and supports pattern abstraction and recursive definitions through pattern references. While the semantics of operators is basically the same, the main difference compared to the semantics of the pattern core of Chapter 3 and its realisation in Concat is that transformations are not part of the pattern language. Although transformations are not

pattern expressions, pattern-based transformation is an important part of XMF. The view mechanism relies on two patterns to transform between models and views. Explicit transformation can be expressed by calling an API function.

The same is true for querying models which is not expressed using $find$, $findall$ and $replaceall$ operators but through a query function implementing these path polymorphic patterns. This approach has the advantage that transformations can be triggered by code associated with the user interface and that constraints can be formulated as scripts defining matching and instantiation of DOM tree elements. XMF patterns do not return a result of a match but only bindings that can then be used to instantiate a pattern to define a transformation. Therefore, vertical and diagonal matching is not possible in XMF. It is replaced by the notion of conjunctive matching where two patterns are applied to the same data.

An XPLT pattern is a well-formed XML fragment. It is a mix of the XML elements contained in the data being matched or instantiated, i.e., elements of the data language, and elements of the pattern language. The scope of XPLT patterns is not restricted to XML elements but includes attributes associated with a node. In addition to that, source code fragments can be attached to tree nodes during instantiation. Patterns are surrounded by a `pattern` tag. The id attribute may be used to give a pattern a unique id by which it can be referenced. For example, the following is a pattern with id p1: `<pattern id="p1"><a/></pattern>`. The pattern expressions of XPLT will be introduced by example in the following. The surrounding pattern will be ignored in the following to shorten the presentation.

**Hierarchical** Every regular XML element is treated as an element pattern that has the semantics of matching or instantiating its children sequentially. This ensures that regular XML can be mixed with the XPLT pattern elements defined in this section. For example, the pattern `<e>$x <b/></e>` matches `<e><a/><b/></e>` by first ensuring equality of the element `<e>` and then matching the contained children in sequence; in this case, a variable and another element pattern. As a result, `$x` is bound to `<e/>`.

In XPLT, a variable can be used in place of the element tag id. For example, the pattern `<$x><a/></$x>` expresses that the element `<a/>` has a parent node of variable type. The variable name in the start-tag must match that of the end-tag. Any binding of a variable in that position must have a text node as a value, otherwise instantiation fails. The element pattern corresponds to the typed sequence pattern defined in Section 3.2.3 with the restriction that the pattern in the tag position cannot be an arbitrary pattern but only a variable.

**Variable** A variable allows an arbitrary structure in the place where the variable occurs in the pattern. The pattern `<a><variable name="var"/></a>` matches any DOM tree element `<a>` with a single child and binds the variable name to that child. For example, `<a><b/></a>` and `<a><b><c/></b></a>` both match the patterns and yield bindings from `var` to `<b/>` and `<b><c/></b>` respectively. As defined in Chapter 3, variable bindings must be consistent with existing bindings, otherwise matching fails. If the attribute name in the variable element is empty, no bindings are created. Instantiating a named variable yields the value of that variable if a binding exists, otherwise instantiation fails. XPLT provides a shorthand notation for variables: instead of an element, a string starting with a `$` character followed by a variable name may be used to represent a variable. The name must not contain white spaces or any reserved XML characters and may be empty. For instance, the pattern `<a>$var</a>` is equivalent to the pattern above. The XPLT parser transforms the text node with value `"$var"` into a variable node with attribute `name="var"`, hence both notations yield the same DOM tree.

**Conjunction** Variables in the pattern core and in Concat consist of a name and a pattern part. The pattern part can be used to express the type of a variable precisely. An XPLT variable, on the other hand, matches any node indiscriminately. Nevertheless, there is a generic pattern providing the same level of control over variable matching as defined in the pattern core. The conjunctive `<and>` pattern matches patterns sequentially with the same data and succeeds only if all matches succeed. For example, the pattern `<and>$x <a>$</a></and>` matches only `<a>` elements. Thus, the pattern matches `<a><b/><a/>` but it does not match `<c><b/></c>`. For the first case, a binding from `$x$` to `<a><b/></a>` is created. The instantiation semantics of conjunction yields as a result the value of instantiating the first element. Accordingly, the example pattern above may be instantiated to the value of the variable `$x`.

**Choice** Choice expresses matching or instantiating alternative structures. For example, the pattern `<a><choice><b/><c/></choice></a>` matches both fragments `<a><b/></a>` and `<a><c/></a>`. Matching and instantiation of a choice pattern is *prioritised*, i.e., the alternatives are tried from left to right until one can be matched or instantiated. In case of a match, no further alternatives are tried. In the example above, the first choice `<a><b/></a>` is instantiated.

**Sequencing** The sequence pattern matches or instantiates patterns in the order in which they appear. This translates to a match of the corresponding DOM tree elements from left to right. For example, the pattern `<sequence>$x <b/></sequence>` first matches

or instantiates a variable and then an element. The sequence pattern expresses sequencing independent of a parent element. The semantics of the element pattern defined above makes explicit sequencing superfluous when the parent element is part of the pattern.

**Repetition** Repetition of patterns is expressed by the greedy repeat pattern. The following pattern expresses that element `<a>` might have any number of children `<b>`.

```
<a>
  <repeat>
      <b>$x</b>
  </repeat>
</a>
```

The semantics of the repetition is that of the $^\circ$ operator in Section 4.6. An example of a fragment that matches the pattern above is:

```
<a>
  <b><x/></b>
  <b><y/></b>
  <b><z/></b>
</a>
```

Matching the pattern with this data creates a binding from `$x` to a list of DOM tree nodes `[<x/>,<y/>,<z/>]`. Instantiating it creates as many instances of `<b>$x</b>` as there are bindings. This means that first matching and then instantiating the pattern with the example data is an identity operation. The elliptical semantics of repetition are important for model-view transformations. This will become apparent below when transformations are derived from schemas defining the repeated occurrence of elements in a model or view.

**Text** The text element matches and instantiates plain text in text nodes explicitly. It is used to extract information from a text string. For example, matching the pattern of `<a><text>id:</text>$x</a>` with `<a>id:1</a>` produces a binding from `$x` to a DOM tree text node with value `'1'`.

**Reference** Pattern abstraction and modularisation is implemented through pattern references. For example, the pattern `<a><pref id="b"/></a>` defines that the child of an element `<a>` is the pattern with id `b`.The resolution of references is performed at runtime using a lazy semantics: the reference is dynamically replaced with the referenced

pattern when it is needed for matching or instantiation. In conjunction with prioritised choice, the lazy semantics allows recursive pattern definitions.

**Attributes** XPLT provides support for matching and instantiating XML attributes. However, patterns are restricted to the value part of attributes and need to be variables, plain text or code. For example, the pattern `<a b="$x">` matches an element with tag `</>` and an attribute `b`, the value of which is bound to the variable `x`. Instantiation creates an element `<a>` and adds the attribute `b` with the value of variable `$x` to the attribute list of this element. The W3C XML recommendation states that "the order of attribute specifications in a start-tag or empty-element tag is not significant' [173]. XPLT follows this recommendation and ignores the order in which attributes appear in the DOM tree. The matching semantics are also not concerned with completeness of attributes but only check the presence of an attribute.

**Code** The code pattern allows inserting source code into the DOM tree during instantiation. It may only appear at the value position of an attribute. It starts with an exclamation mark, followed by an identifier for the programming language used, a colon and the source code. Code patterns are ignored during matching. The primary use of this pattern in XMF is for implementing editing functionality in HTML views and for triggering transformations. The latter usage is exemplified by the following pattern.

```
<button onclick="!js:transform(this,p1,p2)"</button>
```

The JavaScript code fragment specifies that the element it is contained in will be transformed using the pattern `p1` as the source and `p2` as the target.

### 6.3.2 Transformation Engine

XPLT-patterns are not only used for syntactic validation of XML documents but are the basis for transforming and querying models. The semantics for the patterns defined in the previous section correspond to the semantics defined in Chapter 3. However, XPLT implements only a subset of the pattern functionality as operators and instead provides a functional interface for performing various pattern-based operations. The core functions of this interface are $match$, $instantiate$, $transform$, $query$ and $refine$. In the following, these functions will be discussed independent of an implementation language using concrete XML syntax for parameters and results of functions. The actual API functions operate on DOM tree fragments and are exposed as a JavaScript API. The use of this library to define constraints will be presented in Section 6.6.

**Matching** To verify that an XML fragment is an instance of a pattern, it is matched with the pattern. The result of a successful match is a set of bindings for all the variables in the pattern. The $match$ function has the following signature:

$$match(pattern, data) \mapsto bindings$$

In the following example, an element pattern <a> containing a variable $x$ is matched with an element <a> containing the element <c/>. The result is a binding from $x$ to the node <c/>

$$match(\texttt{<a>\$x</a>},\texttt{<a><c/><a/>}) \mapsto \{(x,\texttt{<c/>})\}$$

**Instantiation** To create an XML fragment based on a pattern, the pattern is instantiated in the context of bindings. The $instantiate$ function has the following signature:

$$instantiate(pattern, bindings) \mapsto data$$

In the next example, an element pattern <a> containing a variable $x$ is instantiated in the context of a binding from $x$ to <c/>. The result is an element <a> containing the element <c/>.

$$instantiate(\texttt{<a>\$x</a>}, \{(x,\texttt{<c/>})\}) \mapsto \texttt{<a><c/><a/>}$$

**Transformation** Based on matching and instantiating patterns, a transformation relationship for patterns can be defined:

$$transform(pattern1, pattern2, data1) \mapsto data2$$

The result $data2$ is an instance of the argument $pattern2$ which is instantiated with bindings created from matching $pattern1$ with $data1$. The function $transform$ relates $match$ and $instantiate$ in the following way:

$$transform(pattern1, pattern2, data1) = \\ instantiate(pattern2, match(pattern1, data1))$$

Unlike in languages such as W3C XSL Transformation language (XSLT) [172], in XPLT transformations are defined implicitly through naming conventions in the source and target patterns. For many structurally similar patterns, it is possible to compute transformations automatically. The following transformation is an example:

$$transform(\texttt{<a>\$x</a>},\texttt{<b>\$x</b>},\texttt{<a><c/><a/>}) \mapsto \texttt{<b></c></b>})$$

When editing data through a view, it is necessary to have transformations working in two directions. Whenever the user switches between views on a model, the data has to be transformed between two representations. As XPLT transformations are not explicitly defined but instead derived from a pair of patterns, there is no problem of the form "calculate an inverse transformation from a given transformation". Instead, it is the responsibility of the language designer to define patterns in such a way that a bidirectional transformation can be derived. Two patterns $p_a$ and $p_b$ define a bidirectional mapping between two data structures $a$ and $b$ if:

$$transform(p_a, p_b, a) = b$$

$$transform(p_b, p_a, b) = a$$

The patterns $p_a$ and $p_b$ define two data languages $L_a$ and $L_b$. Every data structure that matches $p_a$ or $p_b$ is an element of $L_a$ or $L_b$ respectively. A transformation between two languages $L_a$ and $L_b$ is given if the above equations are true for all $a \in L_a$ and all $b \in L_b$.

**Refinement** The function $refine$ implements partial instantiation of pattens as defined in Section 3.6. While the $instantiate$ function fails in the case of a missing binding for a variable, $refine$ returns the variable uninstantiated. The function expects a pattern and bindings as arguments and returns as a result the pattern with all free variables instantiated with values from the bindings. If the pattern is refined to a structure that contains only elements of the data language, the refinement has the effect of instantiating the pattern to data.

$$refine(pattern, bindings) \mapsto pattern/data$$

The following example shows the refinement of the general pattern that expresses "any element with a child element" to "any element <a> with a child element":

$$refine(\texttt{<\$t>\$x</\$t>}, \{(\texttt{\$t,a})\}) \mapsto \texttt{<a>\$x</a>}$$

**Querying** The XPLT query interface allows to express queries against XML data with patterns. The query function returns all instances of its argument pattern in the data as a sequence.

$$query(pattern, data) \mapsto data\text{-}sequence$$

For example, querying the data

<c>

```
 <a>1</a>
 <a>2</a>
</c>
```

with the pattern `<b>$x</b>` has the result `<res><a>1</a><a>2</a></res>`. The query semantics are equivalent to those defined by the $findall$ operator of Section 3.5.1 when it is configured to descent on any node.

## 6.4 Modelling with XMF

This section shows how XMF uses XML to represent models and to encode intra- and inter-model relationships.

### 6.4.1 Model Representation

The XMF framework contains two exemplary implementations of modelling languages, one for class and one for object modelling. The following XML-fragment is an example of a class definition in the abstract XML syntax of the class modelling language. It describes a class with id `1`, name `Person`, a parent class `Object`, two attributes and one operation.

---

```
<class id="1">
  <name>Person</name>
  <parent>Object</parent>
  <attributes>
    <attribute>
      <name>name</name> <type>String</type>
    </attribute>
    <attribute>
      <name>age</name> <type>Integer</type>
    </attribute>
  </attributes>
  <operations>
    <method>
      <name>saySomething</name><params>x</params>
    </method>
```

```
  </operations>
</class>
```

The XML editor built into XMF allows to create such textual model definitions. The left-hand side of Figure 6.6 shows the same class in a UML-style box-representation. This representation is created in the browser by rendering the HTML that results from model-view transformations. The right-hand side of the figure shows that text boxes and buttons can be used for adding, modifying and deleting classes, attributes and methods. The synchronisation between different editing modes is performed automatically when a user saves the model or changes between views.

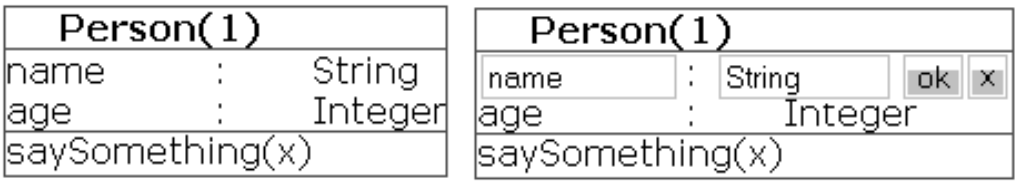


Figure 6.6: HTML User Interface for Displaying and Editing Classes

The element `<classRelationship>` describes relationships between classes. The following XML fragment defines an association ”supervises” between a professor and an arbitrary number of students.

```
<classRelationship>
  <type>association</type>
  <name>supervises</name>
  <source>
    <ref>Professor</ref>
    <multiplicity>1</multiplicity>
    <role>Supervisor</role>
  </source>
  <target>
    <ref>Student</ref>
    <multiplicity>*</multiplicity>
    <role>Supervised</role>
  </target>
</classRelationship>
```

The relationship is of type *association* and has the name *supervises*. A *Professor* has the role *Supervisor* for an arbitrary number of students. Students have the role *Supervised* with exactly one professor. The graphical view on this data is shown in Figure 6.7. The relationship is rendered as an HTML table with similar editing capabilities as described for the class view above.

| Name | Type | SourceRef | SourceMulti | SourceRole | TargetRef | TargetMulti | TargetRole |
|---|---|---|---|---|---|---|---|
| supervises | association | Professor | 1 | Supervisor | Student | * | Supervised |

Figure 6.7: HTML User Interface for Displaying Relationships

The following XML fragment shows how classes and relationships are integrated into a class model.

```
<model id="classModel" name="Class Model">
  <classes>
    <class>...</class>
    ...
  </classes>
  <classRelationships>
    <classRelationship>...</classRelationship>
    ...
  </classRelationships>
</model>
```

The element *model* must be the root element of all models. The class model contains an arbitrary number of classes followed by an arbitrary number of class relationships.

The object model representation follows the same principles as the class model representation. The following is the representation of an object with id “john” that is an instance of the *Professor* class. It contains two attribute values that reference the name and value attributes defined by this class.

```
<object id="john">
  <class>Professor</class>
  <attributeValues>
    <attributeValue>
      <ref>name</ref>
```

```
      <value>John<value>
    </attributeValue>
    <attributeValue>
      <ref>age</ref>
      <value>47<value>
    </attributeValue>
  </attributeValues>
</object>
```

### 6.4.2 Relationships between Models

One way of encoding the relationships between different model elements is by using references. By convention, references are encoded as a `<ref>` element containing the name or id of an element. The target of a reference can be in the same model as the reference itself or in a different model. For example, the following object relationship is part of an object model. It is an instance of a *supervises* relationship in a class model. This is expressed by a reference inside the *instanceof* element. In addition to that, the `<source>` and `<target>` elements are intra-model references to objects.

```
<objectRelationship id="supervises1">
  <instanceof><ref>supervises</ref></instanceof>
  <source><ref>john</ref></source>
  <target><ref>jim</ref></target>
</objectRelationship>
```

A class model defines not only the structure of individual objects but also possible associations between objects. For example, the multiplicities of the $supervises$ relationship in the class model define that student objects in the object model must be connected to exactly one professor object. This kind of relationship is enforced by XMF through constraint-based model checking as will be described in Section 6.6.

## 6.5 Creating Modelling Languages

This section shows how XPLT patterns are used to define the abstract and concrete syntax of modelling languages.

### 6.5.1 Internal XML Representation

The following XML fragment defines a schema for the class representation introduced in the previous section.

```
<pattern id="class">
  <class id="$id">
    <name>$cname</name>
    <parent><ref>$parent</ref></parent>
    <attributes>
      <repeat>
        <pref id="attribute"/>
      </repeat>
    </attributes>
    <operations>
      <repeat>
        <pref id="method"/>
      </repeat>
    </operations>
  </class>
</pattern>
```

The class pattern requires a class to have an id, a name, a parent reference, attributes and methods. The id and name are variables in the pattern. The two repeat elements express that there may be an arbitrary number of attributes and methods. The actual representation of attributes and methods is defined by an attribute pattern that is referenced using the `pref` (pattern reference) element. The referenced attribute pattern is the following:

```
<pattern id="attribute">
  <attribute>
    <name>$aname</name>
    <type>$atype</type>
   </attribute>
</pattern>
```

This pattern defines that an attribute must consist of a name and a type element.

### 6.5.2 Display Views

The graphical representation of classes shown on the left of Figure 6.6 is based on an HTML representation of the class that defines a class to be rendered as a table. The following pattern defines this HTML representation. Together with the pattern for the abstract syntax of a class defined in the previous section, this concrete syntax pattern defines a bidirectional view transformation.

```
<pattern id="classHTML">
 <TABLE cellpadding="2">
  <THEAD>
   <TR><TH colspan="2">
    <SPAN>
     <sequence>
      $cname <text>(</text> $id <text>):</text> $parent
     </sequence>
    </SPAN></TH></TR>
   </THEAD>
   <TBODY>
     <repeat><pref id="attributeHTML"/></repeat>
    </TBODY>
    <TBODY>
      <repeat ><pref id="methodHTML"/></repeat>
    </TBODY>
  </TABLE>
</pattern>
```

The table header is a sequence of class name, opening parenthesis, class id, closing parenthesis, a colon and the name of the parent class. The rest of the table is defined by two body elements that list attributes and methods. Attributes and methods are rows in the table which are defined by the referenced `attributeHTML` and `methodHTML` patterns respectively.

### 6.5.3 Edit Views

XMF supports not only displaying but also editing of models through HTML user interface elements. These elements include text fields, check boxes and buttons. The editing

logic may be added as a separate view definition. Switching between the regular view and the editing view is defined in a fine grained way manner on the element level. For example, in Figure 6.6, an attribute is edited by replacing the plain text in the table fields with input fields. This is done by clicking on the attribute. The display view is defined by the following pattern.

```
<pattern id="attributeHTML">
 <TR onclick="!JS: transform(this,'attributeHTML',
                      'attributeHTMLEdit')">
    <TD>$aname</TD>
    <TD>:</TD>
    <TD>$atype</TD>
 </TR>
</pattern>
```

The table cells for name and type solely consist of the corresponding variable names. A code pattern inserts the transformation logic as a JavaScript call to the transform function with the source and target pattern being the display and edit views for attributes. The edit view is defined by the following pattern:

```
<pattern id="attributeHTMLEdit">
 <TR>
  <TD><input type="text" value="$aname"/></TD>
  <TD>:</TD>
  <TD><input type="text" value="$atype"/></TD>
  <TD>
   <button type="text" onclick=
    "!JS: transform(this.parentNode.parentNode,
        attributeHTMLEdit','attributeHTML')">
    ok
   </button>
  </TD>
  <TD>
   <button type="text" onclick=
     "!JS: removeElem(this.parentNode.parentNode)">
    x
```

```
    </button>
   </TD>
  </TR>
</pattern>
```

A table row in edit mode consists of two text input fields and two buttons *ok* for confirming the editing and *x* for deleting the attribute. The class attribute is edited via the input fields. The button *ok* maps changes back to the display view. This is defined by the JavaScript code fragment that calls $transform$ on the attribute, with the edit view as the source and the display view as the target. If the transformation would be applied to the DOM tree representation directly, changes would not be visible. This is because the value attribute of input elements represents only the *initial value* and not the *current state* of the user interface. To make the current values accessible to pattern matching, XMF traverses the DOM tree before applying a view transformation and replaces the value of the value attribute with the current value of the user interface component.

## 6.6 Relationships and Constraints

Section 6.3.2 discussed a set of functions for matching, instantiating, transforming, querying and refining patterns. These functions are exposed in XMF as a JavaScript API. This API is used for defining intra- and cross model relationships and semantic rules for providing modelling feedback.

### 6.6.1 Defining Intra- and Inter-Model Constraints

Constraints in XMF typically have the form “for all objects of type T in the model M, condition C must hold” or “for all elements O1 of type T1 in model M1 must exist an element O2 of type T2 in model M2 where O1.x=O2.y”. Formulating such constraints is primarily an interplay of the reference, refinement and query functionality: a referenced pattern is first refined by fixing a subset of its variables and then a query is performed with the refined pattern.
For example, the following query returns all classes with name “Student” from a given model:

```
query(refine('class', [{'$name','Student'}]),'classModel')
```

The parameter `'class'` is a string that references the $class$ pattern. The argument `[{'$name','Student'}]` is a binding from `'$name'` to `'Student'` using lit-

eral syntax for an array containing a JavaScript object. The result of $refine$ is the $class$ pattern with the name fixed to `'Student'`. This refined pattern is used for querying the model referenced by the string `'classModel'`. The result is a sequence of class instances represented as DOM tree fragments. There are further helper functions for formulating constraints. The function `queryVar` expects a pattern, a model and a variable name from the pattern. It performs a depth-first search for the pattern in the model, matches the pattern to the first occurrence and returns the value bound to the variable.

The function `chain` abstracts querying structures that are linked using references. For example, classes are linked in an inheritance relationship by adding to each child class of a class a parent reference. Following the parent references from any class to the root class "object" yields a `chain` of classes that are in an inheritance relationship. Retrieving this chain using the basic functions requires repeatedly calling `queryVar` to retrieve the reference, `refine` to refine the class pattern and `query` with the refined class to resolve the reference. The `chain` function hides all this. It expects as arguments a pattern that defines referencing and referenced objects, the variable that is the source of the reference, the variable that is the target of the reference (typically the id or name of a model element), a start value and an end value. It returns all elements between start and end. An example of the use of `chain` will be given below. Other functions of the API are `assert` which throws an exception if its argument does not yield a truth value and `len` which returns the length of a list. The function `equalNode` checks equality between two DOM tree nodes.

**Listing 23** Constraints for InstanceOf Relationships

```
1  function checkInstanceOf(oID,cName){
2   //the object with ID oID
3   var object=
4     query(refine('object',[{'$objectID':oID}]),'objectModel');
5
6   //assert that the object references the class
7   assert(eqNode(queryVar('object',object,'$class'),cName))
8
9   //all classes that are in an inheritance relationship
10  //between cName and Object
11  var classes=chain('class','$cname','$parent',
12         cName,'Object','classModel');
13
14  //attributes of all classes
15  var attributes=query('attribute',classes);
16
17  //all attribute values of the object
18  var attribValues=query('attributeValue',object);
19
20  //assert that the number of attribute values of the
21  //object equals the number of attributes of the classes
22  assert(len(attributes)==len(attribValues));
23
24  //assert that for every attribute value
25  //there is exactly one attribute
26  for(var i=0;i<attributes.childNodes.length;i++) {
27    var attrib=attributes.childNodes[i];
28    var attribName=queryVar('attribute',attrib,'$aname');
29    var avalue=query(refine('attributeValue',
30              [{'$attribName':attribName}]),attribValues);
31    assert(len(avalue)==1);
32  }
33  return 1;
34 }
```

The source code in Listing 23 defines a constraint for validating that an object is an instance of a class. The constraint is defined as a JavaScript function that expects an object

id and a class name as arguments. In lines 3 and 4, the object belonging to the object id argument is queried from the model. This is done by refining the object pattern with a binding from `$objectID` to the argument `oID`. The first check for the "instanceOf" relationship is performed in line 7 by asserting that the class reference in the object is equal to the argument `cName`. In lines 11 and 12, the `chain` function is used to query the class referenced by the `cName` argument and all its superclasses. On this result, a query using the attribute pattern is performed which yields all attributes of the classes (line 15). In line 18, the attribute values are queried from the object. Line 22 asserts that the length of the attributes from classes in the class inheritance chain equals the length of attribute values in the object. Finally, a *for loop* is used to check for every class attribute that there is exactly one attribute value in the object with the same name.

The current implementation of the constraint language as a JavaScript API can be improved by adding a syntactic interface that abstracts from the JavaScript code and defines an actual constraint language. This remains future work.

### 6.6.2 Interactive Modelling

In programming, various stages of program execution, e.g., compiling, running and debugging, provide feedback to the programmer as to the correctness of the program. For the user, this allows an interactive approach where feedback from the system leads to improvements in the program. Modelling, on the other hand, is a far more static process that lacks interactivity and feedback. This makes it difficult to grasp the implications of definitions in a model. One of the goals of XMF is to provide a more interactive modelling process. The use of constraints as defined above plays an important role in this respect. This is because constraints can be used to guide modelling and provide a basis for the modelling system to give feedback if a model violates constraints. Overall, XMF provides the following types of checks on models:

- Well-formedness of XML. Detects syntax errors in XML such as non-matching start- and end-tags in an element definition.
- Modelling language validity. Detects modelling language syntax errors. For example, if an object is missing an object id.
- Intra-model constraint satisfaction. Detects violations of rules in a single model. For example, if parent relationships between classes are circular.
- Inter-model constraint satisfaction. Detects violations of rules across different models. For example, that the number of object relationships in an object model exceeds those defined by the multiplicities in the class model.

XMF offers feedback for all these errors. The first two checks look for syntactic errors by attempting to parse the XML (well-formedness) and by matching the XPLT pattern that defines the language with the model (validity). The latter two checks detect semantic errors and are based on constraint definitions with the scope of single or multiple models.

## 6.7 Summary

This chapter applied the pattern foundation not only to a new problem domain – modelling and meta-modelling – but also to a new technological domain: web technologies. The modelling and meta-modelling framework XMF uses XML as the internal representation of models, HTML for viewing and editing models in the browser and a JavaScript API as a constraint and interaction language. The fact that HTML is a structured representation of a user interface makes it possible to extend the view principle used in Concat on the text-only level to graphical user interfaces. The underlying technique is more fundamental: the HTML representation defines the shared space of an interface. This interface marks the boundaries between a domain of symbolic manipulation and a domain of symbolic interpretation outside of the grasp of symbolic manipulation. The pattern approach in this work can be seen as an attempt to push the boundaries of the symbolic manipulation domain.

The pattern language XPLT plays a central role in XMF. XPLT can be seen as a schema language but it defines schemas in such a way that the schema information is available for more than validating a document. Instead, it is used for transforming and querying models and for refining patterns. This functionality is the basis of the view mechanism and the constraint language. Constraints complement XMF patterns with means to express complex structural properties. The constraint language fills the role of the relationship language in the XMF architecture by allowing intra- and inter-model relationships to be formally defined. It also allows to impose arbitrary semantic constraints. This is the basis of the successful use of XMF as a device for teaching modelling: semantic constraints are defined by teachers to make design errors for a particular modelling task detectable. In this case, the system can give interactive feedback to students.

In the last few years, much research has gone into the development of modelling tools and architectures in the context of model-driven development. The driving force behind this development is the desire to generate full code from models [97]. This turns modelling languages eventually into visual programming languages. XMF was intentionally designed with a different kind of modelling in mind: XMF models can be abstract in the sense that they actually omit irrelevant details and serve as diagrammatic tools for visualising certain structural and operations aspects of systems rather than code. Therefore,

XMF does not define an executable language as a target for transformations and instead only provides a view mechanism. However, given the existing functionality for defining transformations and the generic nature of the XML representation, it would be possible to add a backend for code generation to XMF. An interesting research direction would be the integration of Concat in XMF. This way, the execution language can be defined using Concat and the modelling and editing facilities using XMF.

# Chapter 7

# Analysing Communication Systems

This chapter presents a pattern-based approach that extends the notion of protocol analysis in automotive networks. The approach is the outcome of a collaboration with the Automotive Competence Center at Heilbronn University. The work is motivated by the growing complexity of automotive networks and the underspecification of communication behaviour (Section 7.1). Abstract protocols are introduced as a tool for making application-specific communication behaviour and design intentions accessible on the message level. This allows the specification and analysis of complex scenarios at the appropriate level of abstraction (Section 7.2). Abstract protocols are defined by a mapping of communication patterns in a protocol layer to messages in the abstract protocol layer. The mapping is described as a configuration of message processing units: channels represent a communication medium, filters include or exclude individual messages based on a criterion and rules associacte communication patterns with messages (Section 7.3). These concepts are formalised using pattern expressions. Messages are typed sequences and message formats are typed sequence patterns. Filters are based on typed sequences and pattern-based search. Rules are transformations with a message pattern as source and a message sequence as target. Layering is defined as the vertical combination of channels, filters and rules (Section 7.4). The approach has several application areas in automotive systems engineering including re-engineering, system comprehension and automated testing (Section 7.5). A visual domain specific language based on the approach uses generative techniques to produce executable protocol analysers for different platforms (Section 7.6).

## 7.1 Complexity in Automotive Networks

The amount of electronics and software in vehicles has increased rapidly over the last three decades. Modern cars contain up to one hundred Electronic Control Units (ECUs) that are in charge of different subsystems, ranging from motor control to entertainment [139]. Bus systems connect these distributed ECUs into communication networks and thus allow previously autonomous subsystems to exchange information in order to provide more advanced functionality [4].

Coping with the system complexity resulting from highly interconnected software-intensive subsystems poses one of the great challenges for the automotive industry today [23]. All software running in vehicles has to be reliable as software updates involve costly maintenance. Crucial subsystems such as brakes, steering and airbags require utmost reliability as their failure might cost human lives [18]. Despite that, the millions of lines of source code running in modern cars have not yet reached a sufficient degree of reliability [23] and current testing methodologies are reaching their limits [24]. Problems caused by software are quickly becoming a major reason for car defects.

Figure 7.1 depicts a typical communication context in a modern car. Subsystems in distinct domains such as drive train or multimedia are connected by specialised bus technologies, e.g., CAN (Controller Area Network) [17] or MOST (Media Oriented Systems Transport) [124], to form subnetworks. Communication between components in different subnetworks is established through gateways that bridge technological differences.

An example of advanced functionality that utilises this heterogeneous communication architecture is the adaptive brake light: becoming a standard in today's cars, it warns following drivers through rapid blinking when an emergency brake manoeuvre is executed. To correctly implement its functionality, data from different subsystems in different subnetworks has to be exchanged and compared. This is achieved by sending messages over the communication bus. Among the relevant data might be the speed of the car provided by the instrument cluster and the reaction of the anti-lock brakes provided by the brake system. In addition to that, the lighting system has to be instructed to blink rapidly.

### 7.1.1 Protocols, Layering and Underspecification

As described in the previous section, the different ECUs and buses in modern cars constitute complex, distributed systems of communicating entities [156]. Such systems have been the subject of much research in the telecommunication and computer networking domain and many of the techniques in these domains have been applied to networks in cars as well. This includes the fundamental concepts of protocols and layering. Layering is a

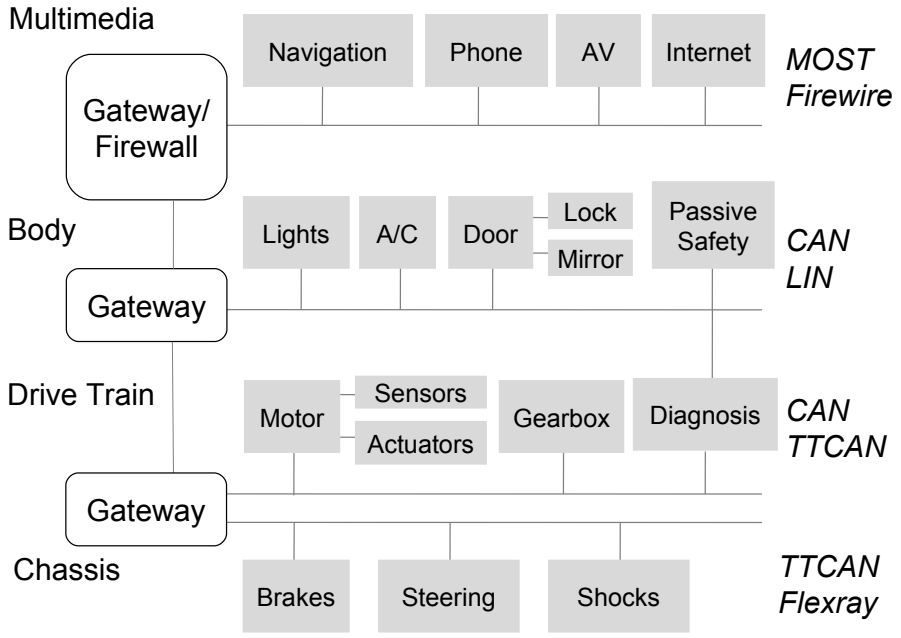


Figure 7.1: Typical Communication Context in a Modern Automotive Network

basic technique for designing distributed systems. It is a method to provide in a stepwise fashion higher-level service to users on the layer above, and to separate levels of services by precisely defined interfaces [71]. This overall design principle is reflected by the use of protocol stacks [87] where higher layer network services rely on lower layer services until a physical layer is reached [79].

Protocols define the format and order of messages that can be exchanged between components. ECUs have to implement these protocols in order to successfully communicate. *Protocol analysers* can help both testing and understanding systems that communicate based on protocols. A protocol analyser scans messages on the network, decodes the messages in accordance with the protocol standard into a reader-friendly textual or graphical representation and offers various monitoring functions.

Protocol analysers can monitor system communication on different layers of a protocol stack [65]. However, in today's complex systems, not all communication behaviour is specified [24]. Instead, communication intentions may be implicit, undocumented or evolved during implementation – with the design intention hidden in the source code. Unspecified communication behaviour that is not part of a protocol is invisible during analysis. Instead, only the effect that the implementation of a design has on a lower protocol layer can be monitored.

For complex communication scenarios such as a braking manoeuvre with an adaptive brake light where a sudden event triggers a chain of messages between different com-

ponents in different subnetworks, underspecification poses a severe problem for system comprehension and testing: communication not specified in a protocol is shown at the "wrong" level of abstraction during analysis and thus contains irrelevant details.

### 7.1.2 Underspecification: An Example

In a request/reply protocol, a valid communication always consists of a request by the client and a reply by the server. This entails that the server does not possess means to contact the client directly without receiving a prior request. A common communication scenario is that a client needs information about a particular state change on the server. The obvious solution – that the server sends a notification message to the client as soon as the state changes – seems impossible because of the restriction on initiating communication of the request/reply protocol.

One solution to this problem is to implement notification services for both the client and the server. The services encapsulate request/reply-based communication and offer interfaces that provide notification capabilities to the rest of the application. To achieve this, the service implements a polling strategy. That is, the service on the client periodically sends requests to the server. Upon receiving a request, the service on the server can reply and transmit either a negative acknowledgement or deliver the notification message. For applications using the notification service, communication is based on notifications, not on requests and replies.

Request/reply protocols and polling techniques for implementing notifications are widely used in automotive networking. The MOST network uses request/reply mechanisms for communication between clients, e.g., Human Machine Interfaces, and servers, e.g., radio tuners [119]. For example, to determine the current radio frequency, a message is sent to the tuner with the content

```
AMFMTuner.Frequency.Get()
```

and the reply is sent back with the current value. On an application level, API calls make it possible to subscribe a client to be notified if a property such as the frequency changes. CAN, on the other hand, is purely built upon broadcast of signal values. However, the Remote Transmission Request Bit that is part of the protocol header can be used to stimulate the transmission of a signal value. This is in effect a request/reply mechanism. The following client-code illustrates how notification based communication may "look like" from an application perspective:

```
fmTuner.regForNotification(this.tunerStateChange())
```

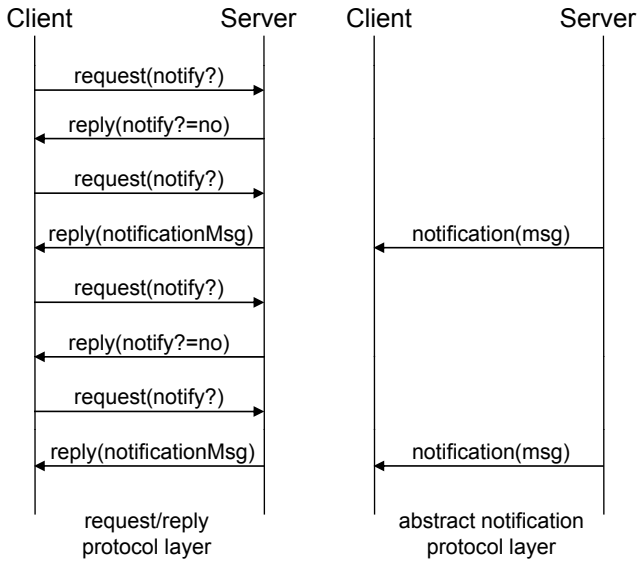


Figure 7.2: Protocol Abstraction: Defining an Abstract Notification Layer

The server object provides a method that allows the client to register a callback for notification in case of a state change. The service on the client starts polling for updates periodically. In the application running on the tuner, code such as the following is used:

```
client.notify(ATFrequency,102.6)
```

The programming interface suggests that the server notifies the client about a change of frequency. What actually happens is that the service that receives client requests sends the state update the next time the client requests an update. The client callback function `tunerStateChange()` then reacts to the state change. The fact that the underlying system uses a polling strategy is abstracted through the programming interface server and client side.

The left sequence chart in Figure 7.2 shows how a possible communication involving two notifications might be traced with a protocol analyser for the request/reply protocol. The communication consists of all request/reply messages between client and server. However, the notification service layer that was introduced in the software is invisible. That means, by using the protocol analyser, it is impossible to view the system behaviour at a level of abstraction where a server sends notifications to a client, although – as just described – this is the way application developers think about the communication behaviour. Viewing communication at this "higher level" is what is typically needed for reasoning about communication. For instance, when the goal is to determine which notifications are

sent from the server and how the client reacts to them, the implementation of the notification mechanism is irrelevant. Instead, the communication behaviour is analysed in terms of *application level abstractions*.

## 7.2 Abstract Protocols and Complex Scenarios

### 7.2.1 Abstract Protocols

How can application-level abstractions be made accessible during analysis and how can they be used in describing communication scenarios? The key idea is the introduction of a separate application-specific protocol layer. A relationship between two layers can be established by relating messages on one layer with messages on an adjacent layer. For example, from the viewpoint of the request/reply protocol notification messages do not exist, they are introduced on the protocol layer above. The term *abstract protocol* will be used to refer to such application-specific protocols. The right-hand side of the sequence chart in Figure 7.2 shows how communication is perceived when analysing the notification layer. It is important to understand, that abstract protocols also introduce the idea of abstract communication channels. That is, the messages shown are independent of the actual physical media on which lower-level messages are sent. An abstract message may actually be implemented through a set of different messages in different sub-networks.

To support abstract protocols, an analyser must be aware of the additional service level introduced by an application. For the notification-example, this means that a notification protocol is defined where valid communication consists only of a server sending a notification message to a client. The protocol is then related to the request/reply protocol by defining how a pattern on the request/reply layer maps to a message on the notification layer: a request message from a client followed by a positive reply from the server to the same client should be interpreted as a notification message from server to client. The notification protocol captures the design intention of the application designer. Abstract protocols may be stacked in the same way as regular protocols. An abstract protocol can be stacked on top of another abstract protocol. Indeed, even the request/reply protocol of the example just discussed may be an abstract protocol based on a yet lower protocol layer. This approach is open ended and can be used to build levels of abstraction in a stepwise fashion, thereby mirroring the abstraction process in the application itself.

### 7.2.2 Complex Scenarios

A complex scenario is a chain of communication events, possibly in different subsystems, that can be associated with a use case. An example of a use case is a braking manoeuvre in a car with adaptive brake lights. If the user – a driver in this case – performs a braking manoeuvre that activates ABS (Anti-lock Braking System), the brake light potentially shows some reaction that deviates from its normal behaviour, depending on speed, strength of braking and the reaction of the driving assistance systems. The use case can be described as: *The driver performs a hard braking manoeuvre*. At the level of ECUs, the following – simplified – scenario can be derived from the use case:

1. The braking system registers blocking of tires and activates the anti-lock brakes.
2. The braking system notifies other components listening on the bus of the anti-lock brake activity.
3. The control unit in charge of the adaptive brake light registers the activity of the ABS and requests the current speed $v_c$ of the car from the instrument cluster.
4. Because $v_c$ is higher than the threshold $v_t$, the control unit decides to instruct the brake light system to flash.

A second scenario associated with the same use case shares the first three steps, only that in step 4 it determines $v_c \leq v_t$ and thus does not interfere with the normal operation of the brake light. While the above natural-language description of scenarios is rather imprecise, scenarios can be precisely described at the level of the actual messages exchanged between components over the communication buses.

Complex scenarios pose a challenge for both specification and testing of ECUs. They may involve a large number of different messages in different subsystems. In addition to that, the resulting communication patterns are likely to be interrupted as vehicle communications systems are heavily impacted by stochastic triggers arising from the system environment [119]. For example, mobile phone calls may appear at any time as well as warnings and recommendations from driver information and assistance systems. Even for moderate scenarios, the amount of communication to process may rise to an extent that makes monitoring and verification of scenarios a complex task. Describing the scenarios at the right level of abstraction is crucial for reducing complexity.

## 7.3 Defining Layers with CFR Models

Abstract protocols can provide a high level of abstraction to specify communication scenarios as patterns on abstract application-specific message sequences rather than on highly interleaved implementation-specific message sequences. The process of creating an abstract protocol can be described in three steps. Firstly, filtering out all messages not pertaining to the protocol implementation. Secondly, describing rules that relate patterns in the remaining messages to the abstract protocol layer. Thirdly, joining the resulting messages from different sources on a new channel.

### 7.3.1 Channels, Filters and Rules

As explained in the context of the request/reply example, it is a common technique to stack protocol layers in such a way that each layer interfaces directly adjacent protocol layers only. In this way, a protocol layer $L_n$ is described by relating its communication patterns to the communication patterns on the next lower layer $L_{n-1}$. Indeed, that is how the abstract notification protocol is informally described: a request for notification message from a client followed by a positive reply from the server to the same client should be rendered on the next higher layer as a notification message from server to client. Examining this natural language description of the protocol layer reveals that the following information is provided:

- Certain messages on the lower protocol layer are identified.
- The sequence in which these messages occur is given.
- A mapping of this message sequence to a message on the higher-level protocol layer is defined.

The follwing conceptual framework for defining protocol layers is the foundation of the language formalised and defined in the next sections. It defines three core concepts:

- A *channel* is a medium that transports a sequence of messages over time. The messages obey a concrete or abstract protocol. A protocol defines the message format and optionally a set of legal communication patterns.
- A *filter* is a passive and stateless message processing unit that redirects messages from a channel to a positive or negative output channel based on a filter criterion. The scope of a filter is a single message.

- A *rule* is an active and stateful message processing unit that consumes messages from one or more inputs and emits messages to at least one output. In opposition to filters, a rule may modify or create messages.

### 7.3.2 Models, Abstraction and Interpretation

CFR models relate protocol layers by composing channels, filters and rules. The inputs and outputs of filters and rules are sequences of messages. Theoretically, this allows arbitrary interconnection. Nevertheless, for methodological reasons, it is forbidden to directly connect filters with rules and vice versa. This constraint has the goal of producing more readable and reusable models by separating the concepts. The restriction does not reduce the expressive power of CFR models because placing an otherwise unconnected channel between a filter and a rule is equivalent to connecting the two elements directly. Inputs and outputs are abstracted by input and output pins. An input pin may be connected to an output pin using a connector. Channels and rules have an arbitrary number of input and output pins while filters have exactly one input and two output pins.

CFR models contain an abstraction mechanism for filters and rules. Compound filters can be defined by connecting existing filters. Rules can be defined by arbitrary CFR models with defined inputs and outputs. This abstraction mechanism allows to define rules and filters based on a set of primitive filters and to reuse compound filters as if they were primitive. Filters and rules that cannot be defined by composition are described using an external language. The requirement for the filter language is to express criteria on a message-level and the requirement for the rule language is to express criteria on the sequence level.

CFR models can be interpreted in two similar but distinct ways: (1) as a specification that defines a higher protocol layer through a lower protocol layer or (2) as instructions for a protocol analyser on how to render communication on a higher layer by observing communication on a lower layer.

Figure 7.3 shows the basic principle of using filters and rules for relating a protocol on a channel (Ch) to a protocol on an adjacent channel. The primary interest when *analysing* systems is the “bottom-up’ mapping. The goal is to find communication patterns and make them visible. In principle, a mapping could be defined “top-down” so that abstract protocol messages can be related to messages on the lower layer. This could for instance be used for describing a message generator on a high level of abstraction. If filters and rules are reversible, a mapping in one direction can be derived from a mapping in the other direction.

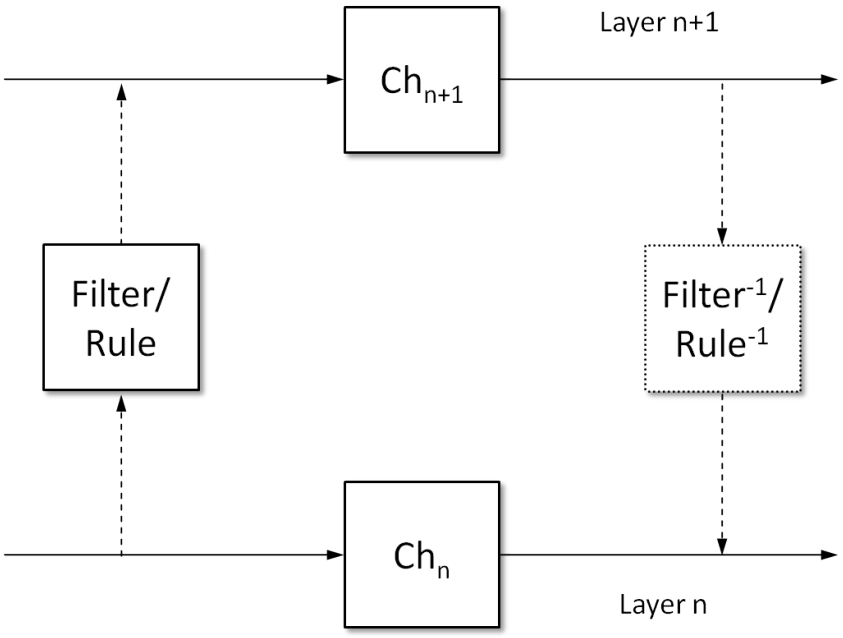


Figure 7.3: Relating Abstract Protocol Layers

### 7.3.3 Example: A CFR Model for the Notification Protocol

Let $C_{rr}$ be a channel over which messages defined by a protocol $rr$ – the request/reply protocol – are sent. These messages include request and reply messages related to notifications but also other request and reply messages not related to notifications. Let $C_{not}$ be a channel over which notifications messages are sent. A CFR model that describes how the message stream on $C_{rr}$ can be transformed into a message stream on $C_{not}$ is defined by the following setup: The output pin of channel $C_{rr}$ is connected to the input pin of a Filter $F_{not}$. The positive output pin of $F_{not}$ is connected to a channel $C_{rrf}$. $F_{not}$ is configured in such a way that it sends all request/reply messages concerned with notifications to its positive output and all other messages to its negative output. Accordingly, the communication on $C_{rrf}$ consists only of the request/replies concerned with notifications and no other messages. The output pin of $C_{rrf}$ is connected to the input pin of rule $R_{not}$. The first output pin of the rule is connected to the input pin of $C_{not}$. The rule is configured in the following way: When a requests message from a client occurs, the rule memorises the request. When a positive reply that fits a previously memorised request is received, a notification message is sent through the output pin onto $C_{not}$. The messages on $C_{not}$ are now notification messages sent from servers to clients. Omitting the filter and rule logic for the moment, the CFR model can be described as follows, whereby the $\rightarrow$ means *connected to* and where $R_{not}(out_1)$ denotes the first output channel of $R_{not}$.

$Channel : C_{rr}, C_{rrf}, C_{not}$

$Filter : F_{not}$
$Rule : R_{not}$
$C_{rr}(out) \rightarrow F_{not}(in)$
$F_{not}(positive) \rightarrow C_{rrf}(in)$
$C_{rrf}(out) \rightarrow R_{not}(in)$
$R_{not}(out, 1) \rightarrow C_{not}(in)$

The example indicates that the restriction on combining filters and rules enforces reusable and clear designs. Channel $C_{rrf}$ is introduced as a helper because the output pin of $F_{not}$ must not be directly connected to the input pin of $R_{not}$. The channel $C_{rrf}$ defines an intermediate step in the abstraction process. It introduces a sub-protocol on a layer that is between the request/reply and notification layers. This layer can be used for monitoring the flow of all messages pertaining to the notification protocol or re-used in further protocol or scenario definitions.

## 7.4 Pattern-based Formalisation

This section formalises the basic concepts introduced in the previous sections – messages, channels, filters, rules and layering – using the pattern approach.

### 7.4.1 Messages and Channels

A *message* is a typed sequence that consists of a unique identifier for the message type, a sender and receiver id and content. A *message format* is a restriction on both the type and the content part of a message and is expressed as a pattern on typed sequences.

**Listing 24** Request/Reply Message Format

```
1 rr_message =>
2 [:(request|reply)
3     [:sender <id>] [:receiver <id>]
4     [:content ([<key> <value>])+]].
```

The recogniser in Listing 24 exemplarily defines the message format for request and reply messages containing at least one key/value pair as input.

A *channel* is a sequence of typed sequences that match a certain message format. The sequence of messages represents a sequentially ordered history of all communication over the channel. The vertical relationship between a channel $C_n$ and $C_{n+1}$ is defined by a transformation $T_{(n,n+1)}$ with the source being a sequence of messages on $C_n$ and

the target being a sequence of messages on $C_{n+1}$. The transitive relationship between a channel $C_n$ and a channel $C_{n+m}$ where $m > 1$ is defined as the vertical combination of transformations $T_{(n,n+1)} \rightarrow ... \rightarrow T_{(n+m-1,n+m)}$.

In order to keep the presentation clear, messages will be represented by items $a$, $b$, $x$ and $y$ instead of typed sequences in some of the examples below.

### 7.4.2 Message Filters

A filter divides the messages on a channel into those that match certain criteria and those that do not. In effect, a filter creates two channels whose messages are mutually exclusive. The criterion is a pattern that matches a single message and has this single message as the result with no effect on the store. A successful match of the criterion $p_c$ always has the form $\langle p_c, m::C_r, \sigma\rangle \xrightarrow{m} \langle m, C_r, \sigma\rangle$ where $m$ is a typed sequence that encodes a message and $C_r$ is the sequence of messages after $m$.

The $findall$ operator introduced in Section 3.5 provides the foundation to express filtering. The key for defining both the positive and negative output of the filter is how the ignore operator `&i` is used with the search and context patterns. In case the context pattern finds all messages and the search pattern does not transform the input it matches, the result of a match is the entire input sequence. For example, matching the pattern `findall a among <any>` with input $[a, b, a, b]$ has the result $[a, b, a, b]$. To obtain only those elements that match the search pattern $a$, the ignore operator is used with the context pattern. Matching `findall a among &i(any)` yields the result $[a, a]$. Conversely, the elements not matched by the search pattern can be obtained by ignoring the result of the search pattern. Matching `findall &i(a) among any` yields the result $[b, b]$.

The pattern operators `filter-pos` and `filter-neg` in Listing 25 define the positive and negative output of a filter based on a filter criterion and a message format. The two productions have two parameters. The format of messages on the channel and the criterion for accepting and rejecting individual messages.

**Listing 25** Definition of Message Filters

```
1 $format $criterion filter-pos =>
2           findall ,$criterion among &i(,$format)
3 $format $criterion filter-neg =>
4           findall &i(,$criterion) among ,$format
```

Filters are combined using the vertical combinator. Abstraction of filters is defined by a

production with a filter name on the left and the vertical combination on the right.

### 7.4.3 Communication Rules

The scope of a filter is a single message only. Rules, on the other hand, have memory, which means they range over a sequence of messages. A rule recognises communication patterns and associates them with a message on an abstract communication channel. In principle, these patterns can be expressed by combining message recognisers with the horizontal operators and unconditional transformations defined in Chapter 3. For example, the transformation of the message sequence $[a\ b\ c]$ to $d$ is defined by the transformation `(a b c => d)`.

However, on a communication channel, the messages that make up a communication pattern are likely to be interspersed with messages not pertaining to the pattern, e.g., messages with differing senders or receivers or messages that belong to a different aspect of the systems operation. For example, applying the transformation defined above to the sequence $[a, x, b, x, y, c]$ results in failure. A general solution is to explicitly express the possible occurrence of interspersed messages in the pattern. For the example transformation, this can be achieved by interleaving its left-hand with the pattern `(x|y)*`. The result of this is the pattern `(a (x|y)* b (x|y)* c => d)`.

While the approach of explicitly defining message interleaving is possible, it leads to less readable definitions. Therefore, it is desirable to separate the aspect of message interleaving from the definition of the communication pattern. This can be achieved by vertically combining the positive part of a filter that removes interleaved messages with the transformation that defines the rule in terms of a message sequence. For the above example, this means that first the filter removes all messages that are not $a$, $b$ or $c$ and then applies the transformation. This configuration is exempli by the following pattern: `<<any> (a|b|c) filter-pos> -> (a b c => d)`. In effect, the transformation is defined on an abstract layer created through the filtering. While this is a basic technique for defining rules, the rule-specific filter and the rule are orthogonal concepts.

**Listing 26** Definition of Communication Rules and Channels

```
1 $pattern rule => findall ,$pattern among &i(<any>).
2
3 $filter $rule abs-rule =>
4       (,$filter -> ,$rule).
5
6 $format channel => (,$format)*.
```

Listing 26 defines a communication rule as a sequence of transformations that are combined by a choice operator and applied to the communication channel with $findall$. Because of the importance of the filter/rule combination and the restriction that filters and rules cannot be directly connected in a model, the concept is explicitly abstracted as an *abstract rule* (abs-rule) that has a filter and a sequence of transformation as arguments and vertically combines them. The previous section defined that filters and rules must not be connected directly but using an intermediate channel. An intermediate channel can be defined by using the channel abstraction shown in Listing 26. A channel can be used for checking the communication between adjacent patterns in a vertical combination. The channel `<<any> channel>` performs no validation. Accordingly, the vertical combination `<filter> -> <<any> channel> -> <rule>` equals `<filter> -> <rule>`. In addition to the validation functionality the channel provides a defined point of access for monitoring the system through side effects such as logging.

### 7.4.4 Application to the Notification Protocol

This section uses the pattern-based formalisation of channels, filters and rules to implement the mapping between the request/reply channel $C_{rr}$ and the notification channel $C_{not}$ via an intermediate channel $C_{rrf}$ that contains only those messages pertaining to the notification service. The recogniser in Listing 24 defines the message format on channel $C_{rr}$. The refined message format for notifications is defined by the production `rr_not_message` in Listing 27 which also contains the rest of the implementation.

**Listing 27** Pattern-based Definition of the Notification Layer

```
not_request =>
[:request
    [:sender <client>] [:receiver <server>]
    [:content ['type 'notification] ['ref n:<nat>] ]]

not_pos_reply =>
[:reply
    [:sender <server>] [:receiver <client>]
    [:content ['type 'notification] ['ref n:nat]
              ['val m:message]]].

not_neg_reply =>
[:reply
    [:sender server] [:receiver client]
    [:content ['type 'notification] ['ref n:nat]
              ['val 'no]]].

rr_not_message =>
    <not_request> | <not_pos_reply> | <not_neg_reply>.

f_not => <<rr_message> <rr_not_message> filter-pos>.

r_not =>
  <((<not_request> <not_pos_reply> => m:message) |
      (<not_request> <not_neg_reply> => )) rule>.

c_rr_c_not =>
     (<f_not> -> <<rr_not_message> channel> -> <r_not>).
```

The mapping between $C_{rr}$ and $C_{rrf}$ is defined by the filter `f_not`. Based on all messages defined by `rr_message`, filter `f_not` returns only those messages that match the criterion `rr_not_message`. The rule `r_not` defines two cases. The first case is a request followed by a positive reply which produces the notification message contained in the reply as the result. The second case is a request followed by a negative reply producing an empty result. The pattern `['ref n:nat]` ensures that a reply is associated

with the right request. Production `c_rr_c_not` relates channels $C_{rr}$ and $C_{not}$ by vertical combination of $f_{not}$ and $r_{not}$ with a channel in between.

## 7.5 Applications

### 7.5.1 Specifying and Monitoring Complex Scenarios

Rules recognise a communication pattern on a protocol layer $L_{i-1}$ and abstract the pattern as a message on the next higher layer $L_i$. By viewing scenarios as communication patterns, it becomes apparent that the approach for specifying protocol layers can be directly mapped to the specification of scenarios: a scenario is an abstract protocol with a single message. The messages of a scenario do not necessarily obey the same protocol. Complex scenarios such as those in the context of the brake light involve messages in different subnetworks. This makes specification using traditional techniques difficult. Abstract protocols accommodate for the heterogeneity of the messages involved in a scenario: first all relevant messages are combined onto a single abstract protocol layer and based on this protocol layer the scenario layer is defined. To define this unifying protocol layer using CFR models, each combination of a bus and a protocol has to be defined as a channel. A configuration of filters selects the relevant messages from each channel and rules convert these messages into messages obeying the protocol of the unifying channel.

What is required is that each scenario is uniquely identifiable via its message sequence. Ambiguities have to be resolved at the specification level. While there usually are a great number of protocol messages on the communication bus, the challenge is to identify only those messages that are relevant for a specific scenario. After these relevant messages have been singled out, they need to be assigned to a distinct scenario. This is the same procedure discussed for protocol abstraction. In this case, the low-level protocol layer may be the message transport protocol, e.g., on a CAN bus, and the abstracted protocol layer is a scenario.

The application of CFR models for defining *scenario analysers* will be explained based on the two example scenarios introduced in Section 7.2.2 in the context of the adaptive brake light. Figure 7.4 shows the message sequence of the two scenarios. Both scenarios are associated with the use case of a strong brake manoeuvre by the driver. Scenario A describes this use case under high speed, where the brake light flashes. Scenario B describes the case where the speed is low and the brake light operates normally. The two scenarios are depicted as sequences of messages whereby both scenarios share a part of the message sequence. Message $B1$ marks the starting point for both scenario $A$ and scenario $B$. Observing the message flow sequentially, starting at $B1$ there is an ambiguity for

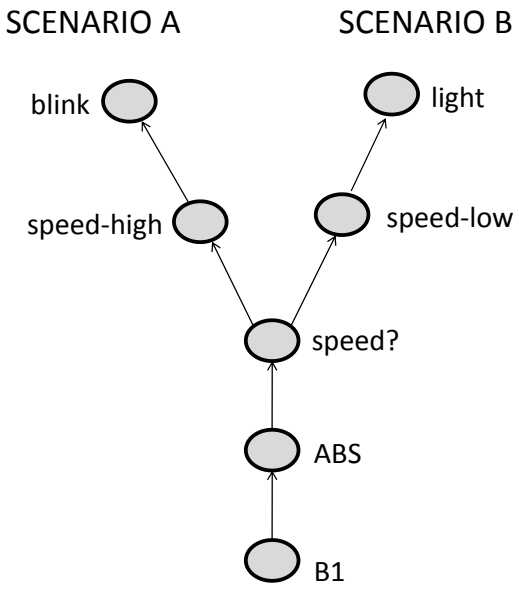


Figure 7.4: Two Scenarios based on the Adaptive Brake Light Use Case

the first three messages. The communication pattern can unambiguously be assigned to scenarios $A$ or $B$ only after the message *speed-high* or *speed-low* can be identified. With each new message, it has to be checked whether this message is the beginning of a new scenario or the continuation of one or more scenarios under monitoring. Unless the path of consecutive messages is not unique, several scenarios remain as candidates.

Under real-world conditions, a large number of different concurrent scenarios may be monitored simultaneously, some still open to a final decision. For example, each occurrence of a $B1$ message in Figure 7.4 initiates a new instance of a scenario monitor. A refinement of analysers is the assignment of time intervals to the arrows connecting two messages. This defines how long the monitor waits for the continuation of a scenario. If the expected future is not confirmed within a given time frame, tracing of the scenario is cancelled and an error is reported: Either the scenario has not been specified properly or the flow of messages between two or more parties is faulty due to misbehaviour of one or more communication partners. Message sequences that cannot be assigned to scenarios indicate underspecification.

The composition and abstraction mechanisms of CFR models may not only be applied to abstract protocols but also to scenarios. A sub-scenario is a communication pattern and abstracted using a message on a higher level. This approach is open-ended and allows stepwise definition of more complex scenarios through layering.

### 7.5.2 Reproducing Complex Scenarios for Test Automation

The previous subsection described monitoring and verifying complex scenarios. These techniques may be used to reproduce complex scenarios with the aim to automate system tests by simulating systems components. The key is to define a trigger message that starts a use case. For example, in the case of a braking manoeuvre the initial trigger is the anti lock brake activation. A certain initiation message and system state define the expected scenario. If this scenario can be reproduced from the messages on the bus the test case succeeds. The testing process can be divided into the following stages:

1. selecting standard protocols on which abstract protocols are defined
2. defining message interfaces, e.g., the output of a protocol analyser
3. defining abstract protocols based on standard protocols
4. specifying test scenarios as communication patterns on abstract protocols
5. associating triggers and system state with expected scenarios
6. sending triggers and attempting to reconstruct the scenario
7. monitoring of the system in case of failure to discover fault

Setting up the system state for a test scenario may involve the use of message generators and simulators that replace actual components. As described earlier, this can be achieved through a "downwards" mapping from an abstract protocol to a concrete protocol.

### 7.5.3 Relevance for Different Stages of the Development Process

The CFR approach provides support for several different stages of the software development process. The contributions are visualised using the V-Model, a widely used process model for software development in the automotive domain in Figure 7.5.

The most apparent contribution of a protocol analyser in the software development process is message level analysis. The message level can be associated to the Module level in the V-Model. This is where unit-testing takes place and where it is assured that system modules act in the desired way. Analysers generated from CFR models validate the correct communication behaviour between modules.

The approach presented extends the scope of protocol analysis to complex scenarios. Analysers for scenarios and abstract protocols validate the correct behaviour of system

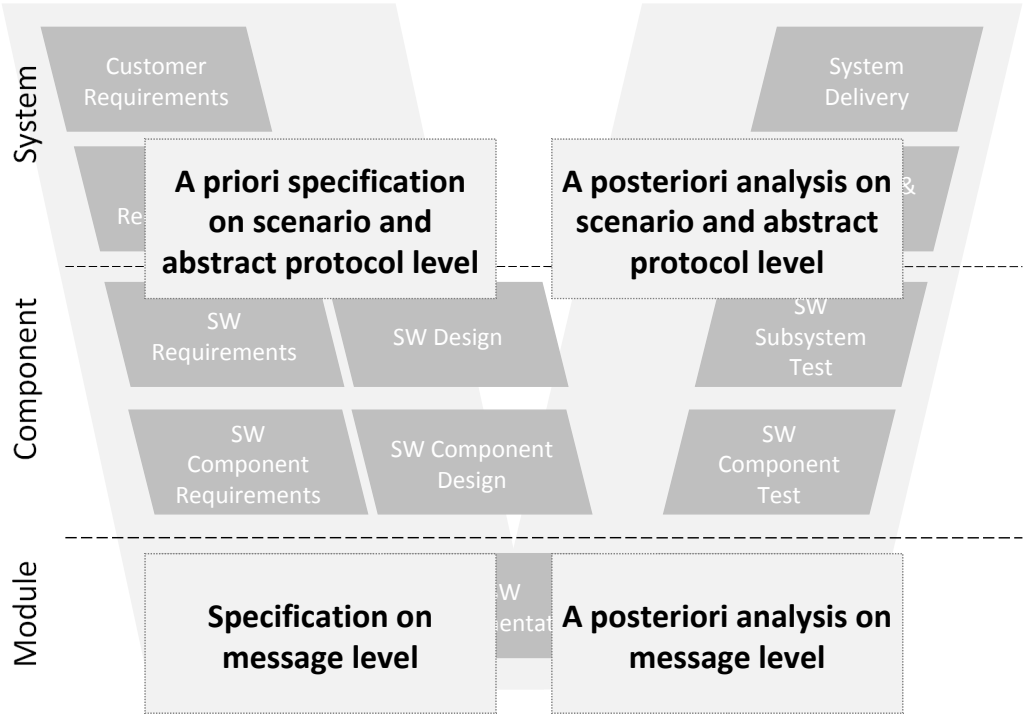


Figure 7.5: Identification of Possible Application Areas using the V-Model

components and the system itself. This contributes to the system test level and also at the component test level of the V-Model. With the capabilities of monitoring message flows and learning from them, it is possible to document and specify different parts of a distributed system. Network protocols may be specified by observing the communication on the message level. Complex Scenarios may be specified by monitoring messages in a broader context, concerning components of a network system or the whole system itself. Accordingly, there are two different types of analysis:

- Analysis with *a priori* knowledge
- Analysis with *a posteriori* knowledge

The challenge in the first case is to uncover underspecification in a given explicit specification. In the second case, design intentions of an implicit specification, e.g., an implementation, are made visible.

## 7.6 A DSL for Protocol Re-Engineering

This section describes the implementation of the CFR approach in the form of a Domain Specific Language (DSL) that generates full code for protocol analysers that target dif-

ferent platforms. The language is based on a visual notation for interconnecting message processing elements. State diagrams are used for visualising rules that are not created by composition. The goal is to provide an intuitive notation for experts in the automotive domain that have little or no background in general purpose programming languages. One of the benefits of the visual notation is the use of encapsulation to hide the details of a filter or rule implementation. This allows working with a model on different levels of abstraction.

### 7.6.1 Syntax and Semantics

This section presents the abstract and concrete syntax of the CFR language. Figure 7.6 shows the CFR model for the notification protocol using visual notation. The filter $F_{not}$ is implemented using a choice pattern based on the recognisers for notification-related messages defined in Listing 27. The state machine visualises the definition of the rule $R_{not}$ that is also part of Listing 27. The language for defining CFR models actually consists of three languages:

1. A language that has channels, filters and rules as basic elements and describes their interconnection. This is the language formalised in this chapter. The other two languages can be considered as languages embedded in this language.

2. A language for defining primitive filter criteria. This language is formalised in Chapter 3 and uses the Concat pattern notation introduced in Section 5.1.

3. A language for defining primitive rules. Two alternatives exist for defining rules. Either using Concat's pattern notation, as shown in Section 7.4, or as state diagrams that have patterns as transitions as depicted in Figure 7.6. The syntax and semantics of the state machine language are adopted from UML state charts [54].

Figure 7.7 shows a partial *meta-model* that defines the abstract syntax of CFR models. A model may contain an arbitrary number of processing units. A processing unit may be a channel, a filter or a rule and may contain at most one model. This containment relation defines structural embedding of CFR models. As discussed earlier, processing units are connected through connectors by using the pins of the processing units.

Inherent to the meta-model design is the philosophy to keep the meta-model itself as simple as possible and to express details with OCL (Object Constraint Language) [128] constraints. OCL Constraints are part of the Unified Modeling Language (UML) and add detail to UML (meta-) models [31]. A constraint defines a predicate on model instances. The constraint that prevents a CFR model from containing its parent model as a child can be defined as follows:

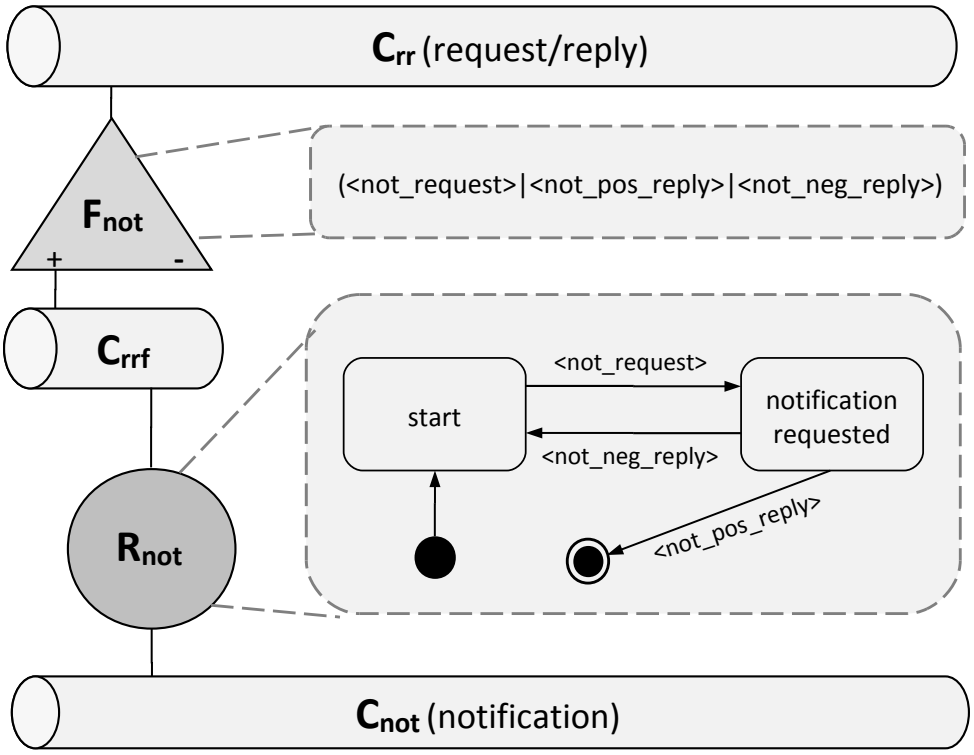


Figure 7.6: CFR Model for the Notification Example in Visual Notation

```
context model:
   inv noSelfContainment:
         processingUnits->forAll(model <> self)
```

The first line defines that the following expressions are applied in the context of a model. The keyword `inv` in the next line stands for *invariant* and states that the ensuing expression must be valid at any time in the system; <> means not equal.

The concrete syntax of the CFR language represents the three basic elements as follows:

- Channels as pipes; pipes are an intuitive metaphor for expressing the capability of messages to flow through channels
- Filters as triangles; a triangle is a natural shape for representing a component with one input and two outputs
- Rules as circles; circles are used because rules can have an arbitrary number of outputs

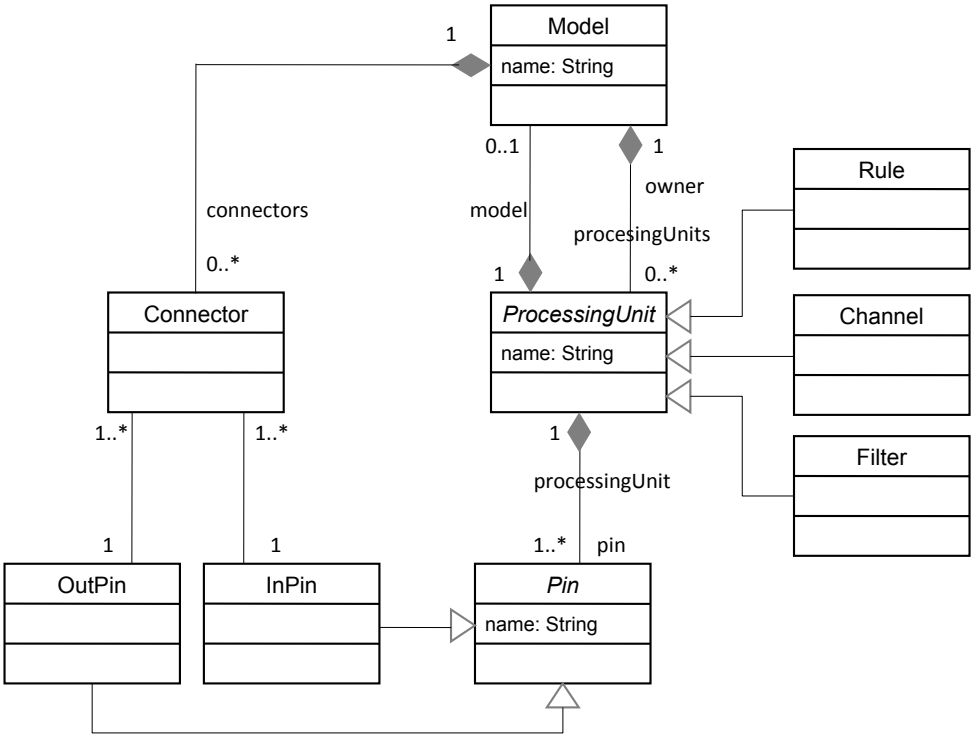


Figure 7.7: Partial Meta-Model of CFR

### 7.6.2 Implementation

As a proof of concept a prototype implementation of CFR using the meta-modelling and code generation tools available for the Eclipse [157] platform was created. An overview of the different parts of the implementation is provided in Figure 7.8. The definition of the abstract syntax of our language is based on a meta-model created using the Eclipse Modeling Framework (EMF) [152]. EMF defines a meta-modelling language and provides a user interface to create *ecore-models* based on Essential MOF (Meta Object Facility) [129], a simple subset of the MOF. EMF also has several built-in facilities for generating code and for editing and creating models based on a meta-model representation.

To define the language constraints and in order to have error detection at edit-time for CFR models, oAW (openArchitectureWare)[1], a modular Model Driven Architecture (MDA) generator framework, was used. Among other functionalities, it provides the definition and checking capabilities for constraints on ecore meta-models. The GMF (Graphical Modeling Framework) [63] was used to create an editor for the CFR language. GMF is based on EMF and provides functionalities for creating graphical editors for ecore-models. On the basis of an existing meta-model the following needs to be defined:

[1] http://www.openarchitectureware.org

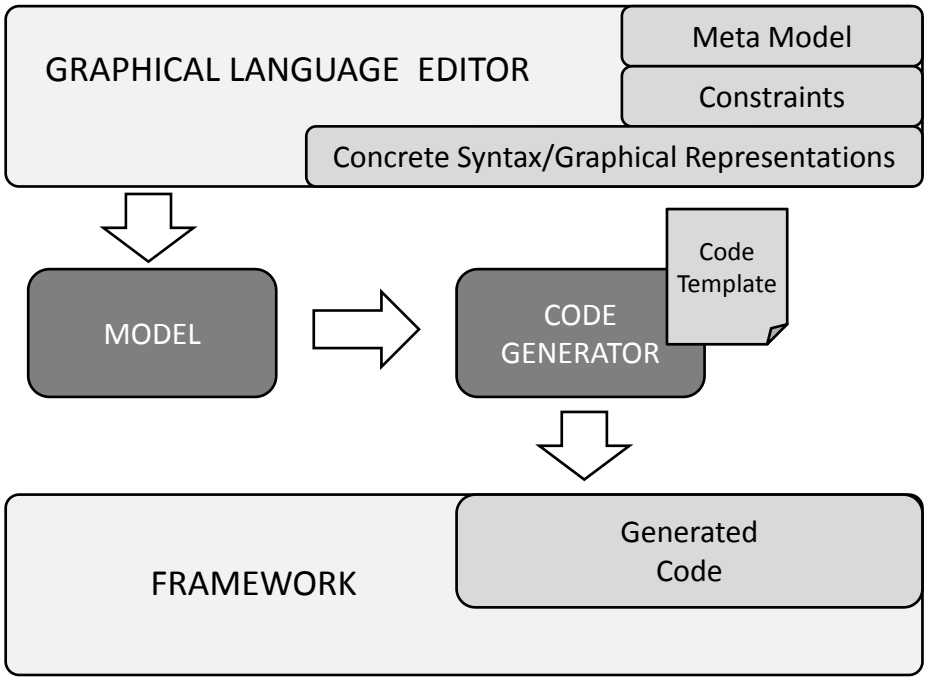


Figure 7.8: CFR Implementation Overview

- Graphical representations for the primitives of the language (concrete syntax)
- A toolbox for using graphical representations in an editor.
- The relationships between toolbox elements and graphical representations of the meta-model elements.
- Settings for language-specific functionalities of the editor.

GMF also provides code generation functionalities. Based on the meta-model and the graphical definitions and settings, it creates Eclipse plugins that can directly be used as a graphical editor for models. Figure 7.9 is a screenshot of the graphical editor prototype that provides "drag&drop" editing of CFR models. A double click on a processing element displays nested definitions which allows navigating through different abstraction levels in the model or to open a rule and edit its logic. The back-end of the implementation is defined using oAW code templates that produce executable analysers from models based on the underlying meta-model. The generator traverses the tree and instantiates code templates associated with the node types. For instance, the initialization of all channels is defined by the following template:

```
<<DEFINE main FOR Model>>
      <<FOREACH procUnits.typeSelect(Channel) AS c>>
```

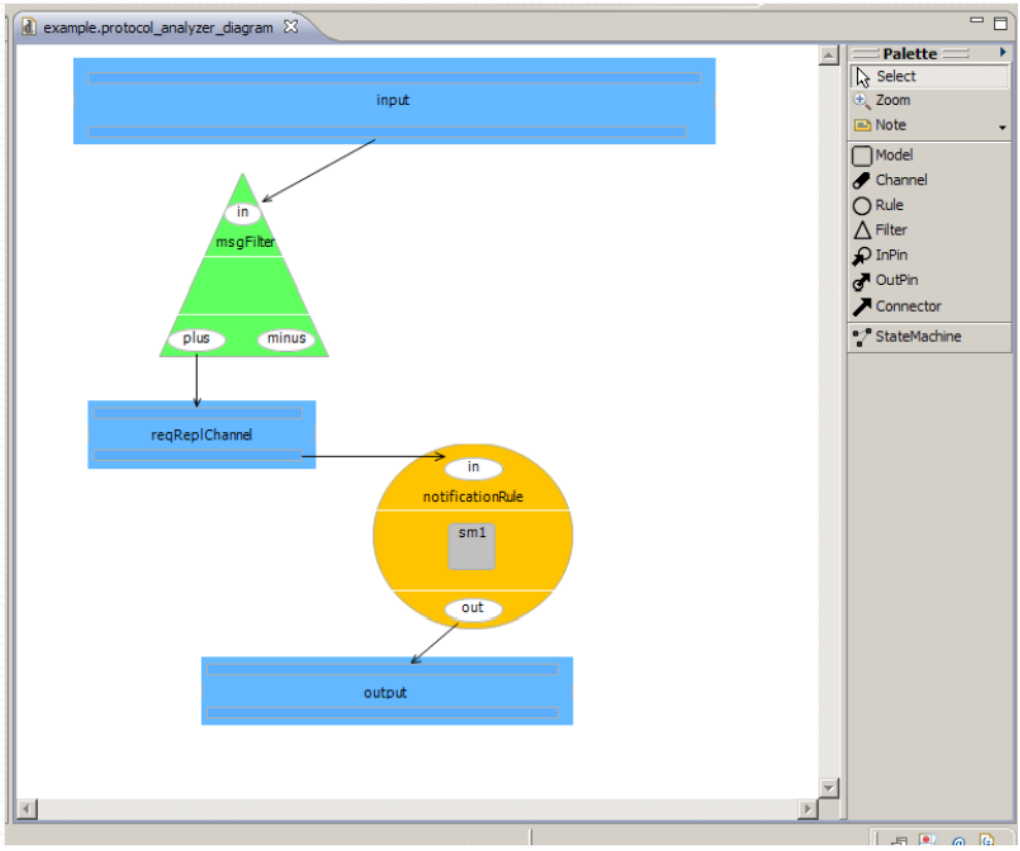


Figure 7.9: Screenshot of the Editor Prototype

```
            <<c.name>> = Channel("<<c.name>>")
    <<ENDFOREACH>>
<<ENDDEFINE>>
```

In the context of a model all processing units are checked for being channels and if they are, a *Channel* object is initialized with the name of the channel. For the example template, the generated source code is configuration code for a Python-based experimentation framework [112]. A second target platform is a framework that reads messages from a MOST network using a commercial protocol analyser [118].

## 7.7 Summary and Conclusions

This chapter presented a novel approach that extends the scope of protocol analysers to include application level communication abstractions and complex scenarios. The approach is implemented by a Domain Specific Language (DSL). The basic idea is to describe the mapping between two adjacent protocol layers bottom up using three basic concepts: channels, filters and rules. The resulting models do not only specify a protocol by relating

its messages to a lower layer but also serve as instructions to a protocol analyser on how to perform the actual mapping. Both concrete and abstract layers are represented as channels in the formalism which means that an abstract layer can always be used to define a yet higher layer.

The DSL presented in this chapter provides an example of how the pattern language can be embedded into another language. CFR provides visual means for connecting and abstracting processing units that are either defined using the pattern language or using a state diagram language. When used with pattern expressions only, the DSL becomes a visual means for pattern combination and pattern abstraction. In other words, the language becomes a visual pattern "programming" system for analysers. Further research could explore a generalisation of this approach to arbitrary application domains – not only analysis – and arbitrary forms of pattern combination – not only vertical combination.

The idea of abstract protocols is closely related to that of views. Views introduce a separate layer for defining a computation process while abstract protocols introduce a separate layer for analysing communication processes. Both layers may be based on their own language for describing computation or communication. The main difference is that abstract protocols may be one-way mappings that do not only restructure but also remove data through filtering. The abstract rule example of Section 7.4.3 suggests a generalisation of the abstract protocol idea for pattern matching. By vertically combining a pattern with a filter, the aspect of interleaved data is separated from the aspect of defining the pattern sequence.

# Chapter 8

# Conclusions and Future Research

The hypothesis underlying this work is that the systematic creation and layering of languages can be reduced to the elementary operations of pattern matching and instantiation and that this pattern-based approach provides a formal and practical foundation for language-driven modelling, programming and analysis. The research approach to support this hypothesis was to define an application-neutral pattern formalism to serve as a foundation and to apply this formalism to three application areas. The previous chapters have presented the pattern formalism and its applications in detail and each chapter provided preliminary conclusions. In this chapter, the work performed is reviewed, limitations are pointed out, final conclusions are drawn with respect to the hypothesis and opportunities for future research are outlined.

## 8.1 Review of Key Pattern Concepts

The pattern formalism of Chapter 3 forms the foundation for the entire work. Therefore, the review of the research starts with a critical reflection of the defining concepts underlying the formalism. The impact of the concepts can be regarded from a standpoint of feature interaction within the formalism, i.e., how core concepts fit together, and from the viewpoint of the application part of the research, i.e., how the core concepts contribute to the goal of building a foundation for LDSE.

**Matching and instantiation semantics** With matching and instantiation, the pattern formalism defines two separate semantics for pattern expressions. This design choice reflects a symmetry underlying the pattern concept: a pattern can be used for the recognition of existing structures as well as for the creation of new ones. Transformations embody this symmetry as, descriptionally, they consist of two pattern expressions based on the same

language that are merely interpreted differently. With elliptical matching, the core pattern operators for atoms, variables, sequencing, repetition, choice, quasiquotation and hierarchy all have matching and instantiation semantics and, therefore, present a significant subset of the pattern language that can be interpreted both ways. In addition to that, partial instantiation provides a dynamic mechanism for the creation of patterns based on the current bindings in the store. XMF utilises (partial) instantiation to implement a query and constraint language. Moreover, in XMF's schema language XPLT, patterns are defined independently of their interpretation and used as a basis for recognition, creation and bidirectional transformation of documents.

**Typed sequences** Typed sequences are the universal data language underlying the pattern formalism. Apart from matching or instantiating the type part, the role of the hierarchical typed sequence operator is to descent one level into a nested structure on both pattern and data level and to match or instantiate the pattern-level content with the data level content. This matching is naturally expressed with the sequential operator. Matching or instantiating nested structures is then an interplay between the sequential operator that operates within a single nesting level and the hierarchical operator that descents into structures. The result is a pattern formalism capable of matching and instantiating arbitrary tree structured data.

While typed sequences appear verbose at first, Section 4.5 defines a view mechanism that allows to "break out" of the syntactic framework predefined by typed sequences. Concat uses the view mechanism consequently. In principle, everything in Concat has to be expressed using typed sequence notation. However, by extending the (un-)parsers for program and meta-level, notational alternatives can be defined and arbitrarily mixed with the default typed sequence notation. The user controls the amount of internal representation to expose. In the implementation of this mechanism, the type identifiers of typed sequences play an important role as they serve as an explicit declaration on how to render the content part and as a dispatch for transformations.

**Transformations are patterns** The pattern formalism defines transformations as patterns that posses a matching semantics. A case could be made that the inclusion of transformations stretches the pattern concept. However, the fact that transformations consist of pattern expressions, that they interact naturally with other operators of the formalism and their importance for formalising core concepts for LDSE makes a convincing case for the validity of this design decision. Rewriting, grammars and pattern abstraction are all formalised through sets of individual transformations that are combined with pattern operators controlling their application. Computation is expressed by matching a compute

pattern with a program, reference resolution is expressed by matching a resolution pattern with the reference and parsing is expressed by matching a grammar pattern with with parser references.

Nevertheless, XMF demonstrates an alternative approach to handling transformations. While utilising the general matching and instantiation semantics of the pattern formalism its pattern language does not explicitly contain the transformation concept. Instead, a JavaScript API exposes a transformation function expecting two patterns and the data to be transformed as parameters. While this approach is less self-contained, i.e., a host language has to be used, it is nonetheless practical because the application of transformations can be scripted and embedded in the user interface's event handling. This indicates that the usefulness of transformations as patterns depends on the purpose. When the goal is, as in the (meta-)programming part of this work, to define a self-contained system that is able to express various forms of computations through patterns, "first class" transformations are crucial.

**Vertical and diagonal composition** Vertical combination defines a staged processing of input where a pattern is matched with the result of a previous pattern. This form of combination is particularly useful in conjunction with transformations. The applications shown in this work range from the configuration of different stages of program execution in Concat (parsing, computation and unparsing) to the interconnection of processing units in CFR. Most profoundly, vertical combination is the key for expressing layering, as demonstrated by the view mechanism. Diagonal combination gives matching a stack semantics enabling concatenative operations to be expressed naturally through matching operations. The result is a particularly concise formalisation of concatenative rewriting systems.

**Arbitrary meta-levels** The pattern formalism defines means to match and instantiate patterns with patterns. To make such meta-manipulation possible, the pattern core defines an advanced form of quasiquotation that (1) provides matching *and* instantiation functionality and (2) allows manipulation of arbitrarily quasiquoted patterns. The result is a significant increase in expressive power. Using this mechanism, the pattern language cannot only manipulate its own pattern expression but also the pattern expressions which, in turn, manipulate those pattern expressions, thus allowing arbitrary layers of meta-functionality.

Several key concepts of this work are based on meta-patterns. Meta-transformations define grammars, pattern abstraction and reference resolution. Meta-meta transformations can be utilised to manipulate the definitions of abstractions. For example, the view abstraction in Section 4.5.3 is defined as a meta-meta-transformation that given internal-

isation and externalisation patterns creates a meta-transformation abstracting the view. This meta-transformation can be parameterised by a computation pattern to yield a computation to be performed on the view. Meta-meta transformations can play an even more profound role, as suggested by Concat: if pattern-based grammars and pattern-based computations define the syntax and semantics of a language, meta-meta transformations can be used to configure and change the language definition. The result is an utmost degree of self-description. Regarding the goal of building a foundation for language-driven software engineering, the meta-facilities are not merely an extension but a key concept of this work.

**Conclusions** The concepts of the formalism can be separated into two categories. Basic pattern expressions, e.g., atomic values and variables, and the horizontal and hierarchical combinators form a sub-language implementing functionality associated with recognition and creation of static structures. Transformations as well as vertical and diagonal combinators extend the language with concepts that are computational in nature, e.g., mapping inputs to a result and passing the result on to the next process. Orthogonal to these categories, meta-patterns provide the means for manipulation of patterns with patterns. The power of the formalism lies in the various ways in which these concepts can be combined and applied to the flexible data representation underlying it. The result is a powerful system capable of expressing structural and computational aspects in a unified way. The applicability of the system was demonstrated in Chapters 4 to 7.

## 8.2 Language Creation and Layering

This section reviews the work from a methodological point of view by discussing how the "creation and layering" of languages is facilitated and utilised in the theoretical and practical part of the research.

**Role of language in the three tools** Creating and relating languages is the principle underlying the three tools presented in this work. In Concat, programming languages are created by defining their syntax and semantics through views, operations and macros. In principle, Concat is a direct implementation of the pattern formalism and the parsing, computing and view mechanisms defined in Chapter 4. XMF allows the creation of modelling languages through schemas, constraints and graphical views. Although XMF is less self-contained than Concat, as discussed above, it nevertheless defines a pattern-based meta-architecture in which all relationships are expressed through pattern matching,

instantiation, transformation, partial instantiation and path polymorphic traversal. The abstract protocols underlying CFR capture the notion of language in a communication network. Abstract protocols are created by specifying message formats and abstraction of communication patterns on a lower protocol layers. At first, the concepts of channels, filters and rules seem different from the concepts of the pattern core. However, the approach is compositional in the sense that a CFR model defines a configuration of processing units. The data passing semantics between the processing units is captured by vertical combination. Therefore, if the pattern language is utilised to define message formats and to specify primitive filters and rules, the CFR language becomes a means of classifying and visually combining pattern expressions.

**Enforcing language semantics and methodology by restriction** A recurring theme in this work is restriction. Rules of a rewriting system are restricted so that transformations behave like functions, the scope of views is restricted to a single typed sequence, the scope of CFR filters is restricted to a single message and CFR filters and rules must not be connected directly. Structurally, the access to the internals of a data type can be restricted by enforcing the use of concrete syntax. On the system level, temporal views restrict access to only selected steps of a computational process. Restricting access to internal structures and processes is a standard technique in software engineering: information hiding. Restricting expressive power of an existing formalism to enforce methodology or to maintain certain behavioural properties of a system is less common [135]. Concat demonstrates how the combination of restrictions and syntactic abstraction lead to a form of pattern-based computing where users have access to parts of the pattern functionality but are encouraged to think about the execution of a system in terms of domain concepts rather than pattern matching and transformations. Internally, the domain concepts of Concat, namely views, internalisation macros, operations, computation macros, and externalisation macros, are transformed into a single rewriting system defined by a single pattern.

**Language layering** The introduction chapter presented a methodological framework for building language-driven systems through layers of languages. A formal definition of language layering in terms of behavioural refinement was given. The basic idea is that a computational process can be decomposed into internalisation, computation and externalisation. The internalisation and externalisation define a mapping of the input and output of a computation to and from a lower layer. The computation defines a computational process on the lower layer. From a "black box" perspective, the lower layer is invisible which creates the "illusion" of a high-level computation. If the actual implementation is exposed,

the system provides different layers for reasoning about its behaviour. Each layer defines its own language according to which computation can be described. Language layering was abstractly defined in Section 1.3 and explored in the application part of the research.

In Section 4.5, it was demonstrated that using meta-transformations and vertical composition operators a view abstraction can be defined that embodies the idea of relating two layers. Arbitrary stacks of layers can be defined by nesting views. Concat utilises views to implement syntax and semantics of languages, as shown in the context of the SKI calculus implementation. It also demonstrates an important principle underlying the methodological framework: the languages used to relate different language layers can themselves be subject to layering. In Concat, internalisation and externalisation are described by views and computations are described by operations. While from the viewpoint of a user these concepts have their own syntax and semantics, they are internally mapped to pattern-based transformations of a rewriting system, i.e., they are mapped to the pattern language. Interestingly, the internalisation process performing this mapping is itself described using the pattern language as shown in Section 5.5.2. Concat is a demonstration of a system that combines the methodological framework of language layering and the pattern approach and takes both to the extreme. This is also expressed in the fact that the lowest computational process for both meta- and program level in Concat is an interpreter for the pattern language and that this interpreter is defined in itself.

**Conclusions** The tools presented in this work are all language-driven as they are inherently concerned with creating and relating languages. The underlying technical basis is pattern matching and instantiation. The methodological approach is defined by language layering. The versatility and usefulness of language layering as a methodological framework lies in the fact that (1) decomposition can be applied recursively to create an arbitrary number of layers and (2) the interpretation of the descriptions of internalisation, computation and externalisation can themselves be described using a layered approach. Pattern-based computing is based on the principles of “everything is visible” and “everything is allowed”. Creating languages with syntax, execution models and certain methodologies can be expressed by a combination of restricting visibility and expressive power and introducing a syntactic layer.

## 8.3 Limitations

**Appeal to Mainstream Software Developers** The most fundamental possible limitation of the work lies in its appeal to mainstream software developers. Many of the technologies that inspired this work, e.g., program transforamtion systems, Lisp, Forth or

Prolog have been around for decades but have not been widely used in the mainstream programming world. After all, these systems require a way of thinking about software development that greatly differs from that of the mainstream languages such as C and Java. This makes it difficult for software developers to switch and utilise these technologies. Likewise, Concat, XMF and CFR are conceptually very different from traditional programming, modelling and analysis. However, in recent years meta-modelling and meta-programming techniques have begun to penetrate mainstream software development and have found their way into more popular programming languages. In the context of these developments the future appeal of this research seems promising.

**Efficiency of Concat** A technical limitation of the work presented in this thesis is the efficiency of the pattern system implementation. The current Concat implementation follows the rules of Chapter 3 closely. The result is an operational system with a sound formal foundation, but not a an efficient one. As described in Section 8.5, future work is required to make Concat sufficiently fast for real world applications.

## 8.4 Final Evaluation of the Hypothesis

The hypothesis underlying this work consists of two parts. The first part requires showing that creating and layering languages can be reduced to matching and instantiating patterns. By utilising the matching and instantiation semantics of Chapter 3, Chapters 4 and 5 defined and implemented fundamental concepts for creating languages including parsing, rewriting and staged processing that clearly supported the validity of the hypothesis for programming. Likewise, the chapters on XMF and CFR demonstrated the validity for modelling and analysis.

As discussed in the introduction chapter of this work, proving the second part of the hypothesis is hardly possible as there is no agreed understanding on what a formal and practical basis for LDSE should look like. However, the previous chapters have attempted to support this part of the thesis by (1) defining a sound foundation for the methodological and technical aspect of the work and (2) applying this foundation to the key domains relevant for LDSE. The methodological framework was defined based on the well established principle of behavioural refinement and formally introduced a precise notion of language layering. The technical framework of the research, the core pattern formalism, was rigorously defined using a formal operational semantics. Based on the rigour of definitions and the wide range of applicability to language engineering, it can be concluded that the work provides a formal foundation for language-driven software engineering.

The most difficult part to show is the practicality of the approach. On the one hand, patterns are an intuitive concept and the three frameworks apply patterns to problems relevant in practice. On the other hand, the discussed domains are complex and it is difficult to show the general practicality of the approach in a single research project. To conclude, both the theoretical and practical part of the research have generated significant evidence to support the hypothesis. To further support the practicality of the approach, additional research is required.

## 8.5 Future Research

The pattern formalism and the existing tools provide a starting point for further research. Concat's combination of concatenative programming and pattern matching defines a new programming paradigm. One possible direction to explore this paradigm is the investigation of type inference for operations. Existing research on typing stack language [154] might be applicable but ways have to be found to deal with the dual nature of quotations representing both data and programs. Furthermore, it would be interesting to determine a minimal set of concatenative combinators [99] that can efficiently implement matching and instantiation for this would allow to bootstrap Concat based on a particularly small kernel. Currently, research on supporting "first class" continuations and parallel execution in Concat is ongoing and produced promising preliminary results that require further substantiation [125].

As discussed in Chapter 6, a combination of Concat's programming facilities with the visual view mechanism of XMF to form an integrated environment for designing visual programming languages is conceivable. Bidirectional transformations in XMF are performed by matching and instantiating the underlying schemas. This entails that two schemas not defining a bidirectional transformation cause a runtime error. Further research could investigate if bidirectionality of a transformation defined by two schemas can be determined by static analysis of pattern expressions. A different and equally interesting direction would follow the approach in [53] and try to define a subset of the pattern formalism for which bidirectionality is guaranteed.

Underlying the CFR approach is the idea of analysing communication by abstracting communication patterns through messages on an abstract protocol layer. As discussed in Chapter 7, while the basic approach is "bottom-up", there are various applications for "top-down" mappings that could be further explored, e.g., specification of message generators and triggering of test scenarios. Another direction of possible research is the application of machine learning techniques for the automatic configuration of filters and rules for complex scenarios. The result could be a valuable technique for re-engineering

underspecified systems. Although the resulting models might not define a complete specification of all possible scenarios between communication partners, the documentation of valid scenarios is already highly valuable. In the context of CFR, abstract rules operating on preprocessed input data hint at a more general mechanism for pattern matching on views. Further research could investigate this mechanism and its applications to separation of concerns in pattern definitions.

A different area of further research is concerned with efficiency of the pattern system. Several systems for parsing and pattern matching support techniques for memoization [11]. These techniques are based on the assumption that an input is consumed in a linear manner and are thus applicable to the horizontal operators in this work. Further research in this direction could investigate memoization techniques for vertically and diagonally combined pattern expressions.

# Bibliography


[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[2] Adobe Systems Inc. *PostScript Language Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.

[4] Jakob Axelsson et al. Correlating Bussines Needs and Network Architectures in Automotive Applications - A Comparative Case Study. In *Proceedings of the 5th IFAC Int. Conference on Fieldbus Systems and Their Applications (FET)*, pages 219–228. Elsevier, 2004.

[5] Franz Baader and Tobias Nipkow. *Term Rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[6] J. Bachrach and K. Playford. D-expressions: Lisp Power, Dylan Style. `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf`, 1999. Retrieved 21 January 2011.

[7] John Backus. Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[8] Henry G. Baker. Pragmatic Parsing in Common Lisp. *SIGPLAN Lisp Pointers*, IV:3–15, April 1991.

[9] Jay Barry. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.

[10] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Ralph Becket and Zoltan Somogyi. DCGs + Memoing = Packrat Parsing – But is it worth it? In Paul Hudak and David Warren, editors, *Practical Aspects of*

*Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2008.

[12] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[13] J Bézivin. On the Unification Power of Models. *Journal on Software and System Modeling, SOSYM 4(2):171-188*, 2005.

[14] Alexander Birman and Jeffrey D. Ullman. Parsing Algorithms with Backtrack. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory*, pages 153–174, Washington, DC, USA, 1970. IEEE Computer Society.

[15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[16] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems*. Monographs in Computer Science. Springer, 1998.

[17] CAN Specification Version 2.0. Technical Specification, Robert Bosch GmbH, 1991.

[18] J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova. Towards Verified Automotive Software. In *SEAS '05: Proceedings of the second international workshop on Software engineering for automotive systems*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[19] Ivan Bratko. *Prolog: Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2001.

[20] Martin Bravenboer, Eric Tanter, and Eelco Visser. Declarative, Formal, and Extensible Syntax Definition for AspectJ. A Case for Scannerless Generalized-LR Parsing. In William R. Cook, editor, *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*, pages 209–228, Portland, Oregon, USA, October 2006. ACM.

[21] Leo Brodie. *Thinking Forth*. Punchy Publishing, 3rd edition, 2004.

[22] Frederick P. Brooks, Jr. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356 –373, feb. 2007.

[24] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM.

[25] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement*. Springer, 2001.

[26] Jean Bézivin, Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev, and William Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, 2005.

[27] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[28] N. Chomsky. Three Models for the Description of Language. *Information Theory, IRE Transactions on*, 2(3):113 –124, sep. 1956.

[29] K. L. Clark. Negation as Failure. *Readings in nonmonotonic reasoning*, pages 311–325, 1987.

[30] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling - A Foundation for Language Driven Development*. Ceteva, 2008.

[31] Tony Clark and Jos Warmer, editors. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. Springer, London, UK, 2002.

[32] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61:190–210, August 2006.

[33] Haskell Curry and Robert Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[34] Krzysztof Czarnecki. Overview of Generative Software Development. In *Unconventional Programming Paradigms*, pages 326–341, 2004.

[35] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[36] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard F. Paige, editor, *International Conference on Model Transformation*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009.

[37] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer, 2003.

[38] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 285–296, New York, NY, USA, 2010. ACM.

[39] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[40] Nachum Dershowitz, Mitsuhiro Okada, and G. Sivakumar. Canonical Conditional Rewrite Systems. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 538–549. Springer, 1988.

[41] Arie Van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[42] Christopher Diggins. Cat: A Functional Stack-Based Little Language. *Dr. Dobb's Journal*, April, 2008.

[43] Sergey Dimitriev. Language Oriented Programming: The Next Programming Paradigm. *JetBrains onBoard Magazine*, 2, 2005.

[44] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp Symb. Comput.*, 5:295–326, December 1992.

[45] Daniel Ehrenberg, Slava Pestov, and Joe Groff. Factor: A Dynamic Stack-based Programming Language. In *Proceedings of the 6$^{th}$ symposium on Dynamic languages*, 2010.

[46] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Secaucus, NJ, USA, 2006.

[47] Andrew D. Eisenberg and Gregor Kiczales. Expressive Programs Through Presentation Extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA, 2007. ACM.

[48] Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. In *ECOOP 2007 - Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[49] Jean Marie Favre. Language Everyware! - Engineering the Tower of Babel through Cartography and Software Linguistics. In *TOWERS 2007, 1st International Workshop on Towers of Models*, 2007.

[50] Jean Marie Favre. Software Linguistics and Software Language Engineering. In *2nd International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE*, 2007.

[51] Matthias Felleisen. On the Expressive Power of Programming Languages. In *Science of Computer Programming*, pages 134–151. Springer, 1990.

[52] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[53] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. *SIGPLAN Notices*, 40:233–246, January 2005.

[54] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[55] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison Wesley, 2010.

[56] Richard Gabriel. On Sustaining Self. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 51–53. Springer, 2008.

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[58] Jeremy Gibbons. A Pointless Derivation of Radix Sort. *Journal of Functional Programing*, 9(3):339–346, 1999.

[59] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[60] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005.

[61] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[62] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[63] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1st edition, 2009.

[64] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer, 2007.

[65] James Hall. Multi-layer Network Monitoring and Analysis. Technical report, University of Cambridge, July 2003.

[66] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF. *SIGPLAN Notices*, 24:43–75, November 1989.

[67] Jack Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[68] Dominikus Herzberg. *Modeling Telecommunication Systems: From Standards to System Architectures*. PhD thesis, RWTH Aachen University, 2003.

[69] Dominikus Herzberg and Manfred Broy. Modeling Layered Distributed Communication Systems. *Formal Aspects of Computing, 28(4):751-763*, May 2005.

[70] Dominikus Herzberg and Tim Reichert. Concatenative Programming - An Overlooked Paradigm in Functional Programming. In *ICSOFT 2009 - Proceedings of the 4th International Conference on Software and Data Technologies, Volume 1, Sofia, Bulgaria, July 26-29, 2009*, pages 257–263. INSTICC Press, 2009.

[71] Dominikus Herzberg and Tim Reichert. Software Engineering for Telecommunication Systems. In *Benjamin W. Wah et al. (Eds): Encyclopedia of Computer Science and Engineering*. Wiley, 2009.

[72] Dominikus Herzberg, Tim Reichert, and Nick Rossiter. Towards Modeling Language Interoperability – Getting Meta-Level Architectures Right. *Forschungsbericht der Hochschule Heilbronn 2008/2009*, 2008.

[73] Dominikus Herzberg and Lars von Wedel. Erweiterungsmechanismen der UML. *OBJEKTspektrum*, pages 56–59, Juli/August (4) 1999.

[74] Mark Hills and Grigore Rosu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In Franz Baader, editor, *Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 246–256. Springer, 2007.

[75] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2008.

[76] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology (JOT) 7(3):125-151*, 2008.

[77] Charles Antony Richard Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[78] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *J. ACM*, 29:68–95, January 1982.

[79] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[80] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, January 2005.

[81] Freeman Huang, Barry Jay, and David Skillicorn. Programming with Heterogeneous Structures: Manipulating XML Data using Bondi. In Vladimir Estivill-Castro and Gillian Dobbie, editors, *29th Australasian Computer Science Conference (ACSC2006)*, volume 48(1) of *Australian Computer Science Communications*, pages 287–296, 2006.

[82] Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[83] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8 (4):437–444, 1998.

[84] Dan Ingalls. Design Principles Behind Smalltalk. *BYTE Magazine*, August 1981.

[85] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM.

[86] *Information Technology – Syntactic Metalanguage – Extended BNF*. International Organization for Standardization, ISO/IEC 14977:1996, December 1996.

[87] *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ITU-T Recommendation X.200, International Telecommunication Union, July 1994.

[88] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[89] C. Barry Jay and Delia Kesner. Pure Pattern Calculus. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 2006.

[90] Mario Jeckle, Stefan Queins, Barbara Zengler, Chris Rupp, and Jürgen Hahn. *UML 2 Glasklar: Praxiswissen für die UML-Modellierung und -Zertifizierung*. Hanser, 2nd edition, 2005.

[91] C B Jones. Formal Development of Correct Algorithms: An Example based on Earley's Recogniser. In *Proceedings of ACM conference on Proving assertions about programs*, pages 150–169, New York, NY, USA, 1972. ACM.

[92] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do Code Clones Matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[93] Wolfram Kahl. Basic Pattern Matching Calculi: A Fresh View on Matching Failure. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 1–5. Springer, 2004.

[94] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

[95] Alan Kay. *The Reactive Engine*. PhD thesis, University of Utah, Department of Computer Science, Salt Lake City, 1969.

[96] Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumarta, and Andreas Raab. Steps Toward The Reinvention of Programming: A Compact and Practical Model of Personal Computing as a Self-Exploratorium (Proposal to NSF). VPRI Research Note RN-2006-002, August 2006.

[97] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[98] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised$^5$ Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[99] Brent Kerby. The Theory of Concatenative Combinators. `http://tunes.org/~iepos/joy.html`, 2002. Retrieved 21 January 2011.

[100] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[101] Gregor Kiczales. It's Not Metaprogramming. *Dr. Dobb's Journal*, November 2004.

[102] Gregor Kiczales et al. Aspect-Oriented Programming. *Proceedings of ECOOP*, 1997.

[103] Gregor Kiczales, Jim Des Rivieres, and Bobrow Daniel G. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[104] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1st edition, 2008.

[105] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodolgy*, 2:176–201, April 1993.

[106] Donald E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1986.

[107] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[108] Alain Laville. Lazy Pattern Matching in the ML Language. In *Proc. of the seventh conference on Foundations of software technology and theoretical computer science*, pages 400–419, London, UK, 1987. Springer.

[109] Qin Ma and Luc Maranget. Algebraic Pattern Matching in Join Calculus. *Logical Methods in Computer Science*, 4(1), 2008.

[110] Ronald Mak. *Writing Compilers and Interpreters: A Software Engineering Approach*. Wiley Publishing, 2009.

[111] David Editor Margolies. *The ANSI Common Lisp Reference Book*. APress, 2008.

[112] Alex Martelli. *Python in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2nd edition, July 2006.

[113] S. Mauw, W. Wiersma, and T. Willemse. Language Driven System Design. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 280.2, Washington, DC, USA, 2002. IEEE Computer Society.

[114] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[115] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[116] M. Douglas McIlroy. Macro Instruction Extensions of Compiler Languages. *Communications of the ACM*, 3(4):214–220, 1960.

[117] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[118] Ansgar Meroth and Dominikus Herzberg. An Open Approach to Protocol Analysis and Simulation for Automotive Applications. In *Embedded World Conference*, 2007.

[119] Ansgar Meroth and Boris Tolg. *Infotainmentsysteme im Kraftfahrzeug: Grundlagen, Komponenten, Systeme und Anwendungen*. Vieweg, Wiesbaden, 2008.

[120] Bruce Mills. *Practical Formal Software Engineering: Wanting the Software You Get*. Cambridge University Press, 2009.

[121] Robin Milner. An Algebraic Definition of Simulation between Programs. In *2nd International Joint Conference on Artificial Intelligence*, pages 481–489. Kaufmann, 1971.

[122] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[123] John C. Mitchell and Krzysztof Apt. *Concepts in Programming Languages*. Cambridge University Press, New York, NY, USA, 2001.

[124] MOST Specification Rev. 3.0. Technical Specification, MOST Cooperation, June 2010.

[125] Aaron Müller and Florian Eitel. Funktionale Meta-Programmierung: Umsetzung eines Konkatenativen Programmiersystems, Bachelor Thesis, Department of Software Engineering, Hochschule Heilbronn, Heilbronn, Germany, July 2010.

[126] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages using Formal Language Theory. *ACM Trans. Internet Technol.*, 5:660–704, November 2005.

[127] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[128] OCL 2.0 OMG Final Adopted Specification. Technical Specification, OMG, October 2003.

[129] Meta Object Facility (MOF) Core Specification, v2.0. Technical Specification, Object Management Group (OMG), January 2006.

[130] Unified Modeling Language: Infrastructure, Version 2.3. Technical Specification, Object Management Group (OMG), May 2010.

[131] Unified Modeling Language: Superstructure, Version 2.3. Technical Specification, Object Management Group (OMG), May 2010.

[132] Angela Orebaugh, Gregor Morris, and Ed Warnicke Gilbert Ramirez. *Ethereal Packet Sniffing*. Syngress, February 2004.

[133] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[134] Terence Parr. *Language Implementation Patterns*. Pragmatic Bookshelf, 1st edition, 2009.

[135] Terence John Parr. Enforcing Strict Model-View Separation in Template Engines. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA, 2004. ACM.

[136] Ian Piumarta and Alessandro Warth. Self-Sustaining Systems. chapter Open, Extensible Object Models, pages 1–30. Springer, Berlin, Heidelberg, 2008.

[137] Ian Piumatra. PEG-based Transformer provides Front-, Middle- and Back-End Stages in a Simple Compiler. Technical Report TR-2010-003, Viewpoints Research Institute, 2010.

[138] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[139] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[140] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The Evolution of Forth. *History of programming languages*, II:625–670, 1996.

[141] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[142] Tim Reichert and Dominikus Herzberg. A Domain Specific Language for Uncovering Abstract Protocols and Testing Message Scenarios. In *Proceedings of Software Engineering 2008 (Workshops)*, pages 427–430, 2008.

[143] Tim Reichert and Dominikus Herzberg. Teaching Language-Driven Software Engineering. In *International Conference of Education, Research and Innovation (ICERI)*, Madrid, Spain, November 2009.

[144] Tim Reichert, Edmund Klaus, Wolfgang Schoch, Ansgar Meroth, and Dominikus Herzberg. A Language for Advanced Protocol Analysis in Automotive Networks. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 593–602, New York, NY, USA, 2008. ACM.

[145] Peter H. Salus. *Little Languages and Tools*. Macmillan Technical Publishing, 1998.

[146] D. V. Schorre. META II: A Syntax-oriented Compiler Writing Language. In *Proceedings of the 19th ACM national conference*, ACM '64, pages 41.301–41.3011, New York, NY, USA, 1964. ACM.

[147] Alex Sellink and Chris Verhoef. Native Patterns. In *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society Press, 1998.

[148] B. Sheil. Environments for Exploratory Programming. *Datamation*, February 1983.

[149] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 451–464, New York, NY, USA, 2006. ACM.

[150] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29 (1-3):17–64, 1996.

[151] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[152] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[153] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 2nd edition, 1994.

[154] Bill Stoddart and Peter J. Knaggs. Type Inference in Stack based Languages. *Formal Aspects of Computing*, 5:289–298, 1993.

[155] Walid Taha and Patricia Johann. Staged Notational Definitions. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 97–116, New York, NY, USA, 2003. Springer.

[156] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 4th edition, 2003.

[157] The Eclipse Foundation. Eclipse IDE. http://www.eclipse.org, 2010.

[158] Ken Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11:419–422, June 1968.

[159] Paul Sammut James Willans Tony Clark, Andy Evans. An eXecutable Metamodelling Facility for Domain Specific Language Design. In *The 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[160] Laurence Tratt. Compile-time Meta-Programming in a Dynamically Typed OO Language. In *Proceedings Dynamic Languages Symposium*, pages 49–64, October 2005.

[161] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *SIGPLAN Notices*, 22:227–242, December 1987.

[162] David Ungar, Adam Spitz, and Alex Ausch. Constructing a Metacircular Virtual machine in an Exploratory Programming Environment. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 11–20, New York, NY, USA, 2005. ACM.

[163] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[164] Eelco Visser. Scannerless Generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[165] Eelco Visser. Meta-Programming with Concrete Object Syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer.

[166] Eelco Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[167] Joost Visser. Matching Objects Without Language Extension. *Journal of Object Technology*, 5(8):81–100, November-December 2006.

[168] Eric van der Vlist. *RELAX NG*. O'Reilly Media, Inc., 2003.

[169] Manfred von Thun. Joy: Forth's Functional Cousin. In *Proceedings of the 17th EuroForth Conference*, 2001.

[170] Manfred von Thun. A Rewriting System for Joy. `http://www.latrobe.edu.au/philosophy/phimvt/joy/j07rrs.html`, 2007. Retrieved 21 January 2011.

[171] Document Object Model (DOM) Level 1 Specification, Version 1.0. Technical Specification, World Wide Web Consortium (W3C), October 1998.

[172] XSL Transformations (XSLT), Version 1.0. Technical Specification, World Wide Web Consortium (W3C), May 1999.

[173] Extensible Markup Language (XML), Version 1.1. Technical Specification, World Wide Web Consortium (W3C), April 2004.

[174] XQuery 1.0: An XML Query Language. Technical Specification, World Wide Web Consortium (W3C), January 2007.

[175] P. Wadler. Views: A way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM.

[176] Philip Wadler. How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 1985.

[177] M. P. Ward. Language Oriented Programming. *Software-Concepts and Tools*, 15:147–161, 1995.

[178] Alessandro Warth. Experimenting with Programming Languages. Technical Report TR-2008-003, Viewpoints Research Institute, 2008.

[179] Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat Parsers can Support Left Recursion. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 103–110, New York, NY, USA, 2008. ACM.

[180] Alessandro Warth and Ian Piumarta. OMeta: An Object-Oriented Language For Pattern Matching. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM.

[181] Ed Wilson. *Network Monitoring and Analysis*. Prentice Hall International, 2000.

[182] Gregory V. Wilson. Extensible Programming for the 21st Century. *ACM Queue*, 2(9):48–57, 2005.