Automated Construction of XSL-Templates

—

An Inductive Programming Approach

Diplomarbeit

im Studiengang Wirtschaftsinformatik

in der Fakultät Wirtschaftsinformatik

und Angewandte Informatik

der

Otto-Friedrich-Universität Bamberg

Verfasser:

Martin Hofmann

Referent:

Prof. Dr. Ute Schmid

Datum:

29. September 2007

Abstract

This diploma thesis bridges the gap between fundamental research in Inductive Program Synthesis and its practical application for end user programming. It demonstrates that it is indeed feasible to automatically generate XSLT stylesheets from a few examples which define the desired input/output behaviour, using the synthesis system for recursive functional programmes IGOR which is set up in a term rewriting framework. The generated XSLT stylesheets apply simple string functions on text nodes of XML documents. To provide the inductive synthesis system IGOR with appropriate I/O examples a prototypical system transforms the strings of an initial I/O pair into a list of substrings as underlying data structure. By recombining the substrings of the input strings, new possible input strings are generated, chosen and completed by the user, and transformed into a specification for IGOR. A parser finally transforms the synthesised functional program into an XSLT stylesheet.

Kurzfassung

Diese Diplomarbeit schlägt die Brücke zwischen Grundlagenforschung auf dem Gebiet der Induktiven Programmsynthese und ihrer praktischen Anwendung für die Endbenutzerprogrammierung. Sie zeigt, dass es durchaus möglich ist, aus wenigen, das Eingabe-/Ausgabeverhalten beschreibenden Bespielen automatisch XSLT Stylesheets zu generieren. Dazu wird das System IGOR verwendet, welches auf der Theorie des "term rewriting" basiert, um rekursive, funktionale Programme zu synthetisieren. Die generierten XSLT Stylesheets führen einfache Funktionen auf Strings in XML Dokumenten aus. Um IGOR geeignete Beispiele zur Verfügung zu stellen, wandelt ein prototypisches Programm die Strings eines initialen Eingabe/Ausgabe Paares in Listen von Substrings als zugrunde liegende Datenstruktur um. Durch Rekombination der Substrings der Eingabe werden weitere, neue Eingabestrings generiert. Der Benutzer wählt geeignete Eingabe Strings aus, vervollständigt diese und erstellt mit Hilfe des Prototypen eine Spezifikation für das System IGOR, welches ein funktionales Programm gemäß der übergebenen Spezifikation erstellt. Ein Parser überführt das funktionale Programm letztendlich in ein XSLT Stylesheet.

Still not knowing what it meant, we dealt with it in the traditional manner for handling very difficult problems. Namely, give them to graduate students and tell them they are easy.

Alan Cypher, Watch What I Do.

Contents

Chapter 1

Introduction

1.1 Thesis

XML, the eXtensible Markup Language, has gone from the latest buzzword to an entrenched eBusiness technology in record time. As HTML is the language of choice for presenting and publishing information online, XML is for structuring and representing data, wherever there is a demand for data interchange in any form. Close together with XML comes XSL and its dialect XSLT (eXtensible Stylesheet Language Transformation), by means of whose transformation and modification of data can be accomplished.

However, despite their eminent importance and omnipresence their support for end-user programming is almost inexistent. Contrary to HTML, where by means of graphical WYSIWYG editors everybody can create web pages, just following the principle "What You See Is What You Get", using XML/XSL is currently reserved to programmers with sound programming skills.

I believe that it is possible to remedy this lack by joining the pragmatic hands-on solutions of Programming by Example and the well-founded theoretical methods of Inductive Programming. This diploma thesis sketches possibilities to automatically generate recursive XSL transformations by collecting examples of user's desired transformation in a plug-in environment of a common programmer's text editor.

1.2 Motivation

XML is a platform-independent way to represent data. Simply put, XML enables you to create data that can be read by any application on any platform. You can even edit and create it by hand, because it is based on the same tag-based technology, i. e. SGML (Standard Generalised Markup Language), that underlies HTML. As a markup language, it permits the separation of text in some document from extra information about the text itself, i. e. information about the text's structure or presentation. The extra infor-

mation is expressed using markup which is intermingled with the primary text. So both, the text and the information about it, are accessible independently and can easily be used and processed by a machine. In the last years XML has been adopted in nearly any field where the interchange of structured information between computer applications is essential and evolved to the lingua franca by reason of its platform independence.

While XML concerns content and structure of information, by means of XSLT it is possible to put this information in any desired context, i. e. transform it into any designated format, rearrange or rewrite it. XSLT, itself written in XML, is a programming language to instruct an XSL interpreter to process XML documents ad libitum. Especially in the field of web design and web development this duo gained in importance by providing the means for customising, adjusting, and changing contents of web pages with little effort.

But also in many domains of eBusiness, Service Oriented Architectures, in any forms of GRID-computing, wherever Electronic Data Interchange is necessary or at Enterprise Application Integration of legacy systems XML/XSL plays a decisive part. So in the future, everybody who works with computers more intensively will come in touch with XML and will use it, work with it, and want to modify it in some way.

However, XSL was designed to increase the ease of processability by computer applications and not readability for human programmers. Writing XSL stylesheets, i. e. an XSL program, is very cumbersome and especially debugging is quite tedious and in particular for end-users almost impossible. By end-user one refers to a user of a computer application, a person who uses a computer as part of his/her daily life or work, but who is not a computer programmer and is not interested in computers per se. The aim of end-user programming is to enable those persons, who have not necessarily been taught how to write code in conventional programming languages, to write computer programs [4].

But why is programming considered as hard and something our average non-programming-user does not want to try? Halbert [10] identifies two obstacles to programming. The first is knowledge and education, the other is skill or talent. To be able to program, a programmer has to keep a mass of detail and idioms about a particular programming language in mind. As a translator has to have learnt the vocabulary and phrases of a foreign language, a programmer has to have learnt the syntax and language pattern of a programming language. Additionally, a programmer has to be able to keep the state of an intangible reality in mind and create plans in an abstract manner which results cannot be revised step-by-step but only as a whole after executing the program.

To reduce the amount of knowledge needed to program, it is necessary to let the user write programs in an environment of a system with which he is already familiar and enable him to use exactly the same operations, he already knows to work with this system. To compensate the lack of skill, it is necessary to help the user avoiding to learn a mass of new detail and help him to create programs more easily. Therefore, one

should let him build them incrementally, so he can test them as he creates them.

Although the readability of HTML is not significantly better than of XSLT, the support for end-user programming is. Different editors enable anybody who cannot or does not want to program directly in HTML to publish online. In WYSIWYG text editors anybody can program, create, and edit HTML pages, just following the principle "What You See Is What You Get" known from everyday's computer usage. So a user operates in a familiar environment and can see and check the result of his operations immediately. Despite its increasing importance, the end user support for XML/XSL is not so broad as it is for HTML. Obviously, to fill this gap there is a need to automatically generate XSL stylesheets to satisfy the desired needs "on the fly".

1.3 How This Could be Done

One possible approach to solve this problem would be the use of deductive methods. The problem itself and its context could be described by a set of axioms specifying all conditions which must hold for the final program, i. e. the desired XSL stylesheet. However, in this context we deal with unskilled or non-programmers and therefore deductive approaches seem to be inappropriate here, as specifying the axioms to describe the problem will be at least as difficult for them as writing the program straight away. So in this case it should be more convenient to synthesise a sufficiently complete and correct program with respect to some examples rather than to a formal specification. In the domain of Machine Learning, two subfields are concerned with such an issue, namely Programming by Demonstration and Inductive Programming. Both are pursuing inductive approaches and are therefore suitable for my problem.

In analogy to HTML and WYSIWIG editors, I propose therefore a WYDIWIYG ("What You Do Is What You Get") editor for XML/XSLT, where the final program the user gets is a realisation of modifications and actions the user performed in a graphical user interface. Due to its layout, an XML document can easily be displayed as a tree-like diagram, where the textual information resides in the leaves. This property makes it possible to establish a "visible world" of XML by creating a user interface in which every operation on this document can be made visible. This, i. e. the visualisation of an intangible state, fulfils one of the main requirements of PbE [10]. Every modification concerning the structure the user could demonstrate by rearranging, cutting off, and adding branches to the tree. Modifications that should affect the textual information in the leaves the user has to demonstrate by appropriate input/output examples (I/O-examples) defining the leaf's content before and after the modification.

However, PbE lacks a comprehensive and stringent theory. Although many systems exhibit remarkable success in different domains, these solutions remain in most instances tailored to a specific problem domain and make intensive use of very domain-specific and specialised heuristics. Additionally, they incorporate only simple forms of

generalisation learning, but typically no or only highly problem-dependent methods for the induction of loops or recursion from samples or traces of repetitive commands [22].

Contrarily, Inductive Programming and especially its subbranch Inductive Program Synthesis are based on a well-founded theoretical basis [35, 1, 52]. Learning generalisations remains largely independent of a specific domain and induction techniques to detect loops and especially recursion are quite powerful. Nevertheless, inductive synthesis systems in particular lack usability, since the prevailing systems are still at syntactically too low a level and only usable for experts with machine learning background [30].

1.4 What I Will Do and What I Won't

For the reason that XSL could be seen as a functional programming language [12] where a template represents a function and recursive template calls are accountable for processing control, I put my program, in the following called ProXSLbE (**P**rogramming **XSL** by **E**xample), on the basis of the inductive synthesis system IGOR which induces recursive functional programs from well chosen examples by detecting syntactical regularities [15].

On top of IGOR a Programming by Example (PbE) component with a graphical user interface is built to assist the user in giving the appropriate examples. As already mentioned, Programming by Example components need to be attached to existing applications, with which the user is already familiar, but they also need access to the internal data of the application [4]. Therefore, my program is incorporated as a plugin in the open source programmer's text editor *jEdit*. Although, this could be any other text editor like *XEmacs* or *Eclipse*, its source code and API should be available.

However, the number of imaginable tasks a user could want to perform on an XML document is huge and therefore are the classes of the underlying programs, i. e. the XSL transformations, arbitrarily complex. Within the scope of my diploma thesis I therefore restrict myself to synthesise recursive functions over strings, particularly over lists of substrings. As a consequence, the stylesheets I will automatically generate can only process the string nodes in the leaves of an XML tree and can only apply functions which underlying data structure could be understood as a list of substrings.

The detection of structural changes in the XML tree and their mapping to input/output examples of a function would have been extremely ambiguous despite the use of tree comparison algorithms and would have gone far beyond the scope of this master thesis. I also do not cover the synthesis or functions processing numbers of functions which take more than one node of an XML tree as its input argument.

Chapter 2

Work by Others

Computers that can program themselves are an old dream of Artificial Intelligence. The Church-Turing thesis states that “every function which would naturally be regarded as computable can be computed by a Turing machine". So in relation to Machine Learning, a computer program, i. e. Turing machine, is the most powerful structure that can be learnt, pushing the final goal well beyond neural networks or decision trees. Two subfields of Machine Learning, namely Programming by Example and Inductive Programming, are currently rising to this challenge.

2.1 Programming by Demonstration

Programming by Example (PbE) or Programming by Demonstration[1] has a quite simple motivation: if a user knows how to perform a task on a computer, this should be enough to create a computer program to perform this task. So instead of directly coding a program by writing instructions in some programming language, it should be more convenient to “teach” the computer the behaviour you desire by just demonstrating the tasks the machine shall perform.

2.1.1 Examples of Programming by Example

The research field of PbE is quite unstructured and defines itself rather through a set of concrete incarnations of different systems than through stringent definitions and an unambiguous nomenclature. Therefore—I think—it will be easier for the reader to get a sense of PbE by first introducing some exemplary systems. However, this is not meant to be a state-of-the-art overview or a chronological description of PbE systems, nor a detailed introduction to the mentioned systems. The reader merely shall get known to

[1] Both terms are mostly used interchangeably in the literature, although slight differences can be identified. In the following, I will use Programming by Example for both terms.

aspects of PbE relevant for this thesis. For continuative information I refer to the specific papers.

2.1.1.1 EAGER

A very prominent example is the system EAGER by Allen Cypher [2, 3] which can program repetitive tasks by example in the Apple's organiser and email environment HyperCard. It constantly monitors the user's activities and when it detects an iterative pattern, it writes a program to complete the iteration. If for example a user browses and reads through today's incoming emails and wants to write a subject's list, he opens a new hyper card (HyperCard's metaphor for a memo), types "1." and copies the subject of the first email. Then he types "2." and copies the subject of the second email. At this point the EAGER icon pops up and suggests "3." as next text to be typed, since it has detected a repetitive task and anticipates that the user will type the third item in his list. The user confirms and copies the subject of the third email. Now again EAGER anticipates a repetitive task and suggests to write "4." and adds the subject of the fourth email. The user, now confident that EAGER has learnt the task clicks on the EAGER icon and completes the subjects list automatically for all mails.

2.1.1.2 SMALLSTAR

The SMALLSTAR system is another, albeit not very up-to-date, but all the more illustrative example [10, 9]. The system was attached to a reimplementation of Star, a general-purpose office system which supported all the usual office functionality as creating, moving, editing, and deleting files and folders, as well as printing, calculating and so on. An additional feature of SMALLSTAR was a programming language called *Cusp* (for Customer Programming) to enable the user to automate repetitive tasks by programming macros. In the small example following, 'CreditBalance' is the name of a field in a table.

IF CreditBalance > 0 **THEN**
  MOVE THE Document **WHOSE NAME IS** 'PleasePay' **TO**
    THE Printer **WHOSE NAME IS** 'Gutenberg'

To render PbE possible SMALLSTAR incorporates also a recording mechanism. Let us assume, the user wants to move all pdf-files in a folder named 'A' to a folder named 'B'. Therefore the user starts the recording utility by pressing *Start recording* and moves the file 'first.pdf' from 'A' to 'B' and then invokes *Stop Recording*. The result would be a similar *Cups* program as above. However, the user wanted to move all pdf-files and not only a single one. For this purpose SMALLSTAR maintains for every referenced object a so called *data description* which is editable by the user. In our case, the user can modify

the data description for the file in such a way, that all pdfs are moved, so he changes the name pattern from ‘first.pdf’ to ‘*.pdf’. The resulting program now moves all pdf-files from ‘A’ to ‘B’.

2.1.1.3 SMARTedit

A program that supports the idea of editing by example is the system SMARTedit [47] which learns simple text-processing tasks. Suppose for example, a user has written some HTML with comments (a string delineated by the tokens `<!--` and `-->`) embedded in it and wants to remove those before publishing. To delete such comments one could use for example the search-and-replace function of let’s say MicrosoftWord, by entering the regular expression `\\<\\!--*--\\>`. Avoiding such arcane syntax in SMARTedit the user again enters a macro recording mode and records an example of the desired task.

In our case he moves the cursor to the beginning of the first comment by either a sequence of cursor-motion keys or mouse clicks, and then deletes the entire comment. Finally he stops the recording. After recording, the user can invoke the recorded macro by pressing a *Step Through Macro* button. SMARTedit now tries to anticipate what action the user is likely to take next. In our example it correctly moves the cursor to the beginning of the next comment. However, if this is not correct, e. g. if his intent was to move the cursor to the end of the last word before a comment, the user can always tell SMARTedit to *Try Another Guess*.

After he verified that SMARTedit did the correct action, the user steps to the next macro action and SMARTedit predicts that he will delete the extent of the HTML comment. The action is visualised by striking the region through that is deleted. This again is correct and the user steps to the next action. Now SMARTedit has used the additional information from the last macro execution and correctly anticipates that the user wants to delete the complete next comment.

2.1.2 Important Aspects of PbE Systems

2.1.2.1 Demonstrating the Task

In so called *demonstrational interfaces*, the user can demonstrate the desired task by performing actions on concrete example objects (often by direct manipulation), while constructing an abstract program. Many researchers tried to identify core concepts and introduced several technical terms. Myers speaks of example-based programming [7] and uses “Programming by Example” and “Programming by Demonstration” synonymously [4], Halbert uses “programming in the user interface” [10], Finzer and Gould named their methodology “Programming by Rehearsal” [51] and Ben Shneiderman

coined the term “Programming by Direct Manipulation” [6]. The differences between these terms are marginal, however, one could look at this issue from two perspectives.

The first perspective emphasises the use of example data, which might represent more general concepts during the development of a program. This is for example the case with SMARTedit and also in my system. Its potential benefit is that concrete examples are easier for people to work with than the abstractions of programming languages.

The second perspective attaches importance to the demonstration of a task and could be described as programming in the user interface. There, the statements in the “program are the same as the commands the user would normally give the system”[sic][10]. A program is built by providing examples of the intended interactions between the user and the application via demonstration. Its potential benefit is that the user can compose the program from commands that are already familiar, both in terms of their functionality and their form. The distinction to the first is that the objects being manipulated might be the actual components of the program rather than examples that represent more general concepts. The SMALLSTAR system would be a representative of this class.

Both of these ideas have well-known problems. Example-based programming is fundamentally intractable because a set of concrete examples can imply infinitely many programs. Programming in the user interface tends towards programs that use interfaces designed for people. People and computers are different, and some programs are awkward from the user interface perspective.

2.1.2.2 Listen to User’s Actions

Another aspect of PbE, which also has to do with demonstrating a task, is when the task has to be demonstrated, i. e. when the system pays attention to the user. EAGER continually logs the user’s actions and examines them for repetitive actions. Only when the system has detected a repetitive task, it comes to the fore and offers its help. The pros and cons are obvious. If the program just pops in every second suggesting useless nonsense, this will be very annoying and the user will deactivate the program after few such blunders. The big advantage is that if the suggestions are correct it comes to the aid just in time. However, it is very problematic to confine the necessary context and to determine when a task is repetitive: once in a minute, once a day or a week?

For this purpose most PbE systems differentiate between a demonstration and an interaction session, as for example SMARTedit and SMALLSTAR. In the further the task is demonstrated and recorded and in the latter it is replayed and adapted by new demonstrations if necessary. Everytime the user has to explicitly start a demonstration session and no ad hoc help is possible, but now the context of the inference is always clear.

2.1.2.3 Guessing User's Intent

This brings us to another aspect of PbE, where the program must guess what the user actually wants to do. In the SMARTedit example mentioned above, the system has to guess to move the cursor at the beginning of the comment and not to the end of the last word before the comment and also to delete the whole comment and not 15 characters starting from cursor position. Some scientists claim, as e. g. Ben Shneiderman on a panel at the SIGCHI '91 conference, that this term is inappropriate, since computers execute deterministic algorithms [8]. However, I agree with Myers who says that it is indeed appropriate "for systems that use heuristics, since it appears to users that the system is guessing their intent."

At this point many systems use methods from Artificial Intelligence to correctly generalise over the provided examples and infer the user's intent. These range from simple hand-coded rules incorporated into the systems as in some of Brad Myers 'gems' (PERIDOT and TOURMALINE) [7, 4] to more sophisticated AI algorithms as for example decision trees (GAMUT [37]) or version space algebras in SMARTedit [47].

However, any system with heuristics will sometimes generate an incorrect result, as it depends on the quality and representativeness of the examples. Some systems as SMALLSTAR do not use any inference and therefore are never wrong, but have to rely on additional information the user provides, as for example through *data descriptions*.

2.2 Inductive Program Synthesis

Program Synthesis or Automatic Programming is the science of automatically creating programs. From the very beginning on, this has always been for the most part a deductive approach, meaning from the general to the specific. So the program is created from a general formal specification, such that the resulting specific program is complete and correct with respect to this specification. However, you can't make a silk purse out of a sow's ear, and so, a formal specification for a program has to be written by a programmer usually following some intended or actual list of specific behaviours, i. e. things the system should do or should not do. This is where Inductive Program Synthesis takes off from. Instead of using this set of behaviours for requirement specification and formal specification to generate a program from, Inductive Programming uses this set of special behaviours in the first place to synthesise a more general program. Figure 2.1 shall illustrate this.

Thus, Inductive Programming is the inference of an algorithm or program, starting from information that is known to be incomplete, called the evidence such as input/output examples. The inferred program must be correct with respect to the provided evidence—and not to a formal specification—in a generalising sense: it should be neither equal to it, nor inconsistent. Contrary to concept learning or classification, which

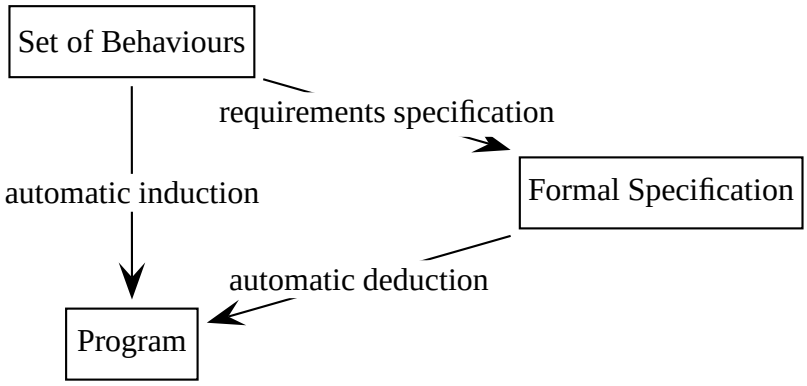

Figure 2.1: Two approaches to programming, inductive and deductive.
(in dependence on [11])

apply a similar inductive approach, it features recursive calls or repetition control structures and is therefore able to infer algorithms and programs.

Although automatic programming is the most important domain of inductive programming methods, it is not the only one. It is also applied in several other areas, such as learning recursive policies for state-based planning [50], the detection of repetitive structures in biology or chemistry [44], or the induction of recursive grammar rules for natural language parsing or translation [25].

2.2.1 Approaches to Inductive Programming

Inductive Programming, as well as any other learning system, is guided explicitly or implicitly by a language bias and a search bias [48]. A language bias is a restriction of the hypothesis language, i.e. the language of hypothesis or possible solutions, whereas a search bias restricts which and/or how many of all possible hypothesis are searched and which are favoured over others.

The inference may additionally draw on background knowledge or query an oracle. Learning cannot take place in empty space, as a learner always needs some form of prior knowledge. This additional information is called background knowledge and is usually provided, dependant on the learning system, in form of additional examples, facts, or functions. A similar source of information is a so called oracle which could answer queries.

Two general forms of Inductive Program Synthesis can be identified, namely a search-based and an analytical approach. In a search-based approach, the construction of hypotheses relies mainly on the search through the hypotheses space, i. e. hypotheses are generated heuristically and tested against given examples. Contrarily, an analytical approach is guided by the structure of the examples and generates programs of a pre-

defined class by detecting recurrences in given examples which are then generalised to recursively defined functions [49].

Search-based methods are applicable for very general program classes, since there are no principal difficulties in enumerating programs. They naturally facilitate usage of predefined, user provided functions, the so called background knowledge, in induced programs. Their drawback is their search intensiveness and consequently their time consumption. These characteristics qualify them for invention of new and efficient algorithms (c. f. [24]), since they are quite expressive but require long computation times.

Analytical approaches however, apply to more restricted program classes since deriving programs by analysing examples is even more complicated than solely generating and testing programs. For the same reason the usage of background knowledge is much more complicated as it is not used by an enumeration algorithm, but must be considered for analysis, too. Moreover, specific input/output-examples are necessary, since analysis is not possible for example specifications based on evaluation functions. In contrast to search-based generate-and-test approaches, analysis minimises search and makes these approaches fast which qualifies them for assisting systems or end-user programming.

2.2.2 Inductive Programming Techniques

Three main techniques to inductively synthesise programs emerged over time, namely Inductive Logic Programming (ILP), Genetic Algorithms and Inductive Functional Programming. In the following they are introduced by concrete example systems.

2.2.2.1 Evolutionary Computation

Evolutionary Computation encompasses methods of simulating evolution on a computer to solve difficult combinatorial optimisation problems. It is inspired by the Darwinian principles of reproduction, random variation, competition, and natural selection (survival of the fittest) and models its various operations on naturally occurring phenomenons, including crossover (sexual recombination), mutation, gene duplication, gene deletion, which are usually performed on bit vectors [29]. Its subdiscipline Genetic Programming, established by Koza [28], employs these methods to find computer programs that perform a user-defined task.

ADATE [23, 24], following the search-based approach, is the most prominent representative of an Inductive Programming system using Genetic Programming. It harnesses the structuredness of a functional programming language to represent program individuals as bit vectors and generates programs in a sublanguage of ML. Starting from an initial rudimentary user provided program as initial population which is directly enclosed in ADATE's ML code, the system randomly applies in each phase genetic operations to modify the population. Then it selects the fittest programs according to an evaluation function and only takes these fittest, together with some randomly selected to freshen the

"gene pool", into the next phase. Usually, such an evaluation function tests the fitness of an individual on a set of input/output pairs, but also any other, more sophisticated ML-function can be applied.

2.2.2.2 Inductive Logic Programming

Another line of research is the field of inductive logic programming (ILP), a term which was first coined by Muggleton [42]. Though ILP has a focus on non-recursive concept learning problems, there has also been research in inducing recursive logic programs on inductive data types in the field of ILP (see [36]). Most of the specialised systems are analytical approaches.

In ILP, all examples, background knowledge, and hypothesis are represented as definite Horn clauses in a subset of first order logic. Definite Horn clauses have the form $H \vee \neg B_1 \ldots \vee \neg B_n$, where the positive literal H is called the head and represents the predicate, or relation to be learnt. Given a set of clauses B representing the background knowledge and a set of positive examples E^+ and a set of negative examples E^-, the task in the normal ILP setting is about finding the simplest consistent hypothesis H such that

$$B \wedge H \models E^+$$

and

$$B \wedge H \not\models E^-.$$

Or in other words, the hypothesis H is complete and consistent with respect to the training data, given B. Finding this simplest consistent hypothesis H is done, as often in Artificial Intelligence, by search. Therefore, the hypothesis space, i. e. all possible Horn Clauses that can be learnt, are partially ordered in a lattice, based on θ-subsumption [21]. Also based on θ-subsumption, additionally a syntactic notion of generality is introduced, which makes it possible to systematically search this lattice, from general-to-specific or vice versa, for an appropriate hypothesis.

Well known ILP systems are Quinlan's FOIL/FFOIL [27] [26], PROGOL developed by Muggleton [33, 32, 43] and GOLEM developed by Muggleton and Feng [34]. A relatively new analytical system is the schema-guided, interactive, inductive, and abductive recursion synthesiser Dialogs-II (Dialogue-based Inductive and Abductive LOGic program Synthesiser)[18, 41].

2.2.2.3 Inductive Functional Programming

Research on the inductive synthesis of recursive functional programs started in the early 1970s and has always been analytical. It was brought on firm theoretical foundations pioneered with the seminal THESYS system of Summers [35] and the work of Biermann [1] and Kodratoff [52], which all concentrated on synthesising LISP programs.

These classical approaches were based on a two step process. First, the given input/output examples are transformed into finite traces and predicates. Hereby a predicate represents the structure of a given input and a trace computes the corresponding output for a given input. Secondly, regularities are searched in traces and predicates respectively. Found regularities then are inductively generalised and expressed in form of the resulting recursive program.

A recent approach of functional program induction inspired by these classical approaches and formulated within the term rewriting framework is the system IGOR [15]. Functional programs are represented as constructor term rewriting systems (CSs) containing recursive rules. I/O-examples for a target function to be implemented are a set of pairs of terms $(F(i_i), o_i)$ meaning that $F(i_i)$—denoting application of function F to input i_i—is rewritten to o_i by a CS implementing the function F. In addition to the examples of the target functions, background knowledge functions that may be called by the induced functions can be supplied in form of ground equations. This algorithm learns several dependent recursive target functions in one step. It can deal with arbitrary user-defined algebraic datatypes and automatically introduces auxiliary subfunctions if needed.

Chapter 3

Inducing Recursive XSL Transformations by Example

3.1 XML, XSL, XSLT, or What?

This section's intent is to give an introduction into XML/XSL technologies, which are an official recommendation of the World Wide Web Consortium (W3C), so it solely describes the underlying technologies as appropriate to understand the following sections. It first gives an introduction of the basic structure of XML documents and namespace declarations 3.1.1 and then sketches XSL, XPath expressions, and XSL Transformations 3.1.2.

3.1.1 Extensible Markup Language

The e**X**tensible **M**arkup **L**anguage (XML) [17] on the one side is an open standard to structure data and format documents, on the other side it is a tool box to process, filter, or get access to such structured data in any form. Notwithstanding the probably deceptive name, XML is not a markup language itself, rather it is a *metalanguage*: a set of rules to create markup languages, tailored to one's current needs. When we talk about *markup*, we mean additional information which is added to a document to enrich its purport. This extra information, for example about the text's structure or presentation, is expressed using markup, which is intermingled with the primary text. Listing 3.1 shows an example XML document.

XML Elements The markup is done using so called *elements* (also called tags). An element, e. g. `<element_name>`, consists of a tag name and opening and closing angled brackets. The text, e. g. the whole recipe in our example, such an element marks is enclosed between an opening (`<recipe>` Listing 3.1, line 2) and a closing element

(`</recipe>` Listing 3.1, line 16). Elements could also occur empty (`<unit/>` Listing 3.1, line 9), if they do not have a *value*. The value of an XML element is any content, text, or XML code, enclosed by its tags.

Listing 3.1: Example XML document of a recipe.

```
1 <?xml version="1.0"encoding="UTF-8"?>
  <recipe>
   <title>Hühnerbrühe</title>
   <ingredients>
5   <position quantity="1">Huhn</position>
    <position quantity ="2">
     <unit>Liter</unit>Wasser
    </position>
    <position><unit/>Salz</position>
10  <!-- weitere Zutaten -->
  </ingredients>
   <preparation>
    Alles in einem <equipment>
    Topf </equipment>kochen.
15  </preparation>
  </recipe>
```

The top most element is called the *document element* which is in our case `<recipe>`. Elements could also contain additional information as so called *attributes*. Attributes have the form `attribute_name="value"` and are infixed between the element name and the closing bracket of an element.

Another important concept of XML are *namespaces* [38]. Elements are naturally identified by their name. However, especially when considering more than one document, i. e. during XSL processing, it could come to mistakes if two elements with semantically different meaning have the same element name, e. g. the person ‘student’ and the person ‘professor’ in a list of persons. To avoid such name clashes, a namespace declaration is added to elements. A definition of a namespace declaration has the form `xmlns:prefix="URI"`, where `xmlns` is the keyword for **xml n**ame**s**pace, `prefix` a user-defined prefix and `URI` a *Uniform Resource Identifier* preferably a web address. It is included in an element like an attribute. Listing 3.2 shows an XML document with two namespaces.

Listing 3.2: XML with namespaces

```
1 <persons
   xmlns:prof="http://www.uni-bamberg.de/prof"
   xmlns:stud="http://www.uni-bamberg.de/stud">
```

```
  <prof:person>Ute Schmid</prof:person>
5 <stud:person>Martin Hofmann</stud:person>
 </persons>
```

Apart from elements and text, an XML document also contains a so called *prolog* which defines, enclosed by special tokens `<?` and `?>`, for example encoding, used XML version or further structural definitions. In Listing 3.1 line 1, only the XML version with which the document is compliant is listed. Comments are written between `<!--` and `-->`.

Well-Formed Documents So far, we introduced the main building blocks of XML documents defined by the XML standard. Additionally, this standard also makes demands on the structure of XML documents, i. e. how elements are allowed to be put together. XML documents have to be (i) well-formed and (ii) the elements must not overlap. Hence, either an element is empty, or an opening tag is followed by a closing tag. Furthermore overlapping elements, i. e. `<e1><e2></e1></e2>` are not allowed.

Although these requirements are not very surprising, they make it possible to represent an XML document as a tree. The so called *Document Object Model* (DOM) [20], an interface for XML processing for several programming languages, make heavily use of this property. Although we do not want to go into details of DOM, we will use the DOM Tree to represent XML documents to describe the functionality of XSL. Figure 3.1 shows the DOM Tree of Listing 3.1. Note the root node of the DOM Tree, the *document node*. This must not be confused with the *document element* or *root element* of the XML document which is in this case `<recipe>`.

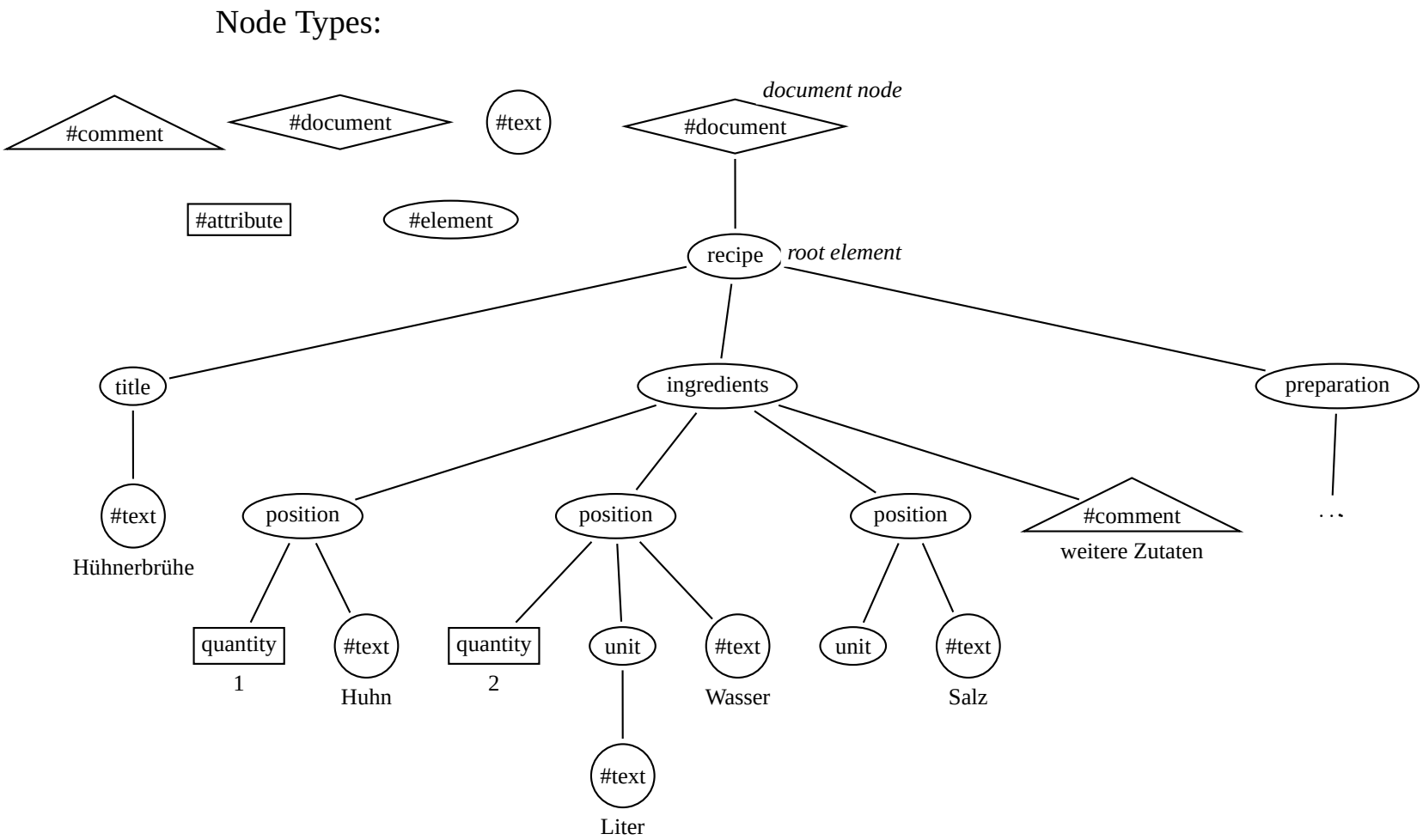

Figure 3.1: Extract of the DOM of Listing 3.1. Formatting whitespaces have been omitted.

3.1.2 Extensible Stylesheet Language

XSL is a family [5] of languages which provide the means to instruct an XSL processor how to transform or format XML standard documents. The first of the following section (3.1.2.1) describes the basics of XPath (XML Path Language), the second sketches the main functionality and concepts of XSL Transformations (XSLT).

3.1.2.1 XPath

XPath (XML Path Language) [40] is an expression language for addressing parts of an XML document or for computing values (strings, numbers, or boolean values) based on the content of an XML document. The XPath language is based on a tree representation of the XML document where every XML element, attribute, comment, or text chunk (including whitespaces) is represented as a node (c. f. Section 3.1.2). Using XPath expressions, it is possible to navigate through the tree, selecting nodes by a variety of criteria, or compute values.

Locating Nodes This is similar to navigate through a file system. We have absolute paths—starting at the root node—and relative paths. The root node is represented by a forward slash '/' and any other element by its `element_name`. Attributes are referred to by an @-sign prefixed to the `attribute_name`, text nodes are simply accessed via `text()`. An absolute path starts with a '`/`', i. e. from the root node, relative paths with an `element_name`. relative paths from the current node start with `//`. To address the current node the period '.' is used, whereas two periods '..' point to the parent node. As a special case, a node name or an attribute name can be represented with an asterisk '*' which serves as a wildcard character. To state alternatives between XPath expression, the vertical bar '|' is used which functions as a logical union operator. Table 3.1 shows some example expressions using the XML document of Listing 3.1.

Computing Values A couple of data types are defined in XPath: string, numeric, Boolean, and nodes (or node-sets). Strings must always be contained in double quotes `"`, if the expression as a whole is contained in double quotes, the string must be enclosed in single quotation marks or apostrophes: `'`. Numeric values are simply written as usual with the common arithmetic operators $+, -, div$ and mod. For Boolean values, two constants `true()` and `false()` are defined but also the correspondent numeric values 1 and 0 apply. Together with Boolean values, the operators `and`, `or`, and `not()` are defined, where `not()` has one Boolean value as argument. Apart from these so called *atomic values* there exists the value type of *nodes* or *node-sets*.

XPath expressions that locate nodes result in *node-sets*, for example the XPath expression `position` (applied on the XML document of Listing 3.1) would result in a node-set consisting of *all* elements with name `position`. Although the term

Table 3.1: Example XPath expressions to navigate through a tree.

`/`	the root node, i. e. `<recipe>`
`position | unit`	any element node, which name is `position` or `unit`
`/*`	any direct descendant of the root node
`/@*`	any attribute of the root node (in this case none)
`position[@quantity="1"]`	any element which name is position and which has an attribute called quantity with value “1”
`title/text()`	any text node that is an immediate descendant of a `title` node

‘set’ is usually associated with an unordered amount of nodes, a node-set also keeps track of a so called *context* of the nodes contained. This context is an internal state, maintained by an XSL processor, holding information as the current node processed, the *size* and the *position* of the nodes relative to their order in the document as well as *namespace bindings* and *family relations*. For example, the node-set defined by `ingredients/*` applied on the XML document of Listing 3.1 includes all nodes enclosed by the `<ingredients>` opening and closing tags. The size of the context of this node-set would be 4 where, `ingredients/*[1]` would indicate the first context element (`<position quantity="1">Huhn</position>`) and `ingredients /*[last()]` the last context element, i. e. the comment. However, not only nodes could be represented in an ordered list, also other atomic values can be combined in a sequence. A *sequence* is a comma-separated list of nodes or values, embraced by two brackets ‘(’ and ‘)’ which could be accessed similar to node sets.

Functions XPath provides also a number of built-in functions. The following table (Table 3.2) gives an outline on the functions used by ProXSLbE.

Table 3.2: Functions used by ProXSLbE.

Function	Returns	Description
Node-set Functions		
`last()`	Number	Returns the number of nodes in the current context node-set.
`position()`	Number	Returns the index position of the node that is currently being processed.
String Functions		
`starts-with(s1,s2)`	Boolean	Returns true if string `s1` starts with string `s2`, otherwise it returns false.
`ends-with(s1,s2)`	Boolean	Returns true if string `s1` ends with string `s2`, otherwise it returns false.
`tokenize(s,p)`	Sequence	Returns the result of tokenizing the string `s` by the pattern string `p`.
Boolean Functions		
`not(b)`	Boolean	Returns true if the boolean value `b` is false, and false if the boolean value `b` is true.
`true()`	Boolean	Returns the boolean value true.
`false()`	Boolean	Returns the boolean value false.
Sequence Functions		
`empty(seq)`	Boolean	Returns true if the value of the sequence `seq` is an empty sequence, otherwise it returns false.
`subsequence(seq,s,l)`	Sequence	Returns a sequence of items from the position specified by `s` and continuing for the number of items specified by `l`. The first item is located at position 1.

3.1.2.2 XSLT

XSLT, the Extensible Sytlesheet Language for Transformations is an official recommendation of the World Wide Web Consortium (W3C) [31]. It provides flexible and powerful means to transform an XML document in any other document format desired, as for example HTML, Portable Document Format (PDF), Scalable Vector Graphics (SVG), XML, or text, to mention only a few. In fact, XSLT is a language, itself compliant to the XML standard, to instruct an XSL processor, as e. g. XALAN or SAXON, how the passed XML document should be transformed. Listing 3.3 shows a simple XSL stylesheet, which copies any XML document but omits comments.

Listing 3.3: Simple XSL stylesheet, copying any element but comments.

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml"indent="yes"/>

  <xsl:template match="/|*|@*|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="comment()">
    <!--noop-->
  </xsl:template>

</xsl:stylesheet>
```

Stylesheet Structure Let us first take a closer look at the structure of an XLST document. As mentioned before, it complies with the XML standard, so also possesses an XML prolog and also has a root element, surrounding the whole content. In case of Listing 3.3—and in fact any XSL document else, compliant to the W3C recommendation—this is the element `<xsl:stylesheet>`. Note that all XSL elements come in the XSL namespace defined in the root element. The element `<output>` defines the format of the target document, i. e. XML and instructs the XSL processor to indent the tags. Beneath the root element there are also two so called *template rules* with an XPath expression as a matching condition. These template rules actually define how an XML document is processed. To make clear how this works, it is necessary to look at the job of an XSL processor in more detail.

Processing First, the XSL processor parses the input XML and internally constructs a tree view of the source. After parsing, the processor starts to process the XSLT instructions with the root node in the context. As already mentioned, the *context* is an internal state of the XSL processor, containing a list of nodes to be processed as well as their relative position in the document and their link to other nodes in the tree. While still remaining nodes are in the context, the processor picks the first node and applies a matching template rule on it. A template rule matches a node, if the XPath expression of the `match` attribute of the template rule applies to the node. If more than one rule would apply, the most specific is executed, if no rule applies, a default rule which in most cases either copies the node or does nothing is executed. Note that this processing model is recursive, since a template rule may invoke other templates again. The concrete functionality of the XSL templates is described in the following paragraph.

XSL Elements This paragraph introduces the XSL elements which are relevant for the system ProXSLbE. The following table 3.3 describes the used XSL elements.

Table 3.3: XSL elements used by ProXSLbE.

Element Name	**Description**
`stylesheet`	Defines the root element of an XSL stylesheet.
`template`	The `<xsl:template>` element contains rules to apply when a specified node is matched. A `xsl:template` element has either a `match` or a `name` attribute. In the first case it is called a *template rule* in the latter it is a *named template*.
Attributes:	
`match`	A XPath expression which specifies the match pattern for the template.
`name`	Specifies a name for the template.
`apply-templates`	The `<xsl:apply-templates>` element applies a template to the current element or to the current element's child nodes, i. e. the stylesheet is invoked recursively on these nodes. If a `select` attribute is added to the `<xsl:apply-templates>` element it will process only the child element that matches the value of the attribute.

	Attribute:	
	`select`	An optional XPath expression which specifies the nodes to be processed. An asterisk selects the entire node-set. If this attribute is omitted, all child nodes of the current node will be selected.
`call-template`		The `<xsl:call-template>` element calls a named template. The value of the `name` attribute must match a `name` in an `<xsl:param>` element. It is allowed within `<xsl:apply-templates>` and `<xsl:call-template>`.
	Attribute:	
	`name`	A required string which specifies the name of the parameter.
`with-param`		The `<xsl:with-param>` element defines the value of a parameter to be passed into a template.
	Attributes:	
	`name`	A required string which specifies the name of the template to be called.
	`select`	An optional XPath expression that defines the value of the parameter.
`param`		The `<xsl:param>` element is used to define the parameter of `<xsl:apply-templates>` and `<xsl:call-template>`.
	Attributes:	
	`name`	Specifies the required name of the parameter.
	`selects`	Specifies an optional XPath expression that specifies a default value for the parameter.
`copy`		The `<xsl:copy>` element creates a copy of the current node.
`choose`		The `<xsl:choose>` element is used in conjunction with <`xsl:when>` and `<xsl:otherwise>` to express multiple conditional tests. If no `<xsl:when>` is true, the content of `<xsl:otherwise>` is processed. If no `<xsl:when>` is true, and no `<xsl:otherwise>` element is present, nothing is created.

`when`		The `<xsl:when>` element is used to specify an action for the `<xsl:choose>` element. The `<xsl:when>` element evaluates an expression and if it returns true, an action is performed.
	Attribute:	
	`test`	Specifies the required Boolean expression to be tested.
`otherwise`		The `<xsl:otherwise>` element specifies a default action for the `<xsl:choose>` element. This action will take place when none of the `<xsl:when>` conditions apply.
`if`		The `<xsl:if>` element contains a template that will be applied only if a specified condition is true.
	Attribute:	
	`test`	Specifies the required condition to be tested.
`value-of`		The `<xsl:value-of>` element extracts the value of a selected node. It can be used to select the value of an XML element and add it to the output.
	Attribute:	
	`select`	A required XPath expression that specifies which node/attribute to extract the value from.
`variable`		The `<xsl:variable>` element is used to declare a variable. Once a variable's value has been set, it cannot be changed or modified again. A variable could be read using the XPath expression `'$variable_name'`
	Attributes:	
	`name`	Specifies the required name of the variable.
	`select`	An optional XPath expression which defines the value of the variable.

3.2 Inductive Program Synthesis with IGOR

IGOR [15, 14, 13] incorporates a technique to induce recursive functional programs over algebraic datatypes from few non-recursive and only positive ground I/O examples. Functional programmes as well as examples are represented as equations between terms. Learning is data-driven and based on structural regularities between the examples. In this context the notion of "functional" does not refer to a programming paradigm or even a concrete functional programming language, but to a class of programmes, that map each input to a unique output.

The following sections give a short introduction into the theoretical foundations of *term rewriting systems* (3.2.1) based on the books [19, 46]. Section 3.2.2.1 deals with the particularities of specifying the input examples for IGOR within the context of this work, and Section 3.2.2 finally gives insights in IGOR's induction algorithm.

3.2.1 Preliminaries

As already mentioned, IGOR's theoretical foundation is the theory of *term rewriting*, which combines elements of logic, universal algebra, automated theorem proving and functional programming and is itself based on equational logic [19, 46]. This logic aims at defining new operations from given ones by stating characteristic identities between them which must hold for the newly defined operation. If interpreted directional from left to the right an equation is considered as a rewrite rule.

In Example 3.2.1 the function $last$, with the usual semantics over lists, is defined using the constant [] denoting the empty list and the function $cons$ for list construction.

Example 3.2.1.

$$\begin{aligned} last(cons(x, [])) &\rightarrow x \\ last(cons(x, xs)) &\rightarrow last(xs) \end{aligned}$$

3.2.1.1 Terms and Signature

The *term rewriting system* (TRS) in Example 3.2.1 consists of two *rewrite rules*, describing in which way a term on the left hand side (*lhs*) can be rewritten (or reduced) to another term on the right hand side (rhs). A *term* consists of *variables*, *function symbols* and *constants*. In Example 3.2.1 $cons$ is a binary function symbol, [] is a constant corresponding to a unary function symbol and x and xs are variables. To make clear which function symbols with accordant arity are applicable in which context, a *signature* is defined.

Definition 3.2.1 (Signature)**.**
*A **signature** Σ is a non-empty set of **function symbols**, each with a fixed arity. The arity of a function symbol F is a non-negative integer, denoting the number of its arguments. Function symbols with arity 0 are called **constants**.*

Terms are strings over an *alphabet*, consisting of the signature and a countably infinite set V of *variables*. The set V is assumed to be disjoint from the set of function symbols in the signature Σ. Variables will be named by x, y, z, x', etc or with indices $x_0, x_1, x_2, \ldots$. For function symbols usually upper case letters are used, or more meaningful names. In the following definition of terms, the prefix notation for function is used, but also the infix notation may be used, as long as it is conform with terms from Definition 3.2.2.

Definition 3.2.2 (Terms)**.**
*Let V be a set of variables disjoint from Σ, then the set of all **terms** over Σ denoted by $T(\Sigma, V)$ is inductively defined:*

- $x \in T(\Sigma)$*, for every* $x \in V$
- *If F is an n-ary $(n \geq 0)$ function symbol and $t_1, \ldots, t_n \in Term(\Sigma)$, then $F(t_1, \ldots, t_n) \in T(\Sigma)$*
- *these are all terms.*

The terms t_i are called *arguments* of the term $F(t_1, \ldots, t_n)$ and the symbol F is called the *head symbol* or *root*. Terms that do not contain any variable are called $ground$ or $closed$. Terms in which every variable occurs only once are called $linear$.

3.2.1.2 Position and Context

To facilitate talking about terms and parts of terms, they can be viewed as a finite, labelled, ordered tree as follows as shown in Figure 3.2 for the term $g(f(x), h(y, z))$.

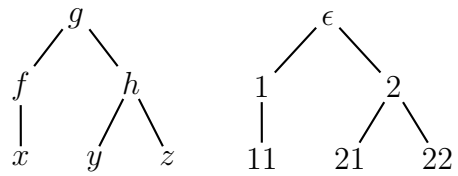

Figure 3.2: AST of $g(f(x), h(y, z)$ (left) and the position its nodes (right)

Definition 3.2.3 (Abstract Syntax Tree of a Term)**.**
The abstract syntax tree (AST) of a term is defined inductively over the structure of terms:

1. *Every variable or constant of a term corresponds to a tree consisting of a single node, labelled with the name of the variable or constant, respectively.*
2. *A term $F(t_1, \ldots, t_n)$ can be represented by a tree, with the root node labelled with F and n child nodes representing the terms $t_1, \ldots, t_n$ from left to right.*
3. *These are all abstract syntax trees of terms.*

To access specific nodes in such a tree, we follow the standard numbering of nodes of a tree denoting a position in a tree by strings of positive integers. The position ϵ is called the root position of a term s. The position ϵ of term $g(f(x), h(y, z))$ is the top-level function symbol g, position 2 is $h(y, z)$ and 11 is x. By induction over the structure of terms, subterms and positions can be defined formally as follows.

Definition 3.2.4 (Tree Numbering)**.**
Let Σ be a signature, V a set of variables disjoint from Σ, and $s, t \in T(\Sigma, V)$.

1. *Let $\mathcal{P}os(s)$ be the set of all* ***positions*** *of a term s*
   - *if $s = x \in V$, then $\mathcal{P}os(s) := \{\epsilon\}$*
   - *if $s = f(t_1, \ldots, t_n)$, then*

$$\mathcal{P}os(s) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip | p \in \mathcal{P}os(t_i)\} :$$

2. *For $p \in \mathcal{P}os(s)$,* ***the subterm of s at position*** *p, denoted by $s|_p$, is defined by induction on the length of p:*

$$\begin{aligned} s|_\epsilon &:= s, \\ f(s_1, \ldots, s_n)|_{iq} &:= s_i|q. \end{aligned}$$

3. *For $p \in \mathcal{P}os(s)$,* ***replacing the subterm of a term*** $\mathbf{s}$ ***at position*** $\mathbf{p}$ ***by*** $\mathbf{t}$ *is denoted by $s[t]_p$,* ***i.e.***

$$\begin{aligned} s[t]_\epsilon &:= t, \\ f(s_1, \ldots, s_n)[t]_{iq} &:= f(s_1, \ldots, s_i[t]q, \ldots s_n). \end{aligned}$$

To access subterms at arbitrary positions of a term, the concept of a *context* is introduced. A context C could be seen as an incomplete term containing one or more empty places, so called *holes*. Formally, it is a term over the extended signature $\Sigma \cup \{\Box\}$, where each $\Box$ denotes a hole. If C is a context containing exactly n holes, then $C[t_{,1}, \ldots, t_n]$ denotes a term resulting from replacing all holes in C from left to right by the terms $t_{,1}, \ldots, t_n$.

3.2.1.3 Substitution, Subsumption, and Unification

A *substitution* σ is a mapping from variables to terms. Assume a signature Σ as given. Usually $t\sigma$ is written instead of $\sigma(t)$, so $t\sigma$ is the result of applying σ to all variables in term t.

Definition 3.2.5 (Substitution)**.**
A ***substitution*** *is a mapping $\sigma : V \mapsto T(\Sigma)$, which satisfies*

$$\sigma(F(t_1, \ldots, t_n)) \equiv F(\sigma(t_1), \ldots, \sigma(t_n))$$

for every n-ary function symbol F $(n \geq 0)$; in particular, $\sigma(F) \equiv F$ if F is a constant.

A substitution from variables to variables is called *variable renaming*. Substitution induces a quasi-order on terms, the so called *subsumption* order.

Definition 3.2.6 (Subsumption)**.**
*Let $s \equiv t\sigma$. Then s is called a (**substitution**) **instance** of t. We say that t **subsumes** s, i. e. it is a **generalisation** of s. The term t is also said to be matched with its instance s by the substitution σ or s matches t by σ.*

Given two terms t_1, t_2 and a substitution σ such that $t_1\sigma = t_2\sigma$, then we say t_1 and t_2 *unify* and call σ a *unifier* of t_1 and t_2.

Definition 3.2.7 (Unifier)**.**
*A **unifier** is a substitution σ such that $t_1\sigma = t_2\sigma$.*

The subsumption relation is generalised to sets of terms and we say that a set of terms T subsumes another set of terms S if each $s \in S$ is subsumed by a term $t \in T$. Given a set of terms $S = \{s, s', s'', \ldots\}$, then there exists a term t which subsumes all terms in S and which is itself subsumed by every other term subsuming all terms in s. Such a term t is called *least general generalisation (lgg)* of the terms in S.

3.2.1.4 Term Rewriting Systems and Constructor TRS

In the introductory Example 3.2.1 we already sketched the idea of a term rewriting system informally. In the previous paragraphs we defined terms and signatures, the last missing building blocks are *rewrite rules*.

Definition 3.2.8 (Rewrite Rule)**.**
*A **rewrite rule** for a signature Σ is a pair $\langle l, r\rangle$ with $l, h \in T(\Sigma, V)$ and will be written $l \to r$. Two restrictions are imposed on rewrite rules:*

1. *the left hand side (lhs) l is not a variable $l \notin V$*
2. *every variable occurring on the right hand side (rhs) r also occurs on the lhs l*

The first property of Definition 3.2.8 simply forbids rewrite rules that match everything, and the second property assures that no rewrite rule introduces new variables. Now we have all parts to give a general definition of a TRS.

Definition 3.2.9 (Term Rewrite System)**.**
*A **term rewrite system** (TRS) is a pair $\mathcal{R} = (\Sigma, R)$ of a signature Σ and a set of rewrite rules R for Σ.*

An important property of the rewrite rules IGOR uses is that every *lhs* of a rule has the form $F(t_1, \ldots, t_n)$ where neither F nor the name of any other of the defined functions occur in t_i. We call such a TRS a *constructor term rewriting system*, since the symbols in the signature Σ can be divided into two disjoint subsets $\mathcal{F}$ of *defined function symbols*, e. g. F and a set $\mathcal{C}$ of *constructors*.

Definition 3.2.10 (Constructor Term Rewriting System)**.**
*A TRS over a signature Σ is a **constructor TRS** (CS) if Σ can be partitioned into two sets of **constructors** $\mathcal{C}$ and **defined functions** $\mathcal{F}$ such that the lhs of every rule has the form $F(t_1, \ldots, t_n)$ with $F \in \mathcal{F}$ and every $t_1, \ldots, t_n \in \Sigma(\mathcal{C}, V)$.*

Terms without defined function symbols are called constructor terms. Ground constructor terms denote values. The constructor terms t_i in the *left hand sides* (*lhss*) of the equations for a defined function F may contain variables and are called *patterns*. This corresponds to the concept of pattern matching in functional programming languages and is the only form of case distinction.

A CS establishes a *rewrite relation* R, denoted $\rightarrow_R$, which is defined as follows:

Definition 3.2.11 (Rewrite Relation)**.**
*Given a TRS R, then a term t rewrites to s according to R, written $t \rightarrow_R s$, iff there exists a rule $l \rightarrow_R r$ in R, a substitution σ, and a context C such that $t = C[l\sigma]$ and $s = C[r\sigma]$. Such a relation between terms is called **rewrite relation**.*

Applying a rewrite rule of a set of rewrite rules R once on a term t_1, i. e. rewriting t_1 to t_2 ($t_1 \rightarrow_R t_2$) is called a *rewrite step*. We also say, t_1 is *reduced* to t_2 or t_1 is reducible to t_2. A sequence of finitely or infinitely many rewrite steps $t_0 \rightarrow_R t_1 \rightarrow_R \ldots$ is called a *derivation*. If a term is not reducible anymore it is called a *normal form*. If a derivation starts with term t and results in a normal form s, then s is called normal form of t, written $t \xrightarrow{!} s$. We say that *tnormalises* to s.

In order to define a (total) function on a domain (a set of ground terms) by a CS, no two derivations starting with the same ground term may lead to different normal forms, i. e. normal forms must be unique. Moreover, the derivations must terminate and the normal forms must be ground constructor terms, i. e. denote values. If each possible derivation terminates, we say that the CS is terminating. A sufficient condition for unique normal forms is that no two *lhss* of a CS unify, then the CS is confluent. A sufficient condition for termination is that the arguments/inputs of recursive calls strictly decrease within each derivation and with regard to a well founded order. In order to get ground normal forms from derivations starting from ground terms, each variable in a *rhs* of a rule must also occur in its lhs.

3.2.2 Function Induction by Pattern Matching

The theoretical foundations have been made clear now, so we can proceed to describe IGOR's functioning. This description is based on the original papers [15, 14] and especially [13].

3.2.2.1 Input Specification

For better readability we write t^n for a list of terms $t_1, \ldots, t_n$. As mentioned before, IGOR expects a set of examples in form of equations as $F(i^n) = o$, where the i^n and o are ground constructor terms and called input and output respectively. Even if an input is a sequence of terms, it is sometimes helpful to regard this sequence as one term. This is done by assuming a distinguished constructor symbol as root which has the inputs as subterms. If the input is a single term, such a root constructor symbol may be omitted. Depending on the context, we speak of equations or rewrite rules or CSs respectively to denote hypotheses. If the current focus is not on term rewriting, the pair $\langle i^n, o \rangle$ is called *input/output pair* (I/O pair) or *input/output example* (I/O example).

If i is an input term to a recursively defined function F with a corresponding output term o and i' is the input to F resulting from a recursive call of F within computing i, then o contains the output term o' for i' as subterm. Compare to the Figure 3.3 where the rhs of equation (j) is contained, allowing for variable renaming, in the rhs of equation ($j+1$), respectively the lhss. To infer a recursive definition from example computations, using this structural regularity between computations of recursively defined functions is the core of analytical function induction as proposed by Summers [35]. A necessary condition for applying this principle is that for each example input, all inputs resulting from recursive calls are also included in the example set. The following definition states this condition formally and is extended to more than one defined function to be induced.

Definition 3.2.12 (Recursively Subsumed Example Equations)**.**
*Let R be a CS which correctly computes a set of example equations. The example equations are called **recursively subsumed** w. r. t. R, iff for all example inputs i^n holds: Let $F(p^n) \to t$ be a rule in R such that i_n matches p_n by substitution σ. Then for each call $F'(r^m)$ of a defined function F' of R in t the instantiation $r^m\sigma$ is contained as an example input in the example equations.*

We require that induced CSs are terminating and that they represent functions, i. e. that they have unique normal forms. With regard to the given examples we require that a hypothesis is correct:

Definition 3.2.13 (Correctness)**.**
*A hypothesis, i. e. a CS R is **consistent/ complete** w. r. t. a set of example equations iff for each example equation $F(i^n) = o$ holds that*

consistent: $F(i^n) \xrightarrow{!}_R o$ *or* $F(i^n) \xrightarrow{!}_R s$ *for a term* $s \notin T(\mathcal{C}, V)$*,*

complete: $F(i^n) \xrightarrow{!}_R s$ *for a term* $s \in T(\mathcal{C}, V)$*.*

*A hypothesis is **correct** iff it is both consistent and complete.*

The consistence condition assures that if the induced function is defined for an input, then the function value on this input is the specified output. The completeness condition assures that the induced function is total on the example inputs. Figure 3.3 shows example equations for the *Reverse*-function and the equations induced by our system. Only the example equations and the corresponding datatype definitions were provided. Note that the example equations contain variables. Using variables where possible reduces the amount of needed example equations and makes the induction more time efficient. Two subfunctions have been introduced automatically, *Last* and *Init*, which compute the last element of a list and the list without the last element respectively. Note that automatically introduced subfunctions are named "Sub1", "Sub2" etc. by the system.

Example Equations:

$$
\begin{array}{lcll}
Reverse([]) & = & [] & (i) \\
Reverse([X]) & = & [X] & (ii) \\
Reverse([X,Y]) & = & [Y,X] & (iii) \\
Reverse([X,Y,Z]) & = & [Z,Y,X] & (iv) \\
Reverse([X,Y,Z,V]) & = & [V,Z,Y,X] & (v)
\end{array}
$$

Induced CS:

$$
\begin{array}{lcl}
Reverse([]) & \rightarrow & [] \\
Reverse([X|Xs]) & \rightarrow & [Last([X|Xs])|Reverse(Init([X|Xs]))] \\
Last([X]) & \rightarrow & [X] \\
Last([X_1,X_2|Xs]) & \rightarrow & Last([X_2|Xs]) \\
Init([X]) & \rightarrow & [] \\
Init([X_1,X_2|Xs]) & \rightarrow & Last([X_1|Init([X_2|Xs])])
\end{array}
$$

Figure 3.3: Non-ground example equations and the induced solution for the function *Reverse*.

The induction of a terminating, confluent, correct CS is organised as a kind of best first search in the hypothesis space. During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating and confluent CS *but potentially with variables in the* rhs *s not occurring in the lhs.* That is, the equations of a hypothesis during search do not necessarily represent *functions*. Such "non-functional" equations are called *unfinished* equations and hypotheses containing unfinished equations are called unfinished hypotheses. A goal state is reached, if at least one of the best—according to a criteria explained below—hypotheses is finished, i. e. does not contain unfinished equations. Such a finished hypothesis is terminating and confluent, since *all* hypotheses during search are confluent and terminating by construction, and it is correct, since its equations entail the example equations and together with termination and confluence all example inputs normalise to their example outputs.

The induction bias is to prefer CSs whose patterns partition the example inputs into fewer subsets. This corresponds to preferring programmes with fewer case distinctions. This leads, in some sense, to a most general hypothesis. Regarding the defined function, this bias prefers a CS with fewer rules, since the pattern of each rule determines one unique subset. But consider the solution for the function *Reverse* with the subfunctions *Last* and *Init* as shown in Figure 3.3. The solution contains six rules but the number of induced example input subsets is only three, because *Last* and *Init* induce the same subsets (pattern $[X]$ subsumes the second example, pattern $[X1, X2|Xs]$ examples $(iii) - (v)$) which are again partitions of the subset induced by pattern $[X|Xs]$ of *Reverse* such that pattern [] from *Reverse* remains and induces the subset containing the first example input. Contrarily, choosing fewer rules as preference bias then obviously the five example equations themselves would have been favoured over the solution with *Init* and *Last* such that no generalisation would have taken place.

With respect to the described bias and in order to get a complete hypothesis w. r. t. the examples, the initial hypothesis is a CS with one rule per target function such that its pattern subsumes all example inputs. In most cases (e. g. for all recursive functions) one rule is not enough and the *rhss* will remain unfinished. Then for one of the unfinished rules successors will be computed which leads to one or more (unfinished) hypotheses. Now repeatedly unfinished rules of currently best hypotheses are replaced until a currently best hypothesis is finished. Since one and the same rule may be member of different hypotheses, the successor rules originate successors of *all* hypotheses containing this rule. Hence, in each induction step several hypotheses are processed.

3.2.2.2 Initial Rules

Given a set of example equations for one target function, the initial rule is constructed by first anti-unifying all example inputs. This leads to the *lgg* (least general generalisation) of the example inputs, i. e. to the most specific pattern subsuming all example inputs. Second, the example outputs are anti-unified w. r. t. the substitutions resulting from anti-unification of the inputs. This gives the *lgg* of all outputs were variables from the pattern are used if possible.

Lemma 3.2.1. *Let R be a CS with non-unifying patterns and which is correct regarding a set of recursively subsumed example equations. Then there exists a CS R' such that R' contains exactly one pattern p' for each pattern p in R, each p' is the* lgg *of all example inputs matching the corresponding pattern p, and R and R' compute the same normal form for each example input.*

Proof 3.2.1. *The proof has been omitted, and the reader is referred to [13] instead.*

That is, R' induces the same partition of example inputs and has only *lggs* of the example input subsets as patterns. This allows to only regard *lggs* as patterns which narrows the search space without loss of completeness regarding the examples.

3.2.2.3 Processing Unfinished Rules

This section introduces three rules to generate successor sets for an unfinished rule, i. e. "non-functional" equations. The successor equations are generated by either splitting rules by pattern refinement, introducing function calls, and introducing subfunctions.

Splitting rules by Pattern Refinement The first method for generating successors of a rule is to replace its pattern p^n by a set of more specific patterns, such that the new patterns induce a partition of the example inputs matching p^n. This results in a set of new rules replacing the original rule and—from a programming point of view—establishes a case distinction.

Suppose a rule with pattern p^n which is the *lgg* of the example inputs matching it. Then the examples whose inputs match p^n have to be partitioned into a minimum number of at least two subsets and p^n has to be replaced by the *lggs* of the inputs of the respective subsets. It has to be assured that no two of the new *lggs* unify.

This is done as follows: First a position u is chosen at which a variable stands in the *lhs* $F(p^n)$. Since p^n is the *lgg* of all inputs matching, it holds that at least two inputs have different constructor symbols at position u. Then respectively all example inputs with the same constructor at position u are taken into the same subset. This leads to a partition of the example inputs. Finally, for each subset the *lgg* is computed. The new *lggs* does not unify, since they have different constructors at least one position.

Possibly different positions of variables in pattern p^n lead to different partitions. Then all partitions and the corresponding sets of specialised patterns are generated. Each new pattern determines the *lhs* of a new rule. The corresponding initial *rhs* s are computed as *lggs* of the respective outputs as described in Section 3.2.2.2. Since the refined patterns subsume fewer examples, the number of variables in the initial *rhs* s which are not contained in the corresponding *lhs* (non-strictly) decreases with each refinement step. Eventually, if no correct hypothesis with fewer partitions exists, each example input is subsumed by itself such that the example equations are simply reproduced.

For example, let

$$\begin{array}{lcll} Reverse([]) & \rightarrow & [] & (i) \\ Reverse([a]) & \rightarrow & [a] & (ii) \\ Reverse([b]) & \rightarrow & [b] & (ii) \\ Reverse([a,b]) & \rightarrow & [b,a] & (iv) \\ Reverse([b,a]) & \rightarrow & [a,b] & (v) \end{array}$$

be some examples for the function *Reverse*. The pattern of the initial rule is simply a variable X, since the example input terms have no common root symbol. Hence, the only position at which the pattern contains a variable and the example inputs have different constructors is the root position. The first example input consists of only the constant $[]$ at the root position. All remaining example inputs have the constructor $cons$ as root.

i. e. two subsets are induced by the root position, one containing the first example, the other containing all remaining examples. The *lggs* of the example inputs of these two subsets are [] and $[X|Xs]$ respectively which are the patterns of the two successor rules.

Introducing Function Calls The second method to generate successor sets for an unfinished rule with pattern p^n for a target function F is to replace its *rhs* by a call to a defined function F', i. e. by a term $F'(R_1(p^n), \ldots, R_m(p^n))$. Each R_i denotes a new introduced defined (sub)function. This finishes the rule, since now the *rhs* does not longer contain variables not contained in the lhs. In order to get a rule leading to a correct hypothesis, for each example equation $F(i^n) = o$ of function F whose input i^n matches p^n with substitution σ must hold: $F'(R_1(p^n), \ldots, R_m(p^n))\sigma \xrightarrow{!} o$. This holds if for each output o an example equation $F'(i'_1, \ldots, i'_m) = o'$ of function F' exists such that $o = o'$ and $R_i(p^n)\sigma \xrightarrow{!} i'_i$ for each R_i and i'_i. That is, if we find example equations of F' such that $o = o'$ for each example output o for the currently considered (unfinished) rule of F and an example output o' of the called function, then we can abduce example equations $R_i(i) = i'$ for the new subfunctions R_i and induce them from these examples. Provided, the final hypothesis is correct for F' and all R_i then it is also correct for F.

In order to assure termination of the final hypothesis it must hold $i' < i$ according to any reduction order $<$ if the function call is recursive.

Introducing Subfunctions The last method to generate successor equations can be applied, if all outputs o of the inputs matching the pattern of the considered unfinished rule have the same constructor c as roots. Let c be of arity m then the *rhs* of the rule is replaced by the term $c(Sub_1(p^n), \ldots, Sub_m(p^n))$ where each Sub_i denotes a new introduced defined (sub)function. This finishes the rule, since all variables from the new *rhs* are contained in the lhs. The examples for the new subfunctions are abduced from the examples of the current function as follows: If $o|_i$ are the i^{th} subterms of the outputs o, then the equations $Sub_i(i) = o|_i$ are the example equations of the new subfunction Sub_i. Thus, correct rules for Sub_i compute the i^{th} subterm of the outputs o such that the term $c(Sub_1(p^n), \ldots, Sub_m(p^n))$ normalises to the outputs o.

A Remark on the Described Successor Functions As described in Section 3.2.2.3, (recursive) calls to defined functions specified by examples are only introduced at the root of a *rhs* (since such calls are introduced by replacing an unfinished *rhs*). Of course generally, function calls can occur at any position in a *rhs*, compare for example the recursive definition for *Init* in Figure 3.3. The reason why e. g. *Init* can be induced by our approach though function calls are only introduced at root positions is that deeper positions are (indirectly) considered as consequence of subprogram introduction as described in Section 3.2.2.3 because *rhs* root positions of such subprogrammes correspond to deeper positions of the *rhs* of the rule calling these subprograms.

3.3 Bridging the Gap – The Core Algorithm of ProXSLbE

This section's intent is to describe, how the system ProXSLbE brings together the world of XML transformation with XSL and the term-rewriting of IGOR. As mentioned in the previous section, IGOR needs the first, in terms of the underlying data structure most simple examples to successfully induce a program, this means, they are recursively subsumed with respect to a constructor term rewriting system R, which correctly computes them. However, for a user with a certain program in mind (in the following called target function), this is rather cumbersome. Suppose for example, the user needs a program to replace the German umlaut 'ü' with 'ue'. The user would rather provide the input/output examples as shown in Example 3.3.1 as initial I/O pair than as shown in Example 3.3.2.

Example 3.3.1.

$$\begin{aligned} \textit{“Brühwürstchen”} &\rightarrow \textit{“Bruehwuerstchen”} \\ \textit{“Hühnerbrühe”} &\rightarrow \textit{“Huehnerbruehe”} \end{aligned}$$

Example 3.3.2.

$$\begin{aligned} \textit{“”} &\rightarrow \textit{“”} \\ \textit{“ü”} &\rightarrow \textit{“ue”} \\ \textit{“Br”} &\rightarrow \textit{“Br”} \\ \textit{“Brü”} &\rightarrow \textit{“Brue”} \\ \textit{“Brühe”} &\rightarrow \textit{“Bruehe”}\ . \end{aligned}$$

However, the latter is exactly what IGOR requires. A program to assist the user to create the adequate examples is, what he needs and here ProXSLbE comes to handy.

3.3.1 Preliminaries

Before we can describe the algorithms in more detail, first some definitions have to be made.

Definition 3.3.1 (Strings)**.**
As usually, let a string be a sequence of characters drawn from an alphabet Σ. Furthermore, we let Σ^* denote the set of all finite-length strings formed using characters from the alphabet Σ. The zero-length ***empty string,*** *denoted by ϵ, also belongs to Σ^*. The length of a string x is denoted $|x|$. The* ***concatenation*** *of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of all characters from x followed by all characters from y.*

Definition 3.3.2 (Parts of String)**.**
*We say a string w is a **prefix** of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note, that if $w \sqsubset x$, then $|w| \leq x$. Similarly, we say that a string w is a **suffix** of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \sqsupset x$ that $|w| \leq |x|$. The empty string ϵ is both a suffix and a prefix of every string. Additionally, we say a string w is a **substring** of a string x, denoted $w \lhd x$ if $x = ywz$ for some $y, z \in \Sigma^*$. Where appropriate, sometimes the notation $sub(i, j, x)$ with $i \geq 0$ and $j \leq |x|$ is used to denote the substring of x from index i to j. If w is neither the string x itself or ϵ, then w is called a **proper substring**.*

As already mentioned, the problems applicable for the system ProXSLbE are of functional, recursive programs with one argument of type *substring list*. Example 3.3.3 shows the input and the output as a substring list of our "Hühnerbrühe" problem from Example 3.3.1. The symbol '␣' distinguishes the concrete elements, i. e. the substrings.

Example 3.3.3.

"H␣ü␣hnerbr␣ü␣he" → "H␣ue␣hnerbr␣ue␣he"

Two types of alternating substrings can be identified, namely $constants$ and $separators$. Constants remain unchanged in both input and output, although their position in the list may vary. Contrarily, separators do not change their position, but may be modified. In Example 3.3.3 the constants 'H', 'hnerbr' and 'he' are in both input and output, but the separators change, namely 'ü' to 'ue'. In Example 3.3.4 the constants '19', '03' and '2006' change their position and the separator '.' is modified to '-'.

Example 3.3.4.

"19␣ . ␣03␣ . ␣2006" → "2006␣ - ␣03␣ - ␣19"

Apart from their feature that constants and separators may, or may not change their position or be modified, they have another crucial property. No separator is a substring of a constant.

Definition 3.3.3 (Substring List)**.**
A substring list L over an alphabet Σ is a sequence of alternating constants c and separators s

$$s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n.$$

The separators s_i, for $i = 0, \ldots, n$, are drawn from Σ^*, the constants c_i, for $i = 1, \ldots, n$, are drawn from $\Sigma^* \backslash \{\epsilon\}$, and the number of separators is greater than or equal to 0.[1]

[1] However, note that if the number of separators is 0, the function to induce is trivial, namely the identity.

Note that the property of alternation is solely required to identify substrings in the initial input/output pair. Although IGOR uses the data type substring list too, it requires only that it is a sequence of strings. This is admissible, since the fact that a string has been treated as a constant or separator should not affect the synthesis. However, to allow for optimisation they are distinguished in the IGOR specification, such that constants can turned into variables by IGOR using anti-unification (c. f. B.1).

Obviously, a substring list is nothing else than a string broken up in its sequential substrings, so let us define the representation of a string as a substring list as well as a multiset of substrings. Note here the different notation with normal brackets '(' and ')' for the set and squared brackets '[' and ']' for the sequence.

Definition 3.3.4 (Substring List of a String)**.**
If x is a string from Σ^* such that $x = x_1x_2 \ldots x_n$, then $\mathbf{Subs[x]}$ is a substring list, such that $Subs[x] = x_1, x_2, \ldots, x_n$.

Definition 3.3.5 (Set of Substring of a String)**.**
If x is a string from Σ^* such that $x = x_1x_2 \ldots x_n$, then $\mathbf{Subs(x)}$ is a set of substrings, such that $Subs[x] = \{x_i | i = 1, \ldots, n\}$.

The entirety of all constants as well of all separators also comes in two flavours, a multiset and a sequence. Most of the time it is more appropriate to use them as a set of substrings, but sometimes we will still need the order. Again, note the different notation with normal and squared brackets for the sequence and the multiset.

Definition 3.3.6 (Sequence of Constants)**.**
The sequence of constants of a substring list $L = s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$, denoted by $c[L]$, is the sequence

$$c[L] = c_1, c_2, \ldots, c_n.$$

Definition 3.3.7 (Sequence of Separators)**.**
The sequence of separators of a substring list $L = s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$, denoted by $s[L]$, is the sequence

$$s[L] = s_0, s_1, \ldots, s_n.$$

Definition 3.3.8 (Set of Constants)**.**
The set of constants of a substring list $L = s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$, denoted by $c(L)$, is the set

$$c(L) = \{c_i | i = 1, \ldots, n\}.$$

Definition 3.3.9 (Set of Separators)**.**
The set of separators of a substring list $L = s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$, denoted by $s(L)$, is the set

$$s(L) = \{c_i | i = 1, \ldots, n\}.$$

At the end of our preliminaries we have created the basis to pinpoint the main terms of ProXSLbE as a set of axioms. Let therefore $L = s_0, c_1, s_1, c_2, s_2, \ldots, c_n, s_n$ be an input argument of a target function and $L' = s'_0, c'_1, s'_1, c'_2, s'_2, \ldots, c'_m, s'_m$ the appropriate output.

Axiom 3.3.1 (Function Type)**.**
Functions applicable for ProXSLbE are of type

$$(\textit{substring list}) \rightarrow (\textit{substring list})$$

Axiom 3.3.2 (Non-empty Input Separators)**.**

$$\forall s \in s(I).s \neq \epsilon$$

Axiom 3.3.3 (Uniqueness between Constants and Separators)**.**

$$\neg\exists s, c.\ s \lhd c \textit{ for } s \in s(L) \cup s(L') \textit{ and } c \in c(L)$$

Axiom 3.3.4 (Constants Preservation)**.**

$$c(L') \subseteq c(L)$$

Axiom 3.3.5 (Separator Modification)**.**
Either

$$s[L] = s[L']$$

or

$$\forall i, j.\ s_i = s_j \rightarrow s'_i = s'_j$$

for $i, j = 1, \ldots, m$, where s_i and s_j have been modified to s'_i and s'_j respectively.

3.3.2 From User's Input to IGOR's Examples

The preliminaries should now have been clarified and we can proceed one step further. We already mentioned, that ProXSLbE assumes a *substring list* as the examples' underlying data structure, whereas the user provides only, as demonstrated in the introductory example 3.3.1, input/output examples in form of strings. ProXSLbE's first task is now, as displayed in Example 3.3.5, to preprocess the provided I/O pair and transform it into a substring list by inferring the constants and separators of the substring list.

Example 3.3.5 (From String to Substring List)**.**

"Hühnerbrühe" $\rightarrow$ *"Huehnerbruehe"*

$\vee$

"H␣ü␣hnerbr␣ü␣he" $\rightarrow$ *"H␣ue␣hnerbr␣ue␣he"*

In Definitions 3.3.8 and 3.3.9 we defined the sets of constants and the set of separators. With the aid of the terms and conditions of ProXSLbE (Axioms 3.3.1 - 3.3.5) we can now state some propositions which help us to identify the necessary building blocks of the input/output substring lists by intersecting $Subs(I)$ and $Subs(O)$. However—unfortunately—we have to distinct two cases, i. e. when they separators do change from I to O and when the do not.

Corollary 3.3.1 (Properties of I/O pairs with unchanged separators)**.**
It holds that $s[L] = s[L']$ *(Axiom 3.3.5).*

$$Subs(I) \cap Subs(O) = s(O) \cup c(O) \tag{3.1}$$
$$Subs(I) \backslash Subs(O) = c(I) \backslash c(O) \tag{3.2}$$
$$Subs(O) \backslash Subs(I) = \emptyset \tag{3.3}$$

Corollary 3.3.2 (Properties of I/O pairs with changing separators)**.**
It holds that $\forall i, j.\ s_i = s_j \rightarrow s'_i = s'_j$ *for* $i, j = 1, \ldots, m$ *(Axiom 3.3.5).*

$$Subs(I) \cap Subs(O) = c(O) \tag{3.4}$$
$$Subs(I) \backslash Subs(O) = s(I) \cup (c(I) \backslash c(O)) \tag{3.5}$$
$$Subs(O) \backslash Subs(I) = s(O) \tag{3.6}$$

However, when regarding the I/O strings, the appropriate split of I and O, i. e. the elements of $Subs(I)$ and $Subs(O)$, is unknown. The goal is to find such a $Subs[I]$ and $Subs[O]$ for a given I and O that the properties from Corollary 3.3.1 and 3.3.2 are fulfilled. The strategy to identify the appropriate split-up of the strings is to successively remove common substrings from I and O.

A brute force approach would be to enumerate all possible substrings of I, test for each substring s if $s = subs(i, j, I)$ for some i and j and assure that there is no other substring $s' = subs(i', j', I)$ such that $s \lhd s'$ and $i > i' \vee j < j'$. However, this (i) has an exponential effort, since this is equal to enumerating the powerset of I and (ii) taking always the longest common substrings as separator is not always appropriate.

Consider following example (3.3.6), where the left hand side represents the input and the right hand side the output.

Example 3.3.6. *Initial I/O Pair:*

$$19.01.2003 \rightarrow 2003.01.19$$

Acceptable Substring List:

$$19 \text{␣} . \text{␣} 01 \text{␣} . \text{␣} 2003 \rightarrow 2003 \text{␣} . \text{␣} 01. \text{␣} 19$$

Refusable Substring List:

$$19 \text{␣} .01. \text{␣} 2003 \rightarrow 2003 \text{␣} .01. \text{␣} 19$$

It is obvious, that the period is a more desirable separator for the substring list representing a date, than the sequence ".01.". Approaches using a difference algorithm based on the longest common subsequence or anti-unification of the initial I/O pair would yield no acceptable result, since the order of the constants or even the separators are allowed to change.

For these reasons, I follow a greedy heuristic-lead best-first search approach (Subsubsection 3.3.2.3). Honestly, in the worst case, this also enumerates all possible substrings of I, but in the average it performs much better by testing the best valued substrings first. The modus operandi is as follows. It (i) more or less "guesses" a possible split into substrings using a heuristic (Subsubsection 3.3.2.2), (ii) tests if some substring of the split is in both I and O (Subsubsection 3.3.2.4), if so, (iii) splits I and O using this substring (Subsubsection 3.3.2.5), and (iv)continues on the resulting substrings, after initialising new search trees for each substring until no common substring is found anymore. Finally (v) some adjustments are necessary (Subsubsection 3.3.2.6) to distinguish between the two cases introduced above in Corollary 3.3.1 and 3.3.2. Figure 3.4 gives an overview over the interaction between the different algorithm parts, described in the following.

The search is organised in a tree-like structure called *tupletree* containing *tuples* as nodes. The tuple tree, i. e. the search tree is successively expanded during search by branching.

Definition 3.3.10 (Tuple)**.**
*A **tuple** is an descending ordered sequence of n integers $i_1, \ldots, i_n$ with $i_m \geq i_{m+1}$.*

A tuple represents the set of all possible splits of a string x with length $|x| = \sum_{j=1}^{n} i_j$ into substring of length i_j. A *split* is a specific permutation of a tuple, i. e. of an integer sequence defined by a tuple.

Definition 3.3.11 (Split)**.**
*Let $t = i_1, \ldots, i_n$ be a tuple. A **split** is a permutation of the tuple $i_1, \ldots, i_n$.*

Similarly, the size of a tuple t is defined as the sum over all its integers.

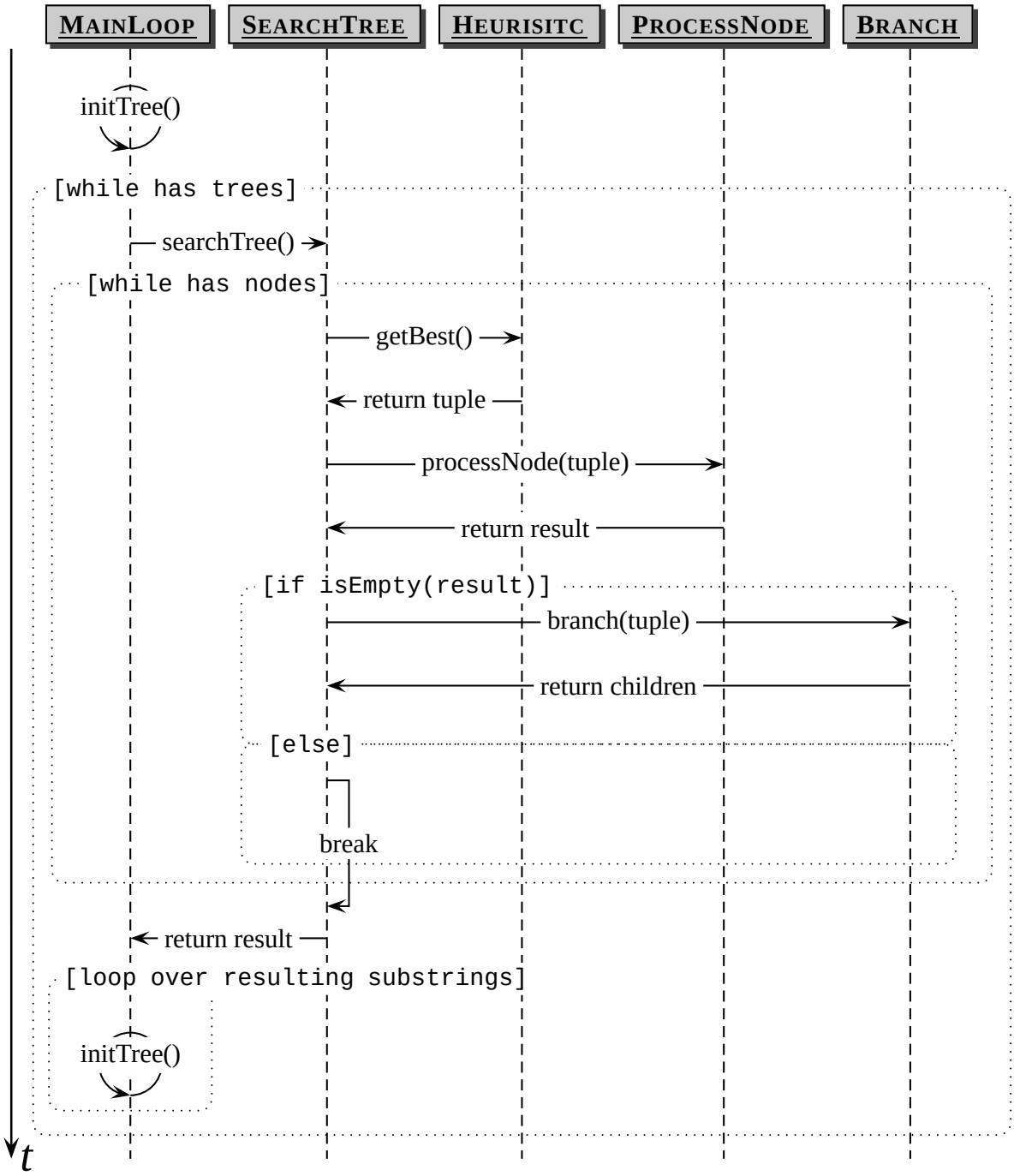

Figure 3.4: Sequence Diagram of Algorithm Interaction

Definition 3.3.12 (Tuple Size)**.**
Let $t = i_1, \ldots, i_n$ be a tuple. The size of t is defined as $\text{SIZE}(t) = \sum_{j=1}^{n} i_j$.

This must not be confused with the length of a tuple $t = i_1, \ldots, i_n$ which is n. Furthermore, if a substring list is represented as a tuple, then the integer i_j's value is the length of the substring at position $j + 1$ in the list.

Definition 3.3.13 (Tupletree)**.**
*A **tupletree** is a tree of all tuples with the same size s. The root node is the 'one'-tuple i_1 with* $\text{SIZE}(i_1) = s$. *All child nodes are created by branching the tuple t using* $\text{BRANCH}(t)$.

Note, that node and tuple in the context of a tupletree are used interchangeably.

3.3.2.1 BRANCH

The function BRANCH applied to a tupletree node t, i. e. a tuple, branches this node and returns a set of the derived child nodes. If the root node is branched, $\text{LENGTH}(t)-1$ new tuples t_i for $i = 2, \ldots, \text{LENGTH}(t)-1$, each with length i and size $\text{SIZE}(t)$ are created. Otherwise, new tuples are created by decreasing an integer i greater than one by one and increasing an integer i' with a lower index than i. This is done for all integers greater than one. Recall hereby, that a tuple is an *descending ordered* sequence of integers.

The function $\text{CREATETUPLE}(n)$ creates an empty tuple of length n. For convenience, in pseudo code a tuple is treated as an array of integers. A sample tupletree for tuples of length is 10 shown in Figure 3.5.

Algorithm 1 BRANCH(t): Branches a tuple t

```
1:  function BRANCH(t)
2:      N ← ∅                                   ▷ initialise return value, i. e. a set of nodes
3:      if LENGTH(t)= 1 then                                                    ▷ t is root
4:          for n = 2, ...,SIZE(t) do
5:              minElemSize ← 1 div n
6:              remainder ← 1 mod n
7:              tuple ← CREATETUPLE(n)
8:              for j = 0, ..., n − 1 do
9:                  tuple[j] ← minElemSize
10:                 if remainder > 0 then                          ▷ distribute remainder
11:                     tuple[j] + +
12:                     remainder− −
13:             N ← N ∪ {tuple}
14:     else                                                          ▷ if t is not root
15:         decrPos ← (index of smallest integer i in t with i > 1) − 1
16:         while decrPos > 0 do
17:             for k = 0, ..., decrPos − 1 do              ▷ iterate over lower indices
18:                 tuple ← t
19:                 tuple[decrPos] − −                          ▷ decrease integer > 1
20:                 tuple[k] + +                ▷ increase integer with lower index
21:                 N ← N ∪ {tuple}
22:             decrPos− −
23:     return N
```

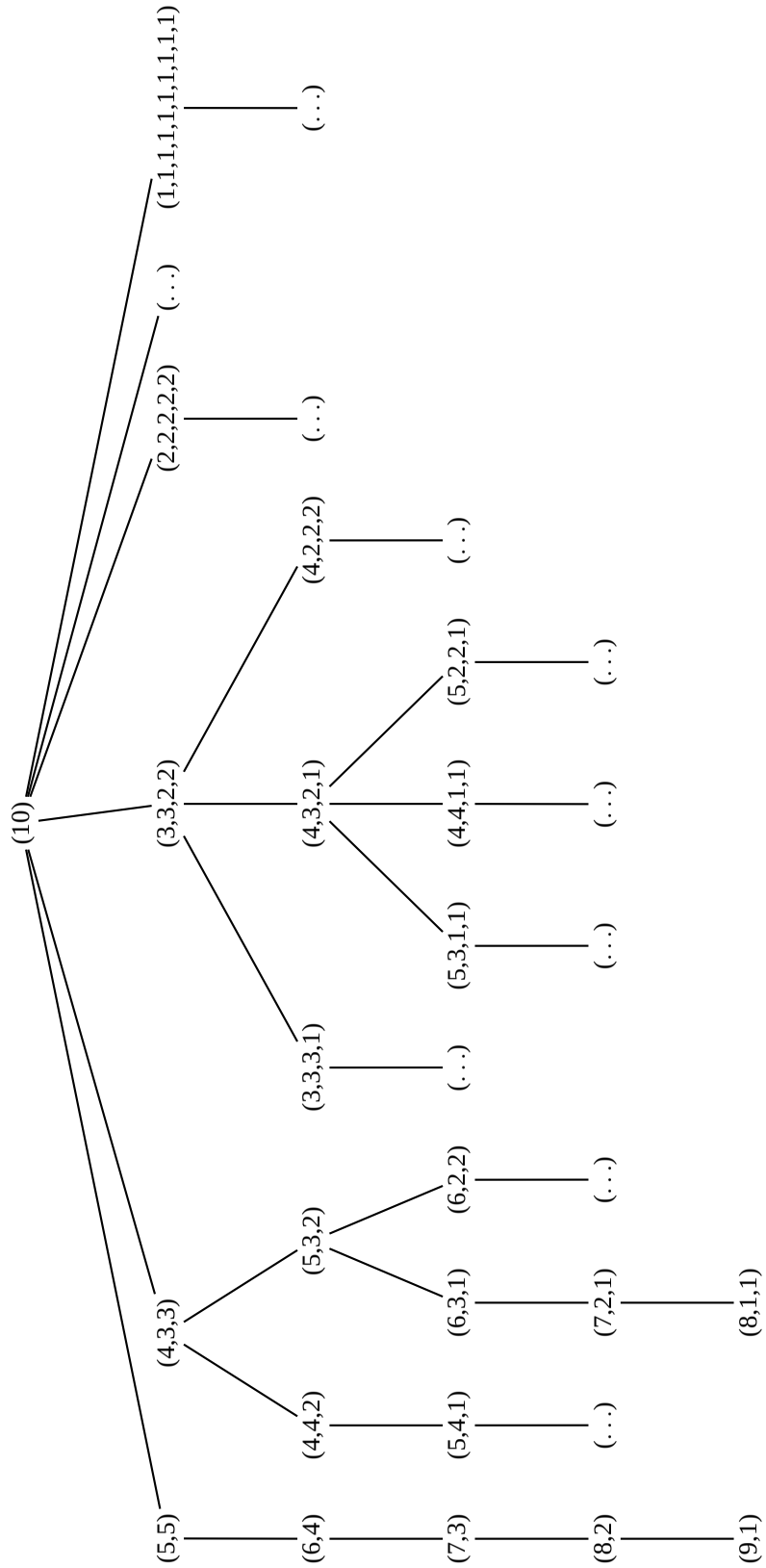

Figure 3.5: The partial tupletree for strings of length 10.

3.3.2.2 HEURISTIC

The heuristic value h of a n-tuple t with n integer $i_1, \ldots, i_n$ is calculated as the root mean square deviation σ_i of the integer values plus the length n of the tuple. The value $\bar{i}$ denotes the mean of the integer values. The heuristic value of a substring list is equal to the heuristic value of the accordant tuple.

$$h(i_1, \ldots, i_n) = \sigma_i + n = \sqrt{\frac{1}{n}\sum_{k=1}^{n}(i_k - \bar{i})^2} + n \tag{3.7}$$

Using the root mean square deviation favours balanced tuples over imbalanced ones. This is motivated by the fact, that highly imbalanced tuples, e. g. the tuple $(9, 1)$, tend to have single-character substrings. This leads to (a) a loss of valuable context information (finding common characters in I and O usually gives no big surprises) and (b) to highly fragmented common substrings, i. e. constants. The latter is especially undesirable, since every constant occurs also in the IGOR input specification and increases its processing time drastically. Also the result of the adjustment tends to become ambiguous the more substrings are involved. Similarly, adding the length of the tuple punishes the long ones to avoid fragmented tuples and many constants. Furthermore, favouring preferably few and balanced substrings attempts to explain as much as possible of O with as few as possible substrings of I.

In order to be admissible for a best-first search, our heuristic h must fulfil the property of monotonicity. So for two tuples t_1 and t_2 with SIZE(t_1) = SIZE(t_2), it must hold that $h(t_1) < h(t_2) \rightarrow b(t_1) < b(t_2)$, where $b(t)$ returns the index of t in the sequence of tuples, created by successively expanding nodes via BRANCH() starting at the root node, i. e. the 'one'-tuple. So, if the heuristic value of a tuple t_1 is smaller than the heuristic value of a tuple t_2 it must be branched before t_2.

Proposition 3.3.1 (Monotonicity)**.** *Let be t_1 and t_2 two arbitrary tuples with equal size. The heuristic h is monotone and it holds that $h(t_1) < h(t_2) \rightarrow b(t_1) < b(t_2)$.*

Proof 3.3.1. *Let us assume the opposite, i. e. Proposition 3.3.1 does not hold and $b(t_1) \not< b(t_2)$ meaning, t_2 has been created before t_1.*

However, this is not possible, when regarding the function BRANCH*(). If t_2 is the root node, this is obviously impossible, as all children $c = c_1, \ldots, c_n$ of the root are already longer so $h(t_1) = 1 < \sigma_{c_i} + n = h(c_i)$ is true for any children c_i of the root node, since $n \geq 2$ and $\sigma_i > 0$. If t_2 is not a root node it is even easier—since in this case—the branching algorithm does not change the tuple size anymore, but increases one element in the tuple and decreases another. This obviously also increases the root mean square deviation of the child tuple.*

This is contradictory to our assumption, so h is monotone.

Proposition 3.3.2 (Optimality with respect to h)**.** *The algorithm is optimal with respect to the heuristic h.*

Proof 3.3.2. *This follows directly from Proposition 3.3.1 and the best-first search. If the best valued tuple, i. e. the tuple with the smallest value of h, is non-empty, a non-empty result is returned, otherwise the next best valued tuple is tested. So the first non-trivial, i. e. non-empty, result is also the best valued result under h.*

3.3.2.3 MAINLOOP

The MAINLOOP procedure invokes successively SEARCHTREE on each tupletree in a queue and initialises new tupletrees for the remaining substrings in the result until the queue is empty. In this procedure, as well as in all following functions, the data type *result* is used to pass the partial or full result after searching a tree.

In pseudo code the fields of a *result* r are accessed via INMINUSOUT(r), OUTMINUSIN(r), and COMMONS(r). The function RESULT() returns an empty result, RESULT(i, o, c) a non-empty one, where i, o, and c are lists of strings and initialise the fields "inMinusOut", "outMinusIn" , and "commons" respectively.

Definition 3.3.14 (Result)**.**
*A **result** is a record-like data structure containing three lists of strings "inMinusOut" (corresponding to Equations (3.5) and (3.2)), "outMinusIn" (corresponding to Equations (3.6)and (3.3)), and "commons" (corresponding to Equations (3.4) and (3.1)).*

If two results r_1 and r_2 are compared, the criteria from Definition 3.3.15 are applied in order as stated. For the ordering relation between results, the symbols $\gg$ and $\ll$ are used with their usual semantics.

Definition 3.3.15 (Result Comparison Criteria)**.**
Let r_1 and r_2 be two results.

1. *An empty result is always inferior than a non-empty one.*
2. *Let $C_1 =$*COMMONS(r_1) *and $C_2 =$*COMMONS(r_2)*, then*

$$r_1 \ll r_2 \text{ iff } \sum_{i=1}^{n} \text{LENGTH}(c_i) < \sum_{j=1}^{m} \text{LENGTH}(c_j),$$

   for all $i = 1, \ldots, |C_1|$ and $j = 1, \ldots, |C_2|$, with c_k denoting the k^{th} element.
3. *Let h_i be the heuristic value of* COMMONS(r_i)*, then*

$$r_1 \ll r_2 \text{ iff } h_1 < h_2.$$

4. $r_1 \ll r_2$ *iff length of* COMMONS(r_1) < COMMONS(r_2).

Algorithm 2 MAINLOOP: The outermost loop over all tupletrees

```
 1: tupleTrees ← FIFOQUEUE
 2: inMinusOut ← LIST                          ▷
 3: outMinusInt ← LIST                         ▷ global variables, read by PROCESSNODE
 4: commons ← LIST                             ▷
 5: procedure MAINLOOP(I, O)
 6:     INSERT(I, inMinusOut)
 7:     INSERT(O, outMinusInt)
 8:     ENQUEUE(INITTREE(I),tupleTrees)
 9:     while ¬ISEMPTY(tupleTrees) do
10:         result ← SEARCHTREE(DEQUEUE(tupleTrees))
11:         if ¬ISEMPTY(result) then
12:             for all Strings s ∈ INMINUSOUT(result) do
13:                 ENQUEUE(INITTREE(s),tupleTrees)
14:             inMinusOut ← INMINUSOUT(result)
15:             outMinusIn ← OUTMINUSIN(result)
16:             INSERT(INMINUSOUT(result), commons)
```

A justification for the first rule is just straight forward. In the second rule, the total length of the found constants is used as an indicator how much of the output is already explained by this result. The third rule favours now less fragmented ones and the last favours results with few constants for the same reason.

The function INITTREE$(string)$ initialises a new tupletree with a 'one'-tuple with size = LENGTH$(string)$ and ENQUEUE(t, q) enqueues a tree t into a queue q, whereas DEQUEUE(q) dequeues the first element from the queue q, and INSERT(e, l) inserts an element e into a list l. I and O are the input and the output string, respectively.

After the algorithm has finished, the value of variable $commons$ conforms to Equation (3.1)/(3.4), $inMinusOut$ conforms to Equation (3.2)/(3.5) and $outMinusIn$ to (3.3)/(3.6) in Corollary 3.3.1/3.3.2.

However, some adjustments are necessary (described in paragraph ADJUSTMENT). For the case that the constants where not modified, the separators sought-after are in $commons$ but still spoilt by the common constants (c. f. Equation 3.1). If the constants where modified, $outMinusIn$ contains the separators for the output (c. f. Equation 3.6), but the separators in $inMinusOut$ are again spoilt with common constants (c. f. Equation 3.5). The adjustment algorithm described in Paragraph ADJUSTMENT 3.3.2.6 now extracts the desired separators. The variable $outMinusIn$ is used as an indicator, whether the constants were modified (it is non-empty) or not (it is empty).

Algorithm 3 SEARCHTREE(tt): Searches a tupletree tt for the best tuple

```
 1: function SEARCHTREE(tupletree)
 2:     result ← RESULT()
 3:     while ¬ ISEMPTY(tupletree) do
 4:         best ← BEST(tupletree)
 5:         target ← TARGET(tupletree)
 6:         result ← PROCESSNODE(first, target)
 7:         if ¬ ISEMPTY(result) then
 8:             return result
 9:         else
10:             BRANCH(best)
11:     return result
```

3.3.2.4 SEARCHTREE

The function SEARCHTREE traverses a tupletree best-first using the heuristic described under paragraph **HEURISTIC**. The function BEST(tt) removes the best valued, i. e. the tuple with the smallest value, from a tree tt and returns it. So successively the currently best tuple is processed until the function PROCESSNODE returned a non-empty result, which is then returned to the caller. This result then is the first non-empty result, i. e. the best with respect to the heuristic h. The function TARGET(tt), where tt is a tupletree, returns the target string with which the tree has been initialised using INITTREE($target\_string$).

3.3.2.5 PROCESSNODE

The function PROCESSNODE is the inner-most algorithm and searches for the best matching substring of a specific split. Recall, that a *tuple* is a set of possible splits, with the same value under the evaluation heuristic h. So for every permutation of $tuple$ it splits the $target\ string$ (i. e. I or some substring of I) into appropriate substrings S and searches for every substring $s \in S$ if it is also a substring of O. So, it performs actually the difference as described in Equation (3.3)/(3.6). However, all permutations have the same value under h, so the results of the tuples are compared (line 18 of algorithm 4) according to Definition 3.3.15 and the currently best result is stored in the variables $tempXYZ$ (line 4-6).

The function SPLIT(s, w) splits a string s by the substring w, so if $s = xwz$, then it returns a list which elements are x and z, where $x, t \neq \epsilon$. If $w \not\triangleleft s$, s is returned. The function SPLITALL(l, w) applies SPLIT(s, w) on each string s in a list of string l.

Algorithm 4 PROCESSNODE(t, s): Process the tuple t using the target string s

```
 1: function PROCESSNODE(tuple, target)
 2:     result ← RESULT()
 3:     for all permutations p of tuple do
 4:         tempCom ← LIST
 5:         tempOMI ← LIST
 6:         tempIMO ← outMinusIn            ▷ global variable, defined in MAINLOOP
 7:         substrings ← SPLIT(target, p)          ▷ split target according to p
 8:         for all String out ∈ outMinusIn do
 9:             for all String sub ∈ substrings do
10:                 if sub ◁ out then
11:                     INSERT(SPLIT(out, sub), tempOMI)
12:                     tempIMO ← SPLITALL(tempIMO, sub)
13:                     INSERT(sub, tempCom)
14:                 else
15:                     INSERT(out, tempOMI)
16:         if ¬ISEMPTY(tempCom) then
17:             tempResult ← RESULT(tempIMO, tempOMI, tempCom)
18:             if tempResult ≫ result then
19:                 result ← tempResult
20:     return result
```

3.3.2.6 ADJUSTMENT

As already mentioned, the final results cannot be used yet. For the case that the constants were not modified, the separators in $commons$ are spoilt by the common constants (c. f. Equation 3.1). If the constants were modified, the separators in $inMinusOut$ are also spoilt with common constants (c. f. Equation 3.5). The adjustment algorithm now tries to detect and correct these flaws. To extract the desired separators the most common prefixes (mcp), or suffixes (mcs) respectively are searched.

But why is it sure, that the sought-after separators are either prefixes or suffixes? To put it in one sentence: It is not! However, if the separators are not too long, experience has shown that they are less likely to be split and a split is performed before or after a separator with a (part of a) constant attached.

As it is usually not known, whether the prefixes P or the suffixes S contain an appropriate separator, both are computed and from these one string s is chosen as a separator according to the rules in Definition 3.3.16. In case of ambiguities, the next rule applies for the remaining candidate separators.

Definition 3.3.16 (Decision Criteria for Selecting Separators)**.**

1. *choose $P \cap I$*
2. *choose the most common string*
3. *choose the longest string*

The adjustment algorithm tests neighboured strings at position i and j in an array for a common prefixes. If the strings at position i and j have a common prefix, it is written at position i, the string at position j is deleted and the index j incremented by one. So now the strings at position i, i. e. the previous prefix, and $j = j + 1$ are compared. If the strings at position i and j have no prefix in common and $i + 1 = j$ (neighboured positions) the string at position i is deleted, if i and j are not neighboured $(i + 1 < j)$ the string at position i already contains a previously found prefix and is therefore not deleted. Finally, both i and j are increased by one. When the algorithm has finished, i. e. all array positions were tested, the array contains either empty strings or prefixes which at least two strings had in common. The function returns an array of pairs, each containing a found prefix and the number of strings that had this prefix in common. For the mcs, the mcp on reversed strings is computed.

The function CHARAT(pos, s) returns the character at position pos in the string s and PROCESS(a) groups and sorts all prefixes in the array, removes empty strings and returns an array of pairs, each containing a found prefix and its occurrence.

Example 3.3.7 shows the current state of affairs. After analysing the input and output strings we have identified the building blocks of our input and output examples. These can now be used to create further I/O examples (last part of Example 3.3.7) which can be processed by IGOR. Therefore, the found building blocks are used to create all substring lists up to a fixed length by combining the found constants and separators. So the first list generated is the empty list, followed by all lists with on constant, then one constant and one separator, two constants and one separators and so on. From these generated input examples, the user chooses appropriately, completes the output and passes them to a component of the program called “specification builder” which writes them into a specification file for IGOR. This step is discussed in more detail in section 4.1 and 4.2. A complete IGOR-specification together with some explanation is given in Appendix B.1.

Algorithm 5 Calculates the most common prefixes of an array of strings

```
1:  function MCP(strings)                                   ▷ array of strings
2:      i = 0
3:      j = 0
4:      while j < LENGTH(strings) do
5:          pre ← ϵ
6:          for pos = 0, ..., min(LENGTH(strings[i]), LENGTH(strings[j])) do
7:              if CHARAT(pos, strings[i]) = CHARAT(pos, strings[j]) then
8:                  pos += CHARAT(pos, strings[i])
9:              else
10:                 break
11:         if pre ≠ ϵ then
12:             strings[i] ← pre
13:             strings[j] ← ϵ
14:         else
15:             if j = i + 1 then
16:                 strings[i] ← ϵ
17:             else
18:                 i ← j
19:         j++
20:     if j=i+1 then
21:         strings[i] ← ϵ
22:     return PROCESS(strings)
```

Example 3.3.7 (Substring List to IGOR's specification)**.**

"Hühnerbrühe" → *"Huehnerbruehe"*

"H␣ü␣hnerbr␣ü␣he" → *"H␣ue␣hnerbr␣ue␣he"*

"" → *""*

"H" → *"H"*

"ü" → *"ue"*

"Hü" → *"Hue"*

"hnerbr␣ü" → *"hnerbr␣ue"*

"H␣ü␣hnerbr" → *"ue␣hnerbr"*

"ü␣hnerbr␣ü' → *"ue␣hnerbr␣ue"*

...

3.3.3 And Further to XSL Transformation

In the next step, after the IGOR specification has been created and IGOR has generated a functional program, this output has to be parsed and transformed into an XSL Transformation template. Examples of IGOR's output are given in Appendix B.2. The grammar in Figure 3.6 describes IGOR's syntax in extended Backus-Naur-Form. Non-terminal symbols are shown slanted, terminal symbols in boldface typewriter font. Capitalised non-terminals resolve to names or identifiers. Since their rules are straight-forward and translate to simple strings in XSLT, they are omitted here. Braces denote expressions that can be omitted or repeated, normal brackets surround alternatives and squared brackets indicate an option. In the following the translation of each non-terminal symbol into XSL fragments is described.

equations → *equation* { *equation* }
equation → **eq** *lhs* **=** *expression* **[none] .**
| **ceq** *lhs* **=** *if_then* **[none] .**
lhs → *Fun_Name* **[** *subslist* **]**
expression → *sub*
| *subslist*
| *function*
function → *Fun_Name* **[** *subslist* **]**
| *Fun_Name* **[** *function* **]**
if_then → *expression* **if** *cond* { ∧ *cond* }
cond → **'_==_[** [**not**] *expression* **,** [**not**] *expression* **]=** *Bool*
subs → *Var_S*
| *Const_S*
subslist → *Var_L*
| *Empty_L*
| **'__[** (*subs* | *function*) **,** (*subslist* | *function*) **]**

Figure 3.6: The grammar of IGOR's syntax in Extended Backus-Naur-Form.

3.3.3.1 Equations

The functional program IGOR returns is represented as a list of equations, which are translated into named templates in XSL for every group of equations with the same function name on the left-hand side. The grammar rule for $equations$ therefore generates the surrounding `<xsl:stylesheet>` which is shown in Listing 3.4. Apart from the usual header it includes an additional namespace declaration (line 3/4) for special IGOR functions defined in a separate stylesheet (line 7). These functions include a pattern matching function (Lisiting A.2 in Appendix A) to simulate the pattern matching

as employed by IGOR, as well as a function to transform a string into a substring list (Lisiting A.2 in Appendix A). For this purpose a global variable is introduced which stores the separators identified during the preprocessing as a sequence of strings (line 8).

Also two default template rules are contained. A most general one (line 11 - 15), which matches everything if no other rule applies. This rule just copies the matched node and applies again all templates on its child nodes. A second, more specific rule matches only the target node, copies it and calls the target function, i. e. the synthesised function, passing the value of its text node transformed into a substring list as argument. For this, the function `igor:separate` is called using the global variable with the collected separators.

Listing 3.4: The surrounding stylesheet definition.

```
1 <?xml version="1.0"encoding="UTF-8"?>
  <xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:igor="http://www.cogsys.wiai.uni-bamberg.de"
5   version="2.0">
    <xsl:output method="xml"indent="yes"/>
    <xsl:include href="patternmatcher.xsl"/>
    <xsl:variable name="seps"select="('S1','S2')"/>

10  <!--Copy non-target nodes-->
    <xsl:template match="/|*|@*|text()">
      <xsl:copy>
        <xsl:apply-templates select="*|@*|text()"/>
      </xsl:copy>
15  </xsl:template>

    <!--Entrance function for target node. Copies everything-->
    <!--but text and passes the text of the node 'target_node'-->
    <!--to the function template-->
20  <xsl:template match="target_node">
      <xsl:copy>
        <xsl:call-template name="target-function">
          <xsl:with-param
           name="input"
25         select="igor:separate(text(),$seps)"/>
        </xsl:call-template>
      </xsl:copy>
    </xsl:template>
```

```
    <!-- other named templates -->
30 </xsl:stylesheet>
```

3.3.3.2 Equation

The translation pattern for the *equation* grammar rule is quite simple (Listing 3.5). A named template element with one argument is created and a `<xsl:param>` element is attached. Also a `<xsl:choose>` element is added which distinguishes later the different alternative patterns. Naturally, only one template for each left hand side function is created. For different equations of the same function, but with different patterns different `<xsl:when>` elements under the `<xsl:choose>` node are created.

Listing 3.5: The translation pattern for an *equation*.

```
1 <xsl:template name="Fun">
    <xsl:param name="input"/>
    <xsl:choose>
    <!-- one alternative for every pattern -->
5   </xsl:choose>
  </xsl:template>
```

3.3.3.3 Left Hand Side

The pattern on the left hand side of an equation is directly transformed into a slightly more simple pattern that is applicable for the function `igor:match-pattern`. The pattern may consist of patterns for constant strings of characters encoded in their hexadecimal value, patterns which match any substring `('X')`, patterns which match any substring list `('Xs')`, and patterns for the empty list `('E')`.

The function `igor:match-pattern` performs a pattern matching and returns a boolean value, indicating if the passed string matches the passed pattern. The computed boolean value is then passed to the `test` attribute of a `<xsl:when>` node. As child elements, `<xsl:variable>`-nodes for every variable in the input pattern are added and their appropriate value in terms of the matched pattern is assigned. The XSL code is shown in Listing 3.6.

Listing 3.6: The translation pattern for *lhs*

```
1<xsl:when test="igor:match-pattern(
                 ('Const','X','Xs'),$input)=true()">
    <xsl:variable name="X0_Subs"select="$input[2]"/>
    <xsl:variable name="X1_SubsList"
```

```
5                 select="subsequence($input,3)"/>
    <!-- results of evaluation of the right hand side -->
  </xsl:when>
```

3.3.3.4 Constants and Variables

The rules *subs* and *subslist* describe the syntax of a substring and a substring list, respectively, in IGOR's output. Except for the last alternative for *subslist*, they consist either of a variable or a constant of particular type, which are simply put into XSL using the element `<xsl:value-of>`. Line 1 in Listing 3.7 shows this for a constant, where `Const` is the string value of this constant. Similarly, line 2 for a variable, where `Var` is the name of a previously defined variable. If the last rule of *subslist* applies, i. e. *subslist* is a more complex construct composed of at least a substring and a rest list, it is translated to XSL by applying the appropriate rules recursively. So Listing 3.7 would also be an example translation of a list with two elements.

Listing 3.7: Variables and constants translated to XSL

```
1 <xsl:value-of select="'Const'"/>
  <xsl:value-of select="$Var"/>
```

3.3.3.5 Functions

Recall, that ProXSLbE can only deal with functions over substring list with a single argument of the same type, so we have two grammar rules to translate into XSL. If the argument is a substring list, first the function's arguments are evaluated and stored in a element `<xsl:variable>`. The corresponding named template for the function itself is called by applying `<xsl:call-template>` and passing the stored function arguments (Listing 3.8). Although we probably stored a distinguishable substring in our variable, i. e. a sequence of strings, when reading it, we unfortunately get again plain text. So we have to transform the argument into a substring list again, by invoking `igor:separate` on the argument.

Listing 3.8: Function with substring list as argument translated to XSL

```
1 <xsl:variable name="Fun_arg">
    <!-- XSL fragments of translated arguments -->
  </xsl:variable>
  <xsl:call-template name="Fun">
```

```
    <xsl:with-param
       name="input"
       select="igor:separate($Fun_arg,$seps)"/>
  </xsl:call-template>
```

If another function is passed as an argument, first this subfunction is evaluated and translated into XSL. Then the result of the subfunction call, i. e. the `<xsl:call-template>` node is wrapped into variable and passed to the template call of the main function after preprocessing the input string. The XSL translation pattern of a function call with a function as argument is shown in Listing 3.9.

Listing 3.9: Function with subfunction as argument translated to XSL

```
<xsl:variable name="SubFun_arg">
    <!-- XSL fragments of arguments of subfunction-->
  </xsl:variable>
  <xsl:variable name="Fun_arg">
    <xsl:call-template name="SubFun">
      <xsl:with-param
       name="input"
       select="igor:separate($SubFun_arg,$seps)"/>
    </xsl:call-template>
</xsl:variable>
  <xsl:call-template name="Fun">
    <xsl:with-param
     name="input"
     select="igor:separate($Fun_arg,$seps)"/>
</xsl:call-template>
```

3.3.3.6 Conditions

According to the grammar rules of Listing 3.6 a conditional right hand side if_then consists of an $expression$ which computes the result of this equation and one or more conditions connected by a logical "and". Each condition $cond$ checks again for equality of two *expressions*. Listing 3.10 shows how this is transformed into XSL. The translation of each such conditional argument is stored in a variable, where the actual conditional check is done using the `<xsl:if>` element. In its `test` attribute the variables holding the conditional arguments are compared accordingly. If there is more than one condition, they are also connected with a logical `and`, and negated using `not` if stated so.

Listing 3.10: XSL translation pattern for Conditional expressions

```
<xsl:variable name="cond_arg_1">
  <!-- result of evaluation of condition argument 1 -->
</xsl:variable>
<xsl:variable name="cond_arg_2">
  <!-- result of evaluation of condition argument 2 -->
</xsl:variable>
<xsl:variable name="cond_arg_3">
  <!-- result of evaluation of condition argument 3 -->
</xsl:variable>
<xsl:variable name="cond_arg_4">
  <!-- result of evaluation of condition argument 4 -->
</xsl:variable>
<xsl:if test="($cond_arg_1=$cond_arg_2)
          and
          not($cond_arg_3=$cond_arg_4)">
  <!-- results of evaluation of 'expression' -->
</xsl:if>
```

Chapter 4

The Learning System ProXSLbE

The methodologies described in Section 3.3 were implemented in a prototypical system called ProXSLbE. The system is integrated as a plugin in the open source programmer's editor *jEdit*.

Section 4.1 gives an overview of the system's architecture and its usage while the second Section 4.2 describes the example problems the system has been tested with and their results.

4.1 ProXSLbE's Mode of Operation

In the previous chapter ProXSLbE's mode of operation has been described from an internal point of view with focus on the algorithms. Now we look at it from the user's point of view and how a user interacts with the system. The usual steps necessary to be undertaken to synthesise an XSLT stylesheet are depicted in Figure 4.1 and described in the following, where the numbers refer to the steps in Figure 4.1. Appendix C shows the relevant screenshots.

❶ When the user opens an XML file with *jEdit*, it is displayed in the editor's buffer. Simultaneously, ProXSLbE parses the file and creates a tree view of it which is displayed on the left side of the main view of ProXSLbE (c.f. Figure C.1 in Appendix C).

❷ By navigating through the tree view, nodes can be chosen. Their content is displayed in the text field at the bottom left of ProXSLbE main view (c.f. Figure C.1 in Appendix C) and if it is a text node, i. e. the value of the selected node is simple text, its value can be modified.

❸ The resulting I/O pair, i.e. the string before and after modification, is sent to ProXSLbE's core algorithm described in Section 3.3 which tries to induce an appropriate split into substrings.

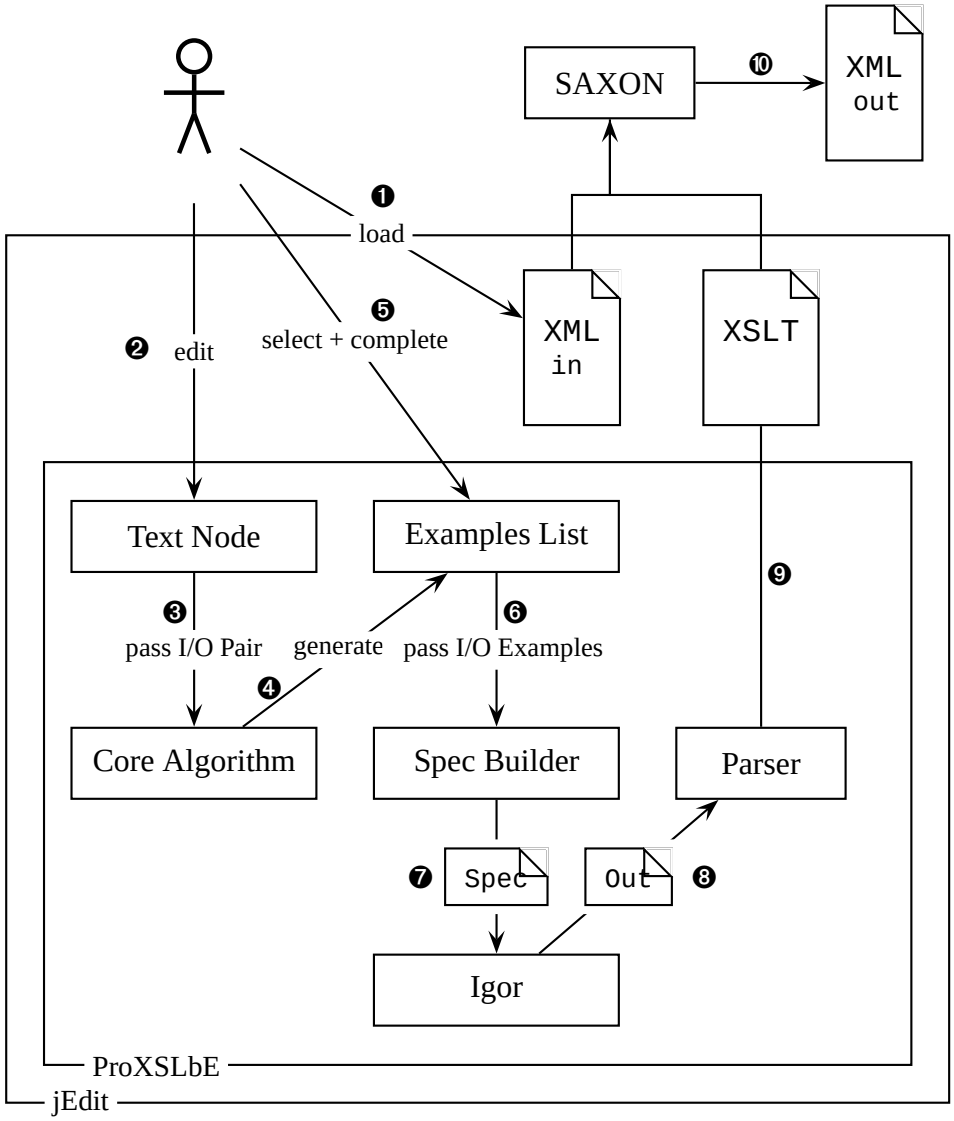

Figure 4.1: ProXSLbE's Architecture and Mode of Operations

❹ From the substrings identified by the core algorithm new example inputs are created. Therefore all substring lists up to a fixed length are generated by combining the found constants and separators. So the first list generated is the empty list, followed by all lists with one constant, then one constant and one separator, two constants and one separators and so on.

❺ The user now chooses appropriate inputs and completes them with a fitting output.

❻ The I/O examples defined by the user are sent to the specification builder.

❼ These I/O examples are written into a specification for the IGOR system.

❽ The created specification file (c.f. Appendix B.1) is then sent to the IGOR system. IGOR turns constants into variables via anti-unification and starts synthesis.

❾ The output is transformed into an XSLT stylesheet by a parser as described in Section 3.3.3.

❿ Finally, the user has to pass the input XML file together with the XSLT document to an XSLT processor (SAXON) to perform the demonstrated transformation.

4.2 Tests

To evaluate the performance of the described methodology, the implemented prototype was tested against various example problems. The following two sections introduce first the problem classes and describe the test results (4.2.1) and then a summary is given (4.3).

4.2.1 Problem Classes

Since we are in a text-editing context, the test problems are categorised in three classes that represent the common editing operations: *replace*, *insert* and *delete*. Furthermore, to make allowance for the special underlying structure of all input examples, namely the substring list, a fourth class with special list functions has been included, too. In the following each tested problem is introduced with a short description and where appropriate the initial input/output pair, the desired separators and constants, and the final I/O examples passed to the system are shown.

4.2.1.1 Replace

The replace-class contains all problems which require to replace a separator substring by another substring. So a function F maps an input substring list to an output substring list

$$F(s_0, c_1, s_1, \ldots, c_n, s_n) \rightarrow s'_0, c_1, s'_1, \ldots, c_n, s'_n$$

such that every separator s_i is modified to a separator s'_i in consideration of Axiom 3.3.5. In the following, three example problems are presented with their initial I/O pair, i.e. the original and the edited string, the separators and the constants found by ProXSLbE, and the final I/O examples passed to IGOR. Keep in mind that their inputs are generated by

the system, but the user chooses the input and completes the output. Within the replace-class, three problems were tested: *replace umlauts*, *reformat a date* and *replace certain separators* in a substring list.

Replace Umlaut aims to replace the umlaut character "ü" by "ue". This is the running example used throughout this text. The user has edited ´´Hühnerbrühe" to "Huehner-bruehe" and ProXSLbE correctly identified the constants and separators stated below.

For every structural unique substring list (separator, constant, separator-constant, constant-separator etc.) two examples are given to allow for optimisation by IGOR using anti-unification. So as already mentioned, IGOR tries first to transform the constants into variables. Also some examples using the separator "ue" in the input as well as the empty string ϵ are provided for completeness.

initial I/O Pair
Hühnerbrühe → Huehnerbruehe

Separators
ü, ue

Constants
H, hnerbr, he

final I/O Examples

ϵ	→ ϵ	Hü	→ Hue
ü	→ ue	Hue	→ Hue
ue	→ ue	ühe	→ uehe
he	→ he	üH	→ ueH
H	→ H	üheü	→ ueheue
heü	→ heue	üHü	→ ueHue
heue	→ heue		

Reformat Date simply tries to change the format of a date and replaces all forward slashes by hyphens. ProXSLbE also correctly identified the desired constants and separators. The choice of the input examples is analogue to the previous *replace umlaut* problem.

initial I/O Pair
10/09/2007 → 10-09-2007

Separators
/, -

Constants
10, 09, 2007

final I/O Examples

ϵ	→ ϵ	09	→ 09
/	→ -	09/	→ 09-
-	→ -	09-	→ 09-
10	→ 10	/09	→ -09
10/	→ 10-	-09	→ -09
10-	→ 10-	/09/	→ -09-
/10	→ -10	/10/	→ -10-
-10	→ -10		

Replace Certain Separators is quite similar to the previous problems. The goal is just to modify certain separators and replace each period by a hyphen. Again, the choice

of the input examples follows the previous problems.

initial I/O Pair
1-2.3-4.5 → 1-2-3-4-5

Separators
-, .

Constants
1, 2, 3, 4, 5

final I/O Examples

$\epsilon \rightarrow \epsilon$	09 → 09
. → -	.1 → -1
- → -	-1 → -1
1 → 1	.2 → -2
2 → 2	-2 → -2
1. → 1-	1.2 → 1-2
1- → 1-	2.1 → 2-1
2. → 2-	.1. → -1-
2- → 2-	.2. → -2-

4.2.1.2 Insert

Among the class of insert-problems are all functions F of the form

$$F(s_0, c_1, s_1, \ldots, c_n, s_n) \rightarrow s'_0, c_1, s'_1, \ldots, c_n, s'_n$$

where some separators s_i are modified according to Axiom 3.3.5 to separators s'_i such that $s'_i = xs_iy$ and x or y could be the empty string. This modification of the separator corresponds to inserting a string before or after the separator. Axiom 3.3.5 demands, that if one separator s of the input is modified, all other separators in the input, which are equal to s, are modified in the same way as well. If for a modification both x and y are empty, the separator remains unchanged.

4.2.1.3 Delete

This kind of problem is on hand if a function F is applied which is of the form

$$F(s_0, c_1, s_1, \ldots, c_n, s_n) \rightarrow s'_0, c_1, s'_1, \ldots, c_n, s'_n,$$

where some separators s_i are modified according to Axiom 3.3.5 to a separator s'_i such that $s'_i = \epsilon$.

4.2.1.4 List Functions

As already mentioned, this section describes the synthesis of typical functions over lists as for example reversing a substring list, deleting all substrings but the last of a list or deleting all substrings on an odd position.

Reverse This function reverses a substring list, i.e computes the palindrome on basis of substrings.

initial I/O Pair
1-2-3-4-5 → 1-2-3-4-5

Separators
-

Constants
1, 2, 3, 4, 5

final I/O Examples

$\epsilon \rightarrow \epsilon$	1-2-3 → 3-2-1
3 → 3	2-3-1 → 1-3-2
2 → 2	1-2-3-4 → 4-3-2-1
1-3 → 3-1	2-3-4-1 → 1-4-3-2
3-2 → 2-3	

Last This function returns the last element of a list. Its intended purpose was to strip away leading parts of formatted text, so for example only retain the year of a date string or something similar.

Odd is another function from the repertoire of list modifications and removes all elements at an odd position from the list.

4.2.2 Results

The Table 4.1 shows the results of the described problems. Since the synthesis performed by IGOR draws on most of the processing time and IGOR finished computation only by two problems, no running times are listed. For *reverse* and *replace umlaut* the system took about half a minute.

A failure due to SAXON occurs when the XSL processor refused the stylesheet due to too many apply-template calls and recursive function calls. IGOR is accountable for the failure, if it did not stop computation, although the provided examples were appropriate. ProXSLbE's underlying theoretic methodology causes a failure if it is not possible for the user to choose the desired examples from the generated inputs due to incorrect identified separators and constants.

4.2.2.1 Replace

Replace Umlaut The listings in Appendix B show the generated specification file (Listing B.1), IGOR's output (Listing B.3), and the generated XSLT stylesheet (Listing B.5).

Inspecting IGOR's output and the generated stylesheet no apparent bugs could be found and the recursion seems to be finite. Only in the if-statements IGOR produced redundant code which consists of duplicate boolean conditions. However, the XSL processor SAXON fails to process this stylesheet due to a stack overflow and stops with

Class	Problem Name	Success	Failure due to ... SAXON	IGOR	methodology
Replace	*Umlaut*		×		
	Reformat Date			×	
	Specific Separators			×	
Insert					×
Delete					×
List Functions	*Reverse*	×			
	Last				×
	Odds				×

Table 4.1: Overview over ProXSLbE's results

the error message "Too many apply-template calls". In several developer mailing lists other people encountered the same problem and finally implemented a work-around, though this was not applicable for the generated stylesheets.

Reformat Date ProXSLbE correctly identified the desired constants and separators, but whatever I/O examples were provided no correct result could be achieved. Either the synthesised function was the identity function, or IGOR did not stop synthesis. This is even more puzzling, since the provided example were did structurally not differ from the example used for *replace umlaut*.

Replace Certain Separators, similarly to the "Reformat Date" problem, was also not successful. Providing less examples the result was the identity function, with more examples IGOR did not stop. It is pleasing that ProXSLbE at least identified the correct constants although the role of the hyphen is ambiguous, because when only regarding the input it could be either a separator or a part of a constant. Only in context with the output it makes more sense to treat it as a separator.

4.2.2.2 Insert

At the first glance there is no reason why ProXSLbE should have difficulties to deal with such problems, since all Axioms (3.3.1–3.3.5) seem to be fulfilled. However it was not possible, apart from the trivial case (i. e. the identity function), to synthesise any of such functions. Substituting s' in the formula above by xsy makes clear why. The I/O pair passed to the core algorithm looks as follows

$$s_0c_1s_1 \ldots c_ns_n \longrightarrow xs_0yc_1xs_1y \ldots c_nxs_ny,$$

but now the substrings which are considered as separators do not change. Depending on where the new substring is inserted, the algorithm attaches the actual separators s_i to the preceding or succeeding constant or identifies it as a sole constant and therefore adds them to the set of common substrings (i. e. $Subs(I) \cap Subs(O)$). Howsoever, the inserted string is identified as a new separator in the output.

So after the algorithm has finished the search for substrings, the variable $constants$ in the final $Result$ (c.f. Definition 3.3.14) contains the intended separators ($s(i)$) and the constants of the input ($c(I) \cup c(O)$) probably concatenated. The field $inMinusOut$ is empty and $outMinusIn$ contains the inserted strings x and y. In the adjustment step $inMinusOut$ is searched for separators, because the heuristic assumes "changed separators" due to the decision criteria ($outMinusIn$ is not empty, c.f. Corollary 3.3.2). Obviously no additional separators are found and input examples are generated using wrong separators and constants.

Probably this could be fixed by making an additional case distinction in the adjustment step and searching for separators in the field $constants$ of the $Result$ if $inMinusOut$ is empty.

4.2.2.3 Delete

At first glance, this looks like a special case of a replace-problem and thus following the properties described in Corollary 3.3.2, but this is not the case.

Consider for example following I/O pair

$$\text{H␣ü␣hnerbr␣ü␣he} \rightarrow \text{H␣hnerbr␣he},$$

with the desired split already illustrated. It is obvious that all substrings in the output are also contained in the input and therefore, after the ProXSLbE's core algorithm has processed this pair, the field $outMinusIn$ is empty. Since the emptiness of $outMinusIn$ serves as an indicator to distinguish between changed or unchanged separators, we have a similar problem as detected for the insert-class. Now "unchanged separators" are assumed, and the adjustment heuristic searches in the common constants for appropriate separators.

This problem could easily resolved if the substrings in $inMinusOut$ would be taken instead which contains in fact exactly the desired separators. However, the problem is to distinguish between the case of deleting separators and the case were a function keeps the separators are unchanged (c.f. *reverse*). This is though, only carried out on basis of the common and non-common substring in input and output not possible. Consequently no function that delete certain parts of the input could be synthesised.

4.2.2.4 List Functions

Reverse This is the only problem which could be completely synthesised by ProXSLbE. The specification passed to IGOR is shown in Listing B.2 and its generated output in Listing B.4 of Appendix B.1. The final XSLT stylesheet can be found in Listing B.6. Remarkable are the comparatively few examples used and the short time required for synthesis. With the setting described above (Section 4.2.2.1), it took only a couple of seconds to generate the correct XSLT stylesheet.

Last However, this kind of problem brought some of ProXSLbE limitations out, i. e. that only one initial I/O pair is often not sufficient. Given a pair as e. g. $1, 2, 3, 4, 5 \rightarrow 5$ it is obviously not possible to identify the desired separator as only two constants were found ('$1, 2, 3, 4,$' and '5'). This is exemplary for ProXSLbE's bias. The more substrings are modified or moved between input and output, the easier it is to correctly identify separators and constants. If only big chunks are removed or added ProXSLbE is misled by its heuristic.

Odd When trying to synthesise this kind of function, similar difficulties arise as described in the previous paragraph. Providing only a short list as I/O example is insufficient to induce the desired separators from. Consider for example the pair $1, 2, 3, 4 \rightarrow 2, 4$. ProXSLbE finds '$2$', '$,$' and '$4$' as common substrings, but has no reason to prefer the comma to the cyphers on basis of the most common prefix/suffix during adjustment (c.f. Section 3.3.2.6). Providing $1, 2, 3, 4, 5, 6 \rightarrow 2, 4, 6$ results in the inadequate constants '2', '$, 4,$' and '6', but only if we provide an example with a longer substring list, e. g. $1, 2, 3, 4, 5, 6, 7, 8 \rightarrow 2, 4, 6, 8$ ProXSLbE finds the desired separators and constants. However, now naively enumerating all possible separator-constants combinations up to a specific length reaches technical limits. Either the user cannot select enough appropriate inputs (described in Step ❺ in Section 4.1) if the maximal length of generated substring lists is not high enough, or the program runs into a stack overflow.

4.3 Summary

The previously described tests illustrated ProXSLbE's capabilities and even more uncovered its limitations and chinks. As the failure in the *insert* and *delete* problem classes has shown, it is not sufficient to solely distinguish between changing and non changing separators. To fix the *insert*-problem it is necessary to introduce a new subcase for "changing separators" which distinguishes a complete modification of the separator from inserting a string before or after the separator. Although this does not simplify the already patchy adjustment phase, it should solve this problem. To enable ProXSLbE to identify delete functions would be much more difficult. Using only the result of the

core algorithm, i. e. the identification of separators and constants, it is not possible to distinguish the I/O pair of a delete function from that of other functions over lists as for example *reverse*.

These problems are mainly accounted for ProXSLbE's underlying data structure. Not that the substring list itself is unsuitable, but the splitting of a string into a substring list according to constants and separators. The reason ProXSLbE mostly fails to transform a string into the appropriate substring list is due to wrongly identified separators. The algorithm assumes that elements of a substring list, i. e. separators and constants, change their context some how from input to output. However, regarding for example the I/O pair for the *last* function this is not always true, because every character in the output still occurs in the same context. These difficulties, as well as the necessary case distinctions, and especially the need for an adjustment phase suggests itself that the splitting of a string according to separators and constants to create a substring list seems not suitable.

Furthermore, the search for separators and constants is highly heuristic. For strings with short separators, e.g. a comma separated list of short substrings, this works comparatively well, but if both constants and separators increase in length, (i) the processing time explodes and (ii) constants and separators are rather split into fragments than found in whole.

Chapter 5

Conclusion

This diploma thesis bridges the gap between fundamental research in Inductive Program Synthesis and its practical application for End User Programming. It demonstrates, that it is indeed feasible to automatically generate XSLT stylesheets from a few examples using the synthesis system for recursive functional programs IGOR which is set up in a term rewriting framework.

The generated XSLT stylesheets apply simple string functions on text nodes of XML documents. To provide the inductive synthesis system IGOR with appropriate I/O examples, a prototypical system transforms the strings of an initial I/O pair into a list of substrings as underlying data structure. Recombining the substrings of the input strings, new possible input strings are generated, chosen and completed by the user, and finally transformed into a specification for IGOR. A parser finally transforms the synthesised functional program into an XSLT stylesheet.

Although the system ProXSLbE does not advance over a prototypical stage, some findings prove of value. The mapping of lexical tokens of IGOR's metalanguage to XSL code fragments has shown to be in particular suitable to translate IGOR's output to XSL. A parser using those flexible XSL blocks can efficiently build the final stylesheet from outwards to inwards. However, there are still possibilities for optimisation, e.g. when inserting and assigning values to variables or creating conditional expressions, they could be combined into one code fragment to reduce the amount of redundant code. Specific attention should be paid to the further development of XSLT processors and their *'tail call optimisation* which is responsible for optimising recursive stylesheets. SAXON sometimes encountered problems when processing the generated stylesheets due to too many apply-template calls. Maybe other processors do not have such problems, or suitable workarounds can be found.

Providing IGOR with appropriate examples is still a problem. Improper, too many, or too few examples lead to wrong result programs or, even worse, to (probably) never ending computing times of IGOR. Naively generating possible inputs is not helpful. Either they are potentially too few and the appropriate were not generated, or the risk of

a stack overflow arises due to combinatorial explosion. After all, a high user expertise is still required to chose those examples which are helpful for IGOR.

However, it should be possible to get around the input generation as applied by ProXSLbE which had its justification from the fact that the user only provides one initial I/O pair from which further example inputs are generated. Maybe it is more efficient to let the user initially provide several different I/O pairs which then are transformed into a specification for IGOR. After collecting more than one initial I/O pair it should be possible to find applicable constants and separators via antiunifying both their inputs and outputs, respectively.

ProXSLbE's main weak spot is its method to transform a string into a list of substrings which is needed for IGOR's input specification. It is important to straighten out that this is not an argument against the substring lists as a data structure to represent strings in IGOR, but against the way an input, respectively output string is transformed into this data structure. The substring list has proven to be quite efficient, because non-function carrying substrings could be transformed into variables and function carrying substrings into constants in IGOR's term rewriting framework. However, a thorough methodology to appropriately transform strings into substring lists on basis of initial I/O pairs without the use of too much "hard coded knowledge" is needed. Promising fields worth to investigate are "Editing by Example" [39] or further development in the fields of "file comparison" and "string difference"[1].

However, it may be less important to provide exactly the right examples in the first place, but enabling the user to quickly provide examples and modify his specification. With short processing times of IGOR it could be therefore even more useful to let the user enter a fast-paced generate and test cycle on basis of trial and error. Put into a feedback loop, the user can quickly provide examples, generate a stylesheet, test it on a target XML document, check the outcome and if necessary modify the examples and start over again.

Finally, it remains to turn reader's attention to those parts of the author's vision described in the introduction that could not be covered here. The synthesis of XSLT stylesheets need not remain restricted to string functions on text nodes. Using drag-and-drop operations, functions over the structure of an XML document could be demonstrated. However, similar problems will arise: (i) detecting the differential between input and output, (ii) extracting from it sufficient information to provide appropriate examples and (iii) transform them into a representation useful for IGOR. Theoretically, IGOR should also be capable of learning functions over tree structures, provided that an appropriate data structure could be found to represent example modifications of XML trees. To detect differentials between an input and an output XML tree current developments in the field of "tree differential algorithms" should be investigated. Useful

[1] The difference algorithm described by Myers [16] itself is not appropriate, since comparison is only conducted sequentially.

information on automatic differential XSL stylesheet generation may be found in [45].

Bibliography

[1] A. BIERMANN: The Inference of Regular LISP Programs from Examples. In: *IEEE Transactions on Systems, Man, and Cybernetics* 8 (1978), Nr. 8, S. 585–600

[2] ALLEN CYPHER: EAGER: Programming Repetitive Tasks by Example. In: *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM Press, 1991. – ISBN 0–89791–383–3, S. 33–39

[3] ALLEN CYPHER: Eager: Programming repetitive tasks by demonstration. In: *Watch What I Do: Programming by Demonstration* (1993), S. 205–217. ISBN 0–262–03213–9

[4] ALLEN CYPHER (Hrsg.): *Watch What I Do – Programming by Demonstration*. Cambridge, MA, USA : MIT Press, 1993

[5] ANDERS BERGLUND: Extensible Stylesheet Language (XSL) Version 1.1 / W3C. 2006. – W3C Recommendation. – URL: `http://www.w3.org/TR/2006/REC-xsl11-20061205/` (accessed: 08/07)

[6] B. SHNEIDERMAN: Direct manipulation: A step beyond programming languages. In: *Human-computer interaction: A multidisciplinary approach* (1987), S. 461–467. ISBN 0–934613–24–9

[7] BRAD A. MYERS: Demonstrational Interfaces: A Step Beyond Direct Manipulation. In: *Computer* 25 (1992), Nr. 8, S. 61–73. – ISSN 0018–9162

[8] BRAD A. MYERS AND ALLEN CYPHER AND DAVID MAULSBY AND DAVID C. SMITH AND BEN SHNEIDERMAN: Demonstrational interfaces: Coming soon? In: *CHI '91: Proceedings of the SIGCHI Conference on Human factors in Computing Systems*. New York, NY, USA : ACM Press, 1991. – ISBN 0–89791–383–3, S. 393–396

[9] DANIEL C. HALBERT: SmallStar: Programming by demonstration in the desktop metaphor. In: *Watch What I Do: Programming by Demonstration* (1993), S. 103–123. ISBN 0–262–03213–9

[10] DANIEL CONRAD HALBERT: *Programming by Example*, University of California, Berkeley, Diss., 1984

[11] DEREK PARTRIDGE: The Case for Inductive Programming. In: *Computer* 30 (1997), Nr. 1, S. 36–41. – ISSN 0018–9162

[12] DIMITRE NOVATCHEV: *The Functional Programming Language XSLT - A proof through examples*. http://fxsl.sourceforge.net/articles/FuncProg/Functional Programming.html, November 2001

[13] EMANUEL KITZELMANN: Data-Driven Induction of Recursive Functions from Input/Output-Examples. In: EMANUEL KITZELMANN AND UTE SCHMID (Hrsg.): *Proceedings of Workshop on Approaches and Applications of Inductive Programming*, 2007, S. 15–27

[14] EMANUEL KITZELMANN AND UTE SCHMID: Inducing Constructor Systems from Example Terms by Detecting Syntactical Regularities. In: *7th International Workshop on Rule-Based Programming, 11th August, Seattle, USA, 2006. see also: Electronic notes in Theoretical Computer Science* 174 (2006), , Nr. 1, S. 49–63

[15] EMANUEL KITZELMANN AND UTE SCHMID: Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach. In: *Journal of Machine Learning Research* 7 (2006), S. 429–454

[16] EUGENE W. MYERS: An O(ND) Difference Algorithm and Its Variations. In: *Algorithmica* 1 (1986), Nr. 2, S. 251–266

[17] EVE MALER AND FRANÇOIS YERGEAU AND TIM BRAY AND JOHN COWAN AND JEAN PAOLI AND C. M. SPERBERG-MCQUEEN: Extensible Markup Language (XML) 1.1 (Second Edition) / W3C. 2006. – W3C Recommendation. – URL: `http://www.w3.org/TR/2006/REC-xml11-20060816` (accessed: 08/07)

[18] FLENER, P.: Inductive logic program synthesis with Dialogs. In: MUGGLETON, S. (Hrsg.): *Proceedings of the 6th International Workshop on Inductive Logic Programming*, Stockholm University, Royal Institute of Technology, 1996, S. 28–51

[19] FRANZ BAADER AND TOBIAS NIPKOW: *Term rewriting and all that*. New York, NY, USA : Cambridge University Press, 1998. – ISBN 0–521–45520–0

[20] GAVIN NICOL AND PHILIPPE LE HÉGARET AND JONATHAN ROBIE AND ARNAUD LE HORS AND STEVE BYRNE AND LAUREN WOOD AND MIKE CHAMPION: Document Object Model (DOM) Level 2 Core Specification / W3C. 2000. – W3C Recommendation. – URL: `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113` (accessed: 08/07)

[21] G.D. PLOTKIN: *Automatic Methods of Inductive Inference*, Edinburgh University, Diss., August 1971

[22] H. LIEBERMAN: *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001

[23] J. R. OLSSON: *Inductive functional programming using incremental program transformation and execution of logic programs by iterative-deepening A* SLD-tree search*. Norway, University of Oslo, Dr scient thesis, 1994. `http://www.ia-stud.hiof.no/~rolando/01-report.ps.Z`

[24] J. R. OLSSON: Inductive functional programming using incremental program transformation. In: *Artificial Intelligence* 74 (1995), Nr. 1, S. 55–83

[25] J. ROLAND OLSSON AND D. M. W. POWERS: Machine Learning of Human Language through Automatic Programming. In: *International Conference on Cognitive Science, University of New South Wales*, 2003, S. 507–512

[26] J. ROSS QUINLAN: Learning First-Order Definitions of Functions. In: *Journal of Artificial Intelligence Research* 5 (1996), S. 139–161

[27] J. ROSS QUINLAN AND R. MIKE CAMERON-JONES: FOIL: A Midterm Report. In: *Machine Learning: ECML-93, European Conference on Machine Learning, Proceedings* Bd. 667, Springer-Verlag, 1993, S. 3–20

[28] JOHN R. KOZA: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA : MIT Press, 1992. – ISBN 0–262–11170–5

[29] MARC SCHOENAUER AND ZBIGNIEW MICHALEWICZ: Evolutionary Computation. In: *Control and Cybernetics* 26 (1997), Nr. 3, S. 307–338

[30] MARTIN HOFMANN AND ANDREAS HIRSCHBERGER AND EMANUEL KITZELMANN AND UTE SCHMID: Inductive Synthesis of Recursive Functional Programs – A Comparison of Three Systems. In: *Proceedings of the 30th Annual German Conference on AI (KI-2007)*, Springer Verlag, 2007, S. 468–472

[31] MICHAEL KAY: XSL Transformations (XSLT) Version 2.0 / W3C. 2007. – W3C Recommendation. – URL: http://www.w3.org/TR/2007/REC-xslt20-20070123/ (accessed: 08/07)

[32] MUGGLETON, S.: Inverse Entailment and Progol. In: *New Generation Computing, Special issue on Inductive Logic Programming* 13 (1995), Nr. 3-4, S. 245–286

[33] MUGGLETON, S.: Learning from positive data. In: MUGGLETON, S. (Hrsg.): *ILP96* Bd. 1314, SV, 1996 (LNAI), S. 358–376

[34] MUGGLETON, S. AND FENG, C.: Efficient induction of logic programs. In: *Proceedings of the 1st Conference on Algorithmic Learning Theory*, Ohmsma, Tokyo, Japan, 1990, S. 368–381

[35] PHILLIP D. SUMMERS: A Methodology for LISP Program Construction from Examples. In: *J. ACM* 24 (1977), Nr. 1, S. 161–175. – ISSN 0004–5411

[36] PIERRE FLENER AND SERAP YILMAZ: Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects. In: *J. Log. Program.* 41 (1999), Nr. 2-3, S. 141–195

[37] RICHARD G. MCDANIEL AND BRAD A. MYERS: Getting more out of programming-by-demonstration. In: *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM Press, 1999. – ISBN 0–201–48559–1, S. 442–449

[38] RICHARD TOBIN AND ANDREW LAYMAN AND DAVE HOLLANDER AND TIM BRAY: Namespaces in XML 1.1 (Second Edition) / W3C. 2006. – W3C Recommendation. – URL: http://www.w3.org/TR/2006/REC-xml-names11-20060816 (accessed: 08/07)

[39] ROBERT P. NIX: Editing by example. In: *ACM Trans. Program. Lang. Syst.* 7 (1985), Nr. 4, S. 600–621. – ISSN 0164–0925

[40] SCOTT BOAG AND MARY F. FERNÁNDEZ AND JÉRÔME SIMÉON AND ANDERS BERGLUND AND MICHAEL KAY AND JONATHAN ROBIE AND DON CHAMBERLIN: XML Path Language (XPath) 2.0 / W3C. 2007. – W3C Recommendation. – URL: http://www.w3.org/TR/2007/REC-xpath20-20070123/ (accessed: 08/07)

[41] SERAP YILMAZ: *Inductive Synthesis of Recursive Logic Programs*, University of Bilkent, Computer Science Department, Diplomarbeit, 1997

[42] S.H. MUGGLETON: Inductive Logic Programming. In: *New Generation Computing* 8 (1991), Nr. 4, 295–318. http://www.doc.ic.ac.uk/~shm/Papers/ilp.pdf

[43] S.H. MUGGLETON AND J. FIRTH: CProgol4.4: a tutorial introduction. Version: 2001. http://www.doc.ic.ac.uk/~shm/Papers/progtuttheo.pdf. In: S. DZEROSKI AND N. LAVRAC (Hrsg.): *Relational Data Mining*. Springer-Verlag, 2001, 160–188

[44] STEPHEN H. MUGGLETON AND C. H. BRYANT AND A. SRINIVASAN: Learning Chomsky-like Grammars for Biological Sequence Families. In: *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, Morgan Kaufmann, 2000, S. 631–638

[45] TAKESHI KATO AND HIDETOSHI UENO AND NORIHIRO ISHIKAWA: An Automatic Generation Method of Differential XSLT Stylesheet from two XML Documents. In: *WEBIST*, 2005, S. 5–12

[46] TERESE: *Cambridge Tracts in Theoretical Computer Science*. Bd. 55: *Term Rewriting Systems*. Cambridge University Press, 2003

[47] TESSA LAU AND STEVEN A. WOLFMAN AND PEDRO DOMINGOS AND DANIEL S. WELD: Learning repetitive text-editing procedures with SMARTedit. In: *Your Wish Is My Command: Programming by Example* (2001), S. 209–226. ISBN 1–55860–688–2

[48] TOM MITCHELL: *Machine Learning*. McGraw Hill, 1997

[49] UTE SCHMID: *LNAI 2654*. Bd. : *Inductive Synthesis of Functional Programs – Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. . Berlin : Springer, 2003

[50] UTE SCHMID AND FRITZ WYSOTZKI: Applying Inductive Program Synthesis to Macro Learning. In: *Artificial Intelligence Planning Systems*, 2000, S. 371–378

[51] WILLIAM F. FINZER AND LAURA GOULD: Rehearsal world: programming by rehearsal. In: *Watch what I do: programming by demonstration* (1993), S. 79–100. ISBN 0–262–03213–9

[52] Y. KODRATOFF: A class of functions synthesized from a finite number of examples and a LISP program scheme. In: *International Journal of Computer and Information Science* 8 (1979), Nr. 7, S. 585–600

Appendix A

patternmatcher.xsl

A.1 igor:separate

The *separate* function works similar to a tokenizer. It splits up the string $toSeparate =' s_1 t_1 \ldots s_n t_n'$, using the tokens $separators = ('t_1', \ldots, 't_n')$, into a sequence of strings $('s_1', 't_1', \ldots, 's_n', 't_n')$ which is returned.

Listing A.1: The separate function of IGOR's namespace

```
<xsl:function name="igor:separate">
    <xsl:param name="toSeparate"/>
    <xsl:param name="separators"/>
    <xsl:choose>
      <xsl:when test=
          "empty(subsequence($separators,2))=true()">
        <xsl:variable
            name="separator"
            select="$separators[1]"/>
        <xsl:for-each select="$toSeparate">
          <xsl:variable name="current"select="."/>
          <xsl:variable
              name="tokens"
              select="tokenize($current,$separator)"/>
          <xsl:if test="starts-with($current,$separator)">
            <xsl:value-of select="$separator"/>
          </xsl:if>
          <xsl:value-of select="$tokens[1]"/>
          <xsl:for-each select="subsequence($tokens,2)">
            <xsl:value-of select="$separator"/>
            <xsl:value-of select="."/>
```

```
            </xsl:for-each>
              <xsl:if test=
                  "ends-with($current,$separator)=true()">
                <xsl:value-of select="$separator"/>
              </xsl:if>
            </xsl:for-each>
          </xsl:when>
          <xsl:otherwise>
            <xsl:variable
                name="partialResult"
                select="igor:separate(
                        $toSeparate,
                        subsequence($separators,2))"/>
            <xsl:variable
             name="separator"
             select="$separators[1]"/>
             <xsl:for-each select="$partialResult">
              <xsl:variable name="current"select="."/>
              <xsl:variable
               name="tokens"
               select="tokenize($current,$separator)"/>
              <xsl:if test=
                  "starts-with($current,$separator)">
                <xsl:value-of select="$separator"/>
              </xsl:if>
              <xsl:value-of select="$tokens[1]"/>
              <xsl:for-each select="subsequence($tokens,2)">
                <xsl:value-of select="$separator"/>
                <xsl:value-of select="."/>
              </xsl:for-each>
                <xsl:if test=
                    "ends-with($current,$separator)">
                  <xsl:value-of select="$separator"/>
                </xsl:if>
            </xsl:for-each>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:function>
```

A.2 igor:match-pattern

The function *match-pattern* tests if the passed substring list matches the passed pattern. This pattern may consist of subpatterns for constant strings of characters encoded with their hexadecimal value, patterns which match any substring (`'X'`), patterns which match any substring list (`'Xs'`), and patterns for the empty list (`'E'`).

Listing A.2: The pattern matching function of IGOR's namespace.

```
<xsl:function name="igor:match-pattern">
  <!-- sequence of strings ('X' matches any element, 'Xs' matches-->
  <!--the whole rest list, 'E' matches the empty list, 'string'-->
  <!-- matches 'string') -->
    <xsl:param name="pattern"/>
    <xsl:param name="substrseq"/>
    <xsl:variable name="Elem"select="'X'"/>
    <xsl:variable name="Empty"select="'E'"/>
    <xsl:variable name="Rest"select="'Xs'"/>

    <xsl:choose>
      <xsl:when test="(
                      empty($pattern) = true())
                  and (empty($substrseq) = true())">
        <xsl:value-of select="true()"/>
      </xsl:when>
      <xsl:when test="empty($pattern) = true()">
        <xsl:value-of select="false()"/>
      </xsl:when>
      <xsl:when test="empty($substrseq) = true()">
        <xsl:choose>
          <xsl:when test="(
                          $pattern[1] = $Empty)
                        or ($pattern[1] = $Rest)">
            <xsl:value-of select="true()"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="false()"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>
      <xsl:when test="$pattern[1] = $Elem">
        <xsl:value-of select="igor:match-pattern(
```

```
                subsequence($pattern,2),
                subsequence($substrseq,2))"/>
      </xsl:when>
      <xsl:when test="$pattern[1] = $Rest">
        <xsl:value-of select="true()"/>
      </xsl:when>
      <xsl:when test="$pattern[1] = $substrseq[1]">
        <xsl:value-of select="igor:match-pattern(
                subsequence($pattern,2),
                subsequence($substrseq,2))"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="false()"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:function>
```

Appendix B

IGOR Files

B.1 IGOR Specification

This section of the appendix describes the specification files for IGOR. First the different parts of a specification are explained using Listing B.1 as an example, then the specification for *reverse* and *replace umlaut* follow.

Since IGOR's programming language is MAUDE, which is an implementation of *rewrite logic*, a specification comes in a MAUDE module enclosed in `fmod MODULE_NAME is ... endfm` (line 1-44 in Listing B.1). The module starts with a definition of the used sorts for the signature in the term rewriting algebra (line 2). The sort *Subs* and *SubsList* are the components of the data structure as described in Section 3.3. The sort *InVec* is a kind of dummy sort which uses IGOR only internally. Then six commented lines starting with `***` (line 4-9) follow which show the internal encoding of characters passed to IGOR. For IGOR a character is encoded in its hexadecimal value. This is simply a standardisation to avoid confusions and clashes due to different character encodings. In line 12 the definitions of the constant terms as unary functions of type `subs`tring follow. The keyword `[ctor]` labels terms as part of the constructor of the underlying data structure. Similar the separators in line 15, which are additionally marked with the metadata `"separator nomatch"` to allow for anti-unification. Line 18 and 19 define the constructors of the data structure substring list (`SubsList`) with the constant `<>` as the empty substring list and the function `__` of type `Subs -> SubsList -> SubsList` as cons-operator. Line 22 defines the type of a helper function for internal use only and line 25 the actual target function, labelled with `metadata "induce"`, of type `SubsList -> SubsList`. The lines 28-42 contain the example equations starting with the keyword `eq`.

B.1.1 ‘Umlaut Replace’ Specification

Listing B.1: IGOR’s specifiaction file for the *umlaut replace* example.

```
fmod PROXSLBE is
 sorts Subs SubsList InVec .

 *** --- String to Hex-Value ---
 ***    hnerb      --> #x0068#x006e#x0065#x0072#x0062#x0072
 ***    he         --> #x0068#x0065
 ***    H          --> #x0048
 ***    ü          --> #x00fc
 ***    ue         --> #x0075#x0065

 *** Substring constants
 ops #x0068#x006e#x0065#x0072#x0062#x0072 #x0068#x0065 #x0048 : -> Subs [ctor] .

 *** Substring separators
 ops #x00fc #x0075#x0065 : -> Subs [ctor metadata "separator nomatch"] .

 *** SubstringList constructors
 op <> : -> SubsList [ctor] .
 op __ : Subs SubsList -> SubsList [ctor] .

 *** input encapsulation
 op in : SubsList -> InVec [ctor] .

 *** function constructor
 op repl : SubsList -> SubsList [metadata "induce"] .
```

```
26
  *** equations
   eq repl(<>) = <> .
   eq repl((#x00fc <>)) = (#x0075#x0065 <>) .
   eq repl((#x0075#x0065 <>)) = (#x0075#x0065 <>) .
31 eq repl((#x0068#x0065 <>)) = (#x0068#x0065 <>) .
   eq repl((#x0068#x0065 (#x00fc <>))) = (#x0068#x0065 (#x0075#x0065 <>)) .
   eq repl((#x0068#x0065 (#x0075#x0065 <>))) = (#x0068#x0065 (#x0075#x0065 <>)) .
   eq repl((#x0048 <>)) = (#x0048 <>) .
   eq repl((#x0048 (#x00fc <>))) = (#x0048 (#x0075#x0065 <>)) .
36 eq repl((#x0048 (#x0075#x0065 <>))) = (#x0048 (#x0075#x0065 <>)) .
   eq repl((#x00fc (#x0068#x0065 <>))) = (#x0075#x0065 (#x0068#x0065 <>)) .
   eq repl((#x00fc (#x0068#x0065 (#x00fc <>)))) = ↵
      (#x0075#x0065 (#x0068#x0065 (#x0075#x0065 <>))) .
   eq repl((#x00fc (#x0048 <>))) = (#x0075#x0065 (#x0048 <>)) .
41 eq repl((#x00fc (#x0048 (#x00fc <>)))) =↵
      (#x0075#x0065 (#x0048 (#x0075#x0065 <>))) .

  endfm
```

B.1.2 'Reverse' Specification

Listing B.2: IGOR's specifiaction file for reversing a substring list.

```
fmod PROXSLBE is
 sorts Subs SubsList InVec .

***     --- String to Hex-Value ---
***    3              --> #x0033
***    2              --> #x0032
***    1              --> #x0031
***    5              --> #x0035
***    4              --> #x0034
***    -              --> #x002d

*** Substring constants
 ops #x0033 #x0032 #x0031 #x0035 #x0034 : -> Subs [ctor] .

*** Substring separators
 ops #x002d : -> Subs [ctor metadata "separator nomatch"] .

*** SubstringList constructors
 op <> : -> SubsList [ctor] .
 op __ : Subs SubsList -> SubsList [ctor] .

*** input encapsulation
 op in : SubsList -> InVec [ctor] .

*** function constructor
```

```
 op reverse : SubsList -> SubsList [metadata "induce"] .

*** equations
 eq reverse(<>) = <> .
 eq reverse((#x0033 <>)) = (#x0033 <>) .
 eq reverse((#x0032 <>)) = (#x0032 <>) .
 eq reverse((#x0031 <>)) = (#x0031 <>) .
 eq reverse((#x0033 (#x002d (#x0031 <>)))) = (#x0031 (#x002d (#x0033 <>))) .
 eq reverse((#x0032 (#x002d (#x0033 <>)))) = (#x0033 (#x002d (#x0032 <>))) .
 eq reverse((#x0032 (#x002d (#x0033 (#x002d (#x0031 <>)))))) = ↵
    (#x0031 (#x002d (#x0033 (#x002d (#x0032 <>))))) .
 eq reverse((#x0031 (#x002d (#x0032 (#x002d (#x0033 <>)))))) = ↵
    (#x0033 (#x002d (#x0032 (#x002d (#x0031 <>))))) .
 eq reverse((#x0032 (#x002d (#x0033 (#x002d (#x0034 (#x002d (#x0031 <>)))))))) = ↵
    (#x0031 (#x002d (#x0034 (#x002d (#x0033 (#x002d (#x0032 <>))))))) .
 eq reverse((#x0031 (#x002d (#x0032 (#x002d (#x0033 (#x002d (#x0034 <>)))))))) = ↵
    (#x0034 (#x002d (#x0033 (#x002d (#x0032 (#x002d (#x0031 <>))))))) .

endfm
```

B.2 IGOR Output

B.2.1 ‘Umlaut Replace’ Output

Listing B.3: IGOR’s output file for the “Hühnerbrühe” example.

```
reduce in IGOR : setup('PROXSLBE, 1, false) .
rewrites: 2756216 in 31741ms cpu (31793ms real) (86831 rewrites/second)
result Hypo: hypo(true, 3, eq 'Sub10['__['#x00fc.Subs,'X0:SubsList]] =
         '['Sub124['__['#x00fc.Subs,'X0:SubsList]]] [none] .
eq 'Sub124['__['#x00fc.Subs,'X0:SubsList]] = 'X0:SubsList [none] .
eq 'Sub143['__['X0:Subs,'X1:SubsList]] = 'X1:SubsList [none] .
eq 'Sub54['__['X0:Subs,'X1:SubsList]] =↵
   '['Sub143['__['X0:Subs,'X1:SubsList]]] [none] .
eq '['<>.SubsList] = '<>.SubsList [none] .
ceq '['__['X0:Subs,'X1:SubsList]] =
         '__['#x0075#x0065.Subs,'Sub10['__['X0:Subs,'X1:SubsList]]]
            if '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] =  'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
               '_==_['X0:Subs,'#x00fc.Subs] = 'true.Bool /\
```

```
                ’_==_[’X0:Subs,’#x00fc.Subs] = ’true.Bool [none] .
  ceq ’[’__[’X0:Subs,’X1:SubsList]] =
26          ’__[’X0:Subs,’Sub54[’__[’X0:Subs,’X1:SubsList]]]
              if ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool /\
                 ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool /\
                 ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool /\
                 ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool /\
31               ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool /\
                 ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool /\
                 ’_==_[’X0:Subs,’#x00fc.Subs] = ’false.Bool [none] .)
```

B.2.2 ‘Reverse’ Output

Listing B.4: IGOR’s output file for the *reverse* problem.

```
eq 'Sub133['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
          '__['X2:Subs,'X3:SubsList]]]]]] =↵
   '__['#x002d.Subs,'Sub169['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,↵
      '__['#x002d.Subs,'__['X2:Subs,'X3:SubsList]]]]]]] [none] .
eq 'Sub169['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
          '__['X2:Subs,'X3:SubsList]]]]]] =↵
   'Sub65['Sub217['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,↵
         '__['#x002d.Subs,'__['X2:Subs,'X3:SubsList]]]]]]] [none] .
eq 'Sub217['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
          '__['X2:Subs,'X3:SubsList]]]]]] =↵
   '__['X1:Subs,'__['#x002d.Subs,'__['X2:Subs,'X3:SubsList]]] [none] .
eq 'Sub40['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'X2:SubsList]]]] =↵
   'reverse['Sub65['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,↵
           'X2:SubsList]]]]] [none] .
eq 'Sub5['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'<>.SubsList]]]] = ↵
          'X1:Subs [none] .
eq 'Sub5['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
         '__['X2:Subs,'X3:SubsList]]]]]] =↵
   'Sub5['Sub59['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
         '__['X2:Subs,'X3:SubsList]]]]]]] [none] .
eq 'Sub59['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
          '__['X2:Subs,'X3:SubsList]]]]]] = ↵
   '__['X0:Subs,'__['#x002d.Subs,'__['X2:Subs,'X3:SubsList]]] [none] .
eq 'Sub6['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'X2:SubsList]]]] =↵
    '__['#x002d.Subs,'Sub40['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,↵
```

```
          'X2:SubsList]]]]] [none] .
eq 'Sub65['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'<>.SubsList]]]] =↵
   '__['X0:Subs,'<>.SubsList] [none] .
eq 'Sub65['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'__['#x002d.Subs,↵
         '__['X2:Subs,'X3:SubsList]]]]]] =↵
   '__['X0:Subs,'Sub133['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,↵
      '__['#x002d.Subs,'__['X2:Subs,'X3:SubsList]]]]]]] [none] .
eq 'reverse['<>.SubsList] = '<>.SubsList [none] .
eq 'reverse['__['X0:Subs,'<>.SubsList]] = '__['X0:Subs,'<>.SubsList] [none] .
eq 'reverse['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'X2:SubsList]]]] =↵
   '__['Sub5['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'X2:SubsList]]]],↵
      'Sub6['__['X0:Subs,'__['#x002d.Subs,'__['X1:Subs,'X2:SubsList]]]]] [none] .
```

B.3 Generated XSLT Stylesheets

B.3.1 Generated Stylesheet for ‘Umlaut Replace’

Listing B.5: The generated XSLT stylesheet for the “Hühnerbrühe” problem.

```
<?xml version="1.0"encoding="UTF-8"?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:igor="http://www.cogsys.wiai.uni-bamberg.de"
   version="2.0">
  <xsl:output
      method="xml"
      indent="yes"
      encoding="UTF-8"/>
  <xsl:include href="patternmatcher.xsl"/>
  <!--Definition of used substringlist separators-->
  <xsl:variable
      name="separators"
      select="('ü', 'ue')"/>
  <!--Copy non-target nodes-->
   <xsl:template match="/|*|@*|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>
  <!--Entrance function for target node-->
  <xsl:template match="replace">
    <!--Copy everything else but text-->
    <xsl:copy>
      <!--Pass text of target node tokenized-->
      <!--by separators to function template-->
      <xsl:call-template name="umlrepl3">
        <xsl:with-param
           name="input"
           select="igor:separate(
                   text(),($separators))"/>
      </xsl:call-template>
    </xsl:copy>
  </xsl:template>
  <xsl:template name="Sub10">
```

```
    <xsl:param name="input"/>
    <xsl:choose>
      <xsl:when test=
          "igor:match-pattern(('ü', 'Xs'), $input)
40              = true()">
        <xsl:variable name="X0_SubsList"
                  select="subsequence($input,2)"/>
        <xsl:variable name="Sub124_arg">
          <xsl:value-of select="'ü'"/>
45        <xsl:value-of select="$X0_SubsList"/>
        </xsl:variable>
        <xsl:variable name="umlrepl3_arg">
          <xsl:call-template name="Sub124">
            <xsl:with-param
50              name="input"
                select="igor:separate(
                        $Sub124_arg,$separators)"/>
          </xsl:call-template>
        </xsl:variable>
55      <xsl:call-template name="umlrepl3">
          <xsl:with-param
              name="input"
              select="igor:separate(
                      $umlrepl3_arg,$separators)"/>
60      </xsl:call-template>
      </xsl:when>
    </xsl:choose>
  </xsl:template>
  <xsl:template name="Sub124">
65  <xsl:param name="input"/>
    <xsl:choose>
      <xsl:when test=
           "igor:match-pattern(('ü', 'Xs'), $input)
                = true()">
70      <xsl:variable
            name="X0_SubsList"
            select="subsequence($input,2)"/>
        <xsl:value-of select="$X0_SubsList"/>
      </xsl:when>
75  </xsl:choose>
```

```
    </xsl:template>
    <xsl:template name="Sub143">
      <xsl:param name="input"/>
      <xsl:choose>
80      <xsl:when test=
            "igor:match-pattern(('X', 'Xs'), $input)
                 = true()">
          <xsl:variable name="X0_Subs"
                        select="$input[1]"/>
85        <xsl:variable name="X1_SubsList"
                        select="subsequence($input,2)"/>
          <xsl:value-of select="$X1_SubsList"/>
        </xsl:when>
      </xsl:choose>
90  </xsl:template>
    <xsl:template name="Sub54">
      <xsl:param name="input"/>
      <xsl:choose>
        <xsl:when test=
95          "igor:match-pattern(('X', 'Xs'), $input)
                 = true()">
          <xsl:variable name="X0_Subs"
                        select="$input[1]"/>
          <xsl:variable name="X1_SubsList"
100                     select="subsequence($input,2)"/>
          <xsl:variable name="Sub143_arg">
            <xsl:value-of select="$X0_Subs"/>
            <xsl:value-of select="$X1_SubsList"/>
          </xsl:variable>
105       <xsl:variable name="umlrepl3_arg">
            <xsl:call-template name="Sub143">
              <xsl:with-param
                 name="input"
                  select="igor:separate(
110                         $Sub143_arg,$separators)"/>
            </xsl:call-template>
          </xsl:variable>
          <xsl:call-template name="umlrepl3">
            <xsl:with-param
115           name="input"
```

```
                select="igor:separate(
                        $umlrepl3_arg,$separators)"/>
            </xsl:call-template>
          </xsl:when>
        </xsl:choose>
      </xsl:template>
      <xsl:template name="umlrepl3">
        <xsl:param name="input"/>
        <xsl:choose>
          <xsl:when test=
              "igor:match-pattern(('E'), $input)
                  = true()">
            <xsl:value-of select=""""/>
          </xsl:when>
          <xsl:when test=
              "igor:match-pattern(('X', 'Xs'), $input)
                  = true()">
            <xsl:variable name="X0_Subs"
                          select="$input[1]"/>
            <xsl:variable name="X1_SubsList"
                          select="subsequence($input,2)"/>
            <xsl:variable name="cond_arg_0">
              <xsl:value-of select="$X0_Subs"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_1">
              <xsl:value-of select="'ü'"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_2">
              <xsl:value-of select="$X0_Subs"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_3">
              <xsl:value-of select="'ü'"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_4">
              <xsl:value-of select="$X0_Subs"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_5">
              <xsl:value-of select="'ü'"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_6">
```

```
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_7">
    <xsl:value-of select="'ü'"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_8">
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_9">
    <xsl:value-of select="'ü'"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_10">
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_11">
    <xsl:value-of select="'ü'"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_12">
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_13">
    <xsl:value-of select="'ü'"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_14">
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_15">
    <xsl:value-of select="'ü'"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_16">
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_17">
    <xsl:value-of select="'ü'"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_18">
    <xsl:value-of select="$X0_Subs"/>
  </xsl:variable>
  <xsl:variable name="cond_arg_19">
    <xsl:value-of select="'ü'"/>
```

```
        </xsl:variable>
        <xsl:variable name="cond_arg_20">
          <xsl:value-of select="$X0_Subs"/>
        </xsl:variable>
        <xsl:variable name="cond_arg_21">
          <xsl:value-of select="'ü'"/>
        </xsl:variable>
        <xsl:variable name="cond_arg_22">
          <xsl:value-of select="$X0_Subs"/>
        </xsl:variable>
        <xsl:variable name="cond_arg_23">
          <xsl:value-of select="'ü'"/>
        </xsl:variable>
        <xsl:variable name="cond_arg_24">
          <xsl:value-of select="$X0_Subs"/>
        </xsl:variable>
        <xsl:variable name="cond_arg_25">
          <xsl:value-of select="'ü'"/>
        </xsl:variable>
        <xsl:if test="($cond_arg_0=$cond_arg_1) and
                      ($cond_arg_2=$cond_arg_3) and
                      ($cond_arg_4=$cond_arg_5) and
                      ($cond_arg_6=$cond_arg_7) and
                      ($cond_arg_8=$cond_arg_9) and
                       ($cond_arg_10=$cond_arg_11) and
                      ($cond_arg_12=$cond_arg_13) and
                      ($cond_arg_14=$cond_arg_15) and
                      ($cond_arg_16=$cond_arg_17) and
                      ($cond_arg_18=$cond_arg_19) and
                       ($cond_arg_20=$cond_arg_21) and
                      ($cond_arg_22=$cond_arg_23) and
                      ($cond_arg_24=$cond_arg_25)">
          <xsl:value-of select="'ue'"/>
          <xsl:variable name="Sub10_arg">
            <xsl:value-of select="$X0_Subs"/>
            <xsl:value-of select="$X1_SubsList"/>
          </xsl:variable>
          <xsl:call-template name="Sub10">
            <xsl:with-param
              name="input"
```

```
            select="igor:separate(
                    $Sub10_arg,$separators)"/>
      </xsl:call-template>
    </xsl:if>
    <xsl:variable name="X0_Subs"select="$input[1]"/>
    <xsl:variable name="X1_SubsList"
              select="subsequence($input,2)"/>
    <xsl:variable name="cond_arg_26">
      <xsl:value-of select="$X0_Subs"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_27">
      <xsl:value-of select="'ü'"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_28">
      <xsl:value-of select="$X0_Subs"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_29">
      <xsl:value-of select="'ü'"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_30">
      <xsl:value-of select="$X0_Subs"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_31">
      <xsl:value-of select="'ü'"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_32">
      <xsl:value-of select="$X0_Subs"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_33">
      <xsl:value-of select="'ü'"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_34">
      <xsl:value-of select="$X0_Subs"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_35">
      <xsl:value-of select="'ü'"/>
    </xsl:variable>
    <xsl:variable name="cond_arg_36">
      <xsl:value-of select="$X0_Subs"/>
    </xsl:variable>
```

```
            <xsl:variable name="cond_arg_37">
              <xsl:value-of select="'ü'"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_38">
              <xsl:value-of select="$X0_Subs"/>
            </xsl:variable>
            <xsl:variable name="cond_arg_39">
              <xsl:value-of select="'ü'"/>
            </xsl:variable>
            <xsl:if test="not($cond_arg_26=$cond_arg_27) and
                          not($cond_arg_28=$cond_arg_29) and
                          not($cond_arg_30=$cond_arg_31) and
                          not($cond_arg_32=$cond_arg_33) and
                          not($cond_arg_34=$cond_arg_35) and
                           not($cond_arg_36=$cond_arg_37) and
                          not($cond_arg_38=$cond_arg_39)">
              <xsl:value-of select="$X0_Subs"/>
              <xsl:variable name="Sub54_arg">
                <xsl:value-of select="$X0_Subs"/>
                <xsl:value-of select="$X1_SubsList"/>
              </xsl:variable>
              <xsl:call-template name="Sub54">
                <xsl:with-param name="input"
                  select="igor:separate(
                          $Sub54_arg,$separators)"/>
              </xsl:call-template>
            </xsl:if>
          </xsl:when>
        </xsl:choose>
      </xsl:template>
    </xsl:stylesheet>
```

B.3.2 Generated Stylesheet for ‘Reverse’

Listing B.6: The generated XSLT stylesheet for the “Reverse” problem.

```
<?xml version="1.0"encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:igor="http://www.cogsys.wiai.uni-bamberg.de"
    version="2.0">
  <xsl:output
      method="xml"
      indent="yes"
      encoding="ISO-8859-1"/>
  <xsl:include href="patternmatcher.xsl"/>
  <!--Definition of used substringlist separators-->
  <xsl:variable name="separators"select="('-')"/>
  <!--Copy non-target nodes-->
  <xsl:template match="/|*|@*|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>
  <!--Entrance function for target node-->
  <xsl:template match="list_2">
    <!--Copy everything else but text-->
    <xsl:copy>
 <!--Pass text of target node tokenized by separators to function template-->
      <xsl:call-template name="reverse">
        <xsl:with-param
           name="input"
           select="igor:separate(
                   text(),($separators))"/>
      </xsl:call-template>
    </xsl:copy>
  </xsl:template>
  <xsl:template name="Sub133">
    <xsl:param name="input"/>
    <xsl:choose>
      <xsl:when test="igor:match-pattern
                      (('X', '-', 'X', '-', 'X', 'Xs'),
                      $input)
```

```
                    = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:variable name="X2_Subs"select="$input[5]"/>
          <xsl:variable
             name="X3_SubsList"
             select="subsequence($input,6)"/>
          <xsl:value-of select="'-'"/>
          <xsl:variable name="Sub169_arg">
            <xsl:value-of select="$X0_Subs"/>
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X2_Subs"/>
            <xsl:value-of select="$X3_SubsList"/>
          </xsl:variable>
          <xsl:call-template name="Sub169">
            <xsl:with-param
               name="input"
               select="igor:separate(
                       $Sub169_arg,$separators)"/>
          </xsl:call-template>
        </xsl:when>
      </xsl:choose>
    </xsl:template>
    <xsl:template name="Sub169">
      <xsl:param name="input"/>
      <xsl:choose>
        <xsl:when test="igor:match-pattern
                         (('X', '-', 'X', '-', 'X', 'Xs'),
                         $input)
                        = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:variable name="X2_Subs"select="$input[5]"/>
          <xsl:variable
             name="X3_SubsList"
             select="subsequence($input,6)"/>
          <xsl:variable name="Sub217_arg">
            <xsl:value-of select="$X0_Subs"/>
```

```
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X2_Subs"/>
            <xsl:value-of select="$X3_SubsList"/>
          </xsl:variable>
          <xsl:variable name="Sub65_arg">
            <xsl:call-template name="Sub217">
              <xsl:with-param
                 name="input"
                 select="igor:separate(
                         $Sub217_arg,$separators)"/>
            </xsl:call-template>
          </xsl:variable>
          <xsl:call-template name="Sub65">
            <xsl:with-param
               name="input"
               select="igor:separate(
                       $Sub65_arg,$separators)"/>
          </xsl:call-template>
        </xsl:when>
      </xsl:choose>
    </xsl:template>
    <xsl:template name="Sub217">
      <xsl:param name="input"/>
      <xsl:choose>
        <xsl:when test="igor:match-pattern
                        (('X', '-', 'X', '-', 'X', 'Xs'),
                        $input)
                       = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:variable name="X2_Subs"select="$input[5]"/>
          <xsl:variable
             name="X3_SubsList"
             select="subsequence($input,6)"/>
          <xsl:value-of select="$X1_Subs"/>
          <xsl:value-of select="'-'"/>
          <xsl:value-of select="$X2_Subs"/>
          <xsl:value-of select="$X3_SubsList"/>
```

```
      </xsl:when>
    </xsl:choose>
</xsl:template>
<xsl:template name="Sub40">
    <xsl:param name="input"/>
    <xsl:choose>
      <xsl:when test="igor:match-pattern
                        (('X', '-', 'X', 'Xs'),
                        $input)
                       = true()">
        <xsl:variable name="X0_Subs"select="$input[1]"/>
        <xsl:variable name="X1_Subs"select="$input[3]"/>
        <xsl:variable
           name="X2_SubsList"
           select="subsequence($input,4)"/>
        <xsl:variable name="Sub65_arg">
          <xsl:value-of select="$X0_Subs"/>
          <xsl:value-of select="'-'"/>
          <xsl:value-of select="$X1_Subs"/>
          <xsl:value-of select="$X2_SubsList"/>
        </xsl:variable>
        <xsl:variable name="reverse_arg">
          <xsl:call-template name="Sub65">
            <xsl:with-param
               name="input"
               select="igor:separate(
                       $Sub65_arg,$separators)"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:call-template name="reverse">
          <xsl:with-param
             name="input"
             select="igor:separate(
                     $reverse_arg,$separators)"/>
        </xsl:call-template>
      </xsl:when>
    </xsl:choose>
</xsl:template>
<xsl:template name="Sub5">
    <xsl:param name="input"/>
```

```
      <xsl:choose>
        <xsl:when test="igor:match-pattern
160                     (('X', '-', 'X', 'E'),
                        $input)
                       = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
165       <xsl:value-of select="$X1_Subs"/>
        </xsl:when>
        <xsl:when test="igor:match-pattern
                        (('X', '-', 'X', '-', 'X', 'Xs'),
                        $input)
170                    = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:variable name="X2_Subs"select="$input[5]"/>
          <xsl:variable
175          name="X3_SubsList"
             select="subsequence($input,6)"/>
          <xsl:variable name="Sub59_arg">
            <xsl:value-of select="$X0_Subs"/>
            <xsl:value-of select="'-'"/>
180         <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X2_Subs"/>
            <xsl:value-of select="$X3_SubsList"/>
          </xsl:variable>
185       <xsl:variable name="Sub5_arg">
            <xsl:call-template name="Sub59">
              <xsl:with-param
                 name="input"
                 select="igor:separate(
190                     $Sub59_arg,$separators)"/>
            </xsl:call-template>
          </xsl:variable>
          <xsl:call-template name="Sub5">
            <xsl:with-param
195           name="input"
              select="igor:separate(
                      $Sub5_arg,$separators)"/>
```

```
            </xsl:call-template>
          </xsl:when>
200       </xsl:choose>
        </xsl:template>
        <xsl:template name="Sub59">
          <xsl:param name="input"/>
          <xsl:choose>
205         <xsl:when test="igor:match-pattern
                            (('X', '-', 'X', '-', 'X', 'Xs'),
                            $input)
                           = true()">
            <xsl:variable name="X0_Subs"select="$input[1]"/>
            <xsl:variable name="X1_Subs"select="$input[3]"/>
210         <xsl:variable name="X2_Subs"select="$input[5]"/>
            <xsl:variable
               name="X3_SubsList"
               select="subsequence($input,6)"/>
            <xsl:value-of select="$X0_Subs"/>
215         <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X2_Subs"/>
            <xsl:value-of select="$X3_SubsList"/>
          </xsl:when>
220       </xsl:choose>
        </xsl:template>
        <xsl:template name="Sub6">
          <xsl:param name="input"/>
          <xsl:choose>
225         <xsl:when test="igor:match-pattern
                            (('X', '-', 'X', 'Xs'),
                            $input)
                           = true()">
            <xsl:variable name="X0_Subs"select="$input[1]"/>
            <xsl:variable name="X1_Subs"select="$input[3]"/>
230         <xsl:variable
               name="X2_SubsList"
               select="subsequence($input,4)"/>
            <xsl:value-of select="'-'"/>
            <xsl:variable name="Sub40_arg">
235           <xsl:value-of select="$X0_Subs"/>
              <xsl:value-of select="'-'"/>
```

```
            <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="$X2_SubsList"/>
          </xsl:variable>
          <xsl:call-template name="Sub40">
            <xsl:with-param
               name="input"
               select="igor:separate(
                       $Sub40_arg,$separators)"/>
          </xsl:call-template>
        </xsl:when>
      </xsl:choose>
    </xsl:template>
    <xsl:template name="Sub65">
      <xsl:param name="input"/>
      <xsl:choose>
        <xsl:when test="igor:match-pattern
                        (('X', '-', 'X', 'E'),
                        $input)
                       = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:value-of select="$X0_Subs"/>
          <xsl:value-of select=""""/>
        </xsl:when>
        <xsl:when test="igor:match-pattern
                        (('X', '-', 'X', '-', 'X', 'Xs'),
                        $input)
                       = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:variable name="X2_Subs"select="$input[5]"/>
          <xsl:variable
             name="X3_SubsList"
             select="subsequence($input,6)"/>
          <xsl:value-of select="$X0_Subs"/>
          <xsl:variable name="Sub133_arg">
            <xsl:value-of select="$X0_Subs"/>
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="'-'"/>
```

```
            <xsl:value-of select="$X2_Subs"/>
            <xsl:value-of select="$X3_SubsList"/>
          </xsl:variable>
          <xsl:call-template name="Sub133">
            <xsl:with-param
               name="input"
               select="igor:separate(
                       $Sub133_arg,$separators)"/>
          </xsl:call-template>
        </xsl:when>
      </xsl:choose>
    </xsl:template>
    <xsl:template name="reverse">
      <xsl:param name="input"/>
      <xsl:choose>
        <xsl:when test="igor:match-pattern
                        (('E'),
                        $input)
                       = true()">
          <xsl:value-of select=""""/>
        </xsl:when>
        <xsl:when test="igor:match-pattern
                        (('X', 'E'),
                        $input)
                       = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:value-of select="$X0_Subs"/>
          <xsl:value-of select=""""/>
        </xsl:when>
        <xsl:when test="igor:match-pattern
                        (('X', '-', 'X', 'Xs'),
                        $input)
                       = true()">
          <xsl:variable name="X0_Subs"select="$input[1]"/>
          <xsl:variable name="X1_Subs"select="$input[3]"/>
          <xsl:variable
             name="X2_SubsList"
             select="subsequence($input,4)"/>
          <xsl:variable name="Sub5_arg">
            <xsl:value-of select="$X0_Subs"/>
```

```
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="$X2_SubsList"/>
          </xsl:variable>
          <xsl:call-template name="Sub5">
            <xsl:with-param
                name="input"
                select="igor:separate(
                        $Sub5_arg,$separators)"/>
          </xsl:call-template>
          <xsl:variable name="Sub6_arg">
            <xsl:value-of select="$X0_Subs"/>
            <xsl:value-of select="'-'"/>
            <xsl:value-of select="$X1_Subs"/>
            <xsl:value-of select="$X2_SubsList"/>
          </xsl:variable>
          <xsl:call-template name="Sub6">
            <xsl:with-param
                name="input"
                select="igor:separate(
                        $Sub6_arg,$separators)"/>
          </xsl:call-template>
        </xsl:when>
      </xsl:choose>
    </xsl:template>
  </xsl:stylesheet>
```

Appendix C

Screenshots

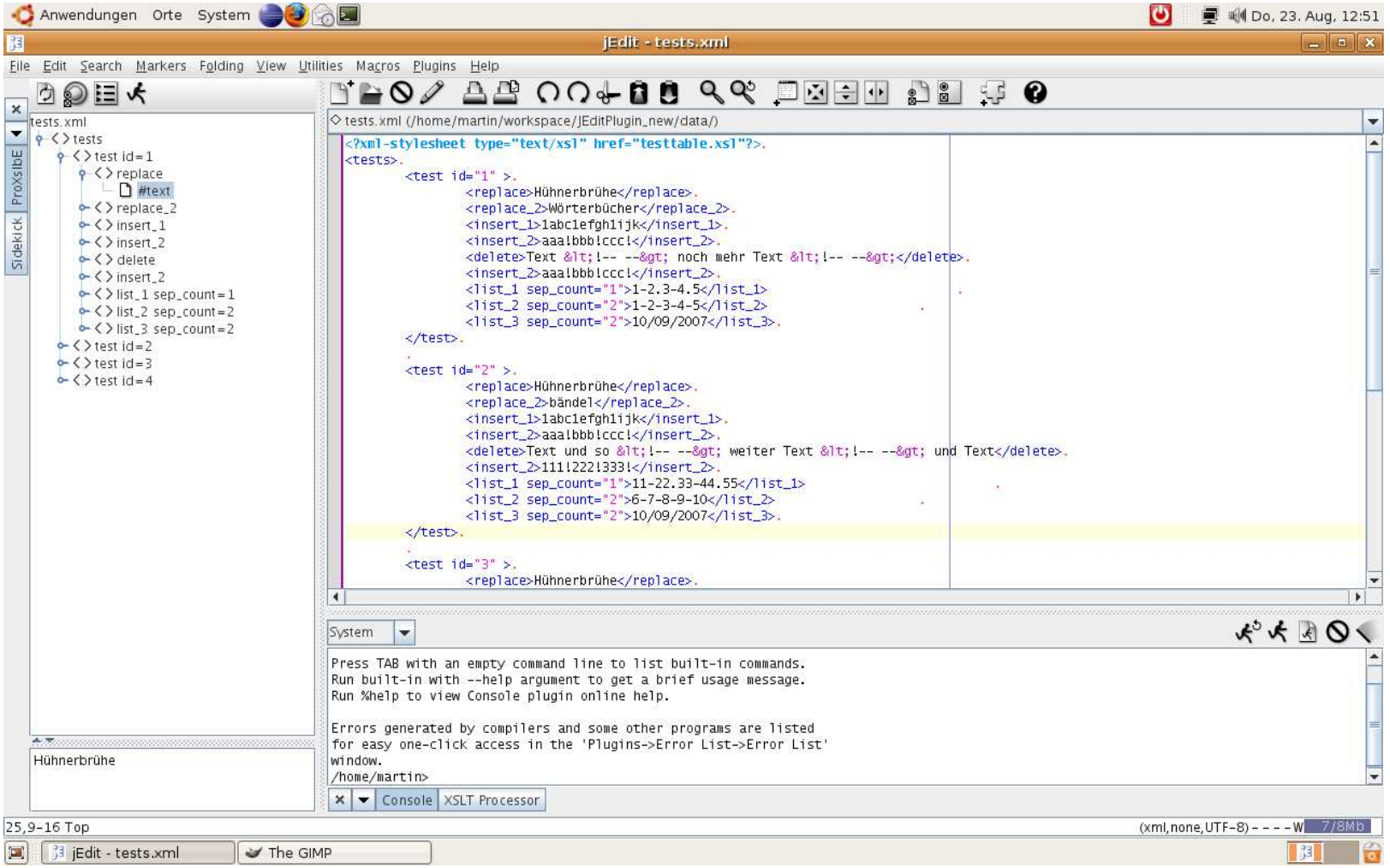

Figure C.1: ProXSLbE's main view

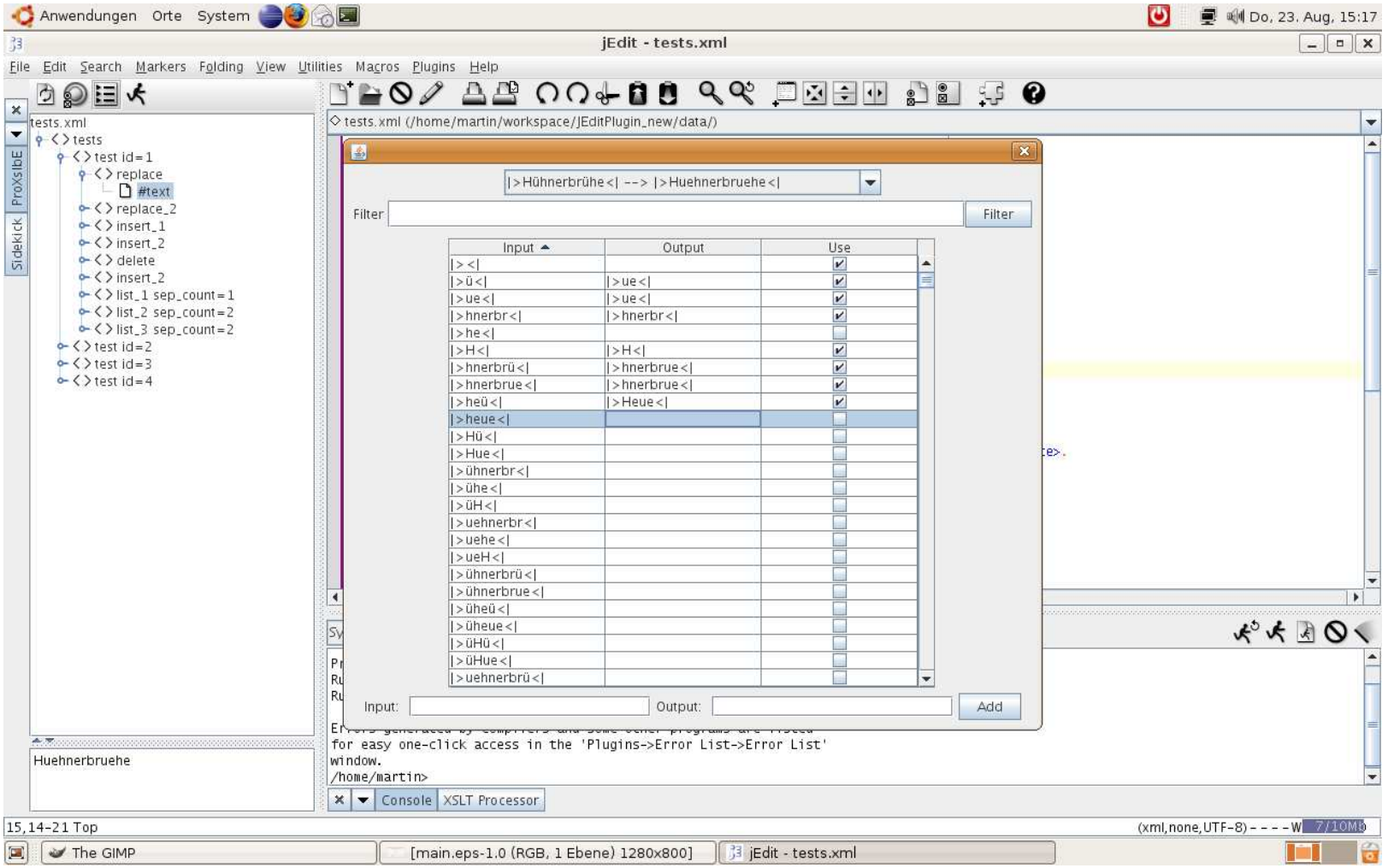

Figure C.2: Selecting and completing the generated input examples.

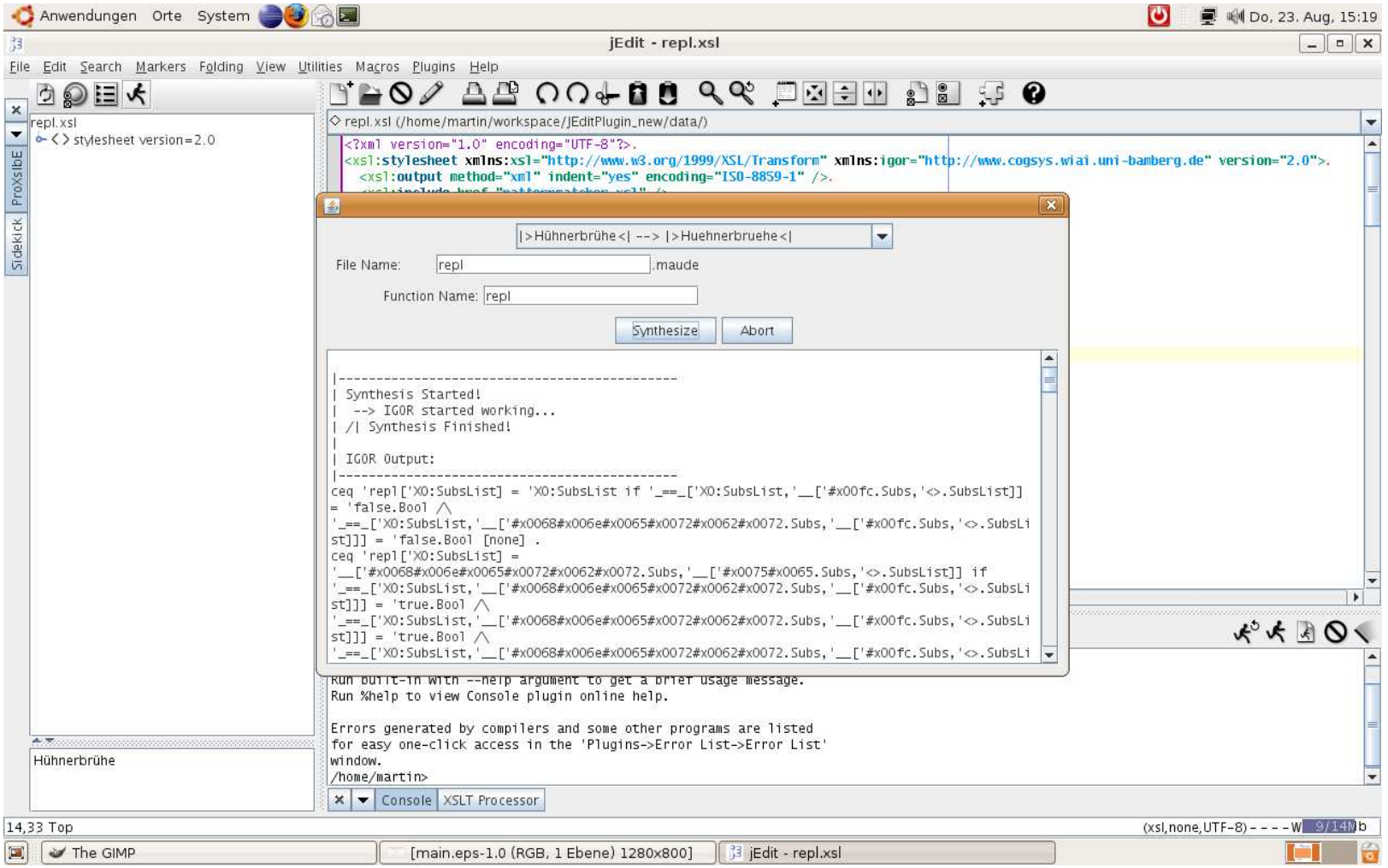

Figure C.3: ProXSLbE at work.

Appendix D

Lists of Figures, Tables, Listings, and Algorithms

List of Figures

List of Tables

Listings

List of Algorithms

“Ich erkläre hiermit gemäß § 27 Abs. 2 APO, dass ich die vorstehende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.”

Bamberg, den 14. September 2007

Martin Hofmann