Analysis and Transformation of Source Code
by
Parsing and Rewriting

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. mr. P.F. van der Heijden
ten overstaan van een door het
college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op dinsdag 15 november 2005, te 10:00 uur
door

Jurgen Jordanus Vinju

geboren te Ermelo

Promotor:	prof. dr. P. Klint
Co-promotor:	dr. M.G.J. van den Brand
Faculteit:	Natuurwetenschappen, Wiskunde en Informatica

The work in this thesis has been carried out at Centrum voor Wiskunde en Informatica (CWI) in Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Preface

Before consuming this manuscript the reader should know that I owe gratitude to many people. First of all, it is a family accomplishment. I want to thank my mother, Annelies, for being so strong and for always motivating me to do what I like best. I thank my father, Fred, for always supporting me. My sister, Krista, is my soul mate. We are so much alike. The love of my live, Rebecca, has been my support and inspiration for the past six years.

I would like to thank my best friends: Arjen Koppen, Bas Toeter, Coen Visser, Hugo Loomans, Mieke Schouten, Warner Salomons, and Winfried Holthuizen. You don't know how much of you is a part of me.

My supervisors at CWI are Mark van den Brand and Paul Klint. Mark inspired me to study computer science in Amsterdam, and to start a PhD project at CWI. He sparked my interest in ASF+SDF already at the age of 17. Thank you for teaching me, caring for me, and for motivating me all these years. Paul is a great mentor and role model. I admire him for his insight in so many issues and for his endless enthusiasm for research. He is the most productive man I have ever seen. Thanks for your time teaching me.

Many of my colleagues have become my friends. Thank you for the teamwork, for providing an inspiring work environment, and for the relaxing times we spent in bars and restaurants. They are in alphabetical order: Ali Mesbah, Anamaria Martins Moreira, Anderson Santana, Anthony Cleve, Arie van Deursen, Li Bixin, Chris Verhoef, David Déharbe, Diego Ordonez Camacho, Eelco Visser, Ernst-Jan Verhoeven, Gerald Stap, Gerco Ballintijn, Hayco de Jong, Jan Heering, Jan van Eijck, Jeroen Scheerder, Joost Visser, Jørgen Iversen, Steven Klusener, Leon Moonen, Magiel Bruntink, Martin Bravenboer, Merijn de Jonge, Niels Veerman, Pierre-Etienne Moreau, Pieter Olivier, Ralf Lämmel, Rob Economopoulos, Slinger Jansen, Taeke Kooiker, Tijs van der Storm, Tobias Kuipers, Tom Tourwé, Vania Marangozova.

I would like to thank Claude Kirchner for allowing me to work an inspiring and productive three month period at INRIA-LORIA. Finally, I thank the members of my reading committee for reading the manuscript and providing valuable feedback: prof. dr. J.A. Bergstra, prof. dr. M. de Rijke, prof. dr. C.R. Jesshope, prof. dr. K.M. van Hee and prof. dr. J.R. Cordy.

The CWI institute is a wonderful place to learn and produce computer science.

Contents

Part I

Overview

CHAPTER 1

Introduction

In this thesis the subject of study is source code. More precisely, I am interested in tools that help in describing, analyzing and transforming source code.

The overall question is how well qualified and versatile the programming language ASF+SDF is when applied to source code analysis and transformation. The main technical issues that are addressed are ambiguity of context-free languages and improving two important quality attributes of analyses and transformations: conciseness and fidelity.

The overall result of this research is a version of the language that is better tuned to the domain of source code analysis and transformation, but is still firmly grounded on the original: a hybrid of context-free grammars and term rewriting. The results that are presented have a broad technical spectrum because they cover the entire scope of ASF+SDF. They include disambiguation by filtering parse forests, the type-safe automation of tree traversal for conciseness, improvements in language design resulting in higher resolution and fidelity, and better interfacing with other programming environments. Each solution has been validated in practice, by me and by others, mostly in the context of industrial sized case studies.

In this introductory chapter we first set the stage by sketching the objectives and requirements of computer aided software engineering. Then the technological background of this thesis is introduced: generic language technology and ASF+SDF. We zoom in on two particular technologies: parsing and term rewriting. We identify research questions as we go and summarize them at the end of this chapter.

1.1 Computer aided software engineering

There are many operations on source code that are usually not catered for in the original design of programming languages, but are nevertheless important or even vital to the software life-cycle. In many cases, CASE tools can be constructed to automate these operations.

The underlying global motivation is cost reduction of the development of such tools, but we do not go into cost analyses directly. Instead we focus on simplicity and the level of automation of the method for constructing the tools and assume that related costs will diminish as these attributes improve.

Particular tools that describe, analyze or transform source code are considered to be case studies from the perspective of this thesis. The techniques described here can be applied to construct them. Apart from this tool construction domain, the tools themselves are equally worthy of study and ask for case studies. We will only occasionally discuss applications of these tools.

1.1.1 Source code

What we call *source code* are all sentences in the languages in which computer programs are written. The adjective "source" indicates that such sentences are the source of a translation to another format: *object code*.

By this definition, object code can be source code again, since we did not specify who or what wrote the source code in the first place. It can be produced by a human, a computer program or by *generatio spontanea*; it does not matter. The key feature of source code is that it defines a computer program in some language, and that this program is always subject to a translation. This translation, usually called compilation or interpretation, is meant to make execution of the described program possible.

Software engineering is the systematic approach to designing, constructing, analyzing and maintaining software. Source code is one of the raw materials from which software is constructed. The following software engineering disciplines particularly focus on source code:

Model driven engineering [79] to develop applications by first expressing them in a high level descriptive and technology independent format. An example is the UML language [80]. Then we express how such a definition gives rise to source code generators by making a particular selection of technologies.

Generative programming [60] to model similar software systems (families) such that using a concise requirements specification, customized software can automatically be constructed.

Programming language definition [67] to formally define the syntax, static semantics and dynamic semantics of a programming language. From such formal definitions programming language tools such as parsers, interpreters and editors can be generated.

Compiler construction [2] to build translators from high level programming languages to lower level programming languages or machine instructions.

Software maintenance and evolution [119] to ensure the continuity of software systems by gradually updating the source code to fix shortcomings and adapt to altering circumstances and requirements. Refactoring [74] is a special case of maintenance. It is used for changing source code in a step-by-step fashion, not

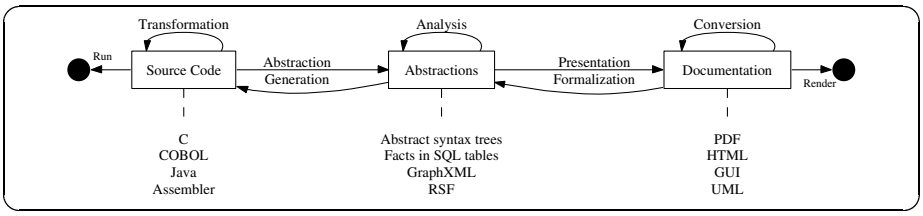

Figure 1.1: Three source code representation tiers and their (automated) transitions.

to alter its behavior, but to improve non-functional quality attributes such as simplicity, flexibility and clarity.

Software renovation [35] Reverse engineering is to analyze the source code of legacy software systems in order to retrieve their high-level design, and other relevant information. Re-engineering continues after reverse engineering, to adapt the derived abstractions to radically improve the functionality and non-functional quality attributes of a software system, after which an improved system will be derived.

For any of the above areas it is interesting to maximize the number of tasks that are automated during the engineering processes. Automation of a task is expected to improve both efficiency of the task itself, and possibly also some quality attributes of the resulting software. Examples of such attributes are correctness and tractability: trivial inconsistencies made by humans are avoided and automated processes can be traced and repeated more easily than human activities. We use the term *meta program* to refer to programs that automate the manipulation of source code. Thus we call the construction of such programs *meta programming*.

Many meta programs have been and will be developed to support the above engineering disciplines. Figure 1.1 sketches the domain, displaying all possible automated transitions from source code, via abstract representations, to documentation. Each of the above areas highlights and specializes a specific part of the graph in Figure 1.1. For example, reverse engineering is the path from source code, via several abstractions to documentation. In reverse engineering, extensive analysis of source code abstractions is common, but the other edges in the graph are usually not traversed. On the other hand, in model driven engineering we start from documentation, then formalize the documentation towards a more machine oriented description, before we generate actual source code.

The example languages for each tier are meant to be indicative, but not restrictive. A specific language might assume the role of source code, abstraction or documentation depending on the viewpoint that is imposed by a particular software engineering task. Take for example a context-free grammar written in the EBNF language. It is source code, since we can generate a parser from it using a parser generator. It is also an abstraction, if we would have obtained it from analyzing a parser written in Java source code. It serves as documentation when a programmer tries to learn the syntax of a language from it.

Each node in Figure 1.1 represents a particular collection of formalisms that are typically used to represent source code in that tier. Each formalism corresponds to a particular language, and thus each transition between these formalisms corresponds to a language translation. Even though each transition may have very specific properties, on some level of abstraction all of them are translations.

1.1.2 Source code analysis and transformation

The above described software engineering tasks define sources and targets, but they do not reveal the details or the characteristics of the translations they imply. Before we consider technical solutions, which is the main purpose of this thesis, we sketch the application domain of translation a bit further. We consider the kinds of translations that are depicted in Figure 1.1.

1.1.3 Translation distance

A coarse picture of a translation is obtained by visualizing what the distance is between the source and the target language. For example, by analyzing attributes of the syntax, static and dynamic semantics of languages they can be categorized into dialect families and paradigms. One might expect that the closer the attributes of the source and target languages are, the less complex a translation will be.

A number of language attributes are more pressing when we consider translation. Firstly, the application scope can range from highly domain specific to completely general purpose. Translations can stay within a scope or cross scope boundaries. Secondly, the level of embedding of a language is important. The level of embedding is a rough indication of the number of translation or interpretation steps that separate a language from the machine. Examples of high level languages with deep embeddings are Java and UML, while byte-code is a low level language. Translations can be vertical, which means going up or down in level, or horizontal, when the level remains equal. Thirdly, the execution mechanism can range from fully compiled, by translation to object code and linking with a large run-time library, to fully interpreted, by direct source execution. Translations that move from one mechanism to another can be hampered by bottlenecks in efficiency in one direction, or lack of expressivity in the other direction. Finally, the size of a language in terms of language constructs counts. A translation must deal with the difference in expressivity in both languages. Sometimes we must simulate a construct of the source language in the target language, compiling one construct into several constructs. Sometimes we must reverse simulate an idiom in the source language to a construct in the target language, condensing several constructs into one.

However, these attributes and the way they differ between source and target does not fully explain how hard a translation will be. A small dialect translation can be so intrinsic that it is almost impossible to obtain the desired result (e.g., COBOL dialect translations [145]). On the other hand, a cross paradigm and very steep translation can be relatively easy (e.g., COBOL source code to hypertext documentation). Clearly the complexity of a translation depends as much on the requirements of a translation as on the details of the source and target language.

1.1.4 Goals and requirements

The requirements of any CASE tool depend on its goal. We can categorize goals and requirements of CASE tools using the three source code representation tiers from Figure 1.1. We discuss each of the seven possible transitions from this perspective:

Transformation: translation between executable languages. This is done either towards runnable code (compilation), or to obtain humanly readable and maintainable code again (e.g., refactoring, de-compilation and source-to-source transformation). With transformation as a goal, the requirement is usually that the resulting code has at least the same observable behavior. With compilation, an additional requirement is that the result is as fast as possible when finally executed. In refactoring and source-to-source transformation, we want to retain as much properties from the original program as possible. Examples of problematic issues are restoring preprocessor macros and normalized code to the original state, and retaining the original layout and comments.

Abstraction: translation from source code to a more abstract representation of the facts that are present in source code. The abstract representation is not necessarily executable, but it must be sound with respect to the source code. The trade-off of such translations is the amount of information against the efficiency of the extraction.

Generation: translation from high level data to executable code. The result must be predictable, humanly readable, and sometimes even reversible (e.g., round-trip engineering). Sometimes it is even required to generate code that emits error messages on the level of abstraction of the source language instead of the target language.

Analysis: extension and elaboration of facts. We mean all computations that reorganize, aggregate or extrapolate the existing facts about source code. These computations are required to retain fact consistency. Also, speed of the process is usually a key factor, due to the usually high amount of facts and computational complexity of fact analysis. Note that we do not mean transformations of the input and output formats of all kinds of fact manipulation languages.

Presentation: compilation of facts into document formats or user-interface descriptions. The requirements are based on human expectations, such as user-friendliness and interactivity.

Formalization: extraction of useful facts from document formats or user-interfaces. For example, to give UML pictures a meaning by assigning semantics to diagrams. In this case the requirement is to extract the necessary information as unambiguously as possible. Sometimes, the goal is to extract as much information as possible. If possible this information is already consistent and unambiguous. If this is not the case, an analysis stage must deal with that problem. Formalization is a most tricky affair to fully automate. User-interaction or explicit adding of annotations by the user is usually required.

Conversion: transformation of one document format into another. The conversion must usually retain all available information, and sometimes even preserve the exact typographic measures of the rendered results.

Most CASE tools are staged into several of the above types of source code transitions. The requirements of each separate stage are simply accumulated. For example, in modern compilers there are separate stages for abstraction and analysis, that feed back information to the front end for error messages and to the back end for optimization. From the outside, these stages implement a transformation process, but internally almost all other goals are realized.

1.1.5 Mechanics

The mechanics of all CASE tools are also governed by the three source code representation tiers in Figure 1.1. Source code will be transposed from one representation to another, either within a tier, or from tier to tier. This induces the three basic stages of each CASE tool: input one representation, compute, and output another representation.

With each source code representation tier a particular class of data structures is typically associated. The source code tier is usually represented by files that contain lists of characters, or syntax trees that very closely correspond to these files. The abstract representation tier contains more elaborately structured data, like annotated trees, graphs, or tables. The documentation tier is visually oriented, containing descriptions of pictures basically.

Input and output of source code representations is about serialization and deserialization. Parsing is how to obtain a tree structured representation from a serial representation. Unparsing is the reverse. The mechanics of parsing and unparsing depend on the syntactic structure of the input format. For some languages in combination with some goals, regular expressions are powerful enough to extract the necessary structure. Other language/goal combinations require the construction of fully detailed abstract syntax trees using parsing technology. The mechanics of parsing have been underestimated for a while, but presently the subject is back on the agenda [7].

Computations on structured data come in many flavors. Usually tools specialize on certain data structures. For example, term rewriting specializes on transforming tree-structured data, while relational algebra deals with computations on large sets of tuples. The most popular quality attributes are *conciseness*, *correctness* and *efficiency*. Other important quality attributes are *fidelity* and *resolution*. High fidelity computations have less noise, because they do not loose data, or introduce junk. For example, we talk about a noisy analysis when it introduces false positives or false negatives and about a noisy transformation when all source code comments are lost. Resolution is the level of detail that a computation can process. High resolution services high-fidelity computations, but it must usually be traded for efficiency. For example, to be maximally efficient, compilers for programming languages work on *abstract* syntax trees. As a result the precision of the error messages they produce with respect to source code locations may be less precise.

In [152], the mechanics of tree transformation are described in a technology independent manner. Three aspects are identified: scope, direction and staging. Here

we use the same aspects to describe any computation on source code representations. *Scope* describes the relation between source and target structures of a computation on source code. A computation can have local-to-local, local-to-global, global-to-local, and global-to-global scope, depending on the data-flow within a single computation step. The *direction* of a computation is defined as being either *forward* (source driven) or *reverse* (target driven). Forward means that the target structure is generated while the source structure is traversed. Reverse means that a target template is traversed while the source structure is queried for information. The *staging* aspect, which is also discussed in [145], defines which intermediate results separate a number of passes over a structured representation. Disentangling simpler subcomputations from more complex ones is the basic motivation for having several stages.

The final challenge is to compose the different computations on source code representations. The mechanical issue is how to consolidate the different data structures that each tier specializes on. Parsing technology is a good example of how to bridge one of these gaps. The natural inclusion of trees into graphs is another. However, there are numerous trade-offs to consider. This subject is left largely untouched in this thesis.

1.1.6 Discussion: challenges in meta programming

Common ground. The above description of meta programming unifies a number of application areas by describing them from the perspective of source code representation tiers (Figure 1.1). In reality, each separate application area is studied without taking many results of the other applications of meta programming into account. There is a lack of common terminology and an identification of well known results and techniques that can be applied to meta programming in general.

The application of general purpose meta programming frameworks (Section 1.5) may offer a solution to this issue. Since each such framework tries to cover the entire range of source code manipulation applications, it must introduce all necessary conceptual abstractions that are practical to meta programming in general. This thesis contributes to such a common understanding by extending one particular framework to cover more application areas. The next obvious step is to identify the commonalities between all such generic meta programming frameworks.

Automation without heuristics. Large parts of meta programs can be generated from high level descriptions or generic components can be provided to implement these parts. It is easy to claim the benefit of such automation. On the other hand, such automation often leads to disappointment. For example, a parser generator like Yacc [92], a powerful tool in the compiler construction field, is not applicable in the reverse engineering field.

On the one hand, the fewer assumptions tools like Yacc make, the more generically applicable they are. On the other hand the more assumptions they make, the more automation they provide for a particular application area. The worst scenario for this trade-off is a tool that seems generically applicable, but nevertheless contains heuristic choices to automate certain functionality. This leads to blind spots in the understanding of the people that use this tool and inevitable errors.

For constructing meta programming tools the focus should be on exposing all parameters of certain algorithms and not on the amount of automation that may be provided in a certain application context. That inevitably results in less automation, but the automation that is provided is robust. Therefore, in this thesis I try to automate without introducing too many assumptions, and certainly without introducing any hidden heuristic choices.

High-level versus low-level unification. In the search for reuse and generic algorithms in the meta programming field the method of *unification* is frequently tried. A common *high-level* representation is searched for very similar artifacts. For example, Java and C# are so similar, we might define a common high-level language that unifies them, such that tools can be constructed that work on both languages. Usually, attempts at unification are much more ambitious than that. The high-level unification method is ultimately self-defeating: the details of the unification itself quickly reach and even surpass the complexity of the original tasks that had to be automated. This is an observation solely based on the success rate of such unification attempts.

The alternative is to not unify in high-level representations, but unify to much more low-level intermediate formats. Such formats are for example standardized parse tree formats, fact representation formats and byte-code. Common run-time environments, such as the Java Virtual Machine and the .NET Common Language Runtime are good examples. This is also the method in this thesis. We unify on low level data-structures that represent source code. The mapping of source code to these lower levels is done by algorithms that are configured on a high level by a language specialist. Orthogonally, we let specialists construct *libraries* of such configurations for large collections of languages. The challenge is to optimize the engineering process that bridges the gap between high-level and low-level source code representations, in both directions.

1.2 Technological background

Having explored the subject of interest in this thesis, we will now explain the technological background in which the research was done.

1.2.1 Generic language technology

With generic language technology we investigate whether a completely language-oriented viewpoint leads to a clear methodology and a comprehensive tool set for efficiently constructing meta programs.

This should not imply a quest for one single ultimate meta-programming language. The domain is much too diverse to tackle with such a unified approach. Even a single translation can be so complex as to allow several domains. The common language-oriented viewpoint does enable us to reuse components that are common to translations in general across these domains.

Each language, library or tool devised for a specific meta-programming domain should focus on being generic. For example, a parser generator should be able to deal with many kinds of programming languages and a transformation language should be

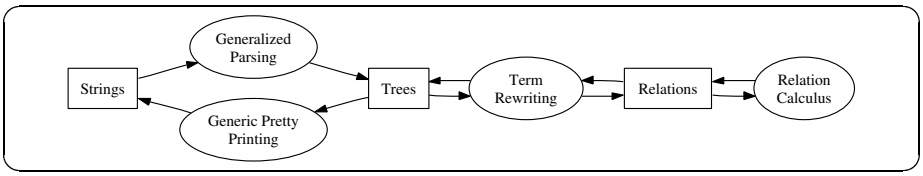

Figure 1.2: Generalized parsing, term rewriting, relational calculus and generic pretty-printing: a meta-programming framework.

able to deal with many different kinds of transformations. That is what being generic means in this context. It allows the resulting tool set to be comprehensive and complementary, as opposed to extensive and with much redundancy.

Another focus of Generic Language Technology is compositionality. As the sketch of Figure 1.1 indicates, many different paths through this graph are possible. The tools that implement the transitions between the nodes are meant to be composable to form complex operations and to be reusable between different applications and even different meta-programming disciplines. For example, if carefully designed, a parser generator developed in the context of a reverse engineering case study can be designed such that it is perfectly usable in the context of compiler construction as well.

1.2.2 A meta-programming framework

This thesis was written in the context of the Generic Language Technology project at CWI, aimed at developing a complete and comprehensive set of collaborating meta-programming tools: *the ASF+SDF Meta-Environment* [99, 42, 28].

Figure 1.2 depicts how the combination of the four technologies in this framework can cover all transitions between source code representations that we discussed. These technologies deal with three major data structures for language manipulation: strings, trees and relations. In principle, any translation expressed using this framework begins and ends with the string representation and covers one of the transitions in Figure 1.1.

Generalized parsing [141] offers a declarative mechanism to lift the linear string representation to a more structured tree representation.

Term rewriting [146] is an apt paradigm for deconstructing and constructing trees.

Relational calculus [61, 101] is designed to cope with large amounts of facts and the logic of deriving new facts from them. The link between term rewriting and relational calculus and back is made by encoding facts as a specific sort of trees.

Unparsing and generic pretty-printing [51, 64] A generic pretty-printer allows the declarative specification of how trees map to tokens that are aligned in two dimensions. Unparsing simply maps trees back to strings in a one-dimensional manner.

Paths through the framework in Figure 1.2 correspond to the compositionality of tools. For example, a two-pass parsing architecture (pre-processing) can be obtained

Technology	Language	References	Goal
Generalized parsing	SDF	[87, 157]	Mapping strings to trees
Term rewriting	ASF	[30, 67]	Tree transformation
Generic pretty-printing	BOX	[51, 27, 63]	Mapping trees to strings
Relational calculus	RScript	[101]	Analysis and deduction
Process algebra	TScript	[12]	Tool composition

Table 1.1: Domain specific languages in the Meta-Environment.

by looping twice through generalized parsing via term rewriting and pretty-printing. Several analyses can be composed by iteratively applying the relational calculus. The enabling feature in any framework for such compositionality is the rigid standardization of the string, tree, and relational data formats.

The programming environment that combines and coordinates the corresponding tools is called the ASF+SDF Meta-Environment. This system provides a graphical user-interface that offers syntax-directed editors and other visualizations and feedback of language aspects. It integrates all technologies into one meta-programming workbench.

Table 1.1 introduces the domain specific languages that we use for each technology in our framework. The Syntax Definition Formalism (SDF) for the generation of parsers, the Algebraic Specification Formalism (ASF) for the definition of rewriting, BOX for the specification of pretty-printing and RScript implements a language for relational calculus.

TScript offers a general solution for component composition for applications that consist of many programming languages. The language is based on process algebra. In the Meta-Environment this technology is applied to compose the separate tools. Note that TScript is a general purpose component glue, not limited to meta-programming at all.

1.2.3 Historical perspective

The original goal of the ASF+SDF Meta-Environment is *generating interactive programming environments* automatically from programming language descriptions.

SDF was developed to describe the syntax of programming languages, and ASF to describe their semantics. From these definitions parsers, compilers, interpreters and syntax-directed editors can be generated. The combination of these generated tools forms a programming environment for the described language [99].

At the starting point of this thesis, ASF, SDF, and the Meta-Environment existed already and had been developed with generation of interactive programming environments in mind. As changing requirements and new application domains for this system arose, the need for a complete redesign of the environment was recognized. For example, in addition to the definition of programming languages, renovating COBOL systems became an important application of the ASF+SDF Meta-Environment. To accommodate these and future developments its design was changed from a closed homogeneous Lisp-based system to an open heterogeneous component-based environment

written in C, Java, TScript and ASF+SDF [28].

While the ASF+SDF formalism was originally developed towards generating interactive programming environments, a number of experiences showed that it was fit for a versatile collection of applications [29]. The following is an incomplete list of examples:

- Implementation of domain specific languages [6, 68, 69, 67],
- Renovating Cobol legacy systems [143, 48, 34, 49],
- Grammar engineering [47, 114, 102]
- Model driven engineering [19]

Driven by these applications, the focus of ASF+SDF changed from generating interactive programming environments to interactive implementation of meta-programming tools. This focus is slightly more general in a way, since interactive programming environments are specific collections of meta-programming tools. On the other hand, re-engineering, reverse engineering and source-to-source transformation were pointed out as particularly interesting application areas, which has led to specific extensions to term rewriting described in this thesis.

1.2.4 Goal

The overall question is how well qualified and versatile ASF+SDF really is with respect to the new application areas. The goal is to cast ASF+SDF into a general purpose meta programming language. In the remainder of this introduction, we describe ASF+SDF and its technological details. We will identify issues in its application to meta programming. Each issue should give rise to one or more improvements in the ASF+SDF formalism or its underlying technology. For both SDF (parsing) and ASF (rewriting), the discussion is organized as follows:

- The mechanics of the domain,
- The formalism that captures the domain,
- The technology that backs up the formalism,
- The bottlenecks in the application to meta programming.

The validation of the solutions presented in this thesis is done by empirical study. First a requirement or shortcoming is identified. Then, we develop a solution in the form of a new tool or by adapting existing tools. We test the new tools by applying them to automate a real programming task in a case study. The result is judged by quality aspects of the automated task, and compared with comparing or otherwise relating technologies. Success is measured by evaluating the gap between requirements of each case study and the features that each technological solution provides.

1.3 Parsing

1.3.1 Mechanics

A parser must be constructed for every new language, implementing the mapping from source code in string representation to a tree representation. A well known solution for automating the construction of such a parser is by generating it from a context-free grammar definition. A common tool that is freely available for this purpose is for example Yacc [92].

Alternatively, one can resort to lower level techniques like scanning using regular expressions or manual construction of a parser in a general purpose programming language. Although these approaches are more lightweight, we consider generation of a parser from a grammar preferable. Ideally, a grammar can serve three purposes at the same time:

- Language documentation,
- Input to a parser generator,
- Exact definition of the syntax trees that a generated parser produces.

These three purposes naturally complement each other in the process of designing meta programs [93]. There are also some drawbacks from generating parsers:

- A generated parser usually depends on a parser driver, a parse table interpreter, which naturally depends on a particular programming environment. The driver, which is a non-trivial piece of software, must be ported if another environment is required.
- Writing a large grammar, although the result is more concise, is not less of an intellectual effort than programming a parser manually.

From our point of view the first practical disadvantage is insignificant as compared to the conceptual and engineering advantages of parser generation. The second point is approached by the Meta-Environment which provides a domain specific user-interface with visualization and debugging support for grammar development.

1.3.2 Formalism

We use the language SDF to define the syntax of languages [87, 157]. From SDF definitions parsers are generated that implement the SGLR parsing algorithm [157, 46]. SDF and SGLR have a number of distinguishing features, all targeted towards allowing a bigger class of languages to be defined, while allowing the possibility for automatically generating parsers.

SDF is a language similar to BNF [11], based on context-free production rules. It integrates lexical and context-free syntax and allows modularity in syntax definitions. Next to production rules SDF offers a number of constructs for declarative grammar

disambiguation, such as priority between operators. A number of short-hands for regular composition of non-terminals are present, such as lists and optionals, which allow syntax definitions to be concise and intentional.

The most significant benefit of SDF is that it does not impose *a priori* restrictions on the grammar. Other formalisms impose grammar restrictions for the benefit of efficiency of generated scanners and parsers, or to rule out grammatical ambiguity beforehand. In reality, the syntax of existing programming languages does not fit these restrictions. So, when applying such restricted formalisms to the field of meta-programming they quickly fall short.

By removing the conventional grammar restrictions and adding notations for disambiguation next to the grammar productions, SDF allows the syntax of more languages to be described. It is expressive enough for defining the syntax of real programming languages such as COBOL, Java, C and PL/I. The details on SDF can be found in [157, 32], and in Chapter 3.

We discuss the second version of SDF, as described by Visser in [157]. This version improved on previous versions of SDF [87]. A scannerless parsing model was introduced, and with it the difference in expressive power between lexical and context-free syntax was removed. Its design was made modular and extensible. Also, some declarative grammar disambiguation constructs were introduced.

1.3.3 Technology

To sustain the expressiveness that is available in SDF, it is supported by a scannerless generalized parsing algorithm: SGLR [157]. An architecture with a scanner implies either restrictions on the lexical syntax that SDF does not impose, or some more elaborate interaction between scanner and parser (e.g., [10]). Instead we do not have a scanner. A parse table is generated from an SDF definition down to the character level and then the tokens for the generated parser are ASCII characters.

In order to be able to deal with the entire class of context-free grammars, we use generalized LR parsing [149]. This algorithm accepts all context-free languages by administrating several parse stacks in parallel during LR parsing. The result is that GLR algorithms can overcome parse table conflicts, and even produce *parse forests* instead of parse trees when a grammar is ambiguous. We use an updated GLR algorithm [130, 138] extended with disambiguation constructs for scannerless parsing. Details about scannerless parsing and the aforementioned disambiguations can be found in Chapter 3 of this thesis.

Theme: disambiguation is a separate concern Disambiguation should be seen as a separate concern, apart from grammar definition. However, a common viewpoint is to see ambiguity as an error of the production rules. From this view, the logical thing to do is to fix the production rules of the grammar such that they do not possess ambiguities. The introduction of extra non-terminals with complex naming schemes is often the result. Such action undermines two of the three aforementioned purposes of grammar definitions: language documentation and exact definition of the syntax trees. The grammar becomes unreadable, and the syntax trees skewed.

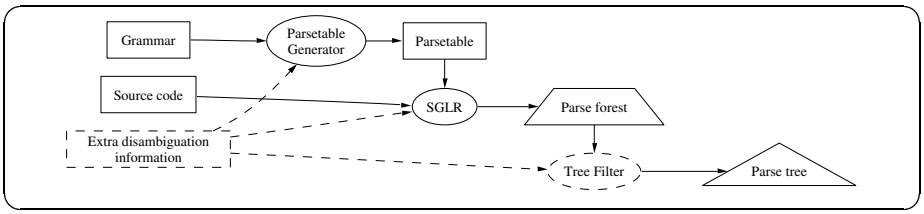

Figure 1.3: Disambiguation as a separate concerns in a parsing architecture.

Our view is based on the following intuition: grammar definition and grammar disambiguation, although related, are completely different types of operations. In fact, they operate on different data types. On the one hand a grammar defines a mapping from strings to parse trees. On the other hand disambiguations define choices between these parse trees: a mapping from parse forests to smaller parse forests. The separation is more apparent when more complex analyses are needed for defining the correct parse tree, but it is just as real for simple ambiguities.

This viewpoint is illustrated by Figure 1.3. It is the main theme for the chapters on disambiguation (Chapters 3, 4, and 5). The method in these chapters is to attack the problem of grammatical ambiguity sideways, by providing external mechanisms for filtering parse forests.

Also note the difference between a *parse table conflict* and an *ambiguity* in a grammar. A parse table conflict is a technology dependent artifact, depending on many factors, such as the details of the algorithm used to generate the parse table. It is true that ambiguous grammars lead to parse table conflicts. However, a non-ambiguous grammar may also introduce conflicts. Such conflicts are a result of the limited amount of lookahead that is available at parse table generation time.

Due to GLR parsing, the parser effectively has an unlimited amount of lookahead to overcome parse table conflicts. This leaves us with the real grammatical ambiguities to solve, which are not an artifact of some specific parser generation algorithm, but of context-free grammars in general. In this manner, GLR algorithms provide us with the opportunity to deal with grammatical ambiguity as a separate concern even on the implementation level.

1.3.4 Application to meta-programming

The amount of generality that SDF and SGLR allow us in defining syntax and generating parsers is of importance. It enables us to implement the syntax of real programming languages in a declarative manner, that would otherwise require low level programming. The consequence of this freedom is however syntactic ambiguity. An SGLR parser may recognize a program, but produce several parse trees instead of just one because the grammar allows several derivations for the same string.

In practice it appears that many programming languages do not have an unambiguous context-free grammar, or at least not a readable and humanly understandable one. An unambiguous scannerless context-free grammar is even harder to find, due to the

absence of implicit lexical disambiguation rules that are present in most scanners. Still for most programming languages, there is only one syntax tree that is defined to be the "correct" one. This tree corresponds best to the intended semantics of the described language. Defining a choice for this correct parse tree is called *disambiguation* [104].

So the technique of SGLR parsing allows us to generate parsers for real programming languages, but real programming languages seem to have ambiguous grammars. SGLR is therefore not sufficiently complete to deal with the meta-programming domain. This gives rise to the following research question which is addressed in Chapters 3 and 4:

Research Question 1

How can disambiguations of context-free grammars be defined and implemented effectively?

1.4 Rewriting

1.4.1 Mechanics

After a parser has produced a tree representation of a program, we want to express analyses and transformations on it. This can be done in any general purpose programming language. The following aspects of tree analyses and transformation are candidates for abstraction and automation:

- Tree construction: to build new (sub)trees in a type-safe manner.
- Tree deconstruction: to extract relevant information from a tree.
- Pattern recognition: to decide if a certain subtree is of a particular form.
- Tree traversal: to locate a certain subtree in a large context.
- Information distribution: to distribute information that was acquired elsewhere to specific sites in a tree.

The term rewriting paradigm covers most of the above by offering the concept of a rewrite rule [16]. A rewrite rule $l \rightarrow r$ consists of two tree patterns. The left-hand side of a rule matches tree patterns, which means identification and deconstruction of a tree. The right-hand side then constructs a new tree by instantiating a new pattern and replacing the old tree. A particular traversal strategy over a subject tree searches for possible applications of rewrite rules, automating the tree traversal aspect. By introducing conditional rewrite rules and using function symbols, or applying so-called rewriting strategies [20, 159, 137], the rewrite process is controllable such that complex transformations can be expressed in a concise manner.

Term rewriting specifications can be compiled to efficient programs in a general purpose language such as C [33]. We claim the benefits of generative programming: higher intentionality, domain specific error messages, and generality combined with efficiency [60].

Other paradigms that closely resemble the level of abstraction that is offered by term rewriting are attribute grammars and functional programming. We prefer term rewriting because of the more concise expressiveness for matching and construction complex tree patterns that is not generally found in these other paradigms. Also, the search for complex patterns is automated in term rewriting. As described in the following, term rewriting allows a seamless integration of the syntactic and semantic domains.

1.4.2 Formalism

We use the Algebraic Specification Formalism (ASF) for defining rewriting systems. ASF has one important feature that makes it particularly apt in the domain of meta-programming: the terms that are rewritten are expressed in user-defined concrete syntax. This means that tree patterns are expressed in the same programming language that is analyzed or transformed, extended with pattern variables (See Chapter 5 for examples).

The user first defines the syntax of a language in SDF, then extends the syntax with notation for meta variables in SDF, and then defines operations on programs in that language using ASF. Because of the seamless integration the combined language is called ASF+SDF. Several other features complete ASF+SDF:

- Parameterized modules: for defining polymorphic reusable data structures,
- Conditional rewrite rules: a versatile mechanism allowing for example to define the preconditions of rule application, and factoring out common subexpressions,
- Default rewrite rules: two level ordering of rewrite rule application, for prioritizing overlapping rewrite rules.
- List matching: allowing concise description of all kinds of list traversals. Computer programs frequently consist of lists of statements, expressions, or declarations, so this feature is practical in the area of meta-programming,
- Layout abstraction: the formatting of terms is ignored during matching and construction of terms,
- Statically type-checked. Each ASF term rewriting system is statically guaranteed to return only programs that are structured according to the corresponding SDF syntax definition.

ASF is basically a functional language without any built-in data types: there are only terms and conditional rewrite rules on terms available. Parameterized modules are used to create a library of commonly used generic data structures such as lists, sets, booleans, integers and real numbers.

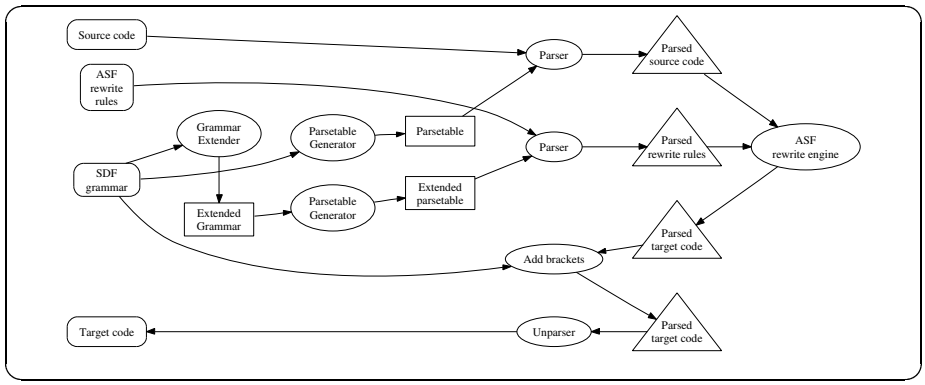

Figure 1.4: The parsing and rewriting architecture of ASF+SDF.

1.4.3 Technology

In ASF+SDF grammars are coupled to term rewriting systems in a straightforward manner: the parse trees of SDF are the terms of ASF. More specifically that means that the non-terminals and productions in SDF grammars are the sorts and function symbols of ASF term rewriting systems. Consequently, the types of ASF terms are restricted: first-order and without parametric polymorphism. Other kinds of polymorphism are naturally expressed in SDF, such as for example overloading operators with different types of arguments, or different types of results. Term rewriting systems also have variables. For this the SDF formalism was extended with variable productions.

Figure 1.4 depicts the general architecture of ASF+SDF. In this picture we can replace the box labeled ““ASF rewrite engine” by either an ASF interpreter or a compiled ASF specification. Starting from an SDF definition two parse tables are generated. The first is used to parse input source code. The second is obtained by extending the syntax with ASF specific productions. This table is used to parse the ASF equations. The rewriting engine takes a parse tree as input, and returns a parse tree as output. To obtain source code again, the parse tree is unparsed, but not before some post-processing. A small tool inserts brackets productions into the target tree where the tree violates priority or associativity rules that have been defined in SDF.

Note that a single SDF grammar can contain the syntax definitions of different source and target languages, so the architecture is not restricted to single languages. In fact, each ASF+SDF module combines one SDF module with one ASF module. So, every rewriting module can deal with new syntactic constructs.

The execution algorithm for ASF term rewriting systems can be described as follows. The main loop is a bottom-up traversal of the input parse tree. Each node that is visited is transformed as many times as possible while there are rewrite rules applicable to that node. This particular reduction strategy is called *innermost*. A rewrite rule is applicable when the pattern on the left-hand side matches the visited node, and all conditions are satisfied. Compiled ASF specifications implement the same algorithm, but efficiency is improved by partial evaluation and factoring out common subcompu-

tations [33].

To summarize, ASF is a small, eager, purely functional, and executable formalism based on conditional rewrite rules. It has a fast implementation.

1.4.4 Application to meta-programming

There are three problem areas regarding the application of ASF+SDF to meta-programming:

Conciseness. Although term rewriting offers many practical primitives, large languages still imply large specifications. However, all source code transformations are similar in many ways. Firstly, the number of trivial lines in an ASF+SDF program that are simply used for *traversing* language constructs is huge. Secondly, passing around *context information* through a specification causes ASF+SDF specifications to look repetitive sometimes. Thirdly, the *generics modules* that ASF+SDF provides can also be used to express generic functions, but the syntactic overhead is considerable. This limits the usability of a library of reusable functionality.

Low fidelity. Layout and source code comments are lost during the rewriting process. From the users perspective, this loss of information is unwanted noise of the technology. Layout abstraction during rewriting is usually necessary, but it can also be destructive if implemented naively. At the very least the transformation that does nothing should leave any program unaltered, including its textual formatting and including the original source code comments.

The interaction possibilities of an ASF+SDF tool with its environment are limited to basic functional behavior: parse tree in, parse tree out. There is no other communication possible. How to integrate an ASF+SDF meta tool in another environment? Conversely, how to integrate foreign tools and let them communicate with ASF+SDF and the Meta-Environment? The above limitations prevent the technology from being acceptable in existing software processes that require meta-programming.

Each of the above problem areas gives rises to a general research question in this thesis.

Research Question 2

How to improve the conciseness of meta programs?

The term rewriting execution mechanism supports very large languages, and large programs to rewrite. It is the size of the specification that grows too fast. We will analyze why this is the case for three aspects of ASF+SDF specifications: tree traversal, passing context information and reusing function definitions.

Traversal. Although term rewriting has many features that make it apt in the meta programming area, there is one particularity. The non-deterministic behavior of term rewriting systems, that may lead to non-confluence[1], is usually an unwanted feature in the meta programming paradigm. While non-determinism is a valuable asset in some other application areas, in the area of meta-programming we need deterministic computation most of the time. The larger a language is and the more complex a transformation, the harder is becomes to understand the behavior of a term rewriting system. This is a serious bottleneck in the application of term rewriting to meta programming.

The non-determinism of term rewriting systems is an intensively studied problem [16], resulting in solutions that introduce term rewriting strategies [20, 159, 137]. Strategies limit the non-determinism by letting the programmer explicitly denote the order of application of rewrite rules. One or all of the following aspects are made programmable:

- Choice of which rewrite rules to apply.
- Order of rewrite rule application.
- Order of tree traversal.

If we view a rewrite rule as a first order function on a well-known tree data structure, we can conclude that strategies let features of functional programming seep into the term rewriting paradigm: explicit function/rewrite rule application and higher order functions/strategies. As a result, term rewriting with strategies is highly comparable to higher order functional programming with powerful matching features.

In ASF+SDF we adopted a functional style of programming more directly. First-order functional programming in ASF+SDF can be done by defining function symbols in SDF to describe their type, and rewrite rules in ASF to describe their effect. This simple approach makes choice and order of rewrite rule application explicit in a straightforward and manner: by functional composition.

However, the functional style does not directly offer effective means for describing tree traversal. Traversal must be implemented manually by implementing complex, but boring functions that recursively traverse syntax trees. The amount and size of these functions depend on the size of the object language. This specific problem of conciseness is studied and resolved in Chapter 6:

Context information. An added advantage of the functional style is that context information can be passed naturally as extra arguments to functions. That does mean that all information necessary during a computation should be carried through the main thread of computation. This imposes bottlenecks on specification size, and separation of concerns because nearly all functions in a computation must thread all information.

Tree decoration is not addressed by the term rewriting paradigm, but can be a very practical feature for dealing with context information [107]. Its main merit is that it allows separation of data acquisition stages from tree transformation stages without the need for constructing elaborate intermediate data structures. It could substantially

[1] See Section 6.1.5 on page 105 for an explanation of confluence in term rewriting systems

alleviate the context information problem. The *scaffolding* technique, described in [142], prototypes this idea by scaffolding a language definition with extension points for data storage.

This thesis does not contain specific solutions to the context information problem. However, traversal functions (Chapter 6) alleviate the problem by automatically threading of data through recursive application of a function. Furthermore, a straightforward extension of ASF+SDF that allows the user to store and retrieve any annotation on a tree also provides an angle for solving many context information issues. We refer to [107] for an analysis and extrapolation of its capabilities.

Parameterized modules. The design of ASF+SDF limits the language to a first-order typing system without parametric polymorphism. Reusable functions that can therefore not easily be expressed. The parameterized modules of ASF+SDF do allow the definition of functions that have a parameterized type, but the user must import a module and bind an actual type to the formal type parameter manually.

The reason for the lack of type inference in ASF+SDF is the following circular dependency: to infer a type of an expression it must be parsed, and to parse the expression its type must be known. Due to full user-defined syntax, the expression can only be parsed correctly after the type has been inferred. The problem is a direct artifact of the architecture depicted in Figure 1.4.

The conciseness of ASF+SDF specifications is influenced by the above design. Very little syntactic overhead is needed to separate the meta level from the object level syntax, because a specialized parser is generated for every module. On the other hand, the restricted type system prohibits the easy specification of reusable functions, which contradicts conciseness. In Chapter 5 we investigate whether we can reconcile syntactic limitations with the introduction of polymorphic functions.

Research Question 3

How to improve the fidelity of meta programs?

A requirement in many meta-programming applications is that the tool is very conservative with respect to the original source code. For example, a common process in software maintenance is updating to a new version of a language. A lot of small (syntactical) changes have to be made in a large set of source files. Such process can be automated using a meta programming tool, but the tool must change only what is needed and keep the rest of the program recognizable to the human maintainers.

The architecture in Figure 1.4 allows, in principle, to parse, rewrite and unparse a program without loss of any information. If no transformations are necessary during rewriting, the exact same file can be returned including formatting and source code comments. The enabling feature is the parse tree data structure, which contains all characters of the original source code at its leaf nodes: a maximally high-resolution data structure. However the computational process of rewriting, and the way a transformation is expressed by a programmer in terms of rewrite rules may introduce unwanted

side-effects:

- ASF abstracts from the particular layout and source code comments of source code patterns during matching.
- ASF abstracts from complex lexical syntax structures and maps them to strings of characters.
- ASF abstracts from applications of bracket productions; they are removed, and necessary brackets are reintroduced after rewriting again.

Each of the above abstractions services the conciseness of ASF+SDF specifications, but hampers their fidelity. Even the identity transformation might result in the loss of some characters in the input code. In Chapter 7 and Chapter 8 we address these issues.

Research Question 4

How to improve the interaction of meta programs with their environment?

Data. One aspect of interaction of a meta program with its environment is the data integration. Since in ASF+SDF the parse tree data structure plays a central role (Figure 1.4), the question is how we can communicate parse trees. The ApiGen tool [62] generates implementations of the exact syntax trees of ASF+SDF in the C programming language. This is the first and foremost step towards connecting ASF+SDF components to C components. The authors of [62] characterize the generated interfaces by efficiency, readability and type-safety.

The Java language is becoming increasingly popular, especially in the area of user interfaces and visualization. The question arises whether the same ApiGen solution can apply to this language, with the requirement of obtaining the same quality attributes. We answer this in Chapter 9.

Coordination. The ApiGen solution services data integration, the question remains how to organize the control flow of ASF+SDF and the Meta-Environment with foreign tools. Several steps towards such integration are taken in Chapter 2. In this chapter, we strongly separate the formalism from its particular programming environment to ensure both are able to communicate with any other partners.

1.5 Related work

The computer aided software engineering or meta-programming field is not identifiable as a single research community. It is divided more or less among the lines of the different *application areas* of meta-programming. To compensate, related work must be

Feature overlap ...	ANTLR	DMS	ELAN	Eli	JastAdd	Maude	Stratego/XT	TXL
Meta-programming system	✓	✓		✓	✓		✓	✓
Uses generalized parsing	✓	✓					✓	✓
Uses term rewriting		✓	✓			✓	✓	✓
Shares components with ASF+SDF			✓				✓	

Table 1.2: Systems related to the ASF+SDF Meta-Environment .

discussed in a highly focused manner. We refer to the relevant chapters for a discussion on the related work on specific research questions.

However, there are a number of highly related systems that compare to the ASF+SDF Meta-Environment at one or more levels: ANTLR [135], DMS [13] , ELAN [23], Eli [96], JastAdd [84], Maude [57] , Stratego/XT [162], and TXL [59]. Of those systems, ASF+SDF, DMS, Eli, JastAdd, TXL and Stratego/XT are designed as language processing toolkits. ELAN and Maude are logical frameworks based on parsing and rewriting that can be applied to language processing just as well. Table 1.2 shows on which level each system compares to the ASF+SDF Meta-Environment.

Note that in Table 1.2 I have used the terms *generalized parsing* and *term rewriting* to denote classes of algorithms and language features. For generalized parsing a whole range of parsing algorithms exist. For example, it can be a non-deterministic LL parser that employs backtracking to produce a single derivation. It can also be a Tomita-style generalized LR parser that produces all possible derivations. Term rewriting is used to denote all algorithms that use basic features available from the term rewriting formalism, such as matching and construction of trees. The actual language may offer only the basics, or also automated traversals, strategies, and side-effects.

The point is that although each of the above systems has claimed a niche of its own, from a meta-programming perspective they are highly comparable. At least algorithms, optimizations and language features carry over easily between these systems conceptually, and sometimes even on the implementation level. We do not explicitly compare the above systems from a general perspective in this thesis, rather one a minute level compare features where appropriate.

1.6 Road-map and acknowledgments

The thesis is organized into mostly previously published chapters that each target specific research questions. Table 1.3 maps the questions to the chapters and corresponding publications. The chapters can be read independently and are grouped into four parts.

The following list details the origins of all chapters including their respective co-authors, and other due acknowledgments. Because in our group we always order authors of a publication alphabetically, it is not immediately obvious who is mainly re-

#	Research questions	Chapter	Publication
1	How can disambiguations of context-free grammars be defined and implemented effectively?	3, 4	[46, 40]
2	How to improve the conciseness of meta programs?	5, 6	[39, 38]
3	How to improve the fidelity of meta programs?	7,8	[50]
4	How to improve the interaction of meta programs with their environment?	2, 9	[45, 44]

Table 1.3: Research questions in this thesis.

sponsible for each result. Therefore, I explain my exact contribution to each publication.

Part I Overview

Chapter 1. Introduction. This chapter provides an introduction and motivation to the subjects in this thesis.

Chapter 2. Environments for Term Rewriting Engines for Free. This chapter was published at RTA in 2002 [44], and is co-authored by Mark van den Brand and Pierre-Etienne Moreau. This work documents an intermediate stage in the process of enabling the ASF+SDF Meta-Environment towards full configurability and extensibility. The architecture described here has been used at CWI to construct several versions of the ASF+SDF Meta-Environment itself, at INRIA-LORIA it was used to construct an IDE for the ELAN language, and later at BRICS to construct an IDE for Action Notation [128, 129].

My contribution consists of the design and implementation of the proposed architectural solution, and translating this result to applications in the term rewriting domain. Mark van den Brand contributed by implementing or enabling all ELAN components for use in the Meta-Environment. Pierre-Etienne Moreau provided the components for the Rho Calculus case-study.

I would like to thank the numerous people that have worked on the design and implementation of the components of all the above systems.

Part II Parsing and disambiguation

Chapter 3. Disambiguation Filters for Scannerless Generalized LR Parsers. I co-authored this chapter with Mark van den Brand, Jeroen Scheerder and Eelco Visser. It was published in CC 2002 [46]. This publication marks a stable implementation of the SGLR parser and disambiguation filters that has since been used in numerous systems and applications.

My contribution to this chapter is not the SGLR algorithm itself. It was published

before [88, 157]. Together with Mark van den Brand, I have contributed to the empirical validation of this algorithm by (re)implementing large parts of it, and completely redesigning the architecture for filtering ambiguities during and after parsing. The initial version of this chapter was written by me, and it has been updated after publication with more discussion on the usefulness of SGLR parsing.

The feedback of many users has been indispensable while developing SGLR. Hayco de Jong and Pieter Olivier dedicated considerable time on improving SGLR efficiency. I would like to thank Martin Bravenboer, Merijn de Jonge, Joost Visser for their use of and feedback on SGLR. I thank Rob van der Leek, Leon Moonen, and Ernst-Jan Verhoeven for putting SGLR to the test with "Island Parsing". Jan Heering and Paul Klint provided valuable input when discussing design and implementation of SGLR.

Chapter 4. Semantics Driven Disambiguation. This chapter was published at LDTA 2003 [40], and co-authored by Mark van den Brand, Steven Klusener, and Leon Moonen. It continues where Chapter 3 stopped: how to deal with the ambiguities that can not be modeled with context-free disambiguation concepts. The subject of the chapter balances between between parsing and term rewriting. The method used is term rewriting, but since the goal is disambiguation it is located in this part.

The algorithm and techniques for disambiguation by rewriting that are proposed in this chapter are my contribution. The work was supervised by Mark van den Brand. Benchmarking and validating them on (industrial) cases was done by Steven Klusener and Leon Moonen.

Chapter 5. A Type-driven approach to Concrete Meta Programming. This chapter reports on experiments with the advanced concrete syntax features of ASF+SDF. A slightly abbreviated version of this chapter has been published in RISE 2005 [155]. This chapter is an extreme application of disambiguation filters as studied in Chapter 3 and Chapter 4: we effectively introduce syntactic ambiguity to optimize syntactic features, only to deterministically filter them later.

Filtering ambiguity by type-checking was independently researched by me at CWI and by Martin Bravenboer at Utrecht University. We decided to join forces resulting in my co-authorship of a GPCE publication on meta programming with concrete syntax in Java [52]. This chapter pushes one step further by introducing the fully automated inference of quoting transitions between meta and object language, including its application to the design of ASF+SDF.

I would like to thank Paul Klint, Mark van den Brand and Tijs van der Storm for their valuable comments on drafts of this chapter. Furthermore, I thank Martin Bravenboer, Rob Vermaas and Eelco Visser for our collaboration on this subject.

Part III Rewriting source code

Chapter 6. Term Rewriting with Traversal Functions. This chapter was published in Transactions on Software Engineering and Methodology (TOSEM) in 2003 [39]. An extended abstract appeared in the proceedings of the Workshop on Rewriting Strategies (WRS) 2002 [38]. Both papers are co-authored by Paul Klint and Mark van den Brand.

The algorithms and techniques in this chapter are my contribution, although shaped by numerous discussions with Paul Klint and Mark van den Brand. As discussed in the chapter, they have an origin in the work by Alex Sellink and Chris Verhoef on Renovation Factories.

I would like to thank the earliest users of traversal functions for their feedback and patience: Eva van Emden, Steven Klusener, Ralf Lämmel, Niels Veerman, Guido Wachsmuth and Hans Zaadnoordijk. Furthermore, I thank Alex Sellink, Ralf Lämmel, Chris Verhoef, Eelco Visser, and Joost Visser for their feedback on this work and their respective studies on the subject of tree traversal mechanisms.

Chapter 7. Rewriting with Layout. This chapter was published in RULE 2000 [50], and co-authored by Mark van den Brand. It has been updated to reflect current developments in ASF+SDF. The results have been used in practice, in particular in the CALCE software renovation project.

This chapter was written by me and supervised by Mark van den Brand.

I would like to thank Steven Klusener for providing data for the case in this chapter, and for his feedback on ASF+SDF with regard to high-fidelity transformations. Together with the results of Chapter 6, this work enables ASF+SDF to be applied to industrial cases.

Chapter 8. First Class Layout. This chapter has not been published outside this thesis. It reports on extensions of ASF+SDF towards full fidelity and resolution for source code transformations.

Magiel Bruntink and I worked together to manually check the results of the two fact extractors for the case study for this chapter. I would like to thank D.G. Waddington and Bin Yoa for coining the term “high-fidelity” in the context of source code transformations [165].

Chapter 9. A Generator of Efficient Strongly Typed Abstract Syntax Trees in Java. This chapter was published in IEE Proceedings – Software in 2005. It is co-authored by Mark van den Brand and Pierre-Etienne Moreau. It is located in the term rewriting part, since syntax trees are the basic data structure that is used in the ASF rewriting engine. As such, the results in Chapter 9 can be used to construct a Java back-end for the ASF compiler.

The algorithms and techniques in this paper are a result of several intensive sessions with Pierre-Etienne Moreau at INRIA-LORIA. He has contributed the idea of a generic shared object factory, while I have designed and implemented the three tier layer of the generated Java API. The JTom compiler that is used as a case study in this chapter is

written by Pierre-Etienne Moreau. I have written the chapter. Note that the subject of API generation was fully inspired by the C version of ApiGen, written by Pieter Olivier and Hayco de Jong.

Part IV

Chapter 10. Conclusions. In this chapter we briefly revisit the research questions and review how they have been answered. We also identify the gaps and opportunities for future research. We finish by summarizing the software developed in the context of this thesis.

Chapter 11. Samenvatting. This chapter summarizes the thesis in Dutch.

CHAPTER 2

Environments for Term Rewriting Engines for Free!

Term rewriting can only be applied if practical implementations of term rewriting engines exist. New rewriting engines are designed and implemented either to experiment with new (theoretical) results or to be able to tackle new application areas. In this chapter we present the Meta-Environment: an environment for rapidly implementing the syntax and semantics of term rewriting based formalisms. We provide not only the basic building blocks, but complete interactive programming environments that only need to be instantiated by the details of a new formalism. [1]

2.1 Introduction

Term rewriting can only be applied if practical implementations of term rewriting engines exist. New rewriting engines are designed and implemented either to experiment with new (theoretical) results or to be able to tackle new application areas, e.g., protocol verification, software renovation, etc. However, rewrite engines alone are not enough to implement real applications.

An analysis of existing applications of term rewriting, e.g., facilitated by formalisms like ASF+SDF [67], ELAN [22], Maude [58], RRL [95], Stratego [159], TXL [59], reveals the following four required aspects:

- a *formalism* that can be executed by a rewriting engine,
- *parsers* to implement the syntax of the formalism and the terms,
- a *rewriting engine* to implement the semantics of the formalism,
- a *programming environment* for supporting user-interaction, which can range from a set of commandline tools to a full-fledged interactive development environment (IDE).

[1] This chapter was published at RTA in 2002 [44], and is co-authored by Mark van den Brand and Pierre-Etienne Moreau.

A formalism introduces the syntactic notions that correspond to the operational semantics of the rewriting engine. This allows the user to write readable specifications. The parsers provide the connection from the formalism to the rewriting engine via abstract syntax trees. The programming environment can be either a set of practical command line tools, an integrated system with a graphical user-interface, or some combination. It offers a user-interface tailored towards the formalism for interacting with the rewriting engine. For a detailed overview of rewriting-based systems we refer to [146].

Implementing the above four entities is usually a major research and software engineering effort, even if we target only small but meaningful examples. It is a long path from a description of a term rewriting engine, via language design for the corresponding formalism, to a usable programming environment.

In this chapter we present the Meta-Environment: *An open architecture of tools, libraries, user-interfaces and code generators targeted to the design and implementation of term rewriting environments.*

We show that by using the Meta-Environment a mature programming environment for a new term rewriting formalism can be obtained in a few steps. Our approach is based on well-known software engineering concepts: standardization (of architecture and exchange format), software reuse (component based development), source code generation and parameterization.

Requirements Real-world examples of term rewriting systems are to be found in many areas, including the following ([146]): rewriting workbenches, computer algebra, symbolic computation, functional programming, definition of programming languages, theorem proving, and generation, analysis, and transformation of programs.

These application areas are quite different, which explains the existence of several formalisms each tailored for a certain application domain. Each area influences the design and implementation of a term rewriting environment in several ways. We identify the following common requirements:

- *Openness.* Collaboration with unforeseen components is often needed. It asks for an open architecture to facilitate communication between the environment, the rewriting engine, and foreign tools.

- *Readable syntax.* Syntax is an important design issue for term rewriting formalisms. Although conceptually syntax might be a minor detail, a formalism that has no practical and readable syntax is not usable.

- *Scalability.* Most real-world examples lead to big specifications or big terms. Scalability means that the implementation is capable of handling such problems using a moderate amount of resources.

- *User Interface.* A textual or graphical user-interface automates the practical use cases of a formalism. An interactive graphical interface also automates as much of the browsing, editing and testing, of specifications as possible. In this work we assume that an interactive GUI is a common requirement for programming environments.

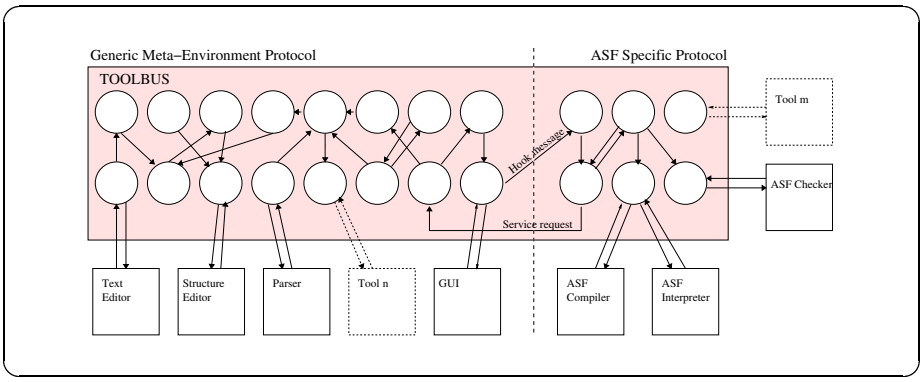

Figure 2.1: A complete environment consisting of a generic part and an ASF specific part

The above four issues offer no deep conceptual challenges, but still they stand for a considerable design and engineering effort. We offer immediately usable solutions concerning each of those issues in this chapter. This paves the way for the application of new experiments concerning term rewriting that would otherwise have cost months to implement. In that sense, this chapter contributes to the promotion and the development of rewriting techniques and their applications.

2.2 Architecture for an open environment

In Section 2.3 we discuss the specific components of the Meta-Environment that can be used to implement a new term rewriting environment. An example environment is implemented in Section 2.4. Here we discuss the general architecture of the Meta-Environment.

The main issue is to separate computation from communication. This separation is achieved by means of a software coordination architecture and a generic data exchange format. An environment is obtained by plugging in the appropriate components into this architecture.

ToolBus. To prevent entangling of coordination with computation in components we introduce a software coordination architecture, the ToolBus [15]. It is a programmable software bus based on Process Algebra. Coordination is expressed by a formal description of the cooperation protocol between components while computation is expressed inside the components that may be written in any language. Figure 2.1 visualizes a ToolBus application (to be discussed below).

Separating computation from communication means that each of these components is made as *independent* as possible from the others. Each component provides a certain service to the other components via the software bus. They interact with each other using messages. The organization of this interaction is fully described using a script

that corresponds to a collection of process algebra expressions.

ATerms. Coordination protocol and components have to share data. We use ATerms [31] for this purpose. These are normal prefix terms with optional annotations added to each node. The annotations are used to store tool-specific information such as text coordinates or proof obligations. All data that is communicated via the ToolBus is encoded as ATerms. ATerms are comparable to XML, both are generic data representations. Although there are tools for conversions between these formats, we prefer ATerms for efficiency reasons. They can be linearized using either a readable representation or a very dense binary encoding.

ATerms can not only be used as a generic data exchange format but also to implement an efficient term data structure in rewriting engines. The ATerm library offers a complete programming interface to the term data structure. It is used to implement term rewriting interpreters or run-time support for compiled rewriting systems. The following three properties of the ATerm library are essential for term rewriting:

- Little memory usage per node.
- Maximal sub-term sharing.
- Automatic garbage collection.

Maximal sharing has proved to be a very good method for dealing with large amounts of terms during term rewriting [30, 159]. It implies that term equality reduces to *pointer equality*. Automatic garbage collection is a very practical feature that significantly reduces the effort of designing a new rewriting engine or compiler.

Meta-Environment Protocol. The ToolBus and ATerms are more widely applicable than just for term rewriting environments. To instantiate this generic architecture, the Meta-Environment ToolBus scripts implement a coordination protocol between its components. Together with the tools, libraries and program generators this protocol implements the basic functionality of an interactive environment.

The Meta-Environment protocol makes no assumptions about the rewriting engine and its coordination with other tools. In order to make a complete term rewriting environment we must complement the generic protocol with specific coordination for every new term rewriting formalism.

For example, the architecture of the ASF+SDF Meta-Environment is shown in Figure 2.1. The ToolBus executes the generic Meta-Environment protocol, depicted by the circles in the left-hand side of the picture. It communicates with external tools, depicted by squares. The right-hand side of the picture shows a specific extension of the Meta-Environment protocol, in this example it is designed for the ASF+SDF rewriting engines. It can be replaced by another protocol in order to construct an environment for a different rewriting formalism.

Hooks. The messages that can be received by the generic part are known in advance, simply because this part of the system is fixed. The reverse is not true, the generic part can make no assumptions about the other part of the system.

Hook	Description
`environment-name(Name)`	The main GUI window will display this name.
`extensions(Sig, Sem, Term)`	Declares the extensions of different file types.
`stdlib-path(Path)`	Sets the path to a standard library.
`semantics-top-sort(Sort)`	Declares the top non-terminal of a specification.
`rewrite(Sig, Sem, Term)`	Rewrite a term using a specification.
`pre-parser-generation(Sig)`	Manipulate the syntax before parser generation.
`rename-semantics(Sig,Binds,Sem)`	Implement module parameterization.
`pre-rewrite(Sig,Spec)`	Actions to do before rewriting.

Table 2.1: The Meta-Environment hooks: the hooks that parameterize the GUI (top half), and events concerning the syntax and semantics of a term rewriting formalism (bottom half).

Tool	Type		Description
`pgen`	`SDF`	→ `Table`	Generates a parse table.
`sglr`	`Table×Str`	→ `AsFix`	parses an input string and yields a derivation.
`implode`	`AsFix`	→ `ATerm`	Maps a parse tree to an abstract term.
`posinfo`	`AsFix`	→ `AsFix`	Adds line and column annotations.
`unparse`	`AsFix`	→ `Str`	Yields a string from a parse tree.

Table 2.2: A list of the most frequently used components for SDF and AsFix

We identify messages that are sent from the generic part of the Meta-Environment to the rewriting formalism part as so-called *hooks*. Each instance of an environment should *at least* implement a receiver for each of these hooks. Table 2.1 shows the basic Meta-Environment hooks. The first four hooks instantiate parameters of the GUI and the editors. The last four hooks are events that need to be handled in a manner that is specific for the rewriting formalisms.

2.3 Reusable components

In this section we present reusable components to implement each aspect of the design of a term rewriting environment. The components are either tools, libraries or code generators. In Section 2.4 we explain how to use these components to create a programming environment using an example term rewriting formalism.

2.3.1 Generalized Parsing for a readable formalism

We offer generic and reusable parsing technology. An implementation of parsing usually consists of a syntax definition formalism, a parser generator, and run-time support

for parsing. Additionally, automated parse tree construction and abstract syntax tree construction are offered. Table 2.2 shows a list of components related to parsing.

Syntax. SDF is a declarative syntax definition formalism used to define modular context-free grammars. Both lexical syntax and context-free syntax can be expressed in a uniform manner. Among other disambiguation constructs, notions for defining associativity and relative priority of operators are present.

Furthermore, SDF offers a simple but effective parameterization mechanism. A module may be parameterized by formal parameters attached to the module name. Using the import mechanism of SDF this parameter can be bound to an actual non-terminal.

Programs that deal with syntax definitions can use the SDF library. It provides a complete high-level programming interface for dealing with syntax definitions.

Concrete syntax. Recall that a syntax definition can serve as a many-sorted signature for a term rewriting system. The grammar productions in the definition are the operators of the signature and the non-terminals are the sorts. The number of non-terminals used in a grammar production is the arity of an operator.

Concrete syntax for any term rewriting formalism can be obtained by simply expressing both the fixed syntax of the formalism and the user defined syntax of the terms in SDF. A parameterized SDF module is used to describe the fixed syntax. This module can be imported for every sort in the user-defined syntax. An example is given in Section 2.4.

SGLR. To implement the SDF formalism, we use scannerless generalized LR parsing [46]. The result is a simple parsing architecture, but capable of handling any modular context-free grammar.

AsFix. SGLR produces parse trees represented as ATerms. This specific class of ATerms is called AsFix. Every AsFix parse tree explains exactly, for each character of the input, which SDF productions were applied to obtain a derivation. A library is offered to be able to create components that deal with AsFix.

2.3.2 Establishing the connection between parsing and rewriting

The SDF library and the AsFix library can be used to implement the connection between the parser and a rewriting engine. Furthermore, we can also automatically generate new libraries specifically tailored towards the rewriting formalism that we want to implement [62] (See also Chapter 9).

We use an SDF definition of the new formalism to generate C or Java libraries that hide the actual ATerm representation of a parse tree of a specification behind a typed interface. The generated interfaces offer: reading in parse trees, constructors, getters and setters for each operator of the new formalism. Apart from saving a lot of time, using these code generators has two major advantages:

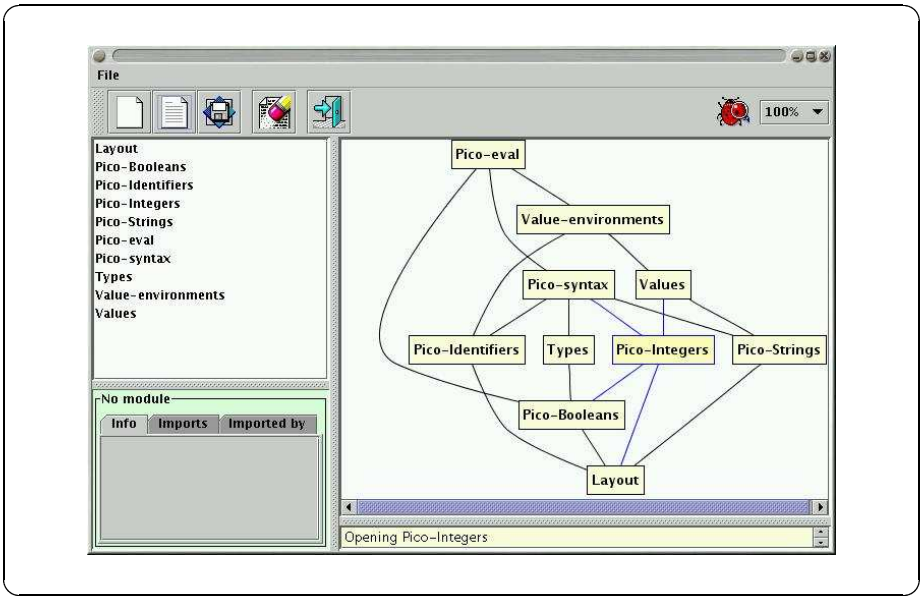

Figure 2.2: GUI of the Meta-Environment displaying an import relation.

- The term rewriter can be developed at a higher level of abstraction.
- Programming errors are prevented by the strictness of the generated types.

2.3.3 Graphical User Interface

MetaStudio. The Meta-Environment contains a user-interface written in Java (Figure 2.2). It can be used to browse modules. Every module has a number of actions that can be activated using the mouse. The actions are sent to the ToolBus. MetaStudio has parameters to configure the name of the environment, the typical file extensions, etc.

Editors. Editing of specifications and terms is done via a generic editor interface. Currently, this interface is implemented by XEmacs [2], but it can be replaced by any editor capable of communicating with other software. To offer structure-editing capabilities, an editor communicates with another component that holds a tree representation of the edited text.

Utilities. Among other utilities, we offer file I/O and in-memory storage that aid in the implementation of an interactive environment.

[2] `http://www.xemacs.org`

2.4 A new environment in a few steps

In this section we show the steps involved in designing a new environment. We take a small imaginary formalism called "RHO" as a running example. It is a subset of the ρ-calculus [56], having first-class rewrite rules and an explicit application operator. The recipe to create a RHO environment is:

1. Instantiate the parameters of the GUI.
2. Define the syntax of RHO.
3. Write some small RHO specifications.
4. Implement and connect a RHO interpreter.
5. Connect other components.

1. Instantiate the parameters of the GUI: We start from a standard ToolBus script that implements default behavior for all the hooks of Table 2.1. We can immediately bind some of the configuration parameters of the GUI. In the case of RHO, we can instantiate two hooks: `environment-name("The RHO Environment")` and `extensions(".sdf",".rho",".trm")`.

Using the RHO Meta-Environment is immediately possible. It offers the user three kinds of syntax-directed editors that can be used to complete the rest of the recipe: SDF editors, editors for the (yet unspecified) RHO formalism, and term editors.

2. Define the syntax of RHO: Figure 2.4 shows how the SDF editors can be used to define the syntax of RHO[3]. It has some predefined operators like assignment (`":="`), abstraction (`"->"`) and application (`"."`), but also concrete syntax for basic terms. So, a part of the syntax of a RHO term is user-defined. The parameterization mechanism of SDF is used to leave a placeholder (`Term`) at the location where user-defined terms are expected[4]. The `Term` parameter will later be instantiated when writing RHO specifications.

To make the syntax-directed editors for RHO files work properly we now have to instantiate the following hook: `semantic-top-sort("Decls")`. The parameter `"Decls"` refers to the top sort of the definition in Figure 2.4.

3. Write some small RHO specifications: We want to test the syntax of the new formalism. Figure 2.3 shows how two editors are used to specify the signature and some rules for the Boolean conjunction. Notice that the `Rho` module is imported explicitly by the `Booleans` module, here we instantiate the `Term` placeholder for the user-defined syntax. In Section 5 we explain how to add the imports automatically.

We can now experiment with the syntax of RHO, define some more operators, basic data types or start a standard library of RHO specifications. For the GUI, the location of the library should be instantiated using the `stdlib-path` hook.

[3] For the sake of brevity, Figure 2.4 does not show any priorities between operators.
[4] Having concrete syntax of terms is not obligatory.

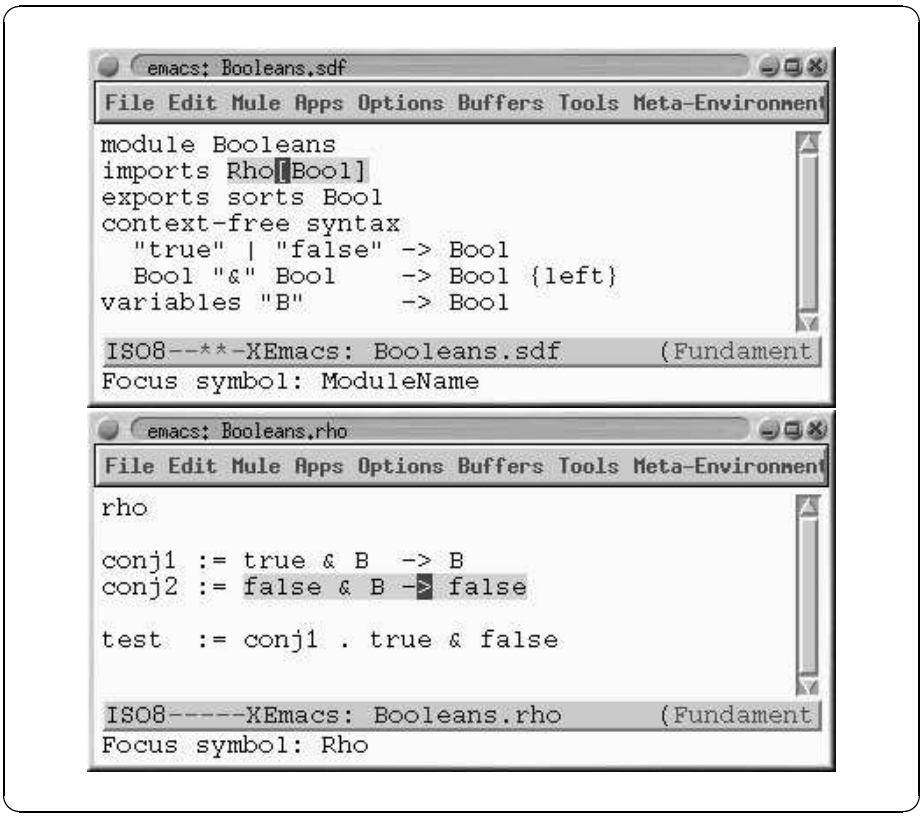

Figure 2.3: A definition of the Boolean conjunction in SDF +RHO.

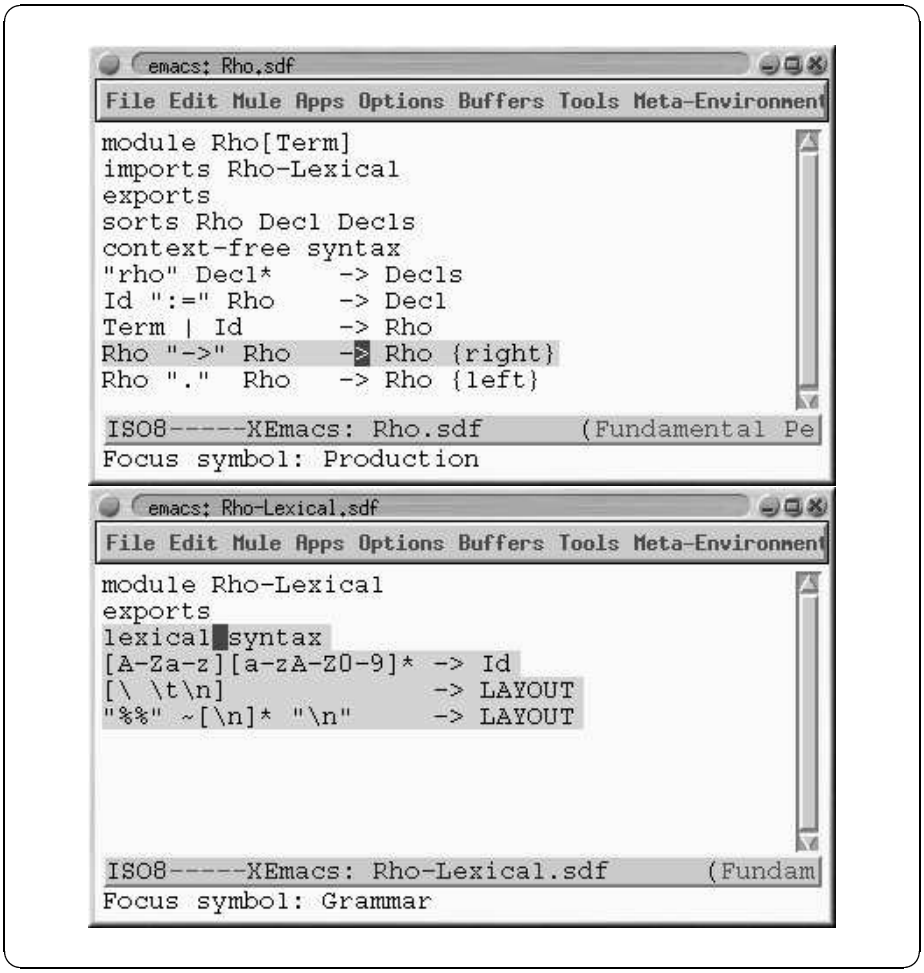

Figure 2.4: A parameterized syntax definition of the formalism RHO.

4. Implement and connect a RHO interpreter: As mentioned in Section 2.2, the ATerm library is an efficient choice for a term implementation. Apart from that we present the details of the connection between a parsed specification and an implementation of the operational semantics of RHO. The algorithmic details of evaluating RHO are left to the reader, because that changes with each instance of a new formalism.

The `rewrite` hook connects a rewriting engine to the RHO environment: `rewrite(Syntax,Semantics,Term)` From this message we receive the information that is to be used by the rewriting engine. Note that this does not prohibit to request any other information from other components using extra messages. The input data that is received can be characterized as follows: `Syntax` is a list of all SDF modules (the parse trees of `Rho.sdf` and `Booleans.sdf`). `Semantics` is a list of all RHO modules (the parse tree of `Booleans.rho`). `Term` is the expression that is to

be normalized (for example a parse tree of a file called `test.trm`).

Two scenarios are to be considered: either a RHO engine already exists, or a new engine has to be designed from scratch. In the first case, the data types of the Meta-Environment will be converted to the internal representation of the existing engine. In the second case, we can implement a new engine based on the data types of the Meta-Environment directly. In both cases the three data types of the Meta-Environment are important: SDF, AsFix and ATerms. The libraries and generators ensure that these cases can be specified on a high level of abstraction. We split the work into the signature and semantics parts of RHO.

Signature. To extract the needed information from the user-defined signature the SDF modules should be analyzed. The SDF library is the appropriate mechanism to inspect them in a straightforward manner.

Semantics. Due to having concrete syntax, the list of parse trees that represent RHO modules is not defined by a fixed signature. We can divide the set of operators in two categories:

- *A fixed* set of operators that correspond to the basic operators of the formalism. Each fixed operator represents a syntactical notion that should be given a meaning by the operational semantics. For RHO, assignment, abstraction, and application are examples of fixed operators.
- *Free* terms occur at the location where the syntax is user-defined. In RHO this is either as the right-hand side of an assignment or as a child of the abstraction or application operators.

There is a practical solution for dealing with each of these two classes of operators. Firstly, from an SDF definition for RHO we generate a library specifically tailored for RHO. This library is used to recognize the operators of RHO and extract information via an abstract typed interface. For example, one of the C function headers in this generated library is: `Rho getRuleLhs(Rho rule)`. A RHO interpreter can use it to retrieve the left-hand side of a rule.

Secondly, the free terms can be mapped to simple prefix ATerms using the component `implode`, or they can be analyzed directly using the AsFix library. The choice depends on the application area. E.g., for source code renovation details such as white space and source code comments are important, but for symbolic computation this information might as well be thrown away in favor of efficiency.

In the case of an existing engine, the above interfaces are used to extract information before providing it to the engine. In the case of a new engine, the interfaces are used to directly specify the operational semantics of RHO.

5. Connect other components: There are some more hooks that can be instantiated in order to influence the behavior of the Meta-Environment. Also, the RHO part of the newly created environment might introduce other components besides the rewriting engine.

We give two examples here. The `pre-parser-generation` hook can be used to extend the user-defined syntax with imports of the RHO syntax automatically for each non-terminal. Secondly, the `pre-rewrite` hook hook can be used to connect an automatic verifier or prover like a Knuth-Bendix completion procedure.

Adding unanticipated tools is facilitated at three levels by the Meta-Environment. Firstly, an SDF production can have any attribute to make it possible to express special properties of operators for the benefit of new tools. An example: `B "&" B -> B { left, lpo-precedence(42) }`. Secondly, any ATerm can be annotated with extra information without affecting the other components. For example: `and(true,false){not-reduced}`. Finally, all existing services of the Meta-Environment are available to the new tool. It can for example open a new editor to show its results using this message: `new-editor(Contents)`

2.5 Instantiations of the Meta-Environment

We now introduce the four formalisms we have implemented so far using the above recipe. We focus on the discriminating aspects of each language.

ASF [67] is a term rewriting formalism based on leftmost-innermost normalization. The rules are called equations and are written in concrete syntax. Equations can have a list of conditions which must all evaluate to true before a reduction succeeds. The operational semantics of ASF also introduces *rewriting with layout* and *traversal functions* [37], operators that traverse the sub-term they are applied to.

The above features correspond to the application areas of ASF. It is mainly used for design of the syntax and semantics of domain specific languages and analysis and transformation of programs in existing programming languages. From the application perspective ASF is an expressive form of first-order functional programming. The Meta-Environment serves as a programming environment for ASF.

ELAN [22] is based on rewrite rules too. It provides a strategy language, allowing to control the application of rules instead of leaving this to a fixed normalization strategy. Primitive strategies are labeled rewrite rules, which can be combined using strategy basic operators. New strategy operators can be expressed by defining them in terms of less complex ones. ELAN supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and offers a modular framework for studying their combination.

In order to improve the architecture, and to make the ELAN system more interactive, it was decided to redesign the ELAN system based on the Meta-Environment. The instantiation of the ELAN environment involved the implementation of several new components, among others an interpreter. Constructing the ELAN environment was a matter of a few months.

The ρ-calculus [56] integrates in a uniform and simple setting first-order rewriting, lambda-calculus and non-deterministic computations. Its abstraction mechanism is

based on the rewrite rule formation. The application operator is explicit, allowing to handle sets of results explicitly.

The ρ-calculus is typically a new rewriting formalism which can benefit from the the Meta-Environment. We have prototyped a workbench for the complete ρ-calculus. After that, we connected an existing ρ-calculus interpreter. This experiment was realized in one day.

The JITty interpreter [137] is a part of the μCRL [18] tool set. In this tool set it is used as an execution mechanism for rewrite rules. JITty is not supported by its own formalism or a specialized environment. However, the ideas of the JITty interpreter are more generally applicable. It implements an interesting normalization strategy, the so-called *just-in-time* strategy. A workbench for the JITty interpreter was developed in a few hours that allowed to perform experiments with the JITty interpreter.

2.6 Conclusions

Experiments with and applications of term rewriting engines are within much closer reach using the Meta-Environment, as compared to designing and engineering a new formalism from scratch.

We have presented a generic approach for rapidly developing the three major ingredients of a term rewriting based formalism: syntax, rewriting, and an environment. Using the scalable technology of the Meta-Environment significantly reduces the effort to develop them, by reusing components and generating others. This conclusion is based on practial experience with building environments for term rewriting languages other than ASF+SDF. We used our approach to build four environments. Two of them are actively used by their respective communities. The others serve as workbenches for new developments in term rewriting.

The Meta-Environment and its components can now support several term rewriting formalisms. A future step is to build environments for languages like Action Semantics [117] and TOM [127].

Apart from more environments, other future work consists of even further parameterization and modularization of the Meta-Environment. Making the Meta-Environment open to different syntax definition formalisms is an example.

Part II

Parsing and disambiguation of source code

CHAPTER 3

Disambiguation Filters for Scannerless Generalized LR Parsers

In this chapter we present the fusion of generalized LR parsing and scannerless parsing. This combination supports syntax definitions in which all aspects (lexical and context-free) of the syntax of a language are defined explicitly in one formalism. Furthermore, there are no restrictions on the class of grammars, thus allowing a natural syntax tree structure. Ambiguities that arise through the use of unrestricted grammars are handled by explicit disambiguation constructs, instead of implicit defaults that are taken by traditional scanner and parser generators. Hence, a syntax definition becomes a full declarative description of a language. Scannerless generalized LR parsing is a viable technique that has been applied in various industrial and academic projects. [1]

3.1 Introduction

Since the introduction of efficient deterministic parsing techniques, parsing is considered a closed topic for research, both by computer scientists and by practitioners in compiler construction. Tools based on deterministic parsing algorithms such as LEX & YACC [120, 92] (LALR) and JAVACC (recursive descent), are considered adequate for dealing with almost all modern (programming) languages. However, the development of more powerful parsing techniques, is prompted by domains such as reverse engineering, domain-specific languages, and languages based on user-definable mixfix syntax.

The field of reverse engineering is concerned with automatically analyzing legacy software and producing specifications, documentation, or re-implementations. This area provides numerous examples of parsing problems that can only be tackled by using powerful parsing techniques.

[1] I co-authored this chapter with Mark van den Brand, Jeroen Scheerder and Eelco Visser. It was published in CC 2002 [46].

Grammars of languages such as Cobol, PL1, Fortran, etc. are not naturally LALR. Much massaging and default resolution of conflicts are needed to implement a parser for these languages in YACC. Maintenance of such massaged grammars is a pain since changing or adding a few productions can lead to new conflicts. This problem is aggravated when different dialects need to be supported—many vendors implement their own Cobol dialect. Since grammar formalisms are not modular this usually leads to forking of grammars. Further trouble is caused by the embedding of 'foreign' language fragments, e.g., assembler code, SQL, CICS, or C, which is common practice in Cobol programs. Merging of grammars for several languages leads to conflicts at the context-free grammar level and at the lexical analysis level. These are just a few examples of problems encountered with deterministic parsing techniques.

The need to tackle such problems in the area of reverse engineering has led to a revival of generalized parsing algorithms such as Earley's algorithm, (variants of) Tomita's algorithm (GLR) [116, 149, 138, 8, 141], and even recursive descent backtrack parsing [59]. Although generalized parsing solves several problems in this area, generalized parsing alone is not enough.

In this chapter we describe the benefits and the practical applicability of *scannerless generalized LR parsing*. In Section 3.2 we discuss the merits of scannerless parsing and generalized parsing and argue that their combination provides a solution for problems like the ones described above. In Section 3.3 we describe how disambiguation can be separated from grammar structure, thus allowing a natural grammar structure and declarative and selective specification of disambiguation. In Section 3.4 we discuss issues in the implementation of disambiguation. In Section 3.5 practical experience with the parsing technique is discussed. In Section 3.6 we present figures on the performance of our implementation of a scannerless generalized parser. Related work is discussed where needed throughout the chapter. Section 3.7 contains a discussion in which we focus explicitly on the difference between backtracking and GLR parsing and the usefulness of scannerless parsing. Finally, we conclude in Section 3.8.

3.2 Scannerless Generalized Parsing

3.2.1 Generalized Parsing

Generalized parsers are a class of parsing algorithms that are not constrained by restrictions on the class of grammars that they can handle, contrary to restricted parsing algorithms such as the various derivatives of the LL and LR algorithms. Whereas these algorithms only deal with context-free grammars in LL(k) or LR(k) form, generalized algorithms such as Earley's or Tomita's algorithms can deal with arbitrary context-free grammars. There are two major advantages to the use of arbitrary context-free grammars.

Firstly, the class of context-free grammars is closed under union, in contrast with all proper subclasses of context-free grammars. For example, the composition of two LALR grammars is very often not a LALR grammar. The compositionality of context-free grammars opens up the possibility of developing modular syntax definition formalisms. Modularity in programming languages and other formalisms is one of the

key beneficial software engineering concepts. A striking example in which modularity of a grammar is obviously practical is the definition of hybrid languages such as Cobol with CICS, or C with assembly. SDF [87, 157] is an example of a modular syntax definition formalism.

Secondly, an arbitrary context-free grammar allows the definition of declarative grammars. There is no need to massage the grammar into LL, LR, LALR, or any other form. Rather the grammar can reflect the intended structure of the language, resulting in a concise and readable syntax definition. Thus, the same grammar can be used for documentation as well as implementation of a language without any changes.

Since generalized parsers can deal with arbitrary grammars, they can also deal with *ambiguous* grammars. While a deterministic parser produces a single parse tree, a non-deterministic parser produces a collection (forest) of trees compactly representing all possible derivations according to the grammar. This can be helpful when developing a grammar for a language. The parse forest can be used to visualize the ambiguites in the grammar, thus aiding in the improvement of the grammar. Contrast this with solving conflicts in a LALR table. Disambiguation filters can be used to reduce a forest to the intended parse tree. Filters can be based on disambiguation rules such as priority and associativity declarations. Such filters solve the most frequent ambiguities in a natural and intuitive way without hampering the clear structure of the grammar.

In short, generalized parsing opens up the possibility for developing clear and concise language definitions, separating the language design problem from the disambiguation problem.

3.2.2 Scannerless Parsing

Traditionally, syntax analysis is divided into a lexical scanner and a (context-free) parser. A scanner divides an input string consisting of characters into a string of tokens. This tokenization is usually based on regular expression matching. To choose between overlapping matches a number of standard lexical disambiguation rules are used. Typical examples are prefer keywords, prefer longest match, and prefer non-layout. After tokenization, the tokens are typically interpreted by the parser as the terminal symbols of an LR(1) grammar.

Although this architecture proves to be practical in many cases and is globally accepted as the standard solution for parser generation, it has some problematic limitations. Only few *existing* programming languages are designed to fit this architecture, since these languages generally have an ambiguous lexical syntax. The following examples illustrate this misfit for Cobol, PL1 and Pascal.

In an embedded language, such as SQL in Cobol, identifiers that are reserved keywords in Cobol might be allowed inside SQL statements. However, the implicit "prefer keywords" rule of lexical scanners will automatically prohibit them in SQL too.

Another Cobol example; a particular "picture clause" might look like `"PIC 99"`, where `"99"` should be recognized as a list of `picchars`. In some other part of a Cobol program, the number `"99"` should be recognized as `numeric`. Both character classes obviously overlap, but on the context-free level there is no ambiguity because picture clauses do not appear where numerics do. See [111] for a Cobol syntax definition.

Another example of scanner and parser interference stems from Pascal. Consider the input sentence `"array [1..10] of integer"`, the range `"1..10"` can be tokenized in two different manners, either as the real `"1."` followed by the real `".10"`, or as the integer `"1"` followed by the range operator `".."` followed by the integer `"10"`. In order to come up with the correct tokenization the scanner must "know" it is processing an array declaration.

The problem is even more imminent when a language does not have reserved keywords at all. PL1 is such a language. This means that a straightforward tokenization is not possible when scanning a valid PL1 sentence such as `"IF THEN THEN = ELSE; ELSE ELSE = THEN;"`.

Similar examples can be found for almost any existing programming language. A number of techniques for tackling this problem is discussed in [10]. Some parser generators provide a complex interface between scanner and parser in order to profit from the speed of lexical analysis while using the power of a parser. Some lexical scanners have more expressive means than regular expressions to be able to make more detailed decisions. Some parser implementations allow arbitrary computations to be expressed in a programming language such as C to guide the scanner and the parser. All in all it is rather cumbersome to develop and to maintain grammars which have to solve such simple lexical disambiguations, because none of these approaches result in declarative syntax specifications.

Scannerless parsing is an alternative parsing technique that does not suffer these problems. The term scannerless parsing was introduced in [139, 140] to indicate parsing without a separate lexical analysis phase. In scannerless parsing, a syntax definition is a context-free grammar with characters as terminals. Such an integrated syntax definition defines all syntactic aspects of a language, including the full details of the lexical syntax. The parser derived from this grammar directly reads the characters of the input string and finds its phrase structure.

Scannerless parsing does not suffer the problems of implicit lexical disambiguation. Very often the problematic lexical ambiguities do not even exist at the context-free level, as is the case in our Cobol, Pascal and PL1 examples. On the other hand, the lack of implicit rules such as "prefer keywords" and "longest match" might give rise to new ambiguities at the context-free level. These ambiguities can be solved by providing *explicit* declarative rules in a syntax definition language. Making such disambiguation decisions explicit makes it possible to apply them selectively. For instance, we could specify longest match for a single specific sort, instead of for the entire grammar, as we shall see in Section 3.3.

In short, scannerless parsing does not need to make any assumptions about the lexical syntax of a language and is therefore more generically applicable for language engineering.

3.2.3 Combining Scannerless Parsing and Generalized Parsing

Syntax definitions in which lexical and context-free syntax are fully integrated do not usually fit in any restricted class of grammars required by deterministic parsing techniques because lexical syntax often requires arbitrary length lookahead. Therefore, scannerless parsing does not go well with deterministic parsing. For this reason the ad-

```
Term ::= Id | Nat | Term Ws Term
Id   ::= [a-z]+
Nat  ::= [0-9]+
Ws   ::= [\ \n]*
%restrictions
Id  -/- [a-z]
Nat -/- [0-9]
Ws  -/- [\ \n]
```

Figure 3.1: Term language with follow restrictions.

jacency restrictions and exclusion rules of [139, 140] could only be partly implemented in an extension of a SLR(1) parser generator and led to complicated grammars.

Generalized parsing techniques, on the other hand, can deal with arbitrary length lookahead. Using a generalized parsing technique solves the problem of lexical lookahead in scannerless parsing. However, it requires a solution for disambiguation of lexical ambiguities that are not resolved by the parsing context.

In the rest of this chapter we describe how syntax definitions can be disambiguated by means of declarative disambiguation rules for several classes of ambiguities, in particular lexical ambiguities. Furthermore, we discuss how these disambiguation rules can be implemented efficiently.

3.3 Disambiguation Rules

There are many ways for disambiguation of ambiguous grammars, ranging from simple syntactic criteria to semantic criteria [104]. Here we concentrate on ambiguities caused by integrating lexical and context-free syntax. Four classes of disambiguation rules turn out to be adequate.

Follow restrictions are a simplification of the adjacency restriction rules of [139, 140] and are used to achieve longest match disambiguation. Reject productions, called exclusion rules in [139, 140], are designed to implement reserved keywords disambiguation. Priority and associativity rules are used to disambiguate expression syntax. Preference attributes are used for selecting a default among several alternative derivations.

3.3.1 Follow Restrictions

Suppose we have the simple context-free grammar for terms as presented in Figure 3.1. An `Id` is defined to be one ore more characters from the class `[a-z]+` and two terms are separated by whitespace consisting of zero or more spaces or newlines.

Without any lexical disambiguation, this grammar is ambiguous. For example, the sentence `"hi"` can be parsed as `Term(Id("hi"))` or as `Term(Id("h")), Ws(""), Term(Id("i"))`. Assuming the first is the intended derivation, we add

```
Star        ::= [\*]
CommentChar ::= ~[\*] | Star
Comment     ::= "(*" CommentChar* "*)"
Ws          ::= ([\ \n] | Comment)*
%restrictions
Star -/- [\)]
Ws   -/- [\ \n] | [\(].[\*]
```

Figure 3.2: Extended layout definition with follow restrictions.

```
Program ::= "begin" Ws Term Ws "end"
Id      ::= "begin" | "end" {reject}
```

Figure 3.3: Prefer keywords using reject productions

a follow restriction, `Id -/- [a-z]`, indicating that an `Id` may not directly be followed by a character in the range `[a-z]`. This entails that such a character should be part of the identifier. Similarly, follow restrictions are added for `Nat` and `Ws`. We have now specified a *longest match* for each of these lexical constructs.

In some languages it is necessary to have more than one character lookahead to decide the follow restriction. In Figure 3.2 we extend the layout definition of Figure 3.1 with comments. The expression `~[\*]` indicates any character except the asterisk. The expression `[\(].[\*]` defines a restriction on two consecutive characters. The result is a *longest match* for the `Ws` nonterminal, including comments. The follow restriction on `Star` prohibits the recognition of the string `"*)"` within `Comment`. Note that it is straightforward to extend this definition to deal with *nested* comments.

3.3.2 Reject Productions

Reject productions are used to implement keyword reservation. We extend the grammar definition of Figure 3.1 with the `begin` and `end` construction in Figure 3.3. The sentence `"begin hi end"` is either interpreted as three consecutive `Id` terms separated by `Ws`, or as a `Program` with a single term `hi`. By *rejecting* the strings `begin` and `end` from `Id`, the first interpretation can be filtered out.

The reject mechanism can be used to reject not only strings, but entire context-free languages from a nonterminal. We focus on its use for keyword reservation in this chapter and refer to [157] for more discussion.

3.3.3 Priority and Associativity

For completeness we show an example of the use of priority and associativity in an expression language. Note that we have left out the `Ws` nonterminal for brevity[2]. In

[2]By doing grammar normalization a parse table generator can automatically insert layout between the members in the right-hand side. See also Section 3.5.

```
Exp ::= [0-9]+
Exp ::= Exp "+" Exp {left}
Exp ::= Exp "*" Exp {left}
%priorities
Exp ::= Exp "*" Exp > Exp ::= Exp "+" Exp
```

Figure 3.4: Associativity and priority rules.

```
Term ::= "if" Nat "then" Term {prefer}
Term ::= "if" Nat "then" Term "else" Term
Id   ::= "if" | "then" | "else" {reject}
```

Figure 3.5: Dangling else construction disambiguated

Figure 3.4 we see that the binary operators + and * are both defined as left associative and the * operator has a higher priority than the + operator. Consequently the sentence "1 + 2 + 3 * 4" is interpreted as "(1 + 2) + (3 * 4)".

3.3.4 Preference Attributes

A preference rule is a generally applicable rule to choose a default among ambiguous parse trees. For example, it can be used to disambiguate the notorious dangling else construction. Again we have left out the Ws nonterminal for brevity. In Figure 3.5 we extend our term language with this construct.

The input sentence "if 0 then if 1 then hi else ho" can be parsed in two ways: if 0 then (if 1 then hi) else ho and if 0 then (if 1 then hi else ho). We can select the latter derivation by adding the prefer attribute to the production without the else part. The parser will still construct an ambiguity node containing both derivations, namely, if 0 then (if 1 then hi {prefer}) else ho and if 0 then (if 1 then hi else ho) {prefer}. But given the fact that the *top* node of the latter derivation tree has the prefer attribute this derivation is selected and the other tree is removed from the ambiguity node.

The dual of {prefer} is the {avoid} attribute. Any other tree is preferred over a tree with an avoided top production. One of its uses is to prefer keywords rather than reserving them entirely. For example, we can add an {avoid} to the Id ::= [a-z]+ production in Figure 3.1 and not add the reject productions of Figure 3.3. The sentence "begin begin end" is now a valid Program with the single derivation of a Program containing the single Id "begin".

Note that naturally the preference attributes can only distinguish among derivations that have different productions at the top. Preference attributes are not claimed to be a general way of disambiguation. Like the other methods, they cover a particular range of disambiguation idioms commonly found in programming languages.

3.4 Implementation Issues

Our implementation of scannerless generalized parsing consists of the syntax definition formalism SDF that supports concise specification of integrated syntax definitions, a grammar normalizer that injects layout and desugars regular expressions, a parse table generator and a parser that interprets parse tables.

The parser is based on the GLR algorithm. For the basic GLR algorithms we refer to the first publication on generalized LR parsing by Lang [116], the work by Tomita [149], and the various improvements and implementations [130, 138, 8, 141]. We will not present the complete SGLR algorithm, because it is essentially the standard GLR algorithm where each character is a separate token. For a detailed description of the implementation of GLR and SGLR we refer to [138] and [156] respectively.

The algorithmic differences between standard GLR and scannerless GLR parsing are centered around the disambiguation constructs. From a declarative point of view each disambiguation rule corresponds to a filter that prunes parse forests. In this view, parse table generation and the GLR algorithm remain unchanged and the parser returns a forest containing all derivations. After parsing a number of filters is executed and a single tree or at least a smaller forest is obtained.

Although this view is conceptually attractive, it does not fully exploit the possibilities for pruning the parse forest *before* it is even created. A filter might be implemented statically, during parse table generation, dynamically, during parsing, or after parsing. The sooner a filter is applied, the faster a parser will return the filtered derivation tree. In which phase they are applicable depends on the particulars of specific disambiguation rules. In this section we discuss the implementation of the four classes of disambiguation rules.

3.4.1 Follow Restrictions

Our parser generator generates a simple SLR(1) parse table, however we deviate at a number of places from standard algorithm [2]. One modification is the calculation of the follow set. The follow set is calculated for each individual production rule instead of for each nonterminal. Another modification is that the transitions between states (itemsets) in the LR-automaton are not labeled with a nonterminal, but with a production rule. These more fine-grained transitions increase the size of the LR-automaton, but it allows us to generate parse tables with fewer conflicts.

Follow restriction declarations with a single lookahead can be used during parse table generation to remove reductions from the parse table. This is done by intersecting the follow set of each production rule with the set of characters in the follow restrictions for the produced nonterminal. The effect of this filter is that the reduction in question cannot be performed for characters in the follow restriction set.

Restrictions with more than one lookahead must be dealt with dynamically by the parser. The parse table generator marks the reductions that produce a nonterminal that has restrictions with more than one character. Then, while parsing, before such a reduction is done the parser must retrieve the required number of characters from the string and check them with the restrictions. If the next characters in the input match these restrictions the reduction is not allowed, otherwise it can be performed. This

parse-time implementation prohibits shift/reduce conflicts that would normally occur and therefore saves the parser from performing unnecessary work.

Note that it is possible to generate the follow restrictions automatically from the lexical syntax definition. Doing so would enforce a global longest match rule.

3.4.2 Reject Productions

Disambiguation by means of reject productions cannot be implemented statically, since this would require computing the intersection of two syntactic categories, which is not possible in general. Even computing such intersections for regular grammars would lead to very large automata. When using a generalized parser, filtering with reject productions can be implemented effectively during parsing.

Consider the reject production `Id ::= "begin" {reject}`, which declares that `"begin"` is not a valid `Id` in *any* way (Figure 3.3). Thus, each and every derivation of the subsentence `"begin"` that produces an `Id` is illegal. During parsing, without the reject production the substring `"begin"` will be recognized both as an `Id` and as a keyword in a `Program`. By adding the reject production to the grammar another derivation is created for `"begin"` as an `Id`, resulting in an ambiguity of two derivations. If one derivation in an ambiguity node is rejected, the entire parse stack for that node is deleted. Hence, `"begin"` is not recognized as an identifier in any way. Note that the parser must wait until each ambiguous derivation has returned before it can delete a stack[3]. The stack on which this substring was recognized as an `Id` will not survive, thus no more actions are performed on this stack. The only derivation that remains is where `"begin"` is a keyword in a `Program`.

Reject productions could also be implemented as a back-end filter. However, by terminating stacks on which reject productions occur as soon as possible a dramatic reduction in the number of ambiguities can be obtained.

Reject productions for keyword reservation can automatically be generated by adding the keyword as a reject production for the nonterminal in the left-hand side of a lexical production rule whenever an overlap between this lexical production rule and a keyword occurs.

3.4.3 Priority and Associativity

Associativity of productions and priority relations can be processed during the construction of the parse table. We present an informal description here and refer to [157] for details.

There are two phases in the parse table generation process in which associativity and priority information is used. The first place is during the construction of the LR-automaton. Item-sets in the LR-automaton contain dotted productions. Prediction of new items for an item-set takes the associativity and priority relations into consideration. If a predicted production is in conflict with the production of the current item, then the latter production is *not* added to the item-set. The second place is when shifting a

[3] Our parser synchronizes parallel stacks on shifts, so we can wait for a shift before we delete an ambiguity node.

dot over a nonterminal in an item. In case of an associativity or priority conflict between a production rule in the item and a production rule on a transition, the transition will not be added to the LR-automaton.

We will illustrate the approach described above by discussing the construction of a part of the LR-automaton for the grammar presented in Figure 3.4. We are creating the transitions in the LR-automaton for state s_i which contains the item-set:

```
[Exp ::= . Exp "+" Exp]
[Exp ::= . Exp "*" Exp]
[Exp ::= . [0-9]+]
```

In order to shift the dot over the nonterminal `Exp` via the production rule `Exp ::= Exp "+" Exp` every item in s_i is checked for a conflict. The new state s_j has the item-set:

```
[Exp ::= Exp . "+" Exp]
```

Note that s_j does not contain the item `[Exp ::= Exp . "*" Exp]`, since that would cause a conflict with the given priority relation `"*" > "+"`.

By pruning the transitions in a parse table in the above manner, conflicts at parse time pertaining to associativity and priority can be ruled out. However, if we want priority declarations to ignore injections (or chain rules) this implementation does not suffice. Yet it is natural to ignore injections when applying disambiguation rules, since they do not have any visible syntax. Priority filtering modulo chain rules require an extension of this method or a post parse-time filter.

3.4.4 Preference Attributes

The preference filter is an typical example of an after parsing filter. In principle it could be applied while parsing, however this will complicate the implementation of the parser tremendously without gaining efficiency. This filter operates on an ambiguity node, which is a set of ambiguous subtrees, and selects the subtrees with the highest preference.

The simplest preference filter compares the trees of each ambiguity node by comparing the `avoid` or `prefer` attributes of the top productions. Each preferred tree remains in the set, while all others are removed. If there is no preferred tree, all avoided trees are removed, while all others remain. Ignoring injections at the top is a straightforward extension to this filter.

By implementing this filter in the back-end of the parser we can exploit the redundancy in parse trees by caching filtered subtrees and reusing the result when filtering other identical subtrees. We use the ATerm library [31] for representing a parse forest. It has *maximal sharing* of sub-terms, limiting the amount of memory used and making subtree identification a trivial matter of pointer equality.

For a number of grammars this simple preference filter is not powerful enough, because the production rules with the `avoid` or `prefer` are not at the root (modulo injections) of the subtrees, but deeper in the subtree. In order to disambiguate these ambiguous subtrees, more subtle preference filters are needed. However, these filters

will always be based on some heuristic, e.g., counting the number of "preferred" and "avoided" productions and applying some selection on the basis of these numbers, or by looking a the depth at which a "preferred" or "avoided" production occurs. In principle, for any chosen heuristic counter examples can be constructed for which the heuristic fails to achieve its intended goal, yielding undesired results.

3.5 Applications

3.5.1 ASF+SDF Meta-Environment

In the introduction of this chapter we claimed that generalized parsing techniques are applicable in the fields of reverse engineering and language prototyping, i.e., the development of new (domain-specific) languages. The ASF+SDF Meta-Environment [28] is used in both these fields. This environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The parser is this environment and in the generated environments is an SGLR parser. The language definitions are written in the ASF+SDF formalism [67] which allows the definition of syntax via SDF (Syntax Definition Formalism) [87] as well as semantics via ASF (Algebraic Specification Formalism).

ASF+SDF has been used in a number of industrial and scientific projects. Amongst others it was used for parsing and compiling ASF+SDF specifications, automatically renovating Cobol code, program analysis of legacy code via so-called island grammars [126], and development of new Action Notation syntax [72].

3.5.2 XT

XT [66] is a collection of basic tools for building program transformation systems including the Stratego transformation language [159], and the syntax definition formalism SDF supported by SGLR. Tools standardize on ATerms [31] as common exchange format. Several meta-tools are provided for generating transformation components from syntax definitions, including a data type declaration generator that generates the data type corresponding to the abstract syntax of an SDF syntax definition, and a pretty-printer generator that generates default pretty-print tables.

To promote reuse and standardization of syntax definitions, the XT project has initiated the creation of the Online Grammar Base[4] currently with some 25 syntax definitions for various general purpose and domain-specific languages, including Cobol, Java, SDL, Stratego, YACC, and XML. Many syntax definitions were semi-automatically re-engineered from LEX/YACC definitions using grammar manipulation tools from XT, producing more compact syntax definitions. SDF/SGLR based parsers have been used in numerous projects built with XT in areas ranging from software renovation and grammar recovery to program optimization and compiler construction.

[4] `http://www.program-transformation.org/gb`

Grammar	Average file size	Tokens/second with filter & tree[5]	Tokens/second w/o filter & tree[5]
ATerms	106,000 chars	108,000	340,000
BibTEX	455,000 chars	85,000	405,000
Box	80,000 chars	34,000	368,000
Cobol	170,000 chars	58,000	146,000
Java	105,000 chars	37,000	210,000
Java (LR1)	105,000 chars	53,000	242,000

Table 3.1: Some figures on SGLR performance.

Grammar	Productions	States	Actions	Actions with conflicts	Gotos
ATerms	104	128	8531	75	46569
BibTEX	150	242	40508	3129	98901
Box	202	385	19249	1312	177174
Cobol	1906	5520	170375	32634	11941923
Java	726	1561	148359	5303	1535446
Java (LR1)	765	1597	88561	3354	1633156

Table 3.2: Some figures on the grammars and the generated parse tables.

3.6 Benchmarks

We have bench-marked our implementation of SGLR by parsing a number of larger files and measuring the `user time`. Table 3.1 shows the results with and without parse tree construction and back-end filtering. All filters implemented in the parse table or during parsing are active in both measurements. The table shows that the parser is fast enough for industrial use. An interesting observation is that the construction of the parse tree slows down the entire process quite a bit. Further speedup can be achieved by optimizing parse tree construction.

Table 3.2 shows some details of the SLR(1) parse tables for the grammars we used. We downloaded all but the last grammar from the Online Grammar Base. ATerms is a grammar for prefix terms with annotations, BibTEX is a bibliography file format, Box is a mark-up language used in pretty-print tools. Cobol and Java are grammars for the well-known programming languages. We have bench-marked two different Java grammars. The first is written from scratch in SDF, the second was obtained by transforming a Yacc grammar into SDF. So, the first is a more natural definition of Java syntax, while the second is in LR(1) form.

The number of productions is measured after SDF grammar normalization[6]. We mention the number of states, gotos and actions in the parse table. Remember that

[5] All benchmarks were performed on a 1200 Mhz AMD Athlon(tm) with 512Mb memory running Linux.
[6] So this number does not reflect the size of the grammar definition.

the parse table is specified down to the character level, so we have more states than usual. Also, actions and gotos are based on productions, not nonterminals, resulting in a bigger parse table. The number of actions with more than one reduce or shift (a conflict) gives an indication of the amount of "ambiguity" in a grammar. The two Java results in Table 3.1 show that ambiguity of a grammar has a limited effect on performance. Note that after filtering, every parse in our test set resulted in a single derivation.

3.7 Discussion

3.7.1 Generalized LR parsing versus backtracking parsers

A Tomita-style GLR parser produces *all* possible derivations[7]. This is a fundamentally different approach than parsing algorithms that employ *backtracking*. Backtracking parsers can also accept all context-free languages[8]. Most backtracking parsers like ANTLR [135] return the *first* valid derivation that they find. The ordering of derivations is implicit in the parsing algorithm, but sometimes it may be influenced by ordering production rules (TXL [59]) or providing other kinds of predicates. Note that backtracking parsers can also produce *all* derivations, but using an impractical (exponential) amount of time. Exponential behavior may also be triggered when an input sentence contains a parse error. In the following we assume backtracking parsers return a single derivation. We also assume that no arbitrary side-effects for influencing the parser are allowed.

So, both approaches can accept all context-free languages. The question remains what the difference is. The most important difference between the two methods is *a priori* disambiguation (backtracking) versus *a posteriori* disambiguation (GLR). With a priori disambiguation the user could never find out that a grammar is in fact ambiguous. With a posteriori disambiguation the user is always confronted with ambiguity after parsing an ambiguous input. After this confrontation she is obliged to provide a conscious choice in the form of a disambiguation rule.

So, with backtracking we get a unique but arbitrary derivation, and with GLR parsing we may get more than one derivation but there is no arbitrary choice made. The difference is the disambiguation method: either heuristically or declaratively. The declarative GLR parsing method may imply more work for the language designer, while the heuristic backtracking method may leave blind spots in her understanding of a language. Neither of the two methods can guarantee full semantically correct and unambiguous derivations (See Chapter 4).

3.7.2 When to use scannerless parsing?

Clearly, not in all cases scannerless parsing is necessary. This is witnessed by the fact that many scanners are still written and used with success. For languages with no

[7] We assume a fix is included for the hidden right recursive grammars [130].

[8] In this discussion we ignore any issues with hidden left recursive grammars.

reserved keywords and irregular rules for longest/shortest match the benefit of scannerless parsing is immediately clear, but what are the other use cases? Apart from the motivating cases described in Section 3.2, there are two use-cases we consider to be important.

The first use-case is when a *declarative* (technology independent) language description is a requirement. Using a scanner there will always be implicit lexical disambiguation heuristics at play, or some arbitrary side-effects must be used. Therefore it is hard to obtain a *full* description of all the properties of a language, since we may have blind spots due to implicit (global) choices that a scanner makes. With a scannerless parser all lexical disambiguation needs to be explained explicitly and precisely in a syntax definition. Especially in the context of language standardization documents, scannerless parsing could be an essential tool for obtaining technology independent descriptions.

The second use-case is when we combine the syntax of two languages. Then, scannerless parsing becomes an essential tool. We combine languages in hybrids, such as COBOL/CICS or COBOL/SQL, or when we translate from one language to another using a meta programming system (like translating C to Java). Scannerless parsing ensures that lexical disambiguations can remain *modular*, with the language they belong to, without influencing the lexical disambiguation rules of the other language. The alternative is to construct one big combined unambiguous scanner, which at the very least will be rather complex, and may not even exist.

3.8 Conclusions

In this chapter we discussed the combination of generalized LR parsing with scannerless parsing. The first parsing technique allows for the development of modular definition of grammars whereas the second one relieves the grammar writer from interface problems between scanner and parser. The combination supports the development of declarative and maintainable syntax definitions that are not forced into the harness of a restricted grammar class such as LL(k) or LR(k). This proves to be very beneficial when developing grammars for legacy languages such as Cobol and PL/I, but it also provides greater flexibility in the development of new (domain-specific) languages.

One of the assets of the SGLR approach is the separation of disambiguation from grammar structure. Thus, it is not necessary to encode disambiguation decisions using extra productions and non-terminals. Instead a number of disambiguation filters, driven by disambiguation declarations solve ambiguities by pruning the parse forest. Lexical ambiguities, which are traditionally handled by ad hoc default decisions in the scanner, are also handled by such filters. Filters can be implemented at several points in time, i.e., at parser generation time, parse time, or after parsing.

SGLR is usable in practice. It has been used as the implementation of the expressive syntax definition formalism SDF. SGLR is not only fast enough to be used in interactive tools, like the ASF+SDF Meta-Environment, but also to parse huge amounts of Cobol code in an industrial environment.

SGLR and the SDF based parse table generator are open-source and can be downloaded from `http://www.cwi.nl/projects/MetaEnv/`.

CHAPTER 4

Semantics Driven Disambiguation

Generalized parsing technology provides the power and flexibility to attack real-world parsing applications. However, many programming languages have syntactical ambiguities that can only be solved using semantical analysis. In this chapter we propose to apply the paradigm of term rewriting to filter ambiguities based on semantical information. We start with the definition of a representation of ambiguous derivations. Then we extend term rewriting with means to handle such derivations. Finally, we apply these tools to some real world examples, namely C and COBOL. The resulting architecture is simple and efficient as compared to semantic directed parsing. [1]

4.1 Introduction

Generalized parsing is becoming more popular because it provides the power and flexibility to deal with real existing programming languages and domain specific languages [9, 46]. It solves many problems that are common in more widely accepted technology based on LL and LR algorithms [7, 92].

We start by briefly recalling the advantages of generalized parsing. It allows arbitrary context-free grammars instead of restricting grammars to classes like LL(k) or LALR(k). Due to this freedom, a grammar can better reflect the structure of a language. This structure can be expressed even better using modularity. Modularity is obtained because context-free grammars are closed under union, as opposed to the more restricted classes of grammars.

An obvious advantage of allowing arbitrary context-free grammars is that the number of grammars accepted is bigger. It seems that real programming languages (e.g., Pascal, C, C++) do not fit in the more restricted classes at all. Without 'workarounds', such as semantic actions that have to be programmed by the user, off-the-shelf parsing technology based on the restricted classes can not be applied to such languages.

[1] This chapter was published at LDTA 2003 [40], and co-authored by Mark van den Brand, Steven Klusener, and Leon Moonen.

The main reason for real programming languages not fitting in the restricted classes is that they are ambiguous in one way or the other. Some grammars have simple conflicts that can be solved by using more look-ahead or by trying more alternative derivations in parallel. Generalized parsing offers exactly this functionality. Other grammars contain more serious ambiguities, which are all accepted as valid derivations. The result is that after parsing with a generalized parser we sometimes obtain a collection of derivations (a parse forest) instead of a single derivation (a parse tree).

4.1.1 Examples

Many examples of the more serious ambiguities can be found in existing programming languages. In this section we will discuss briefly a number of ambiguous constructs which are hard to solve given traditional parsing technology.

Typedefs in C In the C programming language certain identifiers can be parsed as either type identifiers or variable identifiers due to the fact that certain operators are overloaded:

```
Bool *b1;
```

The above statement is either a statement expression multiplying the `Bool` and `b1` variables, or a declaration of a pointer variable `b1` to a `Bool`. The latter derivation is chosen by the C compiler only if `Bool` was declared to be a type using a `typedef` statement somewhere earlier in the program, otherwise the former derivation is chosen. Section 4.4 describes a solution for this problem via the technology presented in this chapter.

Offside rule Some languages are designed to use indentation to indicate blocks of code. Indentation, or any other line-by-line oriented position information is obviously not properly expressible in any context-free grammar, but without it the syntax of such a language is ambiguous. The following quote explains the famous offside rule [115] from the users' perspective:

> "The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase except possibly for bracketed sub-segments."

For example, the following two sentences in typical functional programming style illustrate an ambiguity:

```
a = b                                   a = b
    where b = d                             where b = d
          where d = 1        vs.                    where d = 1
          c = 2                                           c = 2
```

On the left-hand side, the variable `c` is meant to be part of the first `where` clause. Without interpretation of the layout of this example, `c` could just as well part of the second where clause, as depicted by the right-hand side.

There are several languages using some form of offside rule, among others, Haskell [89]. Each of these languages applies "the offside rule" in a different manner making a generic definition of the rule hard to formalize.

Nested dangling constructions in COBOL For C and Haskell we have shown ambiguities that can only be solved using context information. A similar problem exists for COBOL, however we will present a different type of ambiguity here that is based on complex nested statements.[2]

Note that the following example can be parsed unambiguously using some of the existing parser generators. In that case, the parser generators contains implicit heuristics that accidentally fit the disambiguation rules of COBOL. However, we assume the goal is to obtain an explicit definition and implementation of the COBOL disambiguation rules.

The example resembles the infamous *dangling else* construction, but it is more complex due to the fact that more constructs are optional. Consider the following piece of COBOL code in which a nested `ADD` statement is shown:

```
0001 ADD A TO B
0002   SIZE ERROR
0003     ADD C TO D
0004       NOT SIZE ERROR
0005         CONTINUE
0006 .
```

The `SIZE ERROR` and `NOT SIZE ERROR` constructs are optional post-fixes of the `ADD` statement. They can be considered as a kind of exception handling. In order to understand what is going on we will present a tiny part of a COBOL grammar, which is based on [110]:

```
Add-stat           ::= Add-stat-simple Size-err-phrases
Size-err-phrases   ::= Size-err-stats? Not-size-err-stats?
Size-err-stats     ::= "SIZE" "ERROR" Statement-list
Not-size-err-stats ::= "NOT" "SIZE" "ERROR" Statement-list
Statement-list     ::= Statement*
```

The above grammar shows that the COBOL language design does not provide explicit scope delimiters for some deeply nested `Statement-lists`. The result is that in our example term, the `NOT SIZE ERROR` can be either part of the `ADD`-statement on line `0001` or `0003`. The period on line `0006` closes both statements.

The COBOL definition does not have an offside rule. Instead it states that in such cases the "dangling" phrase should always be taken with the innermost construct, which is in our case the `ADD`-statement on line `0003`. There are 16 of such ambiguities in the COBOL definition. Some of them interact because different constructs might be nested.

[2] There are many versions of the COBOL programming language. In this chapter we limit ourselves to IBM VS COBOL II.

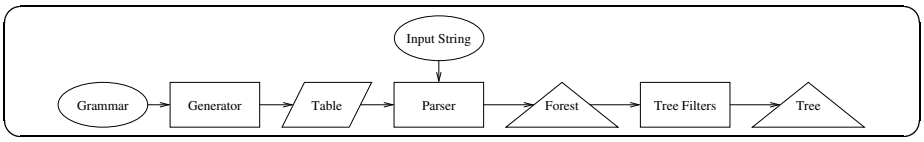

Figure 4.1: General parsing and disambiguation architecture

In some implementations of LL and LR algorithms such dangling cases are implicitly resolved by a heuristic that *always* chooses the deepest derivation. In this chapter we describe a more declarative and maintainable solution that does not rely on such a heuristic.

Discussion The above examples indicate that all kinds of conventions or computed information might have been used in a language design in order to disambiguate its syntax. This information can be derivable from the original input sentence, or from any other source.

Generalized parsing is robust against any grammatical ambiguity. So, we can express the *syntax* of ambiguous programming languages as descriptive context-free grammars. Still, in the end there must be only one parse tree. The structure of this parse tree should faithfully reflect the *semantics* of the programming language. In this chapter we will fill this gap between syntax and semantics, by specifying how to disambiguate a parse forest.

4.1.2 Related work on filtering

The parsing and disambiguation architecture used in this chapter was proposed earlier by [104] and [166]. An overview is shown in Figure 4.1. This architecture clearly allows a separation of concerns between syntax definition and disambiguation. The disambiguation process was formalized using the general notion of a *filter*, quoted from [104]:

> "A filter F for a context-free grammar G is a function from sets of parse trees for G to sets of parse trees for G, where the number of resulting parse trees is equal to or less than the original number."

This rather general definition allows for all kinds of filters and all kinds of implementation methods. In [46] several declarative disambiguation notions were added to context-free grammars (See Chapter 3). Based on these declarations several filter functions were designed that discard parse trees on either *lexical* or *simple structural* arguments. Because of their computational simplicity several of the filters could be implemented early in the parsing process. This was also possible because these filters were based on *common* ambiguity concepts in language design.

In this chapter we target more *complex structural* parse tree selections and selections based on *non-local* information. More important, we aim for *language specific* disambiguations, as opposed to the more reusable disambiguation notions. Such filters

naturally fit in at the back-end of the architecture, just before other semantics based tools will start their job. In fact, they can be considered part of the semantic analysis.

Wagner and Graham [166] discuss the concept of disambiguation filters including an appropriate parse forest formalism, but without presenting a formalism for implementing disambiguation filters. This chapter complements their work by describing a simple formalism based on term rewriting which allows the user to express semantics-guided disambiguation. Furthermore, we give a 'proof of concept' by applying it to real programming languages.

The notion of *semantics/attribute directed parsing* [3, 25] also aims to resolve grammatical ambiguities that can be solved using semantical information. However, the approach is completely different. In case of semantics directed parsing the parser is extended to deal with derived semantic information and directly influence the parsing process. Both in the specification of a language and in the implementation of the technology syntax and semantics become intertwined. We choose a different strategy by clearly separating syntax and semantics. The resulting technology is better maintainable and the resulting language specifications also benefit from this separation of concerns. For example, we could replace the implementation of the generalized parser without affecting the other parts in the architecture[3].

4.1.3 Filtering using term rewriting

Given the architecture described, the task at hand is to find a practical language for implementing language specific disambiguation *filters*. The functionality of every disambiguation filter is similar, it analyzes and prunes parse trees in a forest. It does this by inspecting the structure of sub-trees in the parse forest and/or by using any kind of context information.

An important requirement for every disambiguation filter is that *it may never construct a parse forest that is ill-formed with respect to the grammar of a language*. This requirement ensures that the grammar of a language remains a valid description of the parse forest and thus a valuable source of documentation [93], even after the execution of any disambiguation filters.

The paradigm of *term rewriting* satisfies all above mentioned requirements. It is designed to deal with terms (read trees); to analyze their structure and change them in a descriptive and efficient manner. Term rewriting provides exactly the primitives needed for filtering parse forests. Many implementations of term rewriting also ensure well-formedness of terms with respect to the underlying grammar (a so-called *signature*). Term rewriting provides a solid basis for describing disambiguation filters that are concise and descriptive.

4.1.4 Plan of the chapter

In the rest of this chapter we give the implementation details of disambiguation filters with term rewriting. In Section 4.2, we give a description of the parse tree formalism we use. Section 4.3 briefly describes term rewriting basics before we extend it with the

[3] If the parse forest representation remains the same.

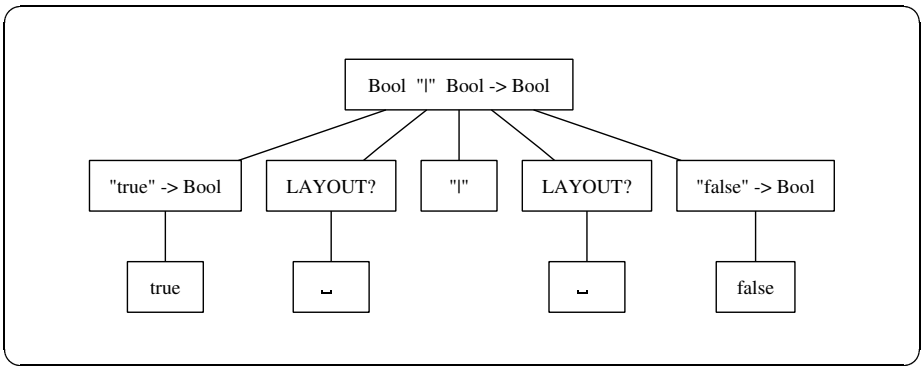

Figure 4.2: An example parse tree.

ability to deal with forests instead of trees. In Section 4.4 we give a number of examples with details to show that it works for real programming languages. In Section 4.5, we discuss how our techniques can be applied to other paradigms and describe our future plans. We present our conclusions in the final section.

4.2 Parse Forest Representation

Based on a grammar, the parser derives valuable information about how a sentence is structured. However, the parser should also preserve any information that might be needed for disambiguation later on. The most obvious place to store all this information is in the syntax tree.

Furthermore, we need a practical representation of the alternative derivations that are the result of grammatical ambiguity. Ambiguities should be represented in such a way that the location of an ambiguous sub-sentence in the input can be pinpointed easily. Just listing all alternative parse trees for a complete sentence is thus not acceptable.

In this section we describe an appropriate parse tree formalism, called AsFix. A more elaborate description of AsFix can be found in [157]. We will briefly discuss its implementation in order to understand the space and time efficiency of the tools processing these parse trees.

AsFix is a very simple formalism. An AsFix tree contains all original characters of the input, including white-space and comments. This means that the *exact* original sentence can be reconstructed from its parse tree in a very straightforward manner. Furthermore, an AsFix tree contains a complete description of all grammar rules that were used to construct it. In other words, all valuable information present in the syntax definition and the input sentence is easily accessible via the parse tree.

Two small examples illustrate the basic idea. Figure 4.2 shows a parse tree of the sentence "`true | false`". Figure 4.3 shows a parse tree of the ambiguous input sentence "`true | false | true`". We have left out the white-space nodes in latter picture for the sake of presentation. The diamond represents an *ambiguity* node which indicates that several derivation are possible for a certain sub-sentence. The

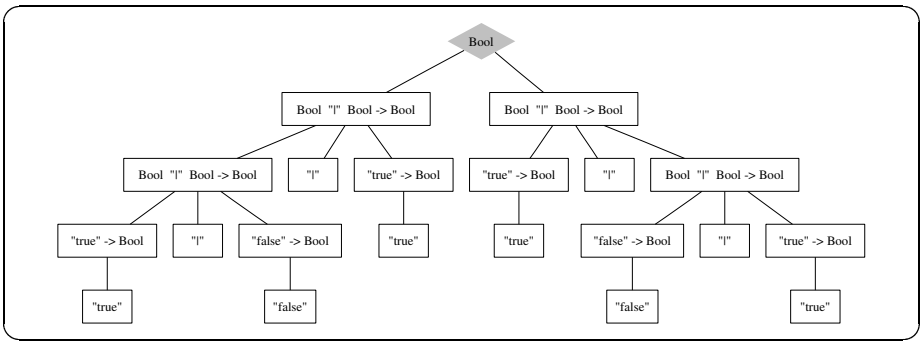

Figure 4.3: An example parse forest.

following grammar (in SDF [87]) was used to parse these two sentences:

```
context-free syntax
  "true"          -> Bool
  "false"         -> Bool
  Bool "|" Bool -> Bool
```

The implementation of AsFix is based on the ATerm library [31]. An AsFix parse tree is an ordinary ATerm, and can be manipulated as such by all utilities offered by the ATerm library. The ATerm library is a library that implements the generic data type ATerm. ATerms are a simple tree-like data structure designed for representing all kinds of trees. The characteristics of the ATerm library are maximal sub-term sharing and automatic garbage collection.

The maximal sharing property is important for AsFix for two reasons. Firstly, the parse trees are completely self-contained and do not depend on a separate grammar definition. It is clear that this way of representing parse trees implies much redundancy. Maximal sharing prevents unnecessary occupation of memory caused by this redundancy. Secondly, for highly ambiguous languages parse forests can grow quite big. The compact representation using ambiguity nodes helps, but there is still a lot of redundancy between alternative parse trees. Again, the ATerm library ensures that these trees can be stored in an minimal amount of memory. To illustrate, Figure 4.4 shows the parse forest of Figure 4.3 but now with full sharing. For the sake of presentation, this picture does not show how even the information in the node labels is maximally shared, for example such that the representation of `Bool` appears only once in memory.

4.3 Extending Term Rewriting

In this section we explain how a parse tree formalism like AsFix can be connected to term rewriting. This connection allows us to use term rewriting directly to specify disambiguation filters. The important novelty is the lightweight technique that is applied to be able to deal with ambiguities. After explaining it we present a small example to

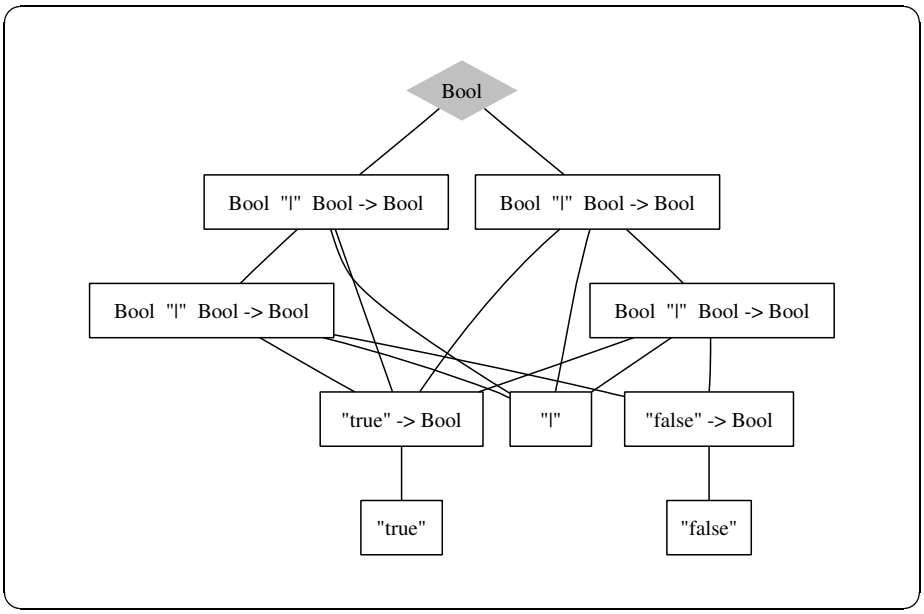

Figure 4.4: A parse forest with maximal sharing.

illustrate the style of specification used for defining disambiguation filters. More elaborate examples are given in Section 4.4. We start by briefly and informally describing the basics of term rewriting.

4.3.1 What is term rewriting?

In short, a Term Rewriting System (TRS) is the combination of a *signature* and a collection of *rewrite rules*. The signature defines the prefix *terms* that are to be rewritten according to the rewrite rules. We refer to [105] for a detailed description of term rewriting.

Signature A many-sorted signature defines all possible terms that can occur in the rewrite rules and the term that is to be rewritten. Usually a signature is extended with a collection of *variables*, which are needed for specifying the rewrite rules. The following is an example signature, with constants (nullary functions), function applications, and a variable definition:

```
signature
  true            -> Bool
  false           -> Bool
  or(Bool,Bool)   -> Bool
variables
  B -> Bool
```

This signature allows ground terms like: "`or(true,or(false,true))`". *Ground* means containing no variables. Terms containing variables are called *open* terms.

Rules A rewrite rule is a pair of such terms `T1 = T2`. Both `T1` and `T2` may be open terms, but `T2` may not contain variables that do not occur in `T1`. Furthermore, `T1` may not be a single variable. A ground term is called a *redex* when it *matches* a left-hand side of any rule. Matching means equality modulo occurrences of variables. The result of a match is a mapping that assigns the appropriate sub-terms to the variable names. A *reduct* can be constructed by taking the right-hand side of the rule and substituting the variables names using the constructed mapping. Replacing the original redex by the reduct is called a *reduction*. Below we give an example of a set of rewrite rules:

```
rules
  or(true, B) = true
  or(false, B) = B
```

The redex "`or(false, false)`" matches the second rule, yielding the binding of `B` to the value `false`. The reduct is `false` after substitution of `false` for `B` in the right-hand side.

In most implementations of term rewriting, the rewrite rules are guaranteed to be *sort-preserving*. This implies that the application of any rewrite rule to a term will always yield a new term that is well-formed with respect to the signature.

Normalization Given a ground term and a set of rewrite rules, the purpose of a rewrite rule interpreter is to find all possible redices in a larger term and applying all possible reductions. Rewriting stops when no more redices can be found. We say that the term is then in *normal form*.

A frequently used strategy to find redices is the *innermost* strategy. Starting at the leafs of the tree the rewriting engine will try to find reducable expressions and rewrite them. For example, "`or(true,or(false,true))`" can be normalized to `true` by applying the above rewrite rules in an innermost way.

Associative matching Lists are a frequently occurring data structure in term rewriting. Therefore, we allow the `*` symbol to represent repetition in a signature:

```
signature
  set(ELEM*) -> SET
variables
  E        -> ELEM
  Es[123] -> ELEM*
```

The argument of the `set` operator is a list of `ELEM` items. By using *list variables*[4], we can now write rewrite rules over lists. The following examples removes all double elements in a `SET`:

[4] `Es[123]` declares three variables, `Es1`, `Es2` and `Es3`, using character class notation.

```
rules
  set(Es1,E,Es2,E,Es3) = set(Es1,E,Es2,Es3)
```

A list variable may bind any number of elements, so left-hand sides that contain list variables may match a redex in multiple ways. One possible choice of semantics is to take the first match that is successful and apply the reduction immediately.

4.3.2 Rewriting parse trees

Grammars as signatures The first step is to exploit the obvious similarities between signatures and context-free grammars. We replace the classical prefix signatures by arbitrary context-free grammars in a TRS. There are three immediate consequences. The non-terminals of a grammar are the sorts. The grammar rules are the function symbols. Terms are valid parse trees over the grammar. Of course, the parse trees can be obtained automatically by parsing input sentences using the user-defined grammar.

Rules in concrete syntax If we want to rewrite parse trees, the left-hand side and right-hand side of rewrite rules should be parse trees as well. We use the same parser to construct these parse trees.[5] In order to parse the variables occurring in the rules, the grammar has to be extended with some variables as well.

Using grammars as signatures and having rules in concrete syntax, the TRS for the `or` can now be written as:

```
context-free syntax
  "true"         -> Bool
  "false"        -> Bool
  Bool "|" Bool -> Bool
variables
  "B"            -> Bool
rules
  true  | B = true
  false | B = B
```

A formalism like this allows us to use term rewriting to analyze anything that can be expressed using an *unambiguous* context-free grammar.

Brackets In order to be able to explicitly express the structure of terms and to be able to express rewrite rules unambiguously, the notion of bracket rules is introduced. The following grammar adds a bracket production to the booleans:

```
context-free syntax
  "(" Bool ")" -> Bool bracket
```

Bracket productions may only be *sort-preserving*. This allows that applications of bracket productions can be removed from a parse tree without destroying the well-formedness of the tree. The result of this removal is a parse tree with the structure that the user intended, but without the explicit brackets.

[5] We implicitly extend the user-defined grammar with syntax rules for the rewrite rule syntax, e.g., `Bool "=" Bool -> Rule`, is added to parse any rewrite rule for booleans.

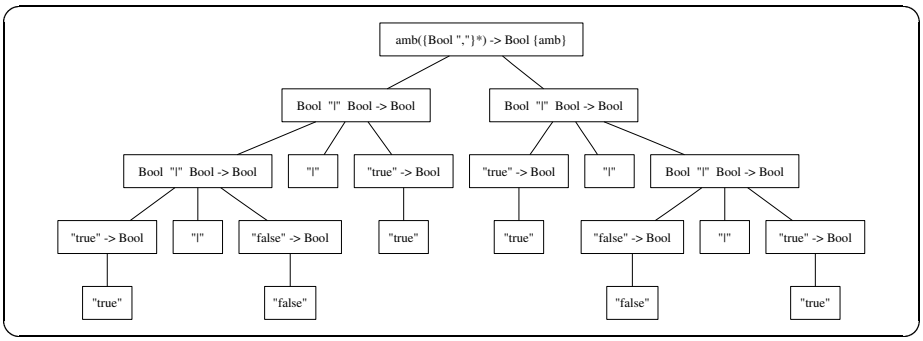

Figure 4.5: Translating an ambiguity node to an ambiguity constructor.

4.3.3 Rewriting parse forests

The step from rewriting parse trees to rewriting parse forests is a small one. If we want to use term rewriting to design disambiguation filters, we want to be able to address ambiguities *explicitly* in a TRS.

Extending the signature The ambiguity nodes that exist in a parse forest are made visible to a TRS by augmenting the signature automatically with a new function symbol for every sort in the signature. The new function symbols are referred to as *ambiguity constructors*. For example, the following ambiguity constructor is added for `Bool` expressions:

context-free syntax
```
amb({Bool ","}*) -> Bool {amb}
```

Each ambiguity constructor has a comma separated list of children. These children represent the original ambiguous derivations for a certain sub-sentence.

Preprocessing before rewriting Just before the normalization process begins we translate each `amb` node in the parse forest of an input term to an application of an ambiguity constructor as described in the previous paragraph. The result of this translation is a *single* parse tree, representing a parse forest, that is completely well-formed with respect to the augmented signature of the TRS.

Figure 4.5 depicts the result of this translation process. It shows how the parse forest from Figure 4.3, containing an ambiguity node, is translated to a parse tree with an explicit ambiguity constructor. Due to the reserved production attribute {`amb`} the translation back is a trivial step. This is done after the normalization process is finished and there are still ambiguities left. This step is necessary in order to make the normal form completely well-formed with respect to the original grammar.

Since the extended signature allows empty ambiguity clusters, e.g., `amb()`, the final translation can sometimes not be made. In this case we return an error message similar to a parse error. An empty ambiguity constructor can thus be used to indicate that a term is semantically incorrect.

Rewrite rules Using the ambiguity constructors, we can now define rewrite rules which process ambiguity clusters. The following example specifies the removal of right-associative application of the "`|`" operator. This is performed by the first rewrite rule

```
variables
  "Bs"[12] -> Bool*
  "B"[123] -> Bool
rules
  amb(Bs1, B1 | (B2 | B3), Bs2) = amb(Bs1, Bs2)
  amb(B1) = B1
```

The second rule transforms an ambiguity cluster containing exactly one tree into an ordinary tree. Using innermost normalization, the above rewrite rules will rewrite a parse forest of the ambiguous term "`true | false | true`" to a parse tree that represents "`(true | false) | true`".

The following features of term rewriting are relevant for this example. Concrete syntax allows specification of the functionality directly in recognizable syntax. The use of brackets is essential to disambiguate the left-hand side of the first rewrite rule. Associative matching is used to locate the filtered derivation directly without explicit list traversal. Finally, the innermost strategy automatically takes care of executing this rewrite rule at the correct location in the parse tree.

4.4 Practical Experiences

The above ideas have been implemented in ASF+SDF [67], which is the combination of the syntax definition formalism SDF and the term rewriting language ASF. ASF+SDF specifications look almost like the last examples in the previous section.

C typedefs To test the concept of rewriting parse forests, we developed a small specification that filters one of the ambiguities in the C language. We started from an ambiguous syntax definition of the C language.

The ambiguity in question was discussed in the introduction. Depending on the existence of a `typedef` declaration an identifier is either interpreted as a type name or as a variable name. The following specification shows how a C `CompoundStatement` is disambiguated. We have constructed an environment containing all declared types. If the first `Statement` after a list of `Declarations` is a multiplication of an identifier that is declared to be a type, the corresponding sub-tree is removed. Variable definitions have been left out for the sake of brevity.

```
context-free syntax
  "types" "[[" Identifier* "]]"  -> Env
  filter(CompoundStatement, Env) -> CompoundStatement
equations
[]Env = types[[Ids1 Id Ids2]]
  ============================
  filter(amb(CSs1,Decls Id * Expr;Stats,CSs2),Env) =
  amb(CSs1,CSs2)
```

Note the use of concrete C syntax in this example. The filter function searches and removes ambiguous block-statements where the first statement uses an identifier as a variable which was declared earlier as a type. Similar rules are added for every part of the C syntax where an ambiguity is caused by the overlap between type identifiers and variable identifiers. This amounts to about a dozen rules. Together they both solve the ambiguities and *document* exactly where our C grammar is ambiguous.

The offside rule in Sasl We have experimented with Sasl [150], a functional programming language, to implement a filter using the offside rule. The following function uses *column* number information that is stored in the parse forest to detect whether a certain parse tree is offside.

```
equations
[] Col = get-start-column(Expr),
   minimal(Expr,Col) < Col = true
   ==============================
   is-offside(NameList = Expr) = offside
```

An expression is offside when a sub-expression is found to the left of the beginning of the expression. The function `minimal` (not shown here) computes the minimal column number of all sub-expressions.

After all offside expressions have been identified, the following function can be used to propagate nested `offside` tags upward in the tree:

```
equations
[] propagate(Expr WHERE offside) = offside
[] propagate(NameList = offside) = offside
[] propagate(offside WHERE Defs) = offside
```

Next, the following two rules are used to remove the offside expressions:

```
equations
[] amb(Expr*1, offside, Expr*2) = amb(Expr*1,Expr*2)
[] amb(Expr) = Expr
```

Note that if no expressions are offside, the ambiguity might remain. Rules must be added that choose the deepest derivation. We have left out these rules here for the sake of brevity because they are similar to the next COBOL example.

Complex nested dangling COBOL statements The nested dangling constructs in COBOL can be filtered using a simple specification. There is no context information involved, just a simple structural analysis. The following rewrite rules filter the derivations where the dangling block of code was not assigned to the correct branch:

```
equations
[] amb(ASs1, AddStatSimple1
              SIZE ERROR Stats1 AddStatSimple2
              NOT SIZE ERROR Stats2,
       ASs2) = amb(ASs1, ASs2)
```

	Size (bytes)	Lines	Parse time (seconds)	Number of ambiguities	Filter time (seconds)
Smallest file	10,454	158	16.57	0	0.46
Largest file	203,504	3,020	36.66	1	10.55
File with most ambiguities	140,256	2,082	28.21	8	7.21
Largest file without ambiguities	124,127	1,844	26.61	0	8.79
Totals of all files	5,179,711	79,667	1818.67	125	259.25
Averages	59,537	916	20.90	1.44	2.98

Table 4.1: Some figures on parsing and filtering performance.

The variable `AddStatSimple2` terminates the nested statement list. In the rule, the `NOT SIZE ERROR` is therefore assigned to the outer `AddStatSimple1` statement instead of the inner `AddStatSimple2`.

This is exactly the alternative that is *not* wanted, so the rule removes it from the forest. We have defined similar disambiguation rules for each of the 16 problematic constructions.

Performance To provide some insight in the computational complexity of the above COBOL disambiguation we provide some performance measures. We used a rewrite rule *interpreter* for these measurements. Compiling these rules with the ASF-compiler [30] would lead to a performance gain of at least a factor 100. However, it is easier to adapt the ASF interpreter when prototyping new language features in ASF. In Table 4.1 we compare the parsing time to time the rewriter used for filtering. The figures are based on a test system of 87 real COBOL sources, with an average file size of almost 1,000 lines of code.

The parse times include reading the COBOL programs and the parse table from disk, which takes approximately 15 seconds, and the construction and writing to disk of the resulting parse forests. The parse table for this full COBOL grammar is really huge, it consists of 28,855 states, 79,081 actions, and 730,390 gotos. The corresponding grammar has about 1,000 productions. It was derived from the original Reference Manual of IBM via the technique described in [110].

The time to execute this set of disambiguation filters for COBOL is proportional to the size of the files and not to the number of ambiguities. The computation visits every node in the parse tree once without doing extensive computations.

4.5 Discussion

Object-oriented programming We demonstrated the concept of semantic filters via rewriting. An important question is what is needed to apply the same idea in a more general setting, for instance using Java or an Attribute Grammar formalism. We will formulate the requirements and needed steps as a recipe:

1. It is necessary to have a parse forest or abstract syntax forest representation which has ambiguity clusters. The amount and type of information stored in the trees influences the expressiveness of the disambiguation filters directly.
2. The ambiguity nodes should be made addressable by the user.
3. It is necessary to create a mapping from the output of the parser to this parse forest representation. This mapping should preserve or derive as much information as possible from the output of the parser.
4. If possible, it is preferable to guarantee that the output of a filter is well-formed with respect to the grammar.

Programming filters becomes a lot simpler if there exists a practical application programming interface (API) to access the information stored in the parse forest representation. ApiGen (See Chapter 9) has a backend that generates Java class hierarchies that provide a mapping from AsFix to a typed abstract syntax tree in Java that is ready for the Visitor design pattern.

Strategic programming In our description of term rewriting we have not addressed the notion of first class rewriting strategies that is present in languages like Stratego [159] and Elan [22]. Rewriting strategies allow the specification writer to explicitly control the application of rewrite rules to terms, as opposed to using the standard innermost evaluation strategy. Ambiguity constructors can be combined seamlessly with rewriting strategies, allowing disambiguation rules to be applied under a certain user-defined strategy. Recently both Elan and Stratego started to use SDF to implement concrete syntax too, [43] and [161], respectively.

4.6 Conclusions

Starting from the notion of generalized parsing we have presented a solution for one of its implications: the ability to accept ambiguous programming languages. Term rewriting can be extended in a simple manner to filter parse forests. Specifying filters by explicitly addressing ambiguity clusters is now as simple as writing ordinary rewrite rules.

The resulting architecture provides a nice separation of concerns and declarative mechanisms for describing syntax and disambiguation of real programming languages.

Practical experience shows that writing filters in term rewriting with concrete syntax is not only feasible, but also convenient. This is due to the seamless integration of context-free syntax definition, parse forests and rewrite rules. Based on a large collection of COBOL programs we have presented performance figures of an interpreter executing a collection of simple disambiguation filters.

CHAPTER 5

A Type-driven Approach to Concrete Meta Programming

Meta programming can be supported by the ability to represent program fragments in concrete syntax instead of abstract syntax. The resulting meta programs are far more self-documenting because "what you see is what you manipulate".

One caveat in concrete meta programming is the syntactic separation between the meta language and the object language. To solve this problem, many meta programming systems use quoting and anti-quoting to indicate precisely where level switches occur. These "syntactic hedges" can obfuscate the concrete program fragments. ASF+SDF has no syntactic hedges, but must trade this for a simple type system, and very basic error messages. Other meta programming systems have more freedom in their typing systems, but must trade this for more syntactic hedges. We analyze this apparent trade-off and bypass it by applying disambiguation by type checking.

This chapter contributes by removing the technical need for quotes and anti-quotes, allowing more "what you see is what you manipulate" than before in meta programming applications. [1]

5.1 Introduction

Applications that manipulate programs as data are called *meta programs*. Examples of meta programs are compilers, source-to-source translators, type-checkers, documentation generators, refactoring tools, and code generators. We call the language that is used to manipulate programs the *meta language*, and the manipulated language the *object language*.

[1] An abridged version of this chapter appeared in RISE 2005 [155].

Any general purpose programming language can be used to write meta programs. The program is represented using the data type constructs available in that language. In general, the data type used is an abstract syntax tree, but sometimes a string representation seems to be sufficient. Many code generators just print out strings for example, while compilers and type-checkers depend on a more structured representation of a program. Although the string representation is very readable, it is to weak to be a device for modeling program structure. Consequently strings do not scale for use in more complex meta programs; there is not enough compile-time checking.

The main disadvantage of using a general purpose programming language for meta programming is that the original program is encoded in a very complex data type. The abstract syntax tree data type has to be designed in advance, and the basic functionality to manipulate this data also needs to be developed before the actual manipulation can be expressed. Tool support is offered to help design the above features, such as code generators for abstract syntax tree representations (see Chapter 9). Some tools provide a typed interface to program structures, modeling programs more faithfully and strictly. The high level of abstraction offered by these tools brings a meta program conceptually closer to the object program. The meta program becomes more self-documenting.

An old idea to facilitate meta programming is the use of *concrete syntax* to represent program fragments [124]. Using concrete syntax, as opposed to abstract syntax, all program fragments in a meta program are represented in the syntax of the object language [144, 87]. Concrete syntax combines the readability of the string representation with the structural and type-safe representation of abstract syntax trees. The meta programmer embeds the actual program fragments literally in his meta program, but the underlying representation of these fragments is an abstract syntax tree. The resulting meta programs are far more self-documenting because "what you see is what you manipulate". Anybody who can program in the object language can understand and even write the embedded program fragments in a concrete meta program. Compare this to learning a complex abstract syntax representation that may be application specific.

One caveat in concrete meta programming is the syntactic separation between the meta language and the object language. Conventional scanning and parsing technologies have a hard time distinguishing the two levels. To solve this problem, many meta programming systems use quoting and anti-quoting to indicate precisely where level switches are made. To further guide the system, sometimes the user is obliged to explicitly mention the type of the following program fragment. These "syntactic hedges" help the parser, but they can obfuscate the concrete program fragments. In practice, it leads to programmers avoiding the use of concrete syntax because the benefit becomes much less clear when it introduces more syntactic clutter than it removes.

Road-map. This chapter contributes by removing the technical need for quotes and anti-quotes, allowing more "what you see is what you manipulate" than before in meta programming applications. We first explore meta programming with concrete syntax in some detail and then describe a number of existing systems that implement it (Section 5.1.2). We shortly evaluate these systems and then describe the goals of our work, before we detail the actual contributions.

We introduce an architecture that automatically detects implicit transitions from

Data-structure	Linear (strings)	vs.	Structured (trees)
Typing	Untyped (homogeneous)	vs.	Typed (heterogeneous)
Syntax	Abstract syntax	vs.	Concrete syntax
Quoting	Explicit quoting	vs.	Implicit quoting
Annotations	Explicit type annotations	vs.	No explicit type annotations
Nested meta code	Not allowed	vs.	Allowed

Table 5.1: Solution space of code fragment representation in meta programming.

meta language to object language (Section 5.2). This architecture is based on generalized parsing and a separate type-checking phase. By making the transitions between meta language and object language implicit we introduce some challenges for the parsing technology: ambiguous and cyclic grammars. In Section 5.3 we address these issues. Sections 5.4 and 5.5 describe experience and conclusions respectively.

5.1.1 Exploring the solution space

Table 5.1 is an overview of the syntactic features of meta programming languages for representing code fragments. We prefer systems that represent program fragments using all the features in the right column, since there is no syntactic distance between the program fragments and the actual object language.

Suppose we use Java as a meta programming language to implement a Java code generator. Consider the following method that generates a Java method. It is written in four different styles: using strings, using some abstract syntax, concrete syntax notation with quotation, and finally concrete syntax without quotation. To demonstrate the expressivity of concrete syntax in a familiar setting, we use an imaginary concrete syntax feature for Java in the last two examples.

String representation

```
String buildSetter(String name, String type) {
  return "public void set" + name + "(" + type + " arg)\n"
       + "  this." + name + " = arg;\n"
       + "}\n";
}
```

The string representation is unstructured, untyped and uses quotes and anti-quotes. There is no guarantee that the output of this method is a syntactically correct Java method. However, the code fragment is immediately recognizable as a Java method.

Abstract syntax representation

```
String buildSetter(String name, String type) {
  Method method = method(
       publicmodifier(), voidType(), identifier("set" + name),
       arglist(formalarg(classType(type),identifier("arg"))),
       statlist(stat(assignment(
             fieldref(identifier("this"),identifier(name)),
             expression(identifier("arg"))))));
  return method.toString();
}
```

This style uses a number of methods for constructing an abstract syntax tree in a bottom-up fashion. If the used construction methods are strictly typed, this style exploits the Java type system towards obtaining a syntactically correct result. That means that if all `toString()` methods of the abstract representation are correct, then the new expression will also generate syntactically correct Java code. An alternative to the above is to use a generic tree data type. This leads to an *untyped* representation, such that there is no guarantee on the syntactic correctness of the result.

Concrete syntax representation with quoting

```
String buildSetter(String name, String type) {
   Method method = [[public void '"set" + name' ( 'type' arg) {
                       this.'name' = arg;
                    }]];
   return method.toString();
}
```

To guide the parser the user explicitly indicates the transition from meta language to object language by quoting (`[[...]]`), and the transition from object language to meta language by anti-quoting (`'...'`). This style with quoting is close to the string representation in terms of readability. It is structured and typed. The implementation of this syntax would introduce compile-time parsing and type-checking of the generated code fragment with respect to the Java grammar. This means that it is guaranteed to generate syntactically correct Java code, except for the parts that are anti-quoted.

Concrete syntax representation without quoting

```
String buildSetter(String name, String type) {
  Method method = public void "set" + name ( type arg ) {
                    this.name = arg;
                  };
  return method.toString();
}
```

The final example shows our ideal notation. There is a seamless integration between meta language and object language. This approach automates the quoting and anti-quoting for the user. Compare this to type-inference in functional programming languages; if a type can be inferred automatically, we do not ask the user to type it in. Similarly, in concrete syntax without quotation we want to infer the quotes automatically without asking the user to express the obvious.

5.1.2 Concrete meta programming systems

The ASF+SDF system is based on scannerless generalized LR parsing (SGLR) [157, 46] and conditional term rewriting. The syntax of the object language is defined in the SDF formalism, after which rewrite rules in ASF in concrete syntax define appropriate transformations [28]. The SGLR algorithm takes care of a number of technical issues that occur when parsing concrete syntax:

- It accepts all context-free grammars, which are closed under composition. This allows the combination of any meta language with any object language.
- Due to scannerless parsing, there are no implicit global assumptions like longest match of identifiers, or reserved keywords. Such assumptions would influence the parsing of meta programs. For example, the combined language would have the union set of reserved keywords, which is incorrect in either separate language.
- Parallel parse stacks take care of local conflicts in the parse table.

ASF+SDF does not have quoting, or anti-quoting. There are two reasons for this. Firstly, within program fragments no nested ASF+SDF constructs occur that might overlap or interfere. Secondly, the ASF+SDF parser is designed in a very specific manner. It only accepts type correct programs because a specialized parser is generated for each ASF+SDF module. The following rephrases the examples of the introduction in ASF+SDF:

```
context-free syntax
   buildSetter(Identifier, Type) -> Method
variables
  "Name"   -> Identifier
  "Type"   -> Type
equations
[] buildSetter(Name, Type) =
   public void set ++ Name (Type arg) {
     this.Name = arg;
   }
```

Note that we have used an existing definition of the concatenation of Identifiers (++).

This notation is achieved by exploiting a one-to-one correspondence between the type system of ASF+SDF and context-free grammars: non-terminals are types, and productions are functions. The type system of ASF+SDF entails that all equations are *type preserving*. To enforce this rule, a special production is generated for

each user-defined non-terminal `X`: `X "=" X -> Equation`. So instead of having one `Term "=" Term -> Equation` production, ASF+SDF generates specialized productions to parse equations. After this syntax generation, the fixed part of ASF+SDF is added. That part contains the skeleton grammar in which the generated syntax for `Equation` is embedded.

The equation shown above has some syntactic ambiguity. The meta variable `Type` for example, may be recognized as a Java class name, or as a meta variable. Another ambiguity is due to the following user-defined *injection* production: `Method -> Declaration`. By applying it to both sides of the equation, it may also range over the `Declaration` type instead of simply over `Method`. To disambiguate such fragments, ASF+SDF uses two so called *meta disambiguation* rules:

Rule 1: Prefer to recognize declared meta variables instead of object syntax identifiers.

Rule 2: Prefer shorter injection chains.

Rule 1 separates meta variables from program fragments. Rule 2 prefers the more specific interpretations of rules, an arbitrary but necessary disambiguation. Note that such ambiguities also occur for productions that can simulate injections. For example in `A X B -> Y`, where both `A`, and `B` are non-terminals that optionally produce the empty string. We call this a *quasi-injection* from `X` to `Y`. Quasi injections are not covered by meta disambiguation rule 2.

Although the above design offers the concrete syntax functionality we seek, the assumptions that are made limit its general applicability:

- The type system of the meta language must be expressible as a context-free grammar. Higher-order functions or parametric polymorphism are not allowed.
- Usually, meta programming languages offer more meta level constructs than meta variables only. Consider for example `let` or `case` expressions.
- Typing errors are reported as parsing errors which makes developing meta programs unnecessarily hard.
- The user is expected to pick meta variable names that limit the amount of ambiguities that can not be solved by the above two meta disambiguation rules. No feedback other than ambiguity reports and parse tree visualizations are given to help the user in this respect.

In Stratego [159] the concrete object syntax feature also uses SDF for syntax definition and SGLR to parse Stratego programs. The separation between the meta language and the object language is done by quoting and anti-quoting. The programmer first defines quotation and anti-quotation notation syntax herself. Then the object language is combined with the Stratego syntax. After parsing, the parse tree of the meta program is mapped automatically to normal Stratego abstract syntax [161].

By letting the user define the quotation operators, Stratego offers a very explicit way of combining meta language and object language. This is natural for Stratego, since:

- There is no type system, so parsing can not be guided by a type context.
- There are meta operators that could appear nested in a program fragment.

The following are example user-defined quotation and anti-quotation operators for a non-terminal in Java, with or without explicit types:

```
context-free syntax
          "|[" Method "]|" -> Term   {cons("toMetaExpr")}
"Method" "|[" Method "]|" -> Term   {cons("toMetaExpr")}
          "~" Term         -> Method {cons("fromMetaExpr")}
 "~Method:"  Term          -> Method {cons("fromMetaExpr")}
```

The productions' attributes are used to guide the automated mapping to Stratego abstract syntax.

The ambiguities that occur in ASF+SDF due to injections and quasi-injections also occur in Stratego, but the user can always use the explicitly typed quoting operators. An example code fragment in Stratego with meta variables defined in SDF is:

```
context-free syntax
  "|[" Method "]|" -> Term        {cons("toMetaExpr")}
  "~" Term         -> Identifier  {cons("fromMetaExpr")}
variables
  "type" -> Type
strategies
builderSetter(|name, type) =
    !|[public void ~<conc-strings> ("set", name)(type arg)
         this.~name = arg;

     ]|
```

In this example, we used both Stratego syntax like the `!` operator and the `conc-strings` library strategy, and Java object syntax. We assume no quotation operator for `Declaration` is present, otherwise an explicitly typed quote should have been used to disambiguate. To indicate the difference, we used an implicit meta variable for the type argument, and a normal explicitly anti-quoted variable for the field name that we set.

The above leaves a part of implementing concrete syntax, namely combining the meta language with the object language to the user. The use of quotation makes this job easier, but the resulting meta programs contain many quoting operators. Questions the user must be able to answer are:

- For which non-terminals should quotation operators be defined.
- When to use explicit typing.
- What quotation syntax will be appropriate for a specific non-terminal.

If not carefully considered, the answers to these questions might differ for different applications for the same object language. A solution proposed in [161] is to generate quoting operators automatically from the syntax definition of the object language. The current solution is to let an expert define the quotation symbols for a certain language, and put this definition in a library. Still, like the ASF+SDF system, the feedback that such a design can offer in case the user or the expert makes a mistake is limited to parse errors and ambiguity reports.

Concrete syntax in Lazy ML. In [1] an approach for adding concrete syntax to Lazy ML is described. This system also uses quotation operators. It employs scannerless parsing with Earley's algorithm, which is roughly equivalent to SGLR. Disambiguation of the meta programs with program fragments is obtained by:

- Full integration of the parser and type-checker of Lazy ML. All type information can be used to guide the parser. So, only type correct derivations are recognized.
- Overlapping meta variables and the injection problem are partially solved by optionally letting the user annotate meta variables explicitly with their type inside program fragments.

This system is able to provide typing error messages instead of parse errors. Both the level of automated disambiguation, and the level of the error messages are high. There is explicit quoting and anti-quoting necessary:

```
fun buildSetter name type =
          [| public void ^(concat "set" name) (^type arg) {
               this.^name = arg;
             }
          |]
```

The Jakarta Tool Suite is for extending programming languages with domain specific constructs. It implements and extends ideas of intentional programming, and work in the field of syntax macros [118]. Language extension can be viewed as a specific form of meta programming, with a number of additional features.

The parser technology used in JTS is based on a separate lexical analyzer and an LL parser generator. This restricts the number of language extensions that JTS accepts, as opposed to scannerless generalized parsing algorithms. The program fragments in JTS are quoted with explicit typing. For every non-terminal there is a named quoting and an anti-quoting operator, for example:

```
public FieldDecl buildSetter(String name, String type) {
   QualifiedName methodName = new QualifiedName("set" + name);
   QualifiedName fieldName = new QualifiedName(name);
   QualifiedName typeName = new QualifiedName(type);
   return mth{public void $id(methodName) ($id(typeName) arg) {
               this.$id(fieldName) = arg;
             }
   }
```

Meta-Aspect/J is a tool for meta programming Aspect/J programs in Java [172]. It employs context-sensitive parsing, similar to the approach taken for ML. As a result, this tool does not need explicit typing:

```
MethodDec buildSetter(String name, String type) {
  String methodName = "set" + name;
  return `[public void #methodName (#type arg) {
              this.#name = arg;
   }
  ];
}
```

Note that Meta Aspect/J offers a fixed combination of one meta language (Java) with one single object language (Aspect/J), while the other systems combine one meta language with many object languages.

TXL [59] is also a meta programming language. It uses backtracking to generalize over deterministic parsing algorithms. TXL has a highly structured syntax, which makes extra quoting not necessary. Every program fragment is enclosed by a certain operator. The keywords of the operators are syntactic hedges for the program fragments:

```
function buildMethod Name [id] Type [type]
    construct MethodName [id]
        _ [+ 'set]  [+ Name]
    replace [opt method]
    by
        public void MethodName (Type arg) {
            this.Name = arg;
        }
end function
```

The example shows how code fragments are explicitly typed, and also the first occurrence of fresh variables. The `[...]` anti-quoting operator is used for explicit typing, but it can also contain other meta level operations, such as recursive application of a rule or function. The keywords `construct`, `replace`, `by`, etc. can not be used inside program fragments, unless they are escaped.

Although technically TXL does use syntactic hedging, the user is hardly aware of it due to the carefully designed syntax of the meta language. The result is that, compared to other meta programming languages, TXL has more keywords.

5.1.3 Discussion

Table 5.2 summarizes the concrete meta programming systems just discussed. The list is not exhaustive, there are many more meta programming systems, or language extension systems out there. Clearly the use of quoting and anti-quoting is a common design decision for meta programming systems with concrete syntax. Explicit typing is also used in many systems. Only ASF+SDF does not use quoting or explicit typing, except

	ASF	**Stratego**	**ML**	**JTS**	**TXL**	**MAJ**
Typed	+	−	+	+	+	+
Implicit quoting	+	Opt.	−	−	+	−
No type annotations	+	Opt.	−	−	−	+
Nested meta code	−	+	+	−	+	−

Table 5.2: Concrete syntax in several systems.

that the meta variable names picked by the user are a form of explicit disambiguation. Type-safety is implemented in most of the systems described.

From studying the above systems and their concrete syntax features, we draw the following conclusions:

- The more implicit *typing context* is provided by the meta programming language, the less syntactic hedges are necessary. Strictly typed languages will therefore be more appropriate for concrete syntax without hedges than other languages.
- Syntactic hedges are not necessarily obfuscating the code patterns. The TXL example shows how a carefully chosen keyword structure provides a solution that does not bother the user too much with identifying the transitions to and from the meta level.
- Even if syntactic hedges are not necessary, some kind of visualization for identifying the transitions to and from the meta level is always beneficial. *Syntax highlighting* is a well known method for visualizing syntactic categories.
- It is hard to validate the claim that less quotation and anti-quotation is better in all cases. Possibly, this boils down to a matter of taste. Evidently *unnecessary* syntactic detail harms programmer productivity, but that argument just shifts the discussion to what is necessary and what is not. A hybrid system that employs both quote inferencing and explicit quoting would offer the freedom to let the user choose which is best.
- The most important shortcoming of any system that employs concrete syntax is the low quality of the error messages that it can provide.

Our goal is now to design a parsing architecture that can recognize concrete syntax without hedges, embedded in languages with non-trivial expression languages, but with strict type systems. Providing a hybrid system that also allows explicit hedges is a trivial extension that we do not discuss further. Also syntax highlighting is well known functionality that does not need further discussion.

5.2 Architecture

We start with a fixed syntax definition for a meta language and a user-defined syntax definition for an object language. In Fig. 5.1 the general architecture of the process

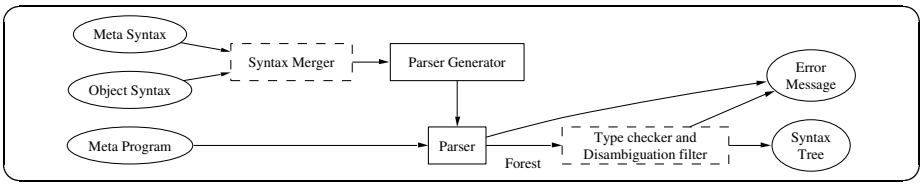

Figure 5.1: Overview: parsing concrete syntax using type-checking to disambiguate.

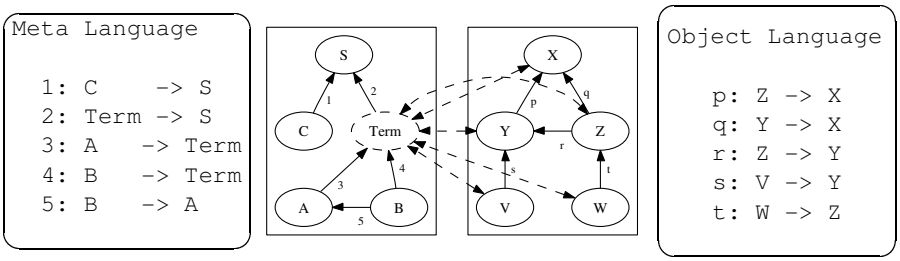

Figure 5.2: A trivial meta syntax and object syntax are merged. Syntax transitions bidirectionally connect all non-terminals in the object language to the `Term` non-terminal in the meta language

starting from these two definitions and a meta program, and ending with an abstract syntax tree is depicted. The first phase, the syntax merger, combines the syntax of the meta language with the syntax of the object language. The second phase parses the meta program. The final phase type-checks and disambiguates the program.

5.2.1 Syntax transitions

The syntax merger creates a new syntax module, importing both the meta syntax and the object syntax. We assume there is no overlap in non-terminals between the meta syntax and the object syntax, or that renaming is applied to accomplish this. It then adds productions that link the two layers automatically. For every non-terminal X in the object syntax the following productions are generated (Fig. 5.2): X `-> Term` and `Term -> X`, where `Term` is a unique non-terminal selected from the meta language. For example, for Java, the `Term` non-terminal would be `Expression`, because expressions are the way to build data structures in Java.

We call these productions the *transitions* between meta syntax and object syntax. They replace any explicit quoting and unquoting operators. In order for easy recognition, we will call the transitions to meta syntax the *quoting transitions* and the transitions to object syntax the *anti-quoting transitions*. Figure 5.3 illustrates the intended purpose of the transitions: nesting object language fragments in meta programs, and nesting meta language fragments again in object language fragments.

The collection of generated transitions from and to the meta language are hazardous for two reasons. They introduce many ambiguities, including cyclic derivations. An *ambiguity* arises when more than one derivation exists for the same substring with the

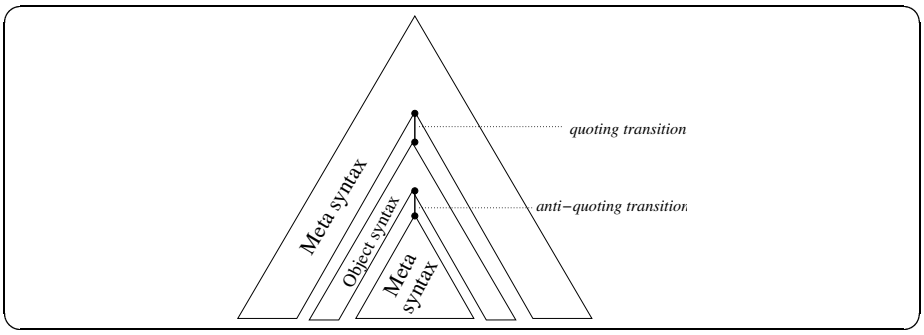

Figure 5.3: A parse tree may contain both meta and object productions, where the transitions are marked by quoting and unquoting transition productions.

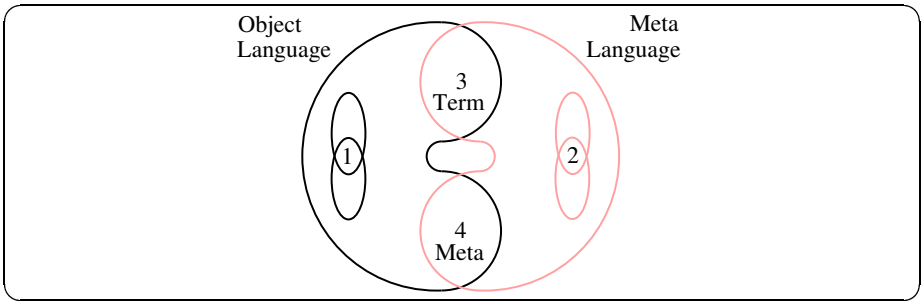

Figure 5.4: Classification of ambiguities caused by joining a meta language with an object language.

same non-terminal. Intuitively, this means there are several interpretations possible for the same substring. A *cycle* occurs in derivations if and only if a non-terminal can produce itself without consuming terminal symbols. Cycles are usually meaningless: they have no semantics. To get a correct parser for concrete meta programs without quoting, we must resolve all cycles and ambiguities introduced by the transitions between meta and object syntax. Figure 5.4 roughly classifies the ambiguities that may occur:

Class 1: Ambiguity in the object language itself. This is an artifact of the user-defined syntax of the object language. Such ambiguity must be left alone, since it is not introduced by the syntax merger. The C language is a good example, with its overloaded use of the `*` operator for multiplication and pointer dereference.

Class 2: Ambiguity of the meta language itself. This is to be left alone too, since it is not introduced by the syntax merger. Usually, the designer of the meta language will have to solve such an issue separately.

Class 3: Ambiguity directly via syntax transitions. The `Term` non-terminal accepts all sub languages of the object language: “everything is a Term”. Parts of the object language that are nicely separated in the object grammar, are now overlaid

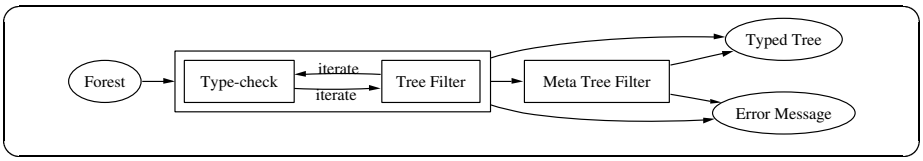

Figure 5.5: The organization of the type-checking and disambiguation approach.

on top of each other. For example, the *isolated* Java code fragment `i = 1` could be a number of things including an assignment statement, or the initializer part of a declaration.

Class 4: Object language and meta language overlap. Certain constructs in the meta language may look like constructs in the object language. In the presence of the syntax transitions, it may happen that meta code can also be parsed as object code. For example, this hypothetical Java meta program constructs some Java declarations: `Declarations decls = int a; int b;`. The `int b;` part can be in the meta program, or in the object program.

We can decide automatically in which class an ambiguity falls. Class 1 or class 2 ambiguities only exercise productions from the object grammar or meta grammar respectively. If the top of the alternatives in an ambiguity cluster exercise the transition productions, it falls into class 3. The other ambiguities fall into class 4, they occur on meta language non-terminals and exercise both the transition productions and object language productions. Note that ambiguities may be nested. Therefore, we take a bottom-up approach in classifying and resolving each separate ambiguity.

5.2.2 Disambiguation by type-checking

Generalized parsing algorithms do not complain about ambiguities or cycles. In case of ambiguity they produce a "forest" of trees, which contain compact representations of the alternative derivations. In case of cycles, parse forests simply contain back edges to encode the cycle, parse graphs.

The construction of parse forests, instead of single trees, enables an architecture in which the disambiguation process is merged with the type checking algorithm rather than integrated in the parsing algorithm. The parser returns a parse forest. After this the type-checker filters out a single type-correct tree or returns a type error. This architecture is consistent with the idea of disambiguation by filtering as described by [104]. Figure 5.5 shows the organization of the type-checking and disambiguation approach.

Type-checking is a phase in compilers where it is checked if all operators are applied to compatible operands. Traditionally, a separate type-checking phase takes an abstract syntax tree as input and one or more symbol tables that define the types of all declared and built-in operators. The output is either an error message, or a new abstract syntax tree that is decorated with typing information [2]. Other approaches incorporate type-checking in the parsing phase [1, 133] to help the parser avoid conflicts. We do

the exact opposite, the parser is kept simple while the type-checker is extended with the ability to deal with alternative parse trees.

Type-checking forests is a natural extension of normal type-checking of trees. A forest may have several sub-trees that correspond to different interpretations of the same input program. Type-checking a forest is the process of selecting the single type correct tree. If no single type correct tree is available then we deal with the following two cases:

- No type correct abstract syntax tree is available; present the collection of error messages corresponding to all alternative trees,
- Multiple type correct trees are available; present an error message explaining the alternatives.

Note that resolving the ambiguities caused by syntax transitions by type-checking is a specific case of type-inference for polymorphic functions [125]. The syntax transitions can be viewed as overloaded (ad-hoc polymorphic) functions. There is one difference: the forest representation already provides the type-inference algorithm with the set of instantiations that is locally available, instead of providing one single abstract tree that has to be instantiated.

Regarding the feasibility of this architecture, recall that the amount of nodes in a GLR parse forest can be bounded by a polynomial in the length of the input string [91, 17]. This is an artifact of smart sharing techniques for parse forests produced by generalized parsers. Maximal sub-term sharing [31] helps to lower the average amount of nodes even more by sharing all duplicated sub-derivations that are distributed across single and multiple derivations in a parse forest.

However, the scalability of this architecture still depends on the size of the parse forest, and in particular the way it is traversed. A maximally shared forest may still be traversed in an exponential fashion. Care must be taken to prevent visiting unique nodes several times. We use *memoization* to make sure that each node in a forest is visited only once.

In the following section we describe the tree filters that are needed to disambiguate the ambiguities that occur after introducing the syntax transitions.

5.3 Disambiguation filters

We will explicitly ignore ambiguity classes 1 and 2, such that the following disambiguation filters do not interfere with the separate definitions of the meta language and the object language. The should only deal with the ambiguities introduced by merging the two languages. We will further analyze ambiguity classes 3 and 4 from Figure 5.4, and explain how the disambiguating type-checker will either resolve these ambiguities or produce an error message.

5.3.1 Class 3. Ambiguity directly via syntax transitions

We further specialize this class intro three parts:

Class 3.1: Cyclic derivations. These are derivations that do not produce any terminals and exercise syntax transitions both to and from the meta grammar. For example, every X has a direct cycle by applying X `-> Term` and `Term -> X`.

Class 3.2: Meaningless coercions. These are derivations that exercise the transition productions to cast any X from the object language into another Y. Namely, every X can be produced by any other Y now by applying `Term -> X` and Y `-> Term`.

Class 3.2: Ambiguous quoting transitions. Several X `-> Term` are possible from different Xs. The ambiguity is on the `Term` non-terminal. For any two non-terminals X and Y that produce languages with a non-empty intersection, the two productions X `-> Term` and Y `-> Term` can be ambiguous.

Class 3.3: Ambiguous anti-quoting transitions. Several `Term -> X` are possible, each to a different X. The ambiguity is on the object language non-terminal X, but the cause is that the `Term` syntax is not specific enough to decide which X it should be. For any two productions of the object language that produce the same non-terminal this may happen. A `-> X` and B `-> X` together introduce an anti-quoting ambiguity with a choice between `Term -> A` and `Term -> B`.

In fact, classes 3.1 and 3.2 consist of degenerate cases of ambiguities that would also exist in classes 3.2 and 3.3. We consider them as a special case because they are easier to recognize, and therefore may be filtered with less overhead. The above four subclasses cover all ambiguities caused directly by the transition productions. The first two classes require no type analysis, while the last two classes will be filtered by type checking.

Class 3.1. Dealing with cyclic derivations

The syntax transitions lead to cycles in several ways. Take for example the two cyclic derivations displayed in Figure 5.6. Such cycles, if introduced by the syntax merger, always exercise at least one production X `-> Term`, and one production `Term -> Y` for any X or Y.

Solution 1. The first solution is to filter out cyclic derivations from the parse forest. With the well known `Term` non-terminal as a parameter we can easily identify the newly introduced cycles in the parse trees that exercise cyclic applications of the transition productions. A single bottom-up traversal of the parse forest that detects cycles by marking visited paths is enough to accomplish this. With the useless cyclic derivations removed, what remains are the useful derivations containing transitions to and from the meta level.

We have prototyped solution 1 by extending the ASF+SDF parser with a cycle filter. Applying the prototype on existing specifications shows that for ASF+SDF such an approach is feasible. However, the large amount of meaningless derivations that

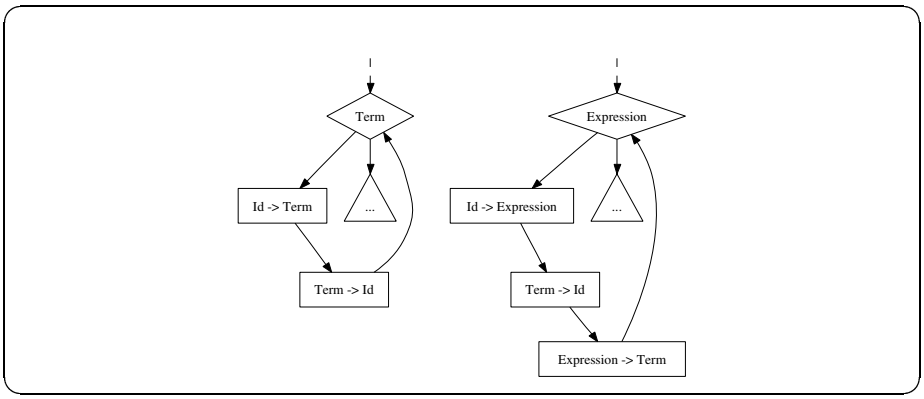

Figure 5.6: Two cyclic derivations introduced by short circuiting quoting and anti-quoting transitions.

are removed later do slow down the average parse time of an ASF+SDF module significantly. To quantify, for smaller grammars with ten to twenty non-terminals we witnessed a factor of 5, while for larger grammars with much more non-terminals we witnessed factors of 20 times slower parsing times.

Solution 2. Instead of filtering the cycles from the parse forest, we can prevent them by filtering reductions from the parse table. This technique is based on the use of a disambiguation construct that is described in Chapter 3. We use *priorities* to remove unwanted derivations, in particular we remove the reductions that complete cycles. The details of this application of priorities to prevent cycles are described in a technical report [154]. The key is to automatically add the following priority for every object grammar non-terminal X: X `-> Term > Term ->` X`.` Because priorities are used to remove reductions from the parse table, many meaningless derivations are not tried at all at parsing time.

Discussion. Prototyping the second scheme resulted in a considerable improvement of the parsing time. The parsing time goes back to almost the original performance. However parse table generation time slows down significantly. So when using solution 2, we trade some compilation time efficiency for run-time efficiency. In a setting with frequent updates to the object grammar, it may pay off to stay with solution 1. The conclusion is that a careful selection of existing algorithms can overcome the cycle challenge for a certain price in runtime efficiency. This price is hard to quantify exactly, since it highly depends on the object grammar. However, the theoretical worst-case upper-bound is given by the polynomial size of the parse forest generated by any Tomita-style generalized parser.

Class 3.2. Dealing with meaningless coercions

For every pair of non-terminals X and Y of the object language that produce languages that have a non-empty intersection, an ambiguity can be constructed by applying the productions X `-> Term` and `Term -> Y .` Effectively, such a derivation casts an X to an Y, which is a meaningless coercion.

These ambiguities are very similar to the cyclic derivations. They are meaningless derivations occurring as a side-effect of the introduction of the transitions. Every direct nesting of an unquoting and a quoting transition falls into this category. As such they are identifiable by structure, and a simple bottom-up traversal of a parse forest will be able to detect and remove them. No type information is necessary for this. Also, introducing priorities to remove these derivations earlier in the parsing architecture is applicable.

Class 3.3. Dealing with ambiguous quoting

So far, no type checking was needed to filter the ambiguities. This class however is more interesting. The X `-> Term` productions allow everything in the object syntax to be `Term`. If there are any two non-terminals of the object language that generate languages with a non-empty intersection, and a certain substring fits into this intersection we will have an ambiguity. This happens for example with all injection productions: X `-> Y ,` since the language accepted by X is the same as the language accepted by Y.

An ambiguity in this class consists of the choice of nesting an X, or an Y object fragment into the meta program. So, either by X `-> Term` or by Y `-> Term` we transit from the object grammar into the meta grammar. The immediate typing context is provided by the meta language. Now suppose this context enforces an X. Disambiguation is obtained by removing all trees that do not have the X `-> Term` production on top.

The example in Fig. 5.7 is a forest with an ambiguity caused by the injection problem. Suppose that from a symbol table it is known that `f` is declared to be a function from `Expression` to `Identifier`. This provides a type-context that selects the transition to `Expression` rather than the transition to `Identifier`.

Class 3.4. Dealing with ambiguous anti-quoting

This is the dual of the previous class. The `Term -> X` productions cause that at any part of the object language can contain a piece of meta language. We transit from the meta grammar into the object grammar. The only pieces of meta language allowed are produced by the `Term` non-terminal. The typing context is again provided by the meta language, but now from below. Suppose the result type of the nested meta language construct is declared X, then we filter all alternatives that do not use the `Term -> X` transition.

Discussion

To implement the above four filters a recursive traversal of the forest is needed. It applies context information on the way down and brings back type information on the

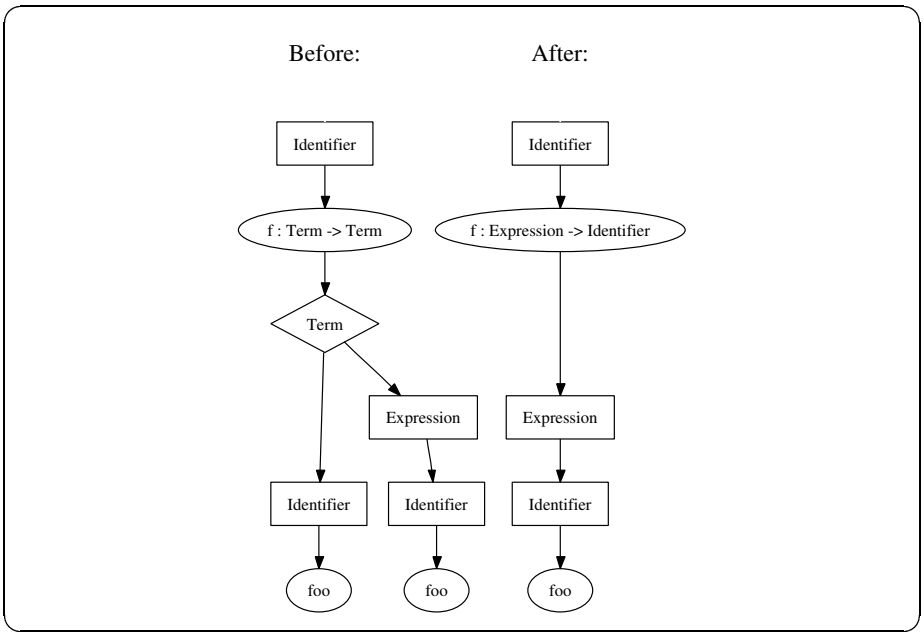

Figure 5.7: An abstract syntax forest is disambiguated by using a type declaration for the function `f`.

way back. On the one hand, the more deterministic decisions can be made on the way down, cutting off branches before they are traversed, the more efficient the type-checking algorithm will be. On the other hand, filtering of nested ambiguities will cut off completely infeasible branches before any analysis needs to be made on the way back.

The above approach assumes all object program fragments are located in a typing context. Language that do not satisfy such an assumption must use either explicit typing for the top of a fragment, or provide a meta disambiguation rule as described in the following. For many meta/object language combinations it is very improbable that after the above type analyses ambiguities still remain. However, it is still possible and we must cover this class as well. The type-checker can produce an accurate error message for them, or apply the meta disambiguation rules that are discussed next.

5.3.2 Class 4. Object language and meta language overlap

The most common example in this class is how to separate meta variables from normal identifiers in the object syntax (Section 5.1.2). Other examples are more complex: the meta and object program fragments must accidentally have exactly the same syntax, and both provide type-correct interpretations. The following example illustrates class 4. The meta language is ASF+SDF, and the object language is Java. The overlapping language constructs are the fragment: "`[]`" and "=". For ASF+SDF, the "`[]`" is an

empty equation tag and "=" is the equation operator, while in Java "`[]`" is a part of an array declarator and "=" is the initializer part.

```
equations
[] int[] foo = int[]
[] foo = bar
```

or is it:

```
equations
[] int[] foo = int[][] foo = bar
```

The parser returns two alternative interpretations. The first has two equations, the second only one. By using suggestive layout, and printing the ASF+SDF symbols in boldface, we illustrate how the right-hand side of the first rule can be extended to be a two dimensional array declarator that is initialized by `bar`: the combination of "`"[]"`" overlapping with array declarators and "`"="`" overlapping with variable initializers leads to the ambiguity. Both interpretations are syntactically correct and type-correct. Note that the above example depends on a particular Java object grammar.

To solve this class of rare and hard to predict ambiguities we introduce a separate meta disambiguation phase. By applying this phase after type-checking we ensure that only type-correct alternatives are subject to this phase (Fig. 5.5). A number of implicit but obvious rules will provide a full separation between meta and object language. The rules are not universal. Each meta programming language may select different ones applied in different orders. Still, the design space can be limited to a number of choices:

Rule 1: Prefer declared meta identifiers over object identifiers, or vice versa.

Rule 2: Maximize or minimize the length of the path of injections and quasi-injections from the object syntax that starts from the `Term` non-terminal and ends with a production that is not an injection or a quasi-injection.

Rule 3: Maximize or minimize the number of meta language productions used in a derivation.

Rule 4: Propose explicit quoting to the user.

Rule 1 is a generalization of the variable preference rule (Section 5.1.2). All meta level identifiers, such as function names and variables are preferred. This rule may involve counting the number of declared meta identifiers in two alternatives and choosing the alternative with the least or most meta identifiers.

Rule 2 is needed only when the type context is not specific down to a first order type, but does impose some constraints. This can happen in systems with polymorphic functions. For example: a function type `f: a -> a -> b` maps two objects of any type `a` to another object of type `b`. Even though the function is parameterized by type, the first two arguments must be of the same type. Choosing the shortest path to derive an object of type `a` for both arguments is then a reasonable choice for most systems. The equations of ASF+SDF are also examples of polymorphic syntax with some constraints: the left-hand side must have an equal type to the right-hand side.

Rule 3 expresses that object language fragments should be either as short, or as long as possible. The more meta productions are used, the shorter object fragments become. This takes care of our earlier example involving the `"[]"` and `"="`.

If Rule 3 fails, Rule 4 provides the final fail-safe to all ambiguities introduced by merging the meta and object syntax.

Discussion The above rules should be tried in order. Rule 1 is a practical heuristic for meta programming without syntactic hedges, it will fire frequently. Rule 2 is needed when a unique type-context is not available. The other rules ensure full disambiguation in the rare cases where syntactic overlap coincides with type correct alternatives and Rule 1 does not apply. A warning message to the programmer in case Rule 3 fires is preferable since this case rarely occurs and is therefore unexpected.

After the type-checking and meta disambiguation phases all ambiguities introduced by the syntax merger have been identified. Only type-correct alternatives remain after type-checking. This is the main "quote inference" functionality. Then, the first two meta rules take care of further inference when necessary. The last two rules are fail-safes for degenerate cases.

Another design decision might be to drop the heuristic rules 1, 2 and 3 and always ask the user to explicitly disambiguate using quoting (rule 4). Using the parse forest representation the introduction of explicit quotes can be automated after the user has expressed her preference. The motivation for this design is that if heuristics are needed for disambiguation, it is probably also unclear to the programmer which interpretation is correct.

5.4 Experience

Parsing. This work has been applied to parsing ASF+SDF specifications. First the syntax of ASF+SDF was extended to make the meta language more complex: a generic expression language was added that can be arbitrarily nested with object language syntax. Before there were only meta variables in ASF+SDF, now we can nest meta language function calls at arbitrary locations into object language patterns, even with parametric polymorphism. Then a syntax merger was developed that generates the transitions, and priorities for filtering cycles. The generated parsers perform efficiently, while producing large parse forests.

Type checking. Both experience with post-parsing disambiguation filters in ASF+SDF (Chapter 4), and the efficient implementation of type-inference algorithms for languages as Haskell and ML suggests that our cycle removal and type-checking disambiguation phase can be implemented efficiently. Knowing the polynomial upper-bound for the size of parse forests, we have implemented a type-checker that applies the basic disambiguation rules for ASF+SDF. Since we do not have a syntax for defining *type parameters* yet, we did not investigate the use of parametric polymorphism.

Furthermore, in [52] we describe a related disambiguation architecture that also employs disambiguation by type checking. In this approach we show how such an

architecture can remove only the need for explicitly typed quotes, not the need for explicit quoting in general. In this work we apply the disambiguation by type checking design pattern to Java as a meta programming language. A Java type checker is used to disambiguate object language patterns. As reported, such an algorithm performs very well.

5.5 Conclusion

An architecture for parsing meta programs with concrete syntax was presented. By using implicit transitions syntactic hedges are not necessary. This might result in more readable meta programs, but we have not substantiated this claim. Instead we offer a "quote inference" algorithm, that allows a programmer to remove syntactic hedges if so desired.

The technical consequences of having implicit transitions, cycles and ambiguities, are solved in a three-tier architecture: syntax merging, parsing, and type-checking. The syntax merger adds priorities to let meaningless derivations be filtered from the parse table. Then the SGLR algorithm produces a forest of trees. Next, using type-inference techniques type incorrect trees are filtered. Finally, a small set of disambiguation rules takes care of making the final separation between meta and object language.

The resulting architecture shows strong separation of concerns. Also, it can be applied to meta programming languages with either simple or complex type systems, and can provide the user of such systems with clear error messages.

Future work entails a full implementation of the described architecture and integrating it in the programming environment of ASF+SDF. This will provide better error messages and will make the type-system of ASF+SDF easily extensible.

Part III

Rewriting source code

CHAPTER 6

Term Rewriting with Traversal Functions

Term rewriting is an appealing technique for performing program analysis and program transformation. Tree (term) traversal is frequently used but is not supported by standard term rewriting. We extend many-sorted, first-order term rewriting with Traversal Functions that automate tree traversal in a simple and type safe way. Traversal functions can be bottom-up or top-down traversals and can either traverse all nodes in a tree or can stop the traversal at a certain depth as soon as a matching node is found. They can either define sort preserving transformations or mappings to a fixed sort. We give small and somewhat larger examples of Traversal Functions and describe their operational semantics and implementation. An assessment of various applications and a discussion conclude the chapter. [1]

6.1 Introduction

6.1.1 Background

Program analysis and program transformation usually take the syntax tree of a program as starting point. Operations on this tree can be expressed in many ways, ranging from imperative or object-oriented programs, to attribute grammars and rewrite systems. One common problem that one encounters is how to express the *traversal* of the tree: visit all nodes of the tree once and extract information from some nodes or make changes to certain other nodes.

The kinds of nodes that may appear in a program's syntax tree are determined by the grammar of the language the program is written in. Typically, each rule in the grammar corresponds to a node category in the syntax tree. Real-life languages are described by grammars containing a few hundred up to over thousand grammar productions. This immediately reveals a hurdle for writing tree traversals: a naive recursive Traversal

[1] This chapter was published in Transactions on Software Engineering and Methodology (TOSEM) in 2003. An extended abstract appeared in the proceedings of the Workshop on Rewriting Strategies (WRS) 2002. Both papers are co-authored by Paul Klint and Mark van den Brand.

Function should consider many node categories and the size of its definition will grow accordingly. This becomes even more dramatic if we realize that the Traversal Function will only do some real work (apart from traversing) for very few node categories.

This problem asks for a form of automation that takes care of the tree traversal itself so that the human programmer can concentrate on the few node categories where real work is to be done. Stated differently, we are looking for a generic way of expressing tree traversals.

From previous experience [29, 34, 36, 101] we know that term rewriting is a convenient, scalable technology for expressing analysis, transformation, and renovation of individual programs and complete software systems. The main reasons for this are:

- Term rewriting provides implicit tree pattern matching that makes it easy to find patterns in program code.
- Programs can easily be manipulated and transformed via term rewriting.
- Term rewriting is rule-based, which makes it easy to combine sets of rules.
- Efficient implementations exist that can apply rewrite rules to millions of lines of code in a matter minutes.

In this chapter we aim at further enhancing term rewriting for the analysis and transformation of software systems and address the question how tree traversals can be added to the term rewriting paradigm.

One important requirement is to have a typed design of automated tree traversals, such that terms are always well-formed. Another requirement is to have simplicity of design and use. These are both important properties of many-sorted first-order term rewriting that we want to preserve.

6.1.2 Plan of the Paper

In the remainder of this introduction we will discuss general issues in tree traversal (Section 6.1.3), briefly recapitulate term rewriting (Section 6.1.4), discuss why Traversal Functions are necessary in term rewriting (Section 6.1.5), explain how term rewriting can be extended (Section 6.1.6), and discuss related work (Section 6.1.7).

In Section 6.2 we present Traversal Functions in ASF+SDF [14, 67] and give various examples. Some larger examples of Traversal Functions are presented in Section 6.3. The operational semantics of Traversal Functions is given in Section 6.4 and implementation issues are considered in Section 6.5. Section 6.6 describes the experience with Traversal Functions and Section 6.7 gives a discussion.

6.1.3 Issues in Tree Traversal

A simple tree traversal can have three possible goals:

(G1) Transforming the tree, e.g., replacing certain control structures that use `goto`'s into structured statements that use `while` statements.

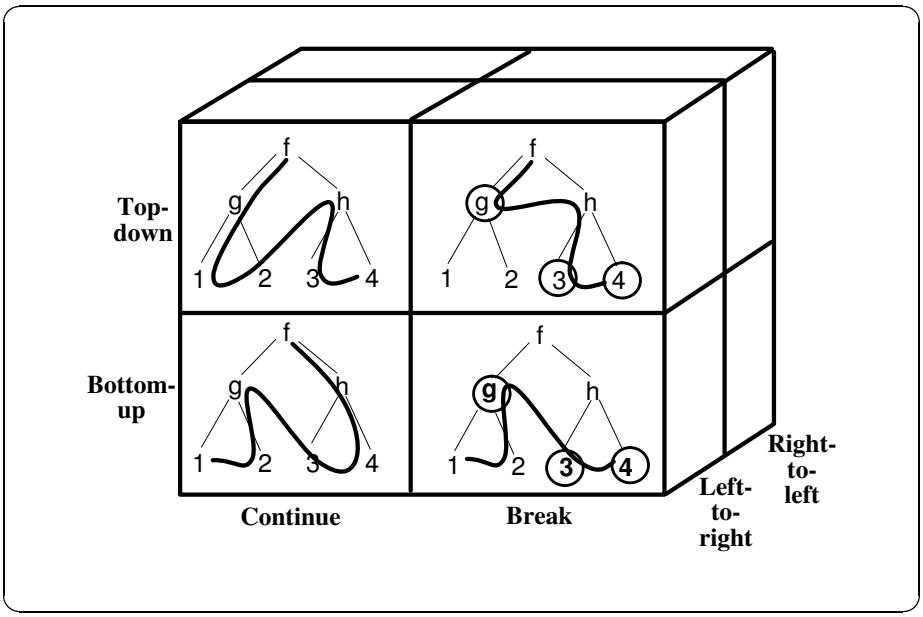

Figure 6.1: The "traversal cube": principal ways of traversing a tree.

(G2) Extracting information from the tree, e.g., counting all `goto` statements.

(G3) Extracting information from the tree and simultaneously transforming it, e.g., extracting declaration information and applying it to perform constant folding.

Of course, these simple tree traversals can be combined into more complex ones.

The goal of a traversal is achieved by visiting all tree nodes in a certain *visiting order* and applying a rewrite rule to each node once.

General properties of tree traversal are shown in the "traversal cube" in figure 6.1. On the first (vertical) axis, we distinguish the standard visiting orders *top-down* (order: root, sub-trees) and *bottom-up* (order: sub-trees, root). Note that for *binary* trees (as shown in the example) there is yet another way of visiting every node once called in-order[2] (order: one sub-tree, root, other sub-tree). In this chapter we target arbitrary tree structures and therefore do not further consider this special case.

On the second (horizontal) axis, we distinguish traversals that *break* the recursion at specific nodes and traversals that always *continue* until all nodes have been visited. In the right half of figure 6.1, these breaks occur at the nodes `g`, `3`, and `4`.

On the third (depth) axis, we distinguish the direction of the traversal: visiting nodes from left-to-right or from right-to-left.

The eight possibilities given in the traversal cube are obvious candidates for abstraction and automation. In this chapter we will focus on the front plane of the cube,

[2] In-order is called post-order in *The Art of Computer Programming, Volume 1* [106], nowadays post-order is used to indicate what is called end-order in that book.

i.e. left-to-right traversals since they are most prominently used in the application areas we are interested in. An extension to the complete cube is, however, straightforward.

During a tree traversal, a rewrite rule should be applied to some or all nodes to achieve the intended effect of the traversal. The type of the Traversal Function depends on the type of the input nodes, which can be one of the following:

- The nodes are untyped. This is the case in, for instance, Lisp or Prolog. Ease of manipulation is provided at the expense of type safety.
- The nodes are typed and the tree is homogeneous, i.e., all nodes have the same type. This is the case when, for instance, C or Java are used and nodes in the tree are represented by a single "tree-node" data type. As with untyped nodes, homogeneous trees are manipulated easily because every combination of nodes is well typed.
- The nodes are typed and the tree is heterogeneous, i.e., nodes may have different types. This is the case when, for instance, C or Java are used and a separate data type is introduced for representing each construct in a grammar (e.g., "declaration_node", "statement_node", "if_node" and so forth).

In this chapter we will focus on the traversal of typed, heterogeneous, trees. Various aspects of traversal functions will be discussed:

- What is the type of their result value?
- What is the type of their other arguments?
- Does the result of the Traversal Function depend only on the current node that is being visited or does it also use information stored in deeper nodes or even information from a global state?

Obviously, tree traversals are heavily influenced by the type system of the programming language in which they have to be expressed.

6.1.4 A Brief Recapitulation of Term Rewriting

A basic insight in term rewriting is important for understanding Traversal Functions. Therefore we give a brief recapitulation of innermost term rewriting. For a full account see [146].

A *term* is a prefix expression consisting of constants (e.g., `a` or `12`), variables (e.g., `X`) or function applications (e.g., `f(a, X, 12)`). For simplicity, we will view constants as nullary functions. A *closed term* (or *ground term*) is a term without variables. A *rewrite rule* is a pair of terms $T_1 \rightarrow T_2$. Both T_1 and T_2 may contain variables provided that each variable in T_2 also occurs in T_1. A term *matches* another term if it is structurally equal modulo occurrences of variables (e.g., `f(a, X)` matches `f(a, b)` and results in a *binding* where `X` is bound to `b`). If a variable occurs more than once in a term, a so-called non-left-linear pattern, the values matched by each occurrence are required to be equal. The bindings resulting from matching can be used for *substitution*, i.e., replace the variables in a term by the values they are bound to.

Algorithm 1 An interpreter for innermost rewriting.

```
function match(term, term) : bindings or NO-MATCH
function substitute(term, bindings) : term

function innermost(t: term, rules : list_of[rule]) : term
begin
  var children, children' : list-of[term];
  var child, reduct, t'   : term;
  var fn                  : function-symbol;

  decompose term t as fn(children);
  children' := nil;
  foreach child in children
  do children' := append(children', innermost(child, rules)) od;
  t' := compose term fn(children');
  reduct := reduce(t', rules);
  return if reduct = fail then t' else reduct fi
end

function reduce(t : term, rules : list-of[rule]) :  term
begin
  var r : rule;
  var left, right : term;

  foreach r in rules
  do decompose rule r as left -> right;
     var b : bindings;
     b := match(t, left);
     if b != NO-MATCH then return innermost(substitute(right, b), rules) fi
  od
  return fail
end
```

Given a ground term T and a set of rewrite rules, the purpose of a rewrite rule interpreter is to find a sub-term that can be reduced: the so-called *redex*. If sub-term R of T matches with the left-hand side of a rule $T_1 \rightarrow T_2$, the bindings resulting from this match can be substituted in T_2 yielding T_2'. R is then replaced in T by T_2' and the search for a new redex is continued. Rewriting stops when no new redex can be found and we say that the term is then in *normal form*.

In accordance with the tree traversal orders described earlier, different methods for selecting the redex may yield different results. In this chapter we limit our attention to leftmost innermost rewriting in which the redex is searched in a left-to-right, bottom-up fashion.

The operation of a rewrite rule interpreter is shown in more detail in Algorithm 1. The functions `match` and `substitute` are not further defined, but have a meaning as just sketched. We only show their signature. Terms can be *composed* from a top function symbol and a list of children, and they can be *decomposed* into their separate parts too. For example, if `fn` has as value the function-name `f`, and `children` has as value the list of terms `[a,b,c]`, then *compose term* `fn(children)` will yield the term `f(a,b,c)`. Decompose works in a similar fashion and also allows more structured

```
module Tree-syntax
imports Naturals
exports
  sorts TREE
  context-free syntax
    NAT            -> TREE
    f(TREE, TREE) -> TREE
    g(TREE, TREE) -> TREE
    h(TREE, TREE) -> TREE
  variables
    N[0-9]*        -> NAT
    T[0-9]*        -> TREE
```

Figure 6.2: SDF grammar for a simple tree language.

term patterns. For example, *decompose term* `t` *into* `fn(child, children)` will result in the assignments `fn := f; child := a, children := [b, c]`. Rules are *composed* from a left-hand side and a right-hand side. They can also be *decomposed* to obtain these distinct parts. The underlying term representation can be either typed or untyped. The *compose* and *decompose* functionality as well as the functions `match` and `substitute` have to take this aspect into account. We use an `append` function to append an element to the end of a list.

Observe how function `innermost` first reduces the children of the current term before attempting to reduce the term itself. This realizes a bottom-up traversal of the term. Also note that if the reduction of the term fails, it returns itself as result. The function `reduce` performs, if possible, one reduction step. It searches all rules for a matching left-hand side and, if found, the bindings resulting from the successful match are substituted in the corresponding right-hand side. This modified right-hand side is then further reduced with innermost rewriting. In Section 6.4 we will extend Algorithm 1 to cover Traversal Functions as well.

In the above presentation of term rewriting we have focused on the features that are essential for an understanding of Traversal Functions. Many other features such as, for instance, conditional rules with various forms of conditions (e.g., equality/inequality, matching conditions), list matching and the like are left undiscussed. In an actual implementation (Section 6.5) they have, of course, to be taken care of.

6.1.5 Why Traversal Functions in Term Rewriting?

Rewrite rules are very convenient to express transformations on trees and one may wonder why Traversal Functions are needed at all. We will clarify this by way of simple trees containing natural numbers. Figure 6.2 displays an SDF [87] grammar for a simple tree language. The leafs are natural numbers and the nodes are constructed with one of the binary constructors `f`, `g` or `h`. Note that numbers (sort `NAT`) are embedded in trees (sort `TREE`) due to the production `NAT -> TREE`. The grammar also defines variables over natural numbers (`N`, `N0`, `N1`, ...) and trees (`T`, `T0`, `T1`, ...). Transfor-

```
module Tree-trafo1
imports Tree-syntax
equations
[t1] f(T1, T2) = h(T1, T2)
```

Figure 6.3: Example equation `[t1]`.

```
module Tree-trafo2
imports Tree-syntax
equations
[t2] f(g(T1, T2), T3) = h(T1, h(T2, T3))
```

Figure 6.4: Example equation `[t2]`.

mations on these trees can now be defined easily. For instance, if we want to replace all occurrences of `f` by `h`, then the single equation `[t1]` shown in figure 6.3 suffices. Applying this rule to the term `f(f(g(1,2),3),4)` leads to a normal form in two steps (using innermost reduction):

```
f(f(g(1,2),3),4) -> f(h(g(1,2),3),4) -> h(h(g(1,2),3),4)
```

Similarly, if we want to replace all sub-trees of the form `f(g(T1, T2), T3)` by `h(T1, h(T2, T3))`, we can achieve this by the single rule `[t2]` shown in figure 6.4. If we apply this rule to `f(f(g(1,2),3),4)` we get a normal form in one step:

```
f(f(g(1,2),3),4) -> f(h(1,h(2,3)),4)
```

Note, how in both cases the standard (innermost) reduction order of the rewriting system takes care of the complete traversal of the term. This elegant approach has, however, three severe limitations:

- First, if we want to have the combined effect of rules `[t1]` and `[t2]`, we get unpredictable results, since the two rules interfere with each other: the combined rewrite system is said to be *non-confluent*. Applying the above two rules to our sample term `f(f(g(1,2),3),4)` may lead to either `h(h(g(1,2),3),4)` or `h(h(1,h(2,3)),4)` in two steps, depending on whether `[t1]` or `[t2]` is applied in the first reduction step. Observe, however, that an interpreter like the one shown in Algorithm 1 will always select one rule and produce a single result.
- The second problem is that rewrite rules cannot access any context information other than the term that matches the left-hand side of the rewrite rule. Especially for program transformation this is very restrictive.
- Thirdly, in ordinary (typed) term rewriting only type-preserving rewrite rules are allowed, i.e., the type of the left-hand side of a rewrite rule has to be equal to

the type of the right-hand side of that rule. Sub-terms can only be replaced by sub-terms of the same type, thus enforcing that the complete term remains well-typed. In this way, one cannot express non-type-preserving traversals such as the (abstract) interpretation or analysis of a term. In such cases, the original type (e.g., integer expressions of type `EXP`) has to be translated into values of another type (e.g., integers of type `INT`).

A common solution to the above three problems is to introduce new function symbols that eliminate the interference between rules. In our example, if we introduce the functions `trafo1` and `trafo2`, we can explicitly control the outcome of the combined transformation by the order in which we apply `trafo1` and `trafo2` to the initial term. By introducing extra function symbols, we also gain the ability to pass data around using extra parameters of these functions. Finally, the function symbols allow to express non-type-preserving transformations by explicitly typing the function to accept one type and yield another. This proposed change in specification style does not yield a semantically equivalent rewriting system in general. It is used as a practical style for specifications, for the above three reasons.

So by introducing new function symbols, three limitations of rewrite rules are solved. The main down side of this approach is that we loose the built-in facility of innermost rewriting to traverse the input term without an explicit effort of the programmer. Extra rewrite rules are needed to define the traversal of `trafo1` and `trafo2` over the input term, as shown in figure 6.5. Observe that equations `[1]` and `[5]` in the figure correspond to the original equations `[t1]` and `[t2]`, respectively. The other equations are just needed to define the tree traversal. Defining the traversal rules requires explicit knowledge of *all* productions in the grammar (in this case the definitions of `f`, `g` and `h`). In this example, the number of rules per function is directly related to the size of the Tree language. For large grammars this is clearly undesirable.

6.1.6 Extending Term Rewriting with Traversal Functions

We take a many-sorted, first-order, term rewriting language as our point of departure. Suppose we want to traverse syntax trees of programs written in a language *L*, where *L* is described by a grammar consisting of *n* grammar rules.

A typical tree traversal will then be described by *m* (*m* usually less than *n*) rewrite rules, covering all possible constructors that may be encountered during a traversal of the syntax tree. The value of *m* largely depends on the structure of the grammar and the specific traversal problem. Typically, a significant subset of all constructors needs to be traversed to get to the point of interest, resulting in tens to hundreds of rules that have to be written for a given large grammar and some specific transformation or analysis.

The question now is: how can we avoid writing these *m* rewrite rules? There are several general approaches to this problem.

Higher-order term rewriting. One solution is the use of higher-order term rewriting [90, 73, 85]. This allows writing patterns in which the context of a certain language construct can be captured by a (higher-order) variable thus eliminating the need to

```
module Tree-trafo12
imports Tree-syntax
exports
context-free syntax
  trafo1(TREE)         -> TREE
  trafo2(TREE)         -> TREE
equations
[0] trafo1(N)          = N
[1] trafo1(f(T1, T2)) = h(trafo1(T1), trafo1(T2))
[2] trafo1(g(T1, T2)) = g(trafo1(T1), trafo1(T2))
[3] trafo1(h(T1, T2)) = h(trafo1(T1), trafo1(T2))

[4] trafo2(N)          = N
[5] trafo2(f(g(T1,T2),T3)) = h(trafo2(T1),
                               h(trafo2(T2), trafo2(T3)))
[6] trafo2(g(T1, T2)) = g(trafo2(T1), trafo2(T2))
[7] trafo2(h(T1, T2)) = h(trafo2(T1), trafo2(T2))
```

Figure 6.5: Definition of `trafo1` and `trafo2`.

explicitly handle the constructs that occur in that context. We refer to [86] for a simple example of higher-order term rewriting.

Higher-order term rewriting is a very powerful mechanism, which can be used to avoid expressing entire tree traversals. It introduces, however, complex semantics and implementation issues. It does not solve the non-confluence problems discussed earlier (see Section 6.1.5). Another observation is that the traversal is done during matching, so for every match the sub-terms might be traversed. This might be very expensive.

Generic traversal or strategy primitives One can extend the rewriting language with a set of generic traversal or strategy primitives as basic operators that enable the formulation of arbitrary tree traversals. Such primitives could for instance be the traversal of one, some or all sub-trees of a node, or the sequential composition, choice or repetition of traversals. They can be used to selectively apply a rewrite rule at a location in the term. Generic traversal primitives separate the application of the rewrite rule from the traversal of the tree as advocated in *strategic programming*. See, for instance, [160] for a survey of strategic programming in the area of program transformations.

The expressivity provided by generic traversals is hard to handle by conventional typing systems [158, 109]. The reason for this is that the type of a traversal primitive is completely independent of the structures that it can traverse. In [109] a proposal is made for a typing system for generic traversal primitives which we will further discuss in Section 6.1.7.

Having types is relevant for static type checking, program documentation, and program comprehension. It is also beneficial for efficient implementation and optimization. In ordinary (typed) term rewriting only type-preserving rewrite rules are allowed, i.e., the type of the left-hand side of a rewrite rule has to be equal to the type of the

	Untyped	**Typed**
Strategy primitives	Stratego [159]	ELAN [22]
Built-in strategies	Renovation Factories [49]	Traversal Functions, TXL [59]

Table 6.1: Classification of traversal approaches.

right-hand side of that rule. Sub-terms can only be replaced by sub-terms of the same type, thus enforcing that the complete term remains well-typed. Type-checking a first-order many-sorted term rewriting system simply boils down to checking if both sides of every rewrite rule yield the same type and checking if both sides are well-formed with respect to the signature.

Traversal functions. Our approach is to allow functions to traverse a tree automatically, according to a set of built-in traversal primitives. In our terminology, such functions are called *Traversal Functions*. They solve the problem of the extra rules needed for term traversal without loosing the practical abilities of functions to carry data around and having non-sort-preserving transformations.

By extending ordinary term rewriting with Traversal Functions, the type-system can remain the same. One can provide primitives that allow type-preserving and even a class of non-type-preserving traversals in a type-safe manner without even changing the type-checker of the language.

6.1.7 Related Work

Directly Related Work

We classify directly related approaches in figure 6.1 and discuss them below.

ELAN [22] is a language of many-sorted, first-order, rewrite rules extended with a strategy language that controls the application of individual rewrite rules. Its strategy primitives (e.g., "don't know choice", "don't care choice") allow formulating non-deterministic computations. Currently, ELAN does not support generic tree traversals since they are not easily fitted in with ELAN's type system.

Stratego [159] is an untyped term rewriting language that provides user-defined strategies. Among its strategy primitives are rewrite rules and several generic strategy operators (such as, e.g., sequential composition, choice, and repetition) that allow the definition of any tree traversal, such as top-down and bottom-up, in an abstract manner. Therefore, tree traversals are first class objects that can be reused separately from rewrite rules. Stratego provides a library with all kinds of named traversal strategies such as, for instance, `bottomup(s)`, `topdown(s)` and `innermost(s)`.

Transformation Factories [49] are an approach in which ASF+SDF rewrite rules are generated from syntax definitions. After the generation phase, the user instantiates an actual transformation by providing the name of the transformation and by updating default traversal behavior. Note that the generated rewrite rules are well-typed, but unsafe general types have to be used to obtain reusability of the generated rewrite rules.

Transformation Factories provide two kinds of traversals: transformers and analyzers. A transformer transforms the node it visits. An analyzer is the combination of a traversal, a combination function and a default value. The generated traversal function reduces each node to the default value, unless the user overrides it. The combination function combines the results in an innermost manner. The simulation of higher-order behavior again leads to very general types.

TXL [59] TXL is a typed language for transformational programming [59]. Like ASF+SDF it permits the definition of arbitrary grammars as well as rewrite rules to transform parsed programs. Although TXL is based on a form of term rewriting, its terminology and notation deviate from standard term rewriting parlance. TXL has been used in many renovation projects.

Discussion

Traversal functions emerged from our experience in writing program transformations for real-life languages in ASF+SDF. Both Stratego and Transformation Factories offer solutions to remedy the problems that we encountered.

Stratego extends term rewriting with traversal strategy combinators and user-defined strategies. We are more conservative and extend first-order term rewriting only with a fixed set of traversal primitives. One contribution of Traversal Functions is that they provide a simple *type-safe* approach for tree traversals in first-order specifications. The result is simple, can be statically type-checked in a trivial manner and can be implemented efficiently. On the down-side, our approach does not allow adding new traversal orders: they have to be simulated with the given, built-in, traversal orders. See [100] for a further discussion of the relative merits of these two approaches.

Recently, in [109] another type system for tree traversals was proposed. It is based on traversal combinators as found in Stratego. While this typing system is attractive in many ways, it is more complicated than our approach. Two generic types are added to a first-order type system: type-preserving (`TP`) and type-unifying (`TU(`τ`)`) strategies. To mediate between these generic types and normal types an extra combinator is offered that combines both a type-guard and a type lifting operator. Extending the type system is not needed in our Traversal Function approach, because the tree traversal is joined with the functional effect in a single Traversal Function. This allows the interpreter or compiler to deal with type-safe traversal without user intervention. As is the case with Traversal Functions, in [109] traversal types are divided into type-preserving effects and mappings to a single type. The tupled combination is not offered.

Compared to Transformation Factories (which most directly inspired our Traversal Functions), we provide a slightly different set of Traversal Functions and reduce the notational overhead. More important is that we provide a fully typed approach. At the level of the implementation, we do not generate ASF+SDF rules, but we have

incorporated Traversal Functions in the standard interpreter and compiler of ASF+SDF. As a result, execution is more efficient and specifications are more readable, since users are not confronted with generated rewrite rules.

Although developed completely independently, our approach has much in common with TXL, which also provides type-safe term traversal. TXL rules always apply a pre-order search over a term looking for a given pattern. For matching sub-terms a replacement is performed. TXL rules are thus comparable with our top-down transformers. A difference is that Traversal Functions perform only one pass over the term and do not visit already transformed subtrees. TXL rules, however, also visit the transformed subtrees. In some cases, e.g., renaming all variables in a program, special measures are needed to avoid undesired, repeated, transformations. In TXL jargon, Traversal Functions are all *one-pass* rules. Although TXL does not support accumulators, it has a notion of global variables that can be used to collect information during a traversal. A useful TXL feature that we do not support is the ability to skip sub-terms of certain types during the traversal.

Other Related Work

Apart from the directly related work already mentioned, we briefly mention related work in *functional languages*, *object-oriented languages* and *attribute grammars*.

Functional languages. The prototypical Traversal Function in the functional setting are the functions `map`, `fold` and relatives. `map` takes a tree and a function as argument and applies the function to each node of the tree. However, problems arise as soon as heterogeneous trees have to be traversed. One solution to this problem are fold algebras as described in [113]: based on a language definition Traversal Functions are generated in Haskell. A tool generates generic folding over algebraic types. The folds can be updated by the user. Another way of introducing generic traversals in a functional setting is described in [112].

Object-oriented languages. The traversal of arbitrary data structures is captured by the *visitor design pattern* described in [75]. Typically, a fixed traversal order is provided as framework with default behavior for each node kind. This default behavior can be overruled for each node kind. An implementation of the visitor pattern is JJForester [108]: a tool that generates Java class structures from SDF language definitions. The generated classes implement generic tree traversals that can be overridden by the user. The technique is related to generating traversals from language definitions as in Transformation Factories, but is tailored to and profits from the object-oriented programming paradigm. In [163] this approach is further generalized to traversal combinators.

Attribute grammars. The approaches described so far provide an operational view on tree traversals. Attribute grammars [4] provide a declarative view: they extend a syntax tree with *attributes* and *attribute equations* that define relations between attribute values. Attributes get their values by solving the attribute equations; this is

achieved by one or more traversals of the tree. For attribute grammars tree traversal is an issue for the implementation and not for the user. Attribute grammars are convenient for expressing analysis on a tree but they have the limitation that tree transformations cannot be easily expressed. However, higher-order attribute grammars [164] remedy this limitation to a certain extent. A new tree can be constructed in one of the attributes which can then be passed on as an ordinary tree to the next higher-order attribute function.

Combining attribute grammars with object orientation. JastAdd [84] is recent work in the field of combining reference attribute grammars [83] with visitors and class weaving. The attribute values in reference attributes may be references to other nodes in the tree. The implicit tree traversal mechanism for attribute evaluation is combined with the explicit traversal via visitors. This is convenient for analysis purposes but it does not solve the problems posed by program transformations.

6.2 Traversal Functions in ASF+SDF

We want to automate tree traversal in many-sorted, first-order term rewriting. We present Traversal Functions in the context of the language ASF+SDF [14, 67], but our approach can be applied to any term rewriting language. No prior knowledge of ASF+SDF is required and we will explain the language when the need arises.

ASF+SDF uses context-free syntax for defining the signature of terms. As a result, terms can be written in arbitrary user-defined notation. This means that functions can have free notation (e.g., `move ... to ...` rather than `move(..., ...)`) and that the complete text of programs can be represented as well. The context-free syntax is defined in SDF [3]. Terms are used in rewrite rules defined in ASF [4]. For the purpose of this chapter, the following features of ASF are relevant:

- Many-sorted (typed) terms.
- Unconditional and conditional rules. Conditions are comparisons between two terms which come in three flavors: equality between terms, inequality between terms, and so-called assignment conditions that introduce new variables. In the first two flavors, no new variables may be introduced on either side. In the last form only one side of the condition may contain new variables, which are bound while matching the pattern with the other side of the condition.
- Default rules that are tried only if all other rules fail.
- Terms are normalized by leftmost innermost reduction.

The idea of Traversal Functions is as follows. The programmer defines functions as usual by providing a signature and defining rewrite rules. The signature of a Traversal Function has to be defined as well. This is an ordinary declaration but it is explicitly

[3] Syntax Definition Formalism.
[4] Algebraic Specification Formalism.

labeled with the attribute `traversal`. We call such a labeled function a Traversal Function since from the user's perspective it automatically traverses a term: the rewrite rules for term traversal do not have to be specified anymore since they are provided automatically by the `traversal` attribute. The specification writer only has to give rewrite rules for the nodes that the Traversal Function will actually visit.

The rewrite rules provided by the `traversal` attribute thus define the *traversal* behavior while rewrite rules provided by the user define the *visit* behavior for nodes. If during innermost rewriting a Traversal Function appears as outermost function symbol of a redex, then that function will first be used to traverse the redex before further reductions occur.

Conceptually, a Traversal Function is a shorthand for a possibly large set of rewrite rules. For every Traversal Function a set of rewrite rules can be calculated that implements both the traversal and the actual rewriting of sub-terms. Expanding a Traversal Function to this set of rewrite rules is a possible way of defining the *semantics* of Traversal Functions, which we do not further pursue here (but see [37]).

We continue our discussion in Section 6.1.6 on how to type generic traversals. The question is what built-in traversals we can provide in our fully typed setting. We offer three types of Traversal Functions (Section 6.2.1) and two types of visiting strategies (Section 6.2.2) which we now discuss in order. In Section 6.2.3 we present examples of Traversal Functions. The merits and limitations of this approach are discussed in Section 6.7.

6.2.1 Kinds of Traversal Functions

We distinguish three kinds of Traversal Functions, defined as follows.

Transformer A sort-preserving transformation, declared as:
$f(S_1,\ldots,S_n) \rightarrow S_1$ {`traversal(trafo)`}

Accumulator A mapping to a single type, declared as:
$f(S_1,S_2,\ldots,S_n) \rightarrow S_2$ {`traversal(accu)`}

Accumulating transformer A sort preserving transformation that accumulates information simultaneously, declared as:
$f(S_1,S_2,\ldots,S_n) \rightarrow < S_1,S_2 >$ {`traversal(accu, trafo)`}

A Transformer will traverse its first argument. Possible extra arguments may contain additional data that can be used (but not modified) during the traversal. Because a transformer always returns the same sort, it is type-safe. A transformer is used to transform a tree and implements goal (G1) discussed in Section 6.1.3.

An Accumulator will traverse its first argument, while the second argument keeps the accumulated value. After each application of an accumulator, the accumulated argument is updated. The next application of the accumulator, possibly somewhere else in the term, will use the *new* value of the accumulated argument. In other words, the accumulator acts as a global, modifiable state during the traversal.

An accumulator function never changes the tree, it only changes its accumulated argument. Furthermore, the type of the second argument has to be equal to the result type. The end-result of an accumulator is the value of the accumulated argument. By these restrictions, an accumulator is also type-safe for every instantiation.

An accumulator is meant to be used to extract information from a tree and implements goal (G2) discussed in Section 6.1.3.

An Accumulating Transformer is a sort preserving transformation that accumulates information while traversing its first argument. The second argument maintains the accumulated value. The return value of an accumulating transformer is a tuple consisting of the transformed first argument and the accumulated value.

An accumulating transformer is used to simultaneously extract information from a tree and transform it. It implements goal (G3) discussed in Section 6.1.3.

Transformers, accumulators, and accumulating transformers may be overloaded to obtain visitors for heterogeneous trees. Their optional extra arguments can carry information down and their defining rewrite rules can extract information from their children by using conditions. So we can express analysis and transformation using non-local information rather easily.

6.2.2 Visiting Strategies

Having these three types of traversals, they must be provided with visiting strategies (recall figure 6.1). Visiting strategies determine the order of traversal. We provide the following two strategies for each type of traversal:

Bottom-up First recur down to the children, then try to visit the current node. The annotation `bottom-up` selects this behavior.

Top-down First try to visit the current node and then traverse to the children. The annotation `top-down` selects this behavior.

Without an extra attribute, these strategies define traversals that visit all nodes in a tree. We add two attributes that select what should happen after a successful visit.

Break Stop visiting nodes on the current branch after a successful visit. The corresponding annotation is `break`.

Continue Continue the traversal after a successful visit. The corresponding annotation is `continue`.

A transformer with a `bottom-up` strategy resembles standard innermost rewriting; it is sort preserving and bottom-up. It is as if a small rewriting system is defined within the context of a transformer function. The difference is that a transformer function inflicts one reduction on a node, while innermost reduction normalizes a node completely.

To be able to `break` a traversal is a powerful feature. For example, it allows the user to continue the traversal under certain conditions.

```
module Tree-trafo12-trav
imports Tree-syntax
exports
context-free syntax
  trafo1(TREE) -> TREE {traversal(trafo,top-down,continue)}
  trafo2(TREE) -> TREE {traversal(trafo,top-down,continue)}
equations

[tr1'] trafo1(f(T1, T2))      = h(T1,T2)
[tr2'] trafo2(f(g(T1,T2),T3)) = h(T1,h(T2,T3))
```

input

```
trafo1(
  trafo2(f(f(g(1,2),3),4)))
```

output

```
h(h(1,h(2,3)),4)
```

Figure 6.6: `trafo1` and `trafo2` from figure 6.5 now using `top-down` Traversal Functions.

```
module Tree-inc
imports Tree-syntax
exports
context-free syntax
  inc(TREE) -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] inc(N) = N + 1
```

input

```
inc( f( g( f(1,2), 3 ),
        g( g(4,5), 6 )) )
```

output

```
  f( g( f(2,3), 4 ),
     g( g(5,6), 7 ))
```

Figure 6.7: Transformer `inc` increments each number in a tree.

6.2.3 Examples of Transformers

In the following subsections, we give some trivial examples of transformers, accumulators, and accumulating transformers. All examples use the tree language introduced earlier in figure 6.2. In Section 6.3 we show some more elaborate examples.

The trafo example from the introduction revised

Recall the definition of the transformations `trafo1` and `trafo2` in the introduction (figure 6.5). They looked clumsy and cluttered the intention of the transformation completely. Figure 6.6 shows how to express the same transformations using two Traversal Functions.

Observe how these two rules resemble the original rewrite rules. There is, however, one significant difference: these rules can only be used when the corresponding function is actually applied to a term.

```
module Tree-incp
imports Tree-syntax
exports
context-free syntax
  incp(TREE, NAT) -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] incp(N1, N2) = N1 + N2
```

input

```
incp( f( g( f(1,2), 3 ),
         g( g(4,5), 6 )),
      7)
```

output

```
f( g( f( 8, 9), 10 ),
   g( g(11,12), 13 ))
```

Figure 6.8: Transformer `incp` increments each number in a tree with a given value.

```
module Tree-frepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   frepl(TREE)    -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] frepl(g(T1, T2)) = i(T1, T2)
```

input

```
frepl( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )) )
```

output

```
f( i( f(1,2), 3 ),
   i( i(4,5), 6 ))
```

Figure 6.9: Transformer `frepl` replaces all occurrences of `g` by `i`.

Increment the numbers in a tree

The specification in figure 6.7 shows the transformer `inc`. Its purpose is to increment all numbers that occur in a tree. To better understand this example, we follow the traversal and rewrite steps when applying `inc` to the tree `f(g(1,2),3)`:

```
inc(f(g(1,2),3))  ->
f(g(inc(1),2),3)  ->
f(g(2,inc(2)),3)  ->
f(inc(g(2,3)),3)  ->
f(g(2,3),inc(3))  ->
inc(f(g(2,3),4))  ->
f(g(2,3),4)
```

We start by the application of inc to the outermost node, then each node is visited in a left-to-right bottom-up fashion. If no rewrite rule is activated, the identity transformation is applied. So, in this example only naturals are transformed and the other nodes are left unchanged.

```
module Tree-frepl2
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE) -> TREE
   frepl2(TREE)  -> TREE {traversal(trafo, top-down, continue)}
equations
[1] frepl2(g(T1, T2)) = i(T1, T2)
```

input

```
frepl2( f( g( f(1,2), 3 ),
           g( g(4,5), 6 )) )
```

output

```
  f( i( f(1,2), 3 ),
     i( i(4,5), 6 ))
```

Figure 6.10: Transformer `frepl2` replaces all occurrences of `g` by `i`.

```
module Tree-srepl
imports Tree-syntax
exports
context-free syntax
  i(TREE, TREE)  -> TREE
  srepl(TREE)    -> TREE {traversal(trafo, top-down, break)}
equations
[1] srepl(g(T1, T2)) = i(T1, T2)
```

input

```
srepl( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )) )
```

output

```
  f( i( f(1,2), 3 ),
     i( g(4,5), 6 ))
```

Figure 6.11: Transformer `srepl` replaces shallow occurrences of `g` by `i`.

Increment the numbers in a tree (with parameter)

The specification in figure 6.8 shows the transformer `incp`. Its purpose is to increment all numbers that occur in a tree with a given parameter value. Observe that the *first* argument of `incp` is traversed and that the second argument is a value that is carried along during the traversal. If we follow the traversal and rewrite steps for `incp(f(g(1,2),3), 7)`, we get:

```
     incp(f(g(1,2),3),7)  ->
     f(g(incp(1,7),2),3)  ->
     f(g(8,incp(2,7)),3)  ->
     f(incp(g(8,9),7),3)  ->
     f(g(8,9),incp(3,7))  ->
     incp(f(g(8,9),10),7) ->
     f(g(8,9),10)
```

```
module Tree-drepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)  -> TREE
   drepl(TREE)    -> TREE {traversal(trafo, bottom-up, break)}
equations
[1] drepl(g(T1, T2)) = i(T1, T2)
```

input

```
drepl( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )) )
```

output

```
  f( i( f(1,2), 3 ),
     g( i(4,5), 6 ))
```

Figure 6.12: Transformer `drepl` replaces deep occurrences of `g` by `i`.

Replace function symbols

A common problem in tree manipulation is the replacement of function symbols. In the context of our tree language we want to replace occurrences of symbol `g` by a new symbol `i`. Replacement can be defined in many flavors. Here we only show three of them: full replacement that replaces all occurrences of `g`, shallow replacement that only replaces occurrences of `g` that are closest to the root of the tree, and deep replacement that only replaces occurrences that are closest to the leafs of the tree.

Full replacement is defined in figure 6.9. We specified a `bottom-up` traversal that continues traversing after a reduction. This will ensure that all nodes in the tree will be visited. Note that in this case we could also have used a top-down strategy and get the same result as is shown in figure 6.10.

Shallow replacement is defined in figure 6.11. In this case, traversal stops at each outermost occurrence of `g` because `break` was given as an attribute. In this case, the top-down strategy is essential. Observe that a top-down traversal with the `break` attribute applies the Traversal Function at an applicable outermost node and does not visit the sub-trees of that node. However, the right-hand side of a defining equation of the Traversal Function may contain recursive applications of the Traversal Function itself! In this way, one can traverse certain sub-trees recursively while avoiding others explicitly.

We use the combination of a bottom-up strategy with the `break` attribute to define deep replacement as shown in figure 6.12. As soon as the rewrite rule applies to a certain node, the traversal visits no more nodes on the path from the reduced node to the root. In this case, the bottom-up strategy is essential.

6.2.4 Examples of Accumulators

So far, we have only shown examples of transformers. In this section we will give two examples of accumulators.

```
module Tree-sum
imports Tree-syntax
exports
context-free syntax
   sum(TREE, NAT) -> NAT {traversal(accu,bottom-up,continue)}
equations
[1] sum(N1, N2) = N1 + N2
```

input

```
sum( f( g( f(1,2), 3 ),
        g( g(4,5), 6 )),
     0)
```

output

```
21
```

Figure 6.13: Accumulator `sum` computes the sum of all numbers in a tree.

```
module Tree-cnt
imports Tree-syntax
exports
context-free syntax
  cnt(TREE, NAT) -> NAT {traversal(accu,bottom-up,continue)}
equations
[1] cnt(T, N) = N + 1
```

input

```
cnt( f( g( f(1,2), 3 ),
        g( g(4,5), 6 )),
     0)
```

output

```
11
```

Figure 6.14: Accumulator `cnt` counts the nodes in a tree.

Add the numbers in a tree

The first problem we want to solve is computing the sum of all numbers that occur in a tree. The accumulator `sum` in figure 6.13 solves this problem. Note that in equation `[1]` variable `N1` represents the current node (a number), while variable `N2` represents the sum that has been accumulated so far (also a number).

Count the nodes in a tree

The second problem is to count the number of nodes that occur in a tree. The accumulator `cnt` shown in figure 6.14 does the job.

6.2.5 Examples of Accumulating Transformers

We conclude our series of examples with one example of an accumulating transformer.

```
module Tree-pos
imports Tree-syntax
exports
context-free syntax
  pos(TREE, NAT) -> <TREE , NAT>
                    {traversal(accu, trafo,bottom-up,continue)}
equations
[1] pos(N1, N2) = <N1 * N2, N2 + 1>
```

input

```
pos( f( g( f(1,2), 3 ),
        g( g(4,5), 6 )),
     0)
```

output

```
<f( g( f(0,2), 6 ),
    g( g(12,20), 30 )),
 6>
```

Figure 6.15: Accumulating transformer `pos` multiplies numbers by their tree position.

Multiply by position in tree

Our last problem is to determine the position of each number in a top-down traversal of the tree and to multiply each number by its position. This is achieved by the accumulating transformer `pos` shown in figure 6.15. The general idea is to accumulate the position of each number during the traversal and to use it as a multiplier to transform numeric nodes.

6.3 Larger Examples

Now we give some less trivial applications of Traversal Functions. They all use the small imperative language PICO whose syntax is shown in figure 6.16. The toy language PICO was originally introduced in [14] and has been used as running example since then. A PICO program consists of declarations followed by statements. Variables should be declared before use and can have two types: natural number and string. There are three kinds of statements: assignment, if-statement and while-statement. In an assignment, the types of the left-hand side and the right-hand side should be equal. In if-statement and while-statement the condition should be of type natural. The arguments of the numeric operators + and − are natural. Both arguments of the string-valued operator are strings.

6.3.1 Type-checking

The example in figure 6.17 defines a type-checker for PICO in a style described in [86]. The general idea is to reduce type-correct programs to the empty program and to reduce programs containing type errors to a program that only contains the erroneous statements. This is achieved by using the information from declarations of variables to replace all variable occurrences by their declared type and by replacing all constants by their implicit type. After that, all type-correct statements are removed from the program. As a result, only type-correct programs are normalized to the empty program.

```
module Pico-syntax
imports Pico-whitespace
exports
  sorts PROGRAM DECLS ID-TYPE ID DECLS STAT STATS EXP
  sorts NAT-CON STR-CON
  lexical syntax
    [a-z] [a-z0-9]*                          -> ID
    [0-9]+                                   -> NAT-CON
    [\"] ~[\"]* [\"]                         -> STR-CON
  context-free syntax
    "begin" DECLS STATS "end"                -> PROGRAM
    "declare" ID-TYPES ";"                   -> DECLS
    ID ":" TYPE                              -> ID-TYPE
    "natural" | "string"                     -> TYPE
    {ID-TYPE ","}*                           -> ID-TYPES

    ID ":=" EXP                              -> STAT
    "if" EXP "then" STATS "else" STATS "fi" -> STAT
    "while" EXP "do" STATS "od"              -> STAT
    {STAT ";"}*                              -> STATS

    ID                                       -> EXP
    NAT-CON                                  -> EXP
    STR-CON                                  -> EXP
    EXP "+" EXP                              -> EXP {left}
    EXP "-" EXP                              -> EXP {left}
    EXP "||" EXP                             -> EXP {left}
    "(" EXP ")"                              -> EXP {bracket}
  context-free priorities
    EXP "||" EXP -> EXP >
    EXP "-" EXP  -> EXP >
    EXP "+" EXP  -> EXP
```

Figure 6.16: SDF grammar for the small imperative language PICO.

```
module Pico-typecheck
imports Pico-syntax
exports
context-free syntax
 type(TYPE)               -> ID
 replace(STATS, ID-TYPE) -> STATS
                            {traversal(trafo,bottom-up,break)}
 replace(EXP  , ID-TYPE) -> STATS
                            {traversal(trafo,bottom-up,break)}
equations
[0] begin declare Id-type, Decl*; Stat* end =
      begin declare Decl*; replace(Stat*, Id-type) end

[1] replace(Id     , Id : Type) = type(Type)
[2] replace(Nat-con, Id : Type) = type(natural)
[3] replace(Str-con, Id : Type) = type(string)

[4] type(string) || type(string)  = type(string)
[5] type(natural) + type(natural) = type(natural)
[6] type(natural) - type(natural) = type(natural)

[7] Stat*1;
      if type(natural) then Stat*2 else Stat*3 fi ;
    Stat*4
    = Stat*1; Stat*2; Stat*3; Stat*4

[8] Stat*1; while type(natural) do Stat*2 od; Stat*3
    = Stat*1; Stat*2; Stat*3

[9] Stat*1; type(Type) := type(Type); Stat*2
    = Stat*1; Stat*2
```

input

```
begin declare x : natural,
              s : string;
      x := 10; s := "abc";
      if x then x := x + 1
           else s := x + 2
      fi;
      y := x + 2;
end
```

output

```
begin
  declare;
  type(string) :=
         type(natural);
end
```

Figure 6.17: A type-checker for PICO.

This approach is interesting from the perspective of error reporting when rewriting is augmented with *origin tracking*, a technique that links back sub-terms of the normal form to sub-terms of the initial term [70]. In this way, the residuals of the type-incorrect statements in the normal form can be traced back to their source. See [147] for applications of this and similar techniques.

The example in figure 6.17 works as follows. First, it is necessary to accommodate the replacement of variables by their type, in other words, we want to replace `x := y` by `type(natural) := type(natural)`, assuming that `x` and `y` have been declared as `natural`. This is achieved by extending the syntax of PICO with the context-free syntax rule

```
type(TYPE) -> ID
```

The actual replacement of variables by their declared type is done by the transformer `replace`. It has to be declared for all sorts for which equations for `replace` are defined, in this case `STATS` and `EXP`. It is a bottom-up, breaking, transformer. The second argument of `replace` is an (identifier, type) pair as it appears in a variable declaration.

Note that for more complex languages a bottom-up breaking transformer might not be sufficient. For example, when dealing with *nested scopes* it is imperative that the type-environment can be updated before going into a new scope. A top-down breaking transformer is used in such a case which stops at the entrance of a new scope and explicitly recurs into the scope after updating the type-environment.

In equation `[0]` a program containing a non-empty declaration section is replaced by a new program with one declaration less. In the statements all occurrences of the variable that was declared in the removed declaration are replaced by its declared type. `replace` is specified in equations `[1]`, `[2]` and `[3]`. It simply replaces identifiers, natural constants and string constants by their type.

Next, all type correct expressions are simplified (equations `[4]`, `[5]` and `[6]`). Finally, type-correct statements are removed from the program (equations `[7]`, `[8]` and `[9]`). As a result, a type correct program will reduce to the empty program and a type incorrect program will reduce to a simplified program that precisely contains the incorrect statements. The example that is also given in figure 6.17 shows how the incorrect statement `s := x + 2` (both sides of an assignment should have the same type) is reduced to `type(string) := type(natural)`.

The traversal order could be both top-down and bottom-up, since `replace` only matches leafs in `[1]`, `[2]` and `[3]`. However, `bottom-up` and `break` make this traversal more efficient because once a leaf has been visited none of its ancestors is visited anymore. This example shows that Traversal Functions can be used for this style of type-checking and that they make this approach feasible for much larger languages.

Equations `[7]` through `[9]` use associative matching (called list matching in ASF+SDF) to concisely express operations on lists of statements. For instance, in `[8]`, the list variables `Stat*1` and `Stat*3` represent the statements surrounding a while statement and `Stat*2` represents the list of statements in the body of the while statement. On the right-hand side of the equation these three lists of statements are concatenated thus effectively merging the body of the while statement with its surroundings.

```
module Pico-usage-inference
imports Pico-syntax
exports
sorts SET SETS
context-free syntax
 "{" EXP* "}"              -> SET
 "[" SET* "]"              -> SETS
 infer-use(PROGRAM,SETS) -> SETS
                              {traversal(accu,top-down,break)}
 infer-use(STAT   ,SETS) -> SETS
                              {traversal(accu,top-down,break)}
 infer-use(EXP    ,SETS) -> SETS
                              {traversal(accu,top-down,break)}
variables
 "Set"[0-9]*  -> SET
 "Set*"[0-9]* -> SET*
 "Exp*"[0-9]* -> EXP*
equations
[0] infer-use(Id := Exp, [ Set* ] ) = [ { Id Exp } Set* ]
[1] infer-use(Exp      , [ Set* ] ) = [ { Exp }    Set* ]

[2] { Exp*1 Exp Exp*2 Exp Exp*3 }   = { Exp*1 Exp Exp*2 Exp*3 }
[3] { Exp*1 Exp1 + Exp2 Exp*2 }     = { Exp*1 Exp1 Exp2 Exp*2 }
[4] { Exp*1 Exp1 - Exp2 Exp*2 }     = { Exp*1 Exp1 Exp2 Exp*2 }
[5] { Exp*1 Exp1 || Exp2 Exp*3 }    = { Exp*1 Exp1 Exp2 Exp*3 }

[6] [ Set*1 { Exp*1 Id Exp*2 } Set*2
            { Exp*3 Id Exp*4 } Set*3 ] =
    [ Set*1 { Exp*1 Id Exp*2 Exp*3 Exp*4 } Set*2 Set*3 ]
```

input

```
infer-use(
  begin declare x : natural,
                y : natural,
                z : natural;
        x := 0;
        if x then y := 1
             else y := 2 fi;
        z := x + 3; y := 4
  end, [])
```

output

```
[ { y 4 2 1 } { z x 3 0 } ]
```

Figure 6.18: Inferring variable usage for PICO programs.

6.3.2 Inferring Variable Usage

The second example in figure 6.18 computes an equivalence relation for PICO variables based on their usage in a program. This technique is known as *type-inference* [55] and can be used for compiler optimization and reverse engineering. Examples are statically inferring variable types in a dynamically typed language such as Smalltalk or in a weakly typed language such as COBOL ([71]).

The analysis starts with the assumption that the input program is correct. Based on their usage in the program variables are related to each other by putting them in the same equivalence class. Finally, the equivalence classes are completed by taking their transitive closure. Variables of the same type that are used for different purposes will thus appear in different classes. In this way one can, for example, distinguish integer variables used for dates and integer variables used for account numbers.

In the specification, notation is introduced for sets of expressions (`SET`) and sets of such sets (`SETS`). The accumulator `infer-type` is then declared that collects identifier declarations, expressions and assignments and puts them in separate equivalence classes represented by `SETS`. This is expressed by equations `[0]` and `[1]`. In `[0]` an assignment statement generates a new set consisting of both sides of the assignment. In `[1]` an expression generates a new set on its own. In equations `[2]` through `[5]`, equivalence sets are simplified by breaking down complex expressions into their constituting operands. Finally, equation `[6]` computes the transitive closure of the equivalence relation.

Note that equations `[2]` through `[6]` use list matching to concisely express operations on sets. For instance, in `[2]` the list variables `Exp*1`, `Exp*2` and `Exp*3`, are used to match elements that surround the two occurrences of the same expression `Exp`. On the right-hand side of the equation, they are used to construct a new list of expressions that contains only a single occurrence of `Exp`. In fact, this equation defines that `SET` actually defines sets! figure 6.18 also shows an example of applying `infer-use` to a small program.

6.3.3 Examples of Accumulating Transformers

We leave examples of accumulating transformers to the reader. They can be found in two directions. Either transformation with side-effects or a transformations with state. A trivial example of the first is to generate a log file of a transformation. Log entries are added to the accumulated argument while the traversed argument is transformed. This functionality can sometimes be split into first generating the log file and then doing the transformation, but that inevitably leads to code duplication and degradation of performance.

An instance of the second scenario is a transformer that assigns a unique identification to some language constructs. The accumulated argument is used to keep track of the identifications that were already used. It is impossible to split this behavior into a separate transformer and accumulator.

Algorithm 2 An interpreter for transformers, Part 1.

```
function traverse-trafo(t : term, rules : list-of[rule]) : term
begin
  var trfn    : function-symbol;
  var subject : term;
  var args    : list-of[term];

  decompose term t as trfn(subject,args)
  return visit(trfn, subject, args, rules);
end

function visit(trfn : function-symbol, subject : term, args : list-of[term],
               rules : list-of[rule]) : term
begin
  var subject', reduct : term;

  if traversal-strategy(trfn) = TOP-DOWN
  then subject' := reduce(typed-compose(trfn, subject, args), rules);
     if subject' = fail
     then return visit-children(trfn, subject, args, rules)
     else if traversal-continuation(trfn) = BREAK
        then return subject'
        else reduct := visit-children(trfn, subject', args, rules)
             return if reduct = fail then subject' else reduct fi
        fi
     fi
  else /* BOTTOM-UP */
     subject' := visit-children(trfn, subject, args, rules);
     if subject' = fail
     then reduct = reduce(typed-compose(trfn, subject, args), rules)
     else if traversal-continuation(trfn) = BREAK
        then return subject'
        else reduct = reduce(typed-compose(trfn, subject',args), rules)
             return if reduct = fail then subject' else reduct fi
        fi
     fi
  fi
end
```

6.4 Operational Semantics

Now we will describe an operational semantics for Traversal Functions. We assume that we have a *fully typed* term representation.

This means that with every function name a first order type can be associated. For example, a function with name f could have type $f : \tau_1 \times \ldots \times \tau_n \to \tau_r$. If $n = 0$, f is a constant of type $f :\to \tau_r$. If $n > 0$, f is either a constructor or a function with its arguments typed by $\tau_1, ..., \tau_n$ respectively. We will call this fully typed version of a function name a *function symbol* and assume that terms only contain function symbols. Of course, the term construction and destruction and matching functionality should be adapted to this term representation.

Note that the typed-term representation is an operational detail of Traversal Functions. It is needed to match the correct nodes while traversing a tree. However, a def-

Algorithm 3 An interpreter for transformers, Part 2.

```
function visit-children(trfn : function-symbol, subject : term,
                        args : list-of[term],
                        rules : list-of[rule]) : term
begin
  var children, children' : list-of[term];
  var child, reduct       : term;
  var fn                  : id;
  var success             : bool;
  decompose term subject as fn(children);
  success := false;
  foreach child in children
  do reduct := visit(trfn, child, args, rules);
     if reduct != fail
     then children' := append(children', reduct);
          success   := true;
     else children' := append(children', child)
     fi
  od;
  return if success = true then compose the term fn(children') else fail fi
end
function typed-compose(trfn : function-symbol, subject : term,
                       args : list-of[term]) : term
begin
  var τ1, τ2, ..., τn, τsubject : type;
  var rsym : function-symbol;
  var fn   : id;
  τsubject := result-type-of(subject);
  decompose function-symbol trfn as fn:τ1 × τ2 × ... × τn -> τ1;
  rsym := compose function-symbol fn: τsubject × τ2 × ... × τn -> τsubject
  return compose term rsym(subject, args);
end
```

inition of a Traversal Function can be statically type-checked (Section 6.2) to ensure that its execution never leads to an ill-formed term.

6.4.1 Extending Innermost

We start with normal innermost rewriting as depicted earlier in Algorithm 1 (see Section 6.1.4). The original algorithm first normalizes the children of a term and relies on `reduce` to reduce the term at the outermost level.

In the modified algorithm, the call to the function `reduce` is replaced by a case distinction depending on the kind of function: a normal function (i.e., not a Traversal Function), a transformer, an accumulator, or an accumulating transformer. For these cases calls are made to the respective functions `reduce`, `traverse-trafo`, `traverse-accu`, or `traverse-accu-trafo`. Note that we describe the three kinds of Traversal Functions here by means of three different functions. This is only done for expository purposes (also see the discussion in Section 6.4.5).

6.4.2 Transformer

The function `traverse-trafo` and its auxiliary functions are shown in Algorithms 2 and 3. Function `traverse-trafo` mainly decomposes the input term into a function symbol (the Traversal Function), the subject term to be traversed and optional arguments. It then delegates actual work to the function `visit`.

Function `visit` distinguishes two major cases: top-down and bottom-up traversal. In both cases the break/continue behavior of the Traversal Function has to be modeled. If an application of a Traversal Function has not failed the recursion either continues or breaks, depending on the annotation of the Traversal Function. If the application has failed it always continues the recursion.

We apply the Traversal Function by reusing the `reduce` function from the basic innermost rewriting algorithm (see Algorithm 1). It is applied either before or after traversing the children, depending on the traversal strategy (bottom-up or or top-down). `visit` depends on `visit-children` for recurring over all the children of the current node. If none of the children are reduced `visit-children` returns `fail`, otherwise it returns the list of new children.

In order to be type-safe, the type of the Traversal Function follows the type of the term is being traversed. Its type always matches the type of the node that is currently being visited. This behavior is encoded by the `typed-compose` function. Transformers are type-preserving, therefore the type of the first argument and the result are adapted to the type of the node that is currently being visited. Note that using this algorithm this we can reuse the existing matching functionality.

The following auxiliary functions are used but not defined in these algorithms:

- `traversal-strategy(fn : function-symbol)` returns the traversal strategy of the given function symbol `fn`, i.e., `top-down` or `bottom-up`.
- `traversal-continuation(fn : function-symbol)` returns the continuation

  style of the given function symbol `fn`, i.e., `break` or `continue`.
- `result-type-of(t : term)` returns the result type of the outermost function symbol of the given term `t`.

6.4.3 Accumulator

The function `traverse-accu` and its auxiliary functions are shown in Algorithms 4 and 5. The definitions largely follow the same pattern as for transformers, with the following exceptions:

- `traverse-accu` not only separates the traversed subject from the arguments of the Traversal Function. It also identifies the second argument as the initial value of the accumulator.
- Both `visit` and `visit-children` have an extra argument for the accumulator.

- In `typed-compose` only the type of the first argument is changed while the type of the accumulator argument remains the same.
- The traversal of children in function `visit-children` takes into account that the accumulated value must be passed on between each child.

6.4.4 Accumulating Transformer

We do not give the details of the algorithms for the accumulating transformer since they are essentially a fusion of the algorithms for accumulators and transformers. Since an accumulating transformer has two input and output values (the initial term and the current accumulator value, respectively, the transformed term and the updated accumulator value), the types of `visit`, `visit-children` and `typed-compose` have to be adjusted to manipulate a pair of terms rather than a single term.

6.4.5 Discussion

In the above presentation we have separated the three cases transformer, accumulator and accumulating transformer. In an actual implementation, these three cases can be implemented by a single function that uses pairs of terms (to accommodate accumulating transformers).

The algorithms become slightly more involved since the algorithms for transformer and accumulator now have to deal with term pairs and in several places case distinctions have to be made to cater for the specific behavior of one of the three algorithms.

6.5 Implementation Issues

The actual implementation of Traversal Functions in ASF+SDF consists of three parts:

- Parsing the user-defined rules of a Traversal Function (Section 6.5.1).
- An interpreter-based implementation of Traversal Functions (Section 6.5.2).
- A compilation scheme for Traversal Functions (Section 6.5.3).

6.5.1 Parsing Traversal Functions

The terms used in the rewrite rules of ASF+SDF have user-defined syntax. In order to parse a specification, the user-defined term syntax is combined with the standard equation syntax of ASF. This combined syntax is used to generate a parser that can parse the specification.

In order to parse the rewrite rules of a Traversal Function we need grammar rules that define them.

A first approach (described in [37]) was to generate the syntax for any possible application of a Traversal Function. This collection of generated functions could be viewed as one *overloaded* function. This simple approach relieved the programmer

from typing in the trivial productions himself. In practice, this solution had two drawbacks:

- The parse tables tended to grow by a factor equal to the number of Traversal Functions. As a result, interactive development became unfeasible because the parse table generation time was growing accordingly.
- Such generated grammars were possibly ambiguous. Disambiguating grammars is a delicate process, for which the user needs complete control over the grammar. This control is lost if generated productions can interfere with user-defined productions.

An alternative approach that we finally adopted is to let the user specify the grammar rule for each sort that is used as *argument* of the Traversal Function: this amounts to *rewrite rules* defining the Traversal Function and *applications* of the Traversal Function in other rules. The amount of work for defining or changing a Traversal Function increases by this approach, but it is still proportional to the number of node types that are actually being visited. The parse table will now only grow proportionally to the number of visited node types. As a result the parse table generation time is acceptable for interactive development.

We have opted for the latter solution since we are targeting industrial size problems with Traversal Functions and solutions that work only for small examples are not acceptable. The above considerations are only relevant for term rewriting with concrete syntax. Systems that have fixed term syntax can generate the complete signature without introducing any significant overhead.

6.5.2 Interpretation of Traversal Functions

The ASF interpreter rewrites *parse trees* directly (instead of abstract terms). The parse trees of rewrite rules are simply matched with the parse trees of terms during rewriting. A reduction is done by substituting the parse tree of the right-hand side of a rule at the location of a redex in the term.

The ASF+SDF interpreter implements the algorithms as presented in Section 6.4.

6.5.3 Compilation of Traversal Functions

In order to have better performance of rewriting systems, compiling them to C has proved to be very beneficial. The ASF+SDF compiler [33, 30] translates rewrite rules to C functions. The compiled specification takes a parse tree as input and produces a parse tree as result. Internally, a more dense abstract term format is used. After compilation, the run-time behavior of a rewriting system is as follows:

1. In a bottom-up fashion, each node in the input parse tree is visited and the corresponding C function is retrieved and called immediately. This retrieval is implemented by way of a pre-compiled dictionary that maps function symbols to the corresponding C function. During this step the conversion from parse tree to abstract term takes place. The called function contains a dedicated matching

automaton for the left-hand sides of all rules that have the function symbol of this node as outermost symbol. It also contains an automaton for checking the conditions. Finally there are C function calls to other similarly compiled rewrite rules for evaluation of the right-hand sides.

2. When an application of a C function fails, this means that this node is in normal form. As a result, the normal form is explicitly constructed in memory. Nodes for which no rewrite rules apply, including the constructors, have this as standard behavior.

3. Finally, the resulting normal form in abstract term format is translated back to parse tree format using the dictionary.

Traversal functions can be fitted in this run-time behavior in the following manner. For every defining rewrite rule of a Traversal Function and for every call to a Traversal Function the type of the overloaded argument and optionally the result type is turned into a single universal type. The result is a collection of rewrite rules that all share the same outermost Traversal Function, which can be compiled using the existing compilation scheme to obtain a matching automaton for the entire Traversal Function.

Figure 6.19 clarifies this scheme using a small example. The first phase shows a module containing a Traversal Function that visits two types `A` and `B`. This module is parsed, type-checked and then translated to the next module (pretty-printed here for readability). In this phase all variants of the Traversal Function are collapsed under a single function symbol. The `"_"` denotes the universally quantified type.

The Traversal Function in this new module is type-unsafe. In `[2]`, the application of the Traversal Function is guarded by the `b` constructor. Therefore, this rule is only applicable to such terms of type `B`. The other rule `[1]` is not guarded by a constructor. By turning the type of the first argument of the Traversal Function universal, this rule now matches terms of *any* type, which is not faithful to the semantics of ASF+SDF.

The solution is to add a run-time type-check in cases where the first argument of a Traversal Function is not guarded. For this we can use the dictionary that was described above to look up the types of symbols. The new module is shown in the third pane of figure 6.19. A condition is added to the rewrite rule, stipulating that the rule may only succeed when the type of the first argument is equal to the expected type. The `type-of` function encapsulates a lookup in the dictionary that was described above. It takes the top symbol of the term that the variable matched and returns its type. This module can now be compiled using the conventional compiler to obtain a type-safe matching automaton for all defining rules of the Traversal Function.

To obtain the tree traversal behavior this automaton is now combined with calls to a small run-time library. It contains functions that take care of actually traversing the tree and optionally passing along the accumulated argument. The fourth pane of figure 6.19 shows the C code for the running example.

Depending on the traversal type there is a different run-time procedure. In this case it is a transformer, so `call_kids_trafo` is used. For a transformer the function is applied to the children, and a new node is created after the children are reduced. For an accumulator the library procedure, `call_kids_accu`, also takes care of passing along the accumulated value between the children. Depending on the traversal order

Algorithm 4 An interpreter for accumulators, Part 1.

```
function traverse-accu(t : term, rules : list-of[rule]) : term
begin
  var trfn    : function-symbol;
  var subject : term;
  var args    : list-of[term];

  decompose term t as trfn(subject, accu, args)
  return visit(trfn, subject, accu, args, rules);
end

function visit(trfn : function-symbol,
               subject : term,
               accu : term,
               args : list-of[term],
               rules : list-of[rule]) : term
begin
  var reduct, accu' : term;

  if traversal-strategy(trfn) = TOP-DOWN
  then accu' := reduce(typed-compose(trfn, subject, accu, args), rules);
     if accu' = fail
     then return visit-children(trfn, subject, accu, args, rules)
     else if traversal-continuation(trfn) = BREAK
        then return accu'
        else reduct = visit-children(trfn, accu', reduct, args, rules)
           return if reduct = fail then accu' else reduct fi
        fi
     fi
  else /* BOTTOM-UP */
     accu'  := visit-children(trfn, subject, accu, args, rules);
     if accu' = fail
     then reduct := reduce(typed-compose(trfn, subject, accu, args),
                           rules);
     else if traversal-continuation(trfn) = BREAK
        then return accu'
        else reduct := reduce(typed-compose(trfn, subject, accu', args),
                              rules);
           return if reduct = fail then accu' else reduct fi
        fi
     fi
  fi
end
```

Algorithm 5 An interpreter for accumulators, Part 2.

```
function visit-children(trfn : function-symbol,
                        subject : term,
                        accu : term,
                        args : list-of[term],
                        rules : list-of[rule]) : term
begin
  var children                : list-of[term];
  var child, accu', reduct    : term;
  var fn                      : id;
  var success                 : bool;

  decompose term subject as fn(children);
  accu' := accu; success := false;
  foreach child in children
  do reduct  := visit(trfn, child, accu', args, rules);
     if reduct != fail then success = true; accu' := reduct fi
  od;
  return if success = true then accu' else fail fi
end

function typed-compose(trfn : function-symbol,
                       subject : term,
                       accu : term,
                       args : list-of[term]) : term
begin
  var τ1, τ2, ..., τn, τsubject : type;
  var rsym : function-symbol;
  var fn   : id;

  τsubject := result-type-of(subject);
  decompose function-symbol trfn as fn: τ1 × τ2 × ... × τn -> τ2 ;
  rsym := compose function-symbol fn: τsubject × τ2 × ... × τn -> τ2;
  return compose term rsym(subject, accu, args);
end
```

```
module Example
exports
  context-free syntax
  a          -> A
  b ( A )    -> B
  example(A) -> A {traversal(trafo,bottom-up,continue)}
  example(B) -> B {traversal(trafo,bottom-up,continue)}
variables
  "VarA" -> A
equations
[1] example(VarA)    = ...
[2] example(b(VarA)) = ...
```

```
module Example
exports
  context-free syntax
  a          -> A
  b ( A )    -> B
  example(_) -> _ {traversal(trafo,bottom-up,continue)}
variables
  "VarA" -> A
equations
[1] example(VarA)    = ...
[2] example(b(VarA)) = ...
```

```
module Example
exports
  context-free syntax
  a          -> A
  b ( A )    -> B
  example(_) -> _ {traversal(trafo,bottom-up,continue)}
equations
[1] type-of(VarA) = A  ===> example(VarA) = ...
[2] example(b(VarA)) = ...
```

```
ATerm example(ATerm arg0)
{
  ATerm tmp0 =
    call_kids_trafo(example, arg0, NO_EXTRA_ARGS);
  if (check_symbol(tmp0, b_symbol)) { /* [2] */
    return ...;
  }
  if (term_equal(get_type(tmp0), type("A"))) { /* [1] */
    return ...;
  }
  return tmp0;
}
```

Figure 6.19: Selected phases in the compilation of a Traversal Function.

the calls to this library are simply made either before or after the generated matching automaton. The `break` and `continue` primitives are implemented by inserting extra calls to the run-time library procedures surrounded by conditionals that check the successful application or the failure of the Traversal Function.

6.6 Experience

Traversal functions have been applied in a variety of projects. We highlight some representative ones.

6.6.1 COBOL Transformations

In a joint project of the Software Improvement Group (SIG), Centrum voor Wiskunde en Informatica (CWI) and Vrije Universiteit (VU) Traversal Functions have been applied to the conversion of COBOL programs [171, 153]. This is based on earlier work described in [143]. The purpose was to migrate VS COBOL II to COBOL/390. An existing tool (CCCA from IBM) was used to carry out the basic, technically necessary, conversions. However, this leaves many constructions unchanged that will obtain the status "archaic" or "obsolete" in the next COBOL standard. In addition, compiler-specific COBOL extensions remain in the code and several outdated run-time utilities can be replaced by standard COBOL features.

Ten transformation rules were formalized to replace all these deprecated language features and to achieve code improvements. Examples of rules are:

- Adding `END-IF` keywords to close `IF`-statements.
- Replace nested `IF`-statements with `EVALUATE`-statements.
- Replace outdated `CALL` utilities by standard COBOL statements.
- Reduce `GO-TO` statements: a goto-elimination algorithm that itself consists of over 20 different transformation rules that are applied iteratively.

After formalization of these ten rules in ASF+SDF with Traversal Functions, and applying them to a test base of 582 programs containing 440000 lines of code, the following results were obtained:

- 17000 `END-IF`s were added.
- 4000 lines were changed in order to eliminate `CALL`-utilities.
- 1000 `GO-TO`s have been eliminated (about 65% of all `GO-TO`s).

Each transformation rule is implemented by means of a Traversal Function defined by only a few equations. Figure 6.20 shows two rewrite rules which add the missing `END-IF` keywords to the COBOL conditionals.

```
module End-If-Trafo
imports Cobol
exports
context-free syntax
  addEndIf(Program)  -> Program
                     {traversal(trafo,continue,top-down)}
variables
 "Stats"[0-9]*     -> StatsOptIfNotClosed
 "Expr"[0-9]*      -> L-exp
 "OptThen"[0-9]*   -> OptThen
equations
[1] addEndIf(IF Expr OptThen Stats)  =
             IF Expr OptThen Stats END-IF

[2] addEndIf(IF Expr OptThen Stats1 ELSE Stats2) =
             IF Expr OptThen Stats1 ELSE Stats2 END-IF
```

Figure 6.20: Definition of rules to add END-IFs.

The complete transformation took two and a half hours using the ASF interpreter.[5] The compiled version of Traversal Functions was not yet ready at the time this experiment was done but it would reduce the time by a factor of at least 30–40 (see Section 6.6.3). The estimated compiled execution time would therefore be under 5 minutes. These results show that Traversal Functions can be used effectively to solve problems of a realistic size.

6.6.2 SDF Re-factoring

In [114] a Framework for SDF Transformations (FST) is described that is intended to support *grammar recovery* (i.e., the process of recovering grammars from manuals and source code) as well as *grammar re-engineering* (transforming and improving grammars to serve new purposes such as information extraction from legacy systems and dialect conversions). The techniques are applied to a VS COBOL II grammar. The experience with Traversal Functions is positive. To cite the authors:

> "At the time of writing FST is described by 24 Traversal Functions with only a few rewrite rules per function. The SDF grammar itself has about 100 relevant productions. This is a remarkable indication for the usefulness of the support for Traversal Functions. In worst case, we would have to deal with about 2400 rewrite rules otherwise."

[5] On a 333 MHz PC with 192 Mb of memory running Linux.

Grammar	**# of productions**	**Interpreted** (seconds)	**Compiled** (seconds)	**Ratio**
SDF	200	35	0.85	42
Java	352	215	1.47	146
Action Semantics	249	212	2.00	106
COBOL	1251	1586	5.16	307

Table 6.2: Performance of the SDF checker.

6.6.3 SDF Well-formedness Checker

SDF is supported by a tool-set[6] containing among others a parse table generator and a well-formedness checker. A considerable part of the parse table generator is specified in ASF+SDF. The well-formedness checker is entirely specified in ASF+SDF and makes extensive use of Traversal Functions. The well-formedness checker analyses a collection of SDF modules and checks, among others, for completeness of the specification, sort declarations (missing, unused, and double), uniqueness of constructors, and uniqueness of labels. The SDF grammar consists of about 200 production rules, the ASF+SDF specification consists of 150 functions and 186 equations, 66 of these functions are Traversal Functions and 67 of the equations have a Traversal Function as the outermost function symbol in the left-hand side and can thus be considered as "traversal" equations.

An indication of the resulting performance is shown in Table 6.2.[7] It shows results for SDF, Java, Action Semantics and COBOL. For each grammar, the number of grammar rules is given as well as execution times (interpreted and compiled) for the SDF checker. The last column gives the interpreted/compiled ration. These figures show that Traversal Functions have a completely acceptable performance. They also show that compilation gives a speed-up of at least a factor 40.

6.7 Discussion

Traversal functions are based on a minimalist design that tries to combine type safety with expressive power. We will now discuss the consequences and the limitations of this approach.

6.7.1 Declarative versus Operational Specifications

Traversal functions are expressed by annotating function declarations. Understanding the meaning of the rules requires understanding which function is a Traversal Function and what visiting order it uses. In pure algebraic specification, it is considered bad practice to depend on the rewriting strategy (i.e., the operational semantics) when writing specifications. By extending the operational semantics of our rewrite system with Traversal Functions, we effectively encourage using operational semantics. However,

[6] `www.cwi.nl/projects/MetaEnv`

[7] On a 333 MHz PC with 192 Mb of memory running Linux.

if term rewriting is viewed as a programming paradigm, Traversal Functions enhance the declarative nature of specifications. That is, without Traversal Functions a simple transformation must be coded using a lot of "operational style" rewrite rules. With Traversal Functions, only the essential rules have to be defined. The effort for understanding and checking a specification decreases significantly. In [37] we show how Traversal Functions in ASF+SDF can be translated to specifications without Traversal Functions in a relatively straightforward manner. So, Traversal Functions can be seen as an abbreviation mechanism.

6.7.2 Expressivity

Recall from figure 6.1 the main left-to-right visiting orders for trees: top-down and bottom-up combined with two stop criteria: stop after first application or visit all nodes. All of these orders can be expressed by Traversal Functions using combinations of `bottom-up`, `top-down`, `break` and `continue`. We have opted for a solution that precisely covers all these possible visiting orders.

One may wonder how concepts like *repetition* and *conditional evaluation*, as used in strategic programming (see Section 6.1.7), fit in. In that case, *all* control structures are moved to the strategy language and the base language (rewrite rules, functions) remains relatively simple. In our case, we use a base language (ASF+SDF) that is already able to express these concepts and there is no need for them to be added to the set of traversal primitives.

6.7.3 Limited Types of Traversal Functions

Accumulators can only map sub-trees to a *single* sort and transformers can only do sort preserving transformations. Is that a serious limitation?

One might argue that general non-sort-preserving transformations cannot be expressed conveniently with this restriction. Such transformations typically occur when translating from one language to another and they will completely change the type of every sub-term. However, in the case of *full* translations the advantage of *any* generic traversal scheme is debatable, since translation rules have to be given for any language construct anyway. A more interesting case are *partial* translations as occur when, for instance, embedded language statements are being translated while all surrounding language constructs remain untouched. In this case, the number of rules will be proportional to the number of *translated* constructs only and not to the total number of grammatical constructs. Most of such partial transformations can be seen as the combination of a sort-preserving transformation for the constructs where the transformation is not defined and a non-sort-preserving transformation for the defined parts. If the sort-preserving part is expressed as a transformer, we have again a number of rewrite rules proportional to the number of translated constructs. It is therefore difficult to see how a generic non-sort-preserving traversal primitive could really make specifications of translations more concise.

6.7.4 Reuse versus Type-safety

We do not separate the traversal strategy from the rewrite rules to be applied. By doing so, we loose the potential advantage of reusing the same set of rewrite rules under different visiting orders. However, precisely the *combination* of traversal strategy and rewrite rules allows for a simple typing mechanism. The reason is that the generic traversal attributes are not separate operators that need to be type-checked. It allows us to ensure well-formedness in both type-preserving transformations and in type-unifying computations without extending the typing mechanisms of our first-order specification language.

6.7.5 Conclusions

We have described term rewriting with Traversal Functions as an extension of ASF+SDF. The advantages of our approach are:

- The most frequently used traversal orders are provided as built-in primitives.
- The approach is fully type-safe.
- Traversal functions can be implemented efficiently.

Traversal functions are thus a nice compromise between simplicity and expressive power.

The main disadvantage of our approach might manifest itself when dealing with visiting orders that go beyond our basic model of tree traversal. Two escapes would be possible in these cases: such traversals could either be simulated as a modification of one of the built-in strategies (by adding conditions or auxiliary functions), or one could fall back to the tedious specification of the traversal by enumerating traversal rules for all constructors of the grammar.

In practice, these scenario's have not occurred and experience with Traversal Functions shows that they are extremely versatile when solving real-life problems.

CHAPTER 7

Rewriting with Layout

In this chapter we assert that term rewriting is an adequate and powerful mechanism to perform source code transformations. However, an important shortcoming of rewriting technology is that source code comments and layout are discarded before rewriting. We propose "rewriting with layout" to solve this problem. We present a rewriting algorithm that keeps the layout of sub-terms that are not rewritten, and reuses the layout occurring in the right-hand side of the rewrite rules. [1]

7.1 Introduction

Rewriting technology has proved to be an adequate and powerful mechanism to tackle all kinds of problems in the field of software renovation. Software renovation is to bring existing source code up to date with new requirements. One of the techniques applied in this field is source code transformation. Source code transformations are transformations applied on the syntax level of programs, usually implemented using string replacement technology.

Such transformations can also conveniently be implemented using parsing and rewriting technology. Using these technologies is safer than using regular string replacement tools because they provide a firmer grip on source code syntax. The parser constructs a tree representation of the source code, which the rewriter traverses and manipulates. The information in the tree safely guides all transformations. For example, it allows us to ignore the particular layout of a code fragment and concentrate on its essence, or know the difference between a normal code fragment and a code fragment that has been commented out. We call this feature *syntax safety*: to be able to statically determine that the input as well as the output source code of a transformation is defined by a context-free grammar.

However, an important shortcoming of rewriting technology is that source code comments and layout are lost during rewriting. Usually, the reason is that this information is *discarded* completely in the tree representation. Note that no maintenance programmer will consider using rewriting technology and as a result loose all of his

[1] This chapter was published in RULE 2000 [50], and coauthored by Mark van den Brand.

source code comments during a large maintenance job. It might be possible to store comments and layout directly in the tree, but this complicates the specification of such a source code transformation tool because every rewrite rule has to take comments and layout explicitly into consideration. In other words, we loose the ability to *ignore* layout. We propose *rewriting with layout* to solve this problem, and try to make rewriting technology a more attractive alternative to conventional software maintenance tooling. Throughout this chapter, the term *layout* will be used to indicate both formatting by whitespace characters as well as source code comments.

We present a rewriting algorithm that *conserves* layout of sub-terms that are not rewritten and reuses the layout occurring in the right-hand side of the rewrite rules. We will analyze its run-time efficiency, in order to find out whether this approach scales to larger applications.

Using this lightweight algorithm, a certain amount of layout could still be lost when a rewrite rule is applied. It also does not provide the ability to *utilize* layout explicitly for software renovation. It is now implicitly conserved by the rewriting algorithm, but still out of reach from the programmer. In Chapter 8 we solve these issues by promoting layout trees to "first class objects", such that they can be manipulated like any other term.

7.1.1 Source code transformations

Maintenance programmers frequently use string replacement tools to automate their software maintenance tasks. For example, they use the regular expressions available in scripting languages like Perl [167] to perform all kinds of syntactical transformations. Naturally, such source code transformations must be precise. Alas, regular string matching alone is not powerful enough to recognize all kinds of syntactical structures commonly found in programming languages. This lack of power is usually solved by extensive programming, or the lack of precision is accepted as a caveat that sometimes introduces false positives and negatives. Some string replacement languages, like SNOBOL [76], would provide the maintenance programmer with low level context-free matching. But these powerful features are hardly used. It seems that practicality does not go hand in hand with precision in this instance.

The focus of source code transformations on the one hand and general program transformations [136] on the other hand is quite different. Source code transformations deal with automatic syntactical transformations. They automate software maintenance tasks, but they do not usually involve any correctness proof of the adaptations to the source code. In contrast, general program transformations can require user interaction, involve proving that the transformations are sound and complete, and do not have to relate closely to the source code of a program.

As opposed to string rewriting, term rewriting technology is a different approach to implement source code transformations. The source code is fully parsed given the context-free grammar of the language, the term representation is transformed according to a set of powerful rules and the result is unparsed to obtain source code again. We use an algebraic specification formalism, ASF+SDF [67], based on term rewriting. Due to recent improvements of its compilation techniques [33] and term representation [31], ASF+SDF can now be applied better to *industry sized* problems such as described

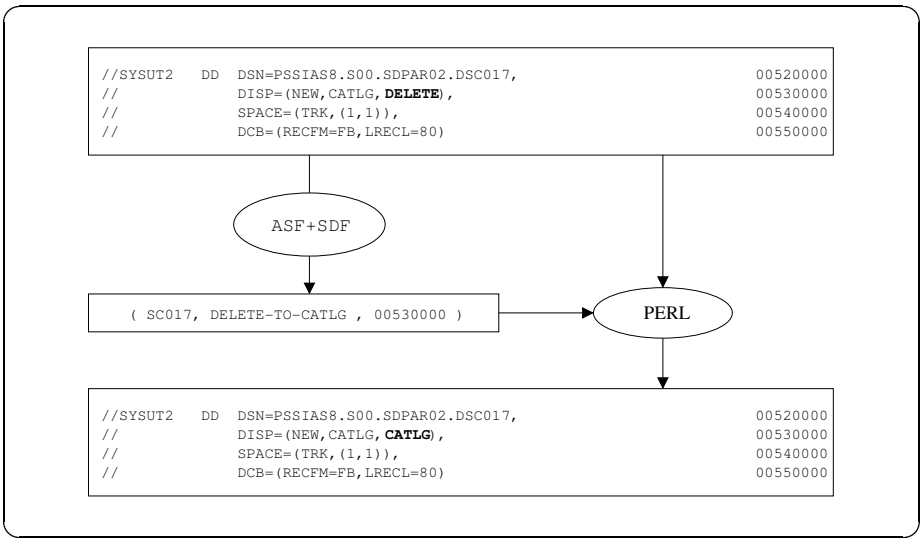

Figure 7.1: Sample input and output of a source code transformation in JCL. The `DELETE` keyword is be replaced by `CATLG`, but in a specific context.

earlier in [36]. For example, COBOL renovation factories have been implemented and used [49]. It can be expected that software maintenance tooling can be developed with more confidence and less effort using such rewriting technology. The benefits are higher level implementations, and more precision.

The rewrite rules in ASF+SDF use concrete syntax. They are defined on the actual syntax of the source language, not some abstract representation. Because the rules are applied to structured terms instead of strings, complex syntactical structures can be grasped by a single rule. A very important feature of rewrite rules is that they ignore the arbitrary layout of the source code, and even source code comments. This is all in favor of simplicity towards the programmer: the distance between source code and the program that transform it is minimal, and he can ignore all irrelevant details.

However, some details that are ignored by default are not irrelevant at all: for example source code comments. This rather practical issue of discarding all layout needs to be resolved before term rewriting can fully deal with the particular requirements of the reverse engineering and re-engineering application domains.

7.1.2 Example

The following example of a source code transformation shows the importance of considering layout while rewriting. This example is a part of a reverse engineering project of JCL scripts[2] in cooperation with a Dutch software house.

[2]JCL stands for Job Control Language and is mainly used in combination with COBOL programs on IBM mainframes.

The interpretation of JCL scripts is sensitive to their particular layout, which is an unfortunate but not uncommon language property. Note that there are numerous examples of languages that depend on layout, although each in their own manner. Examples are COBOL, Haskell, and Python. As an adequate solution to the problem regarding JCL, the source code transformation was performed in two steps:

1. A rewriting language, ASF+SDF, was used to reduce a JCL script to a list of instructions that indicate precisely where to modify the script.

2. The list of instructions was interpreted by a Perl script that used regular string replacements to implement them.

The above situation is depicted in Figure 7.1 for a specific JCL instruction. Obviously, this effective combination of term rewriting and string replacement is neither an attractive nor a generic solution to the problem of transforming layout sensitive source code. It would be preferable to encode the entire transformation in the rewriting language.

7.1.3 Overview

To explore the subject, we have developed an ASF+SDF interpreter that conserves layout. Apart from the actual rewriting algorithm, there are two important prerequisites to the idea of a layout preserving rewriter. Firstly, the parser should produce trees in which the layout is preserved in some way. Secondly, rewriting must be performed on a term representation that also contains all layout.

In Section 7.2 we will introduce the term format. In Section 7.3 we discuss our layout preserving algorithm for ASF+SDF. Section 7.4 describes some benchmark figures. We compare the performance of the layout preserving rewriter with the original ASF+SDF rewriter, and with two other interpreted rewriting systems: ELAN [24] and Maude [58].

Finally, in Section 7.6 we draw some conclusions. Note that in this chapter related work is not discussed. In Chapter 8, we will describe related work that was available after the appearance of the current chapter in RULE 2000.

7.2 Term format

One of the features of rewriting technology is that it implicitly discards all layout of the input and output. It is considered not important, so usually layout information is simply not included in the term format. The same strategy is chosen in most language compilers. For efficiency reasons all "irrelevant" information is discarded from abstract syntax trees. So in rewriting it is also common practice to have a very concise tree representation to represent terms that have to be rewritten. Typical examples of these concise formats are REF [21] used within the ELAN system [24], and μAsf used within the ASF+SDF compiler [33].

We have been exploring another solution: using full parse trees as term format for rewriting. These parse trees contain all information encountered during parsing, e.g.,

```
module Booleans
syntax
  Bool LAYOUT "or" LAYOUT Bool -> Bool {left}
  [\t-\n\ ]                    -> LAYOUT
  "true"                       -> Bool
  "false"                      -> Bool
  [o][r]                       -> "or"
  [t][r][u][e]                 -> "true"
  [f][a][l][s][e]              -> "false"
```

Figure 7.2: An example low-level (kernel) grammar.

```
true
or false
```

Figure 7.3: A term over the Booleans. An ASF+SDF specification of the Booleans can be found in Figure 7.5.

layout, keywords, application of syntax production rules, disregarding none of the information that is present in the original input file. Although much of this information is redundant for the rewriting process itself, it is of importance to the entire transformation process from input to output. In the following two sections we briefly describe our parse trees and the generic term data type it is based on.

7.2.1 ATerm data type

Our representation of parse trees is based on a generic abstract data type called ATerm [31]. The corresponding libraries for this ATerm format have a number of important properties. One of the most important properties is that the ATerm library ensures a maximal sharing of terms, each term is unique. This property results in a memory and execution time efficient run-time behavior. Maximal sharing proofs to be especially beneficial when applied to our parse tree format, which is rather redundant.

A striking consequence of the maximal sharing is that term equality can be implemented as pointer equality. A negative effect of sharing is that the ATerm library allows only functional manipulation of terms. This means that destructive updates on terms can only be implemented by rebuilding the updated term from scratch. But the ATerm library is time efficient nevertheless in the context of term rewriting (see Section 7.4).

7.2.2 Parse trees

Based on the ATerm format we are able to define a simple format to represent parse trees [157]. A parse tree consists of applications of productions and characters. Each application of a production is a node in the tree, and each character is a leaf.

As an example, take the grammar in Figure 7.2. This grammar defines the Booleans with an `or` operator down to the character level, including the whitespace. Using this

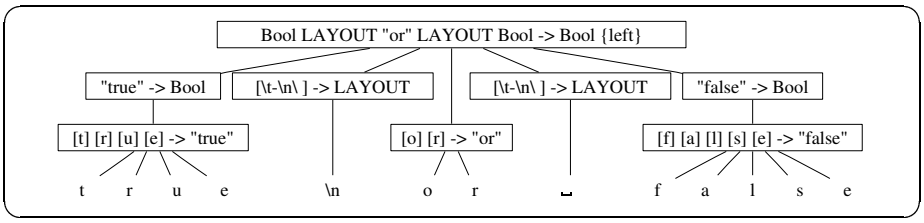

Figure 7.4: A graphical representation of the fully informative parse tree of the term in Figure 7.3.

grammar, we can parse the input boolean term in Figure 7.3, to obtain the parse tree that is depicted graphically in Figure 7.4. Three important characteristics are:

- The structure of the parse tree is fully defined by the original grammar.
- All characters of the input are present as the leafs of the parse tree.
- Maximal sub-term sharing (via ATerms) is used to keep such parse trees small.

Note that in practice we would use the syntax definition formalism (SDF) to define the Boolean language (e.g., Figure 7.5). The implementation of SDF takes care of generating a corresponding character level grammar automatically. This generation process from concise notation to character level notation is straightforward [157]. For example, it introduces a `LAYOUT?` non-terminal in between every two members of a production to indicate that `LAYOUT` can optionally occur. It also adds a production for concatenating `LAYOUT`, namely `LAYOUT LAYOUT -> LAYOUT {left}`, and it implements the literals (keywords) of a language by simple productions with consecutive characters.

7.3 Rewriting with Layout

The rewriting algorithm that is used in the ASF+SDF Meta-Environment [99, 42] operate on the parse tree format described above. We will discuss an adaptation of this algorithm that will preserve as much layout as possible. First we will briefly introduce the reader to the semantics of our rewriting formalism ASF [14]. Then we discuss the interpreter and the adapted interpreter in detail.

7.3.1 Rewriting terms

An example of a basic ASF+SDF specification is presented in Figure 7.5, the equations that define the semantics of the `or` operator are specified. Note that the equations have labels. These labels have no semantics. For more elaborate ASF+SDF examples we refer to [67].

Given an ASF+SDF specification and some term to be normalized, this term can be rewritten by interpreting the ASF equations as rewrite rules. One approach is to

```
module Booleans

imports Layout

exports
  sorts Bool
  context-free syntax
    "true"          -> Bool
    "false"         -> Bool
    Bool "or" Bool -> Bool {left}
  variables
    "Bool"[0-9]* -> Bool

equations
 [or-1] true or Bool = true
 [or-2] false or Bool = Bool
```

Figure 7.5: A more concise definition of boolean syntax, with equations for `or`-operator

compile these equations to C functions [33]. We do this to optimize batch performance of rewrite systems. We also provide a small interpreter that facilitates interactive development of rewrite systems. To explore the subject of rewriting with layout, we have chosen to extend the ASF+SDF interpreter. First we will discuss an interpreter which ignores layout completely in both the term and the equations. Thereafter, we discuss the extension to an interpreter that conserves layout in a specific manner.

The ASF+SDF interpreter takes as input parse trees of both the term and the set of equations. We explicitly do not use some abstract term representation, because we need access to all information that is present in the input. The parse trees are slightly modified, discarding all layout nodes from the parse trees of both the equations and the term. This is an easy and efficient method to ignore layout. The efficiency benefit is due to the ATerm library on which the parse trees are built (Section 7.2.1). Without the layout nodes and with maximal sharing, term equality can be decided by single pointer comparison.

The rewriter operates on these stripped parse trees as an ordinary rewriting engine. Based on the outermost function symbol of some sub-term the appropriate set of rewrite rules (with the same outermost function symbol in the left-hand side) is selected. If the left-hand side matches, variables are instantiated by this match. Then the conditions are evaluated one-by-one using the instantiated variables. Along with the evaluation of the conditions new variables are instantiated. If the evaluation of *all* conditions is successful, the reduct is built by instantiating the variables in the right-hand side of an equation. This reduct replaces the old sub-tree that was matched by the left-hand side of the equation.

We have experimented with a straightforward rewriting with layout algorithm, which proofs to be sufficient. We modified our rewriter in three ways such that it is

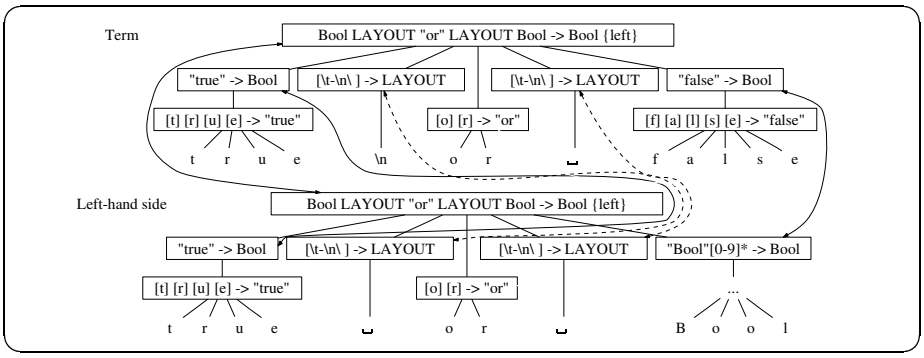

Figure 7.6: Matching the parsed term in Figure 7.3 to the parsed left-hand side of rule `[or-1]` in Figure 7.5. The dotted lines depict matches of layout nodes.

no longer necessary to discard the layout before rewriting:

- *Equality* is implemented modulo layout.
- *Matching* is implemented modulo layout.
- *For construction,* the layout of the right hand sides is reused.

To implement abstraction from layout, term equality can no longer be implemented as pointer equality. An almost full traversal of both trees is now needed to decide term equality. In Section 7.4 we will show what performance penalty is paid now that we need to look deeper for equality modulo layout.

To implement matching modulo layout, the matching algorithm is adapted to identify two classes of nodes: normal nodes and layout nodes. We use a fixed non-terminal (`LAYOUT`) to decide which nodes define layout. Normal nodes can be matched as usual, when the productions are equal. *Layout nodes are defined to always match, completely ignoring their actual contents.* This effect is illustrated by the example in Figure 7.6. Two parse trees are matched by comparing their top node. If their pointers identities are completely equal, we can stop early (maximal sharing). We can also stop if the productions are unequal. If the productions are equal, the match continues recursively. Variables match any subtree of the same top sort. Any two compared layout nodes always match, which implements abstraction from layout.

Finally, the layout occurring in the right-hand side of a successful equation is just left in the normal form. Which effectively means that it is inserted in the reduct. In this way the specification writer can influence the layout of the constructed normal form by formatting the right-hand side of an equation manually.

Note that a logical consequence of this approach is that no layout is conserved from sub-terms that are matched by the left-hand side of an equation. Note that the values of any instantiated variables (sub-terms) do conserve their original layout. So, the new algorithm conserves layout where nothing is rewritten. Where parse trees are rewritten layout is lost permanently, and replaced by the layout of reducts.

```
module BoolList

imports Booleans
exports
  sorts List
  context-free syntax
    "[" Bool ";"* "]" -> List
  variables
    "Bools"[0-9]* -> {Bool ";"}*

equations
 [set-1] [ Bools1 ; Bool ; Bools2 ; Bool ; Bools3 ] =
           [ Bools1 ; Bool ; Bools2 ; Bools3 ]
```

Figure 7.7: An extension of the Boolean syntax and an equation with list variables.

7.3.2 Rewriting lists

A characteristic feature of ASF is list matching. List matching (also called associative matching) enables the specification writer to manipulate elements of a list in a concise manner. The manipulation of the list elements is performed via so-called list patterns, in such a list pattern the individual list elements can be addressed or sublists can be matched via list variables. List matching may involve backtracking. However, the backtracking is restricted to the scope of the rewrite rule in which the list pattern occurs. The possible matches are strictly ordered to enforce deterministic and finite behavior. ASF+SDF supports two variants of lists: lists without separators and lists with separators. A '`*`' indicates zero or more elements and a '`+`' indicates that the list should contain at least one element. For example: `A*` is a list of zero or more elements of sort `A` and `{B ";"}+` represents a list of at least one element of sort `B`. The `B` elements are separated by semicolons.

The ASF+SDF specification in Figure 7.7 demonstrates the use of list matching in an equation. This equation will remove all double occurring `Bool` terms from any `List`. Two equal `Bool` elements of the list are matched anywhere in between the sublists `Bools1`, `Bools2` and `Bools3`. On the right-hand side, one of the instances is removed.

The interpreter implements list matching by means of backtracking. Given a term representing a list and a list pattern all possible matches are tried one after the other until the first successful match and a successful evaluation of all conditions. Backtracking takes only place if more than one list variable occurs in the list pattern.

The list matching algorithm needs some adaptation to deal with layout properly. There are layout nodes between every consecutive element (or separator) in a list. When constructing a sublist to instantiate a list variable these layout nodes have to be incorporated as well.

Special care is needed when constructing a term containing a list variable. If such list variable is instantiated with a sublist consisting of zero elements, the layout occurring before and/or after this list variable must be adapted to ensure the resulting term is

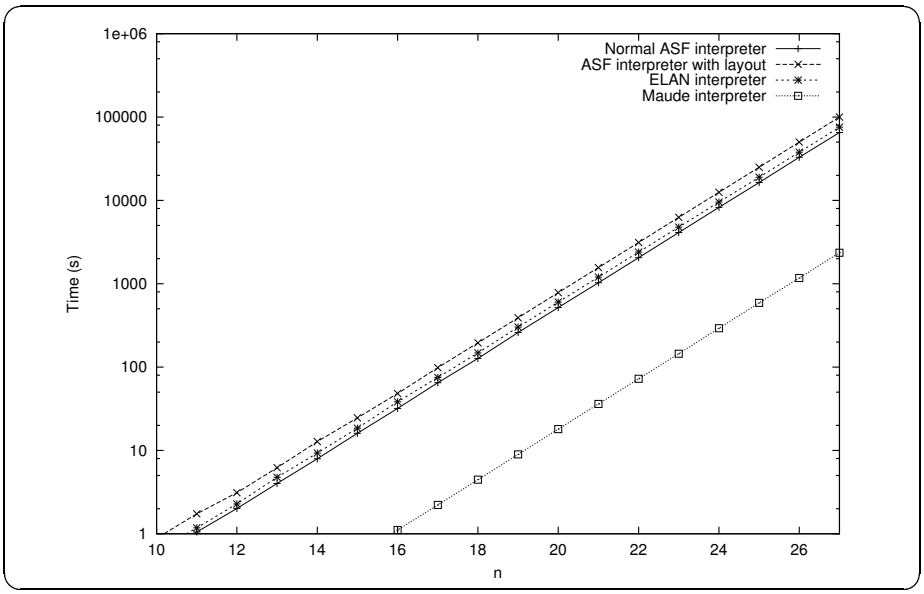

Figure 7.8: Timing results for the `evalsym` benchmark.

well formed with respect to layout again.

Suppose we want to normalize the term `[true; true]` given the specification presented in Figure 7.7. The left-hand side of rule `set-1` matches with this term resulting in the following variable substitutions: *Bools1* = ε, *Bool* = `true`, *Bools2* = ε, *Bool* = `true`, and *Bools3* = ε, where ε represents the empty list. A naive substitution of the variables in the right-hand side of `set-1` would result in: `[ ; true ; ; ]`. The interpreter must dynamically check whether a list variable represents an empty list and decide not to include the redundant separators and layout that occur immediately after it.

Note that it is never necessary for the rewrite engine to invent new layout nodes, it reuses either the layout from the term, or from the right-hand side of an equation. In our example the resulting term will be `[true]`.

7.4 Performance

How much does this rewriting with layout cost? Is this algorithm still applicable in industrial settings? There are three issues which have a negative influence on the performance:

- Matching and equality are more expensive because due to layout nodes the number of nodes roughly doubles in every term.
- The type of each node needs to be inspected to distinguish between layout nodes and normal nodes.

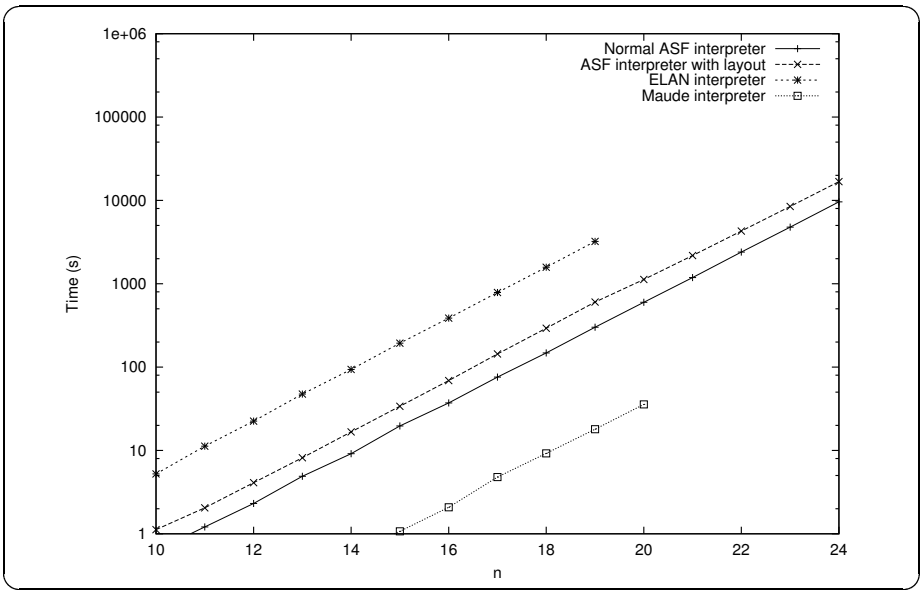

Figure 7.9: Timing results for the `evaltree` benchmark.

- Equality testing is more expensive because tree traversal is needed, with complexity $O(\sharp\text{nodes})$, as opposed to testing for pointer equality which is in $O(1)$.

In order to get insight in the relative performance of rewriting with layout we compare the time and memory usage of the classical ASF interpreter and the layout preserving interpreter. Furthermore, we have run the benchmarks on interpreters of other rule based systems, like ELAN [24] and Maude [58] as well to provide the reader with a better context. Note that for higher execution speed both the ELAN system and the ASF+SDF Meta-Environment also provide compilers, which we do not consider. We have used two simple benchmarks based on the symbolic evaluation of expressions 2^n **mod** 17: `evalsym` and `evaltree`. These benchmarks have been used before for the analysis of compiled rewriting systems in [33]. We reuse them for they isolate the core rewriting algorithm from other language features. All measurements have been performed on a 450 MHz Intel Pentium III with 256 MB of memory and a 500 MB swap disk.

The `evalsym` benchmark The `evalsym` benchmarks computes 2^n **mod** 17 in a memory efficient manner. Its memory complexity is in $O(1)$ for all bench-marked interpreters. From this benchmark we obviously try to learn what the consequences of rewriting with layout are for time efficiency.

The results for this benchmark are in Figure 7.8. The value of n is on the X-axis, and time in seconds on an exponential scale on the Y-axis. The different implementations of rewrite systems all show the same time and memory complexity behavior. Rewriting with layout is a multiplicative factory (1.5) slower. So, we pay a 50% time penalty for

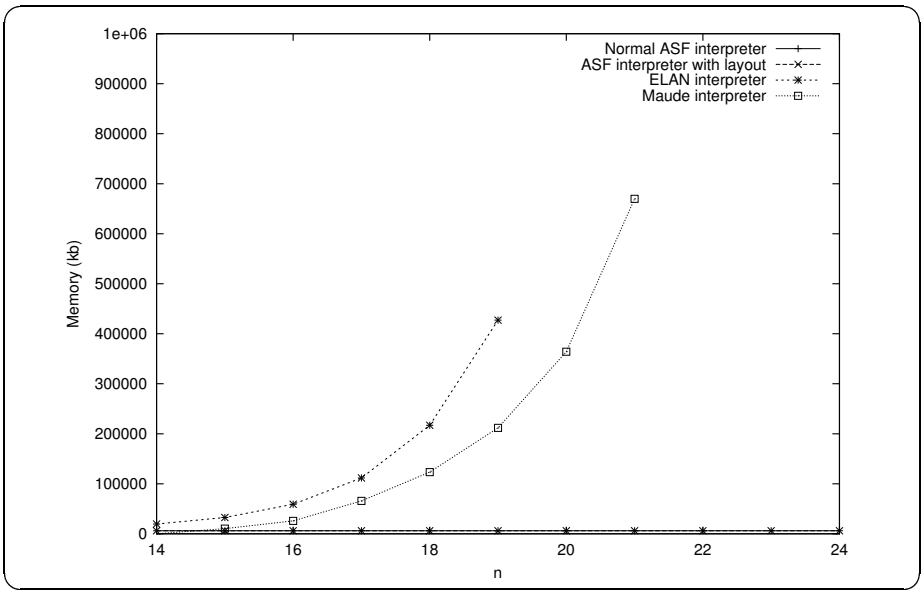

Figure 7.10: Memory profiling results for the `evaltree` benchmark.

rewriting with layout. But this does not change the relative speed to the other systems much. The ASF+SDF system still runs about as fast as the ELAN system.

The `evaltree` benchmark The `evaltree` algorithm generates a huge amount of terms. Real world source code transformations usually involve enormous terms. Therefore scaling up to large terms is an important aspect to source code transformation. So in this benchmark we focus on the space complexity behavior of rewriting with layout.

The results for this benchmark are in Figures 7.9 and 7.10. The ASF+SDF system uses a constant amount of memory, while the other systems show exponential growth in memory usage. This is due to maximal sharing. Obviously, any extra memory allocated for layout is insignificant. Again, we pay a structural 50% time penalty for reducing with layout and the relative speed is not affected significantly.

7.5 Experience

An interpreter for rewriting with layout, as described above, has been in use for several years as the standard ASF interpreter. As such it has been applied to a number of industry sized COBOL renovation applications, for example as described by Veerman [153]. In practice, it was observed that for these applications the speed of the interpreter is acceptable. It scales to large applications, but it could be faster. There is obvious room for improvement, for example by implementing the same algorithm in the ASF compiler.

On the other hand, the conservation of layout was considered nice, but certainly not optimal. Some layout is still lost, which is unacceptable for many applications, especially when source code comments disappear. A method for propagating layout in a selective manner is missing.

Using the same interpreter for completely different application areas also proved to be beneficial, since the readability of output data is far better without the need for a pretty printer. However, for some computationally intensive applications, the overhead for rewriting with layout is too big.

7.6 Conclusions

We have investigated the application of term rewriting to a particular application domain: software renovation. In particular how we can preserve and even investigate whitespace and source code comments while using term rewriting to implement a transformation.

We have investigated the conservation of layout, whitespace and source code comments, during a term rewriting process. A few minimal adaptations were made to the ASF interpreter. It has been applied with success, and appears to be 50% slower.

The resulting rewriting engine has been used in large software renovation applications. We will tackle shortcomings that these applications have identified in Chapter 8.

CHAPTER 8

First Class Layout

The lightweight extension to term rewriting presented in Chapter 7 offers layout conservation during rewriting, while the rewrite rules can still ignore it. However, not 100% of the whitespace and source code comments can be retained, and the technique does not cover software renovation applications that deal explicitly with whitespace or source code comments.

This chapter proposes extensions to ASF+SDF to make "rewriting layout" possible, without sacrificing the ability to ignore it whenever that is appropriate. We introduce two modes of operation: ignoring or utilizing layout. A notation to compose and decompose complex lexical structures in a syntax safe manner is also provided.

We have conducted a case study in which the correctness of source code comments is checked against automatically extracted facts. We have a large embedded software system written in the C programming language, that employs a corporate comment convention for documenting the behavior of each function. A simple analysis of the comments automatically measures the quality of these annotations, and pinpoints the causes of any inconsistencies.[1]

8.1 Introduction

Several case studies into the application of source code transformations have revealed that programmers, and even companies, require that the transformations are *high-fidelity* [165]. This means that the exact set of different characters between the source and target program contains only the intended changes of the transformation. Other unwanted changes would be for example:

- Changes in the layout of the program, missing or added spaces and newlines,
- Disappearing source code comments,
- Normalization of complex expressions to more simple ones, e.g., removal of unnecessary brackets.

[1] This chapter has not been published outside this thesis.

Such changes are unwanted because they enlarge the difference between versions of a program without contributing to any quality attribute: they are simply noise. Moreover, they might violate corporate coding conventions.

Chapter 7 described the basic utility needed for high-fidelity transformations: fully informative parse trees. This solution is now commonly used in ASF+SDF applications, e.g., [153], as well as by other source code transformation systems [169, 165, 13]. However, this is not enough. Layout and source code comments are still lost or changed unintentionally. Much more precision is required to be able to deal with the unexpected effects of source code transformation.

More precision for dealing with layout and source code comments is also required for source code transformations that particularly focus on layout. In this area the layout or comments are the objects of interest instead of the program itself. Examples of this are literate programming tools, such as `javadoc`, and pretty printing tools, such as `indent` and `jindent`. How can we add more precision to ASF+SDF regarding layout to cover both high-fidelity source code transformation and layout transformation?

We list some hypothetical use cases of ASF+SDF that illustrate our intent:

- The layout of a program may be irrelevant for certain applications of ASF+SDF. For example, a documentation generator that transforms a set of programs to a call graph should completely ignore all layout.
- Source code comments are used as documentation, which must not be lost during a software renovation process. For example, an upgrade project that updates Visual Basic applications to the next version.
- Layout or source code comments are to be utilized explicitly during analysis and transformation of source code. For example, when dealing with layout sensitive languages such as Make, or Python, to exploit coding conventions (javadoc), or to generate documentation with generated code.
- Whitespace can be used to analyze, produce or even mimic formatting conventions. For example for authorship analysis of source code, or to implement "pretty printing by example", which can be found in the Eclipse IDE.

In this chapter we will add a generic notation to ASF+SDF that can handle all of the above, and any other unforeseen applications that also deal with layout. As in the previous chapter, we will use the term *layout* to denote both whitespace and source code comments.

The goal of this work is to provide an *object language independent* solution for dealing with layout in source code transformations in a convenient and precise manner. However, to make the following more concrete we first discuss a real-world application in the C programming language.

8.2 Case study: a corporate comment convention

We have studied a component of a large scale embedded software system written in the C programming language. The owner of this code has consistently made use of

```
int unflatten_type_string(
  const char         *type_string,
  const int          length,
  type_description   *desc,
  char               **remaining_type_string)
/* Input(s)  :  type_string
 *                      string width a type description
 *              length
 *                      length of type_string (in bytes)
 * Output(s) :  desc
 *                      parse result of type_string
 *              remaining_type_string
 *                      pointer to first byte not parsed
 *                      NOTE: set parameter to NULL if
 *                      not interested in this value
 * InOut(s)  :  <none>
 * Returns   :  error code
 *              OK:             Function successful
 *              PARAMETER_ERR:  Parameter error
 *              ENCODING_ERR:   Error in type string
 *              MEMORY_ERR:     Memory allocation error
 * Purpose   :  Parse a type string into a structured
 *              type description
 */
{
  ...
  return OK;
}
```

Figure 8.1: An example C function definition with its usage in comments. This function was extracted from an industrial system. The identifiers in the code have been changed to hide the owners identity.

a comment convention to clarify it. Each function is accompanied with a piece of comment to explain its input and output parameters and their typical usage. Also, many formal parameters are qualified with `const` to indicate that their by-reference values are never to be changed. Figure 8.1 shows an example.

Such comments are extremely helpful for code understanding, since the usage of a function may not be immediately clear from the declaration of the formal parameters. This is due to the different intentions a C programmer might have when using a pointer type for a parameter, for instance:

- The parameter is an output variable.
- The parameter is an input value (by reference).
- The parameter is an array of inputs.

- The parameter is an array of outputs.

These intentions are not mutually exclusive, and can be made harder to understand by the use of nested pointer types. The use of the `const` qualifier alleviates this problem, since it documents that the parameter will not change, and the compiler actually asserts this qualification. However, a parameters that does not use the `const` is not always an output parameter. Another complication is that it is not always decided locally, in the function body, which of the above intentions occurs. Pointer values may be passed on to other functions. The comment convention immediately clarifies which of the above is the intended interpretation of a pointer parameter.

Unlike the `const` qualifier, the C compiler can not check the validity of source code comments. So, after the code has been maintained for a while, they might be out-of-date. The same holds for other aspects of the source code, such as parameter checking code and error handling. As the code evolves these aspects might be become inconsistent with corporate conventions, or they may be plainly wrong. This is the disadvantage of idiomatic coding conventions, as opposed to using more strictly enforced built-in language features.

As part of a larger re-engineering effort, a reverse engineering tool was applied to the software system we discussed. This tool computes from the code, among other things, which are the input and output parameters of each function [54]. The goal of this project was to separate common aspects of C functions into separate modules using aspect-oriented programming [97]. Here we will complement this fact extraction tool by also mining the information stored in source code comments.

Formalizing the syntax of comment conventions and extracting information from them would be a natural application of ASF+SDF. We can first define the syntax of a convention in SDF, and then extract the necessary information using ASF. In order to be able to write the extractor, we will need to be able to access layout explicitly. The following sections first describe how we generalize the treatment of layout. Application to the above case is presented in Section 8.8

8.3 Requirements of first class layout

The general idea is that layout will be parsed and rewritten like any other program structure. However, layout is still exceptional as compared to other program structures for two reasons. Firstly, it must be ignored when we are not interested in it. For example, in Figure 8.1 we are interested in the comment between a functions header and its body, but we want to ignore the particular layout of the parameters.

Secondly, an overall consideration is to have *syntax safe* analysis and transformation. A syntax safe program statically guarantees that both its input and output are defined by a context-free grammar. For grammatical reasons layout is usually defined using a lexical syntax definition. This is either for parser efficiency, or to prevent nested comments, or for dealing with other kinds of syntactical complexities. For the sake of simplicity lexical syntax is traditionally represented by flat lists of characters (tokens) in source code transformation systems such as ASF+SDF and StrategoXT. In these systems, the programmer can use arbitrary string matching and string construc-

tion functionality to analyze or transform the tokens of a language. The result is loss of precision and loss of static syntax safety.

The above two issues lead to the following requirements:

R1 We should provide full syntax safety and precision for dealing with layout, even if layout is defined using lexical syntax. We could then guarantee for example that each C comment starts with `/*` and ends with `*/`.

R2 The programmer should to be able to choose between *ignoring* layout, and *utilizing* layout. This would allow her to abstract from irrelevant details, and still allow to focus on layout when necessary. Within a single application, *ignoring* and *utilizing* layout may be used in concert.

R3 When *ignoring* layout, we want to either *discard* it completely for efficiency, or *preserve* it with as much automation as possible. In other words, for reasonably high-fidelity transformations, ASF+SDF may implicitly preserve layout during a transformation that is ignored by the programmer.

R4 When *utilizing* layout, the programmer should be able to explicitly *consume*, *produce*, or *propagate* it. She can consume layout in order to analyze it, and produce it as part of a transformation. Propagating layout, by consuming it from the input and then reproducing it somewhere in the output can be used to make full high-fidelity transformations.

Three aspects need to be tackled in order to make layout a "first class citizen" that meets the above requirements:

Run time environment: How to represent layout in a structured and syntax safe manner in the run-time environment of ASF+SDF (**R1**)? How to represent the difference between ignoring layout and explicit utilization of layout (**R2**)?

Syntax: What is the ASF+SDF notation for first class layout (**R4**)? How to differentiate on a notational level between ignoring layout and utilizing layout (**R2**)?

Compilation: How to parse the new notation and how to map it to the internal representation? How to to guarantee syntax safety all the way (**R1**). How to deal with implicit layout preservation **R3**?

To provide a background for the following, we include an overview of the implementation of ASF+SDF in Figure 8.2. For each source code transformation both a parser and a rewriter are generated from an ASF+SDF description. Generating the rewriter takes an extra step through the SDF parse generator because the ASF equations use concrete syntax. There are two ways for obtaining a rewriting engine, either directly interpreting ASF, or first compiling ASF to a dedicated rewriter. We have included an ASF normalizer in this picture, which will be the main vehicle for implementing first class layout in ASF+SDF.

We first describe fully structured lexicals, then how to deal with the difference between ignoring and utilizing layout and finish with the changes in the type system of ASF+SDF needed to obtain static syntax safety.

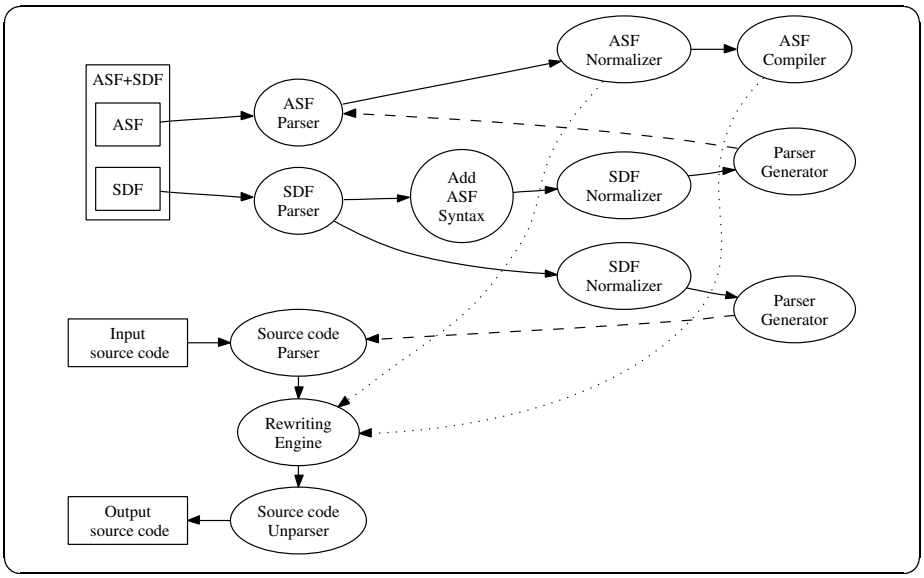

Figure 8.2: The "language parametric" architecture of ASF+SDF.

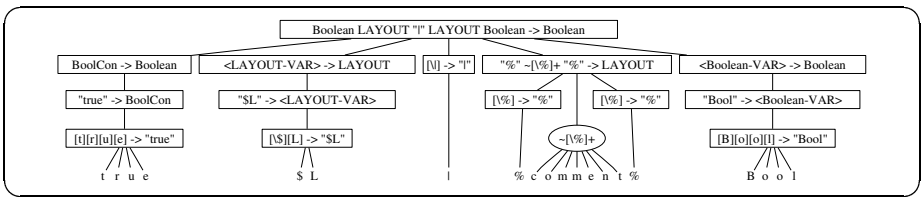

Figure 8.3: A fully structured parse tree of a string containing both a layout variable and a source code comment.

8.4 Fully structured lexicals

To fulfill the requirements of precision and syntax safety, we propose to have to have a fully structured representation of lexical syntax. Remember that most layout is defined using lexical syntax in practice. This feature appears straightforward, but a number of issues arise on both the notational level and the type-system level of ASF+SDF.

8.4.1 Run time environment

We define the syntax of layout using production rules in SDF. There is no principal difference between lexical productions, layout productions, and context-free productions in SDF: they are all grammar productions. Except, the symbols of each context-free production are interleaved with the **LAYOUT** non-terminal (See Section 7.1).

Any syntax production in SDF can be either a lexical, context-free or variable pro-

```
lexical syntax
  "/*" ( ~[\*] | Star )* "*/"    -> LAYOUT
  [\*]                           -> Star
  [\t\n\r\ ]                     -> LAYOUT
lexical restrictions
  Star -/- [\/]
context-free restrictions
  LAYOUT? -/- [\t\n\r\ ]
lexical variables
  "L"[0-9]*         -> LAYOUT
  "Contents"[0-9]* -> ( ~[\*] | Star )*
```

Figure 8.4: The whitespace and comment conventions of C, with meta variables.

duction, and either layout or non-layout, which amounts to six types of productions that might occur in a parse tree. These types have no different semantics at parse time, but might be processed differently at rewriting time. The parse trees produced by the parser naturally contain the full structure of a program, with details down to the structure of lexical syntax, and thus also the structure of source code comments and whitespace. For example, Figure 8.3 contains a parse tree of `%comment%`, structured as a list of characters that are not a % sign, surrounded by two % signs.

8.4.2 Syntax

We will now define a notation for the parse tree structures described above. To define them why not use their *concrete syntax*? This is not feasible for syntactical reasons. Layout syntax, or any lexical syntax in general, usually does not allow straightforward decomposition, which will become clear by the following example.

In reality, layout definitions can be quite complex. As a running example, Figure 8.4 defines the whitespace and comment conventions of C unambiguously and according to the ANSI standard. The lexical restriction prohibits the nesting of `*/` inside a comment. It declares that `Star` may not be followed by a `/`. The context-free restriction denotes a *longest and first match* on optional layout. Together with the production rules, these disambiguation rules encode the ANSI C standard. We have added some meta variables that will be used later to define patterns over this syntax.

This real world example serves to show the complex nature of layout syntax. Simply having concrete syntax as a notation for layout patterns is not the answer. Imagine a layout variable that matches the internals of a C comment using concrete syntax: `/*␣Contents␣*/`. We intend to deconstruct the comment into the opening and closing braces, and the contents. It is unclear whether the spaces around `Contents` are to be taken literally. Do we mean only comments that have them match this pattern, or are they to be ignored? Do we want to match the C comment that has the word `Comment` in it literally, or is `Comment` a meta variable? It is even possible that a language enforces disambiguation rules that make the definition of variable syntax completely impossible. The reason is that any “longest match” disambiguation contradicts decon-

```
equations
  [remove-any-newlines-after-if]
    if L1 layout(\n) L2 (Expression) Statement =
    if L1 L2 (Expression) Statement

  [break-long-comments-and-introduce-stars]
    width(layout(/* Contents1 */)) > 72 == true
    ==========================================
    layout(/* Contents1 Contents2 */) =
    layout(/* Contents1 \n star(\*) Contents2) */
```

Figure 8.5: Example equations using prefix notation for matching and constructing layout nodes.

struction.

This discussion is valid for any lexical syntax definition. Decomposing lexical or layout syntax using concrete syntax is not possible in general, because lexical syntax does not naturally decompose into mutually exclusive syntactic entities. Adding syntax for variables to be able to write patterns even further complicates a grammar, and its ambiguities.

If we let go of concrete syntax we can provide a less elegant, but nevertheless simple declarative notation. Figure 8.5 shows examples of a notation that will serve our purposes. This prefix notation with escaped characters will be *syntactic sugar* for concrete syntax patterns. Each pattern written in prefix notation, corresponds to exactly one hypothetical pattern in concrete notation. This one-to-one correspondence is the vehicle for ensuring syntax safety, as will be become apparent in the following section.

Take for example the equations shown in Figure 8.5. For the purpose of presentation we will ignore all spaces in this example, and consider only the readable notation. We will deal with the difference between ignoring and utilizing layout later. The first equation in Figure 8.5 matches any C conditional that has a newline somewhere in between the `if` keyword and the open parenthesis, and subsequently removes this newline. The second equation uses list matching to split up the characters in a C comment in two arbitrary non-empty parts. When the width of the first part can be longer than 72 characters, we introduce a newline, a space and a `*` in between the two parts. The definition of the `width` function is not shown here.

8.4.3 Compilation

We will parse the prefix syntax that example 8.4 introduced, then translate the resulting parse trees to obtain parse trees that correspond to concrete layout patterns. To remain syntax safe, care must be taken that the prefix notation is *isomorphic* to the parse tree that is computed from it. If a prefix pattern does not contain any variables, it corresponds exactly to a concrete pattern. Parsing the prefix pattern and translating it, should yield the parse tree of the corresponding concrete sentence as if produced by the parser. Figure 8.6 displays how we obtain this isomorphism:

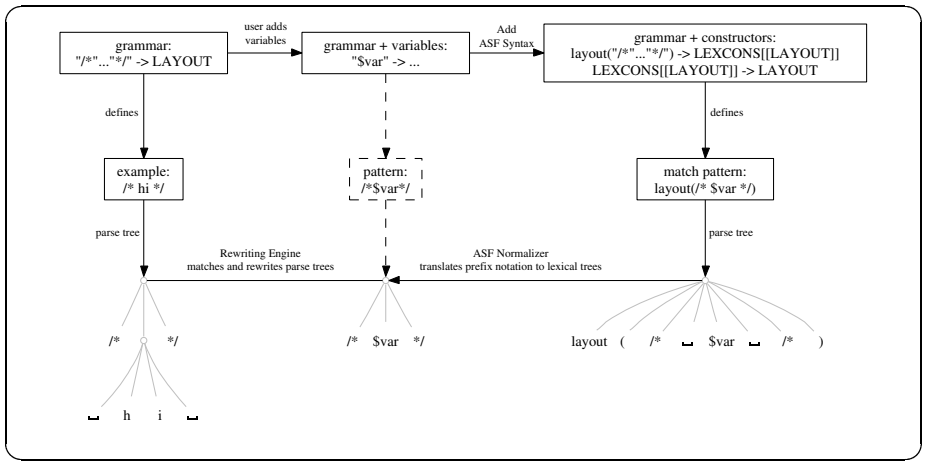

Figure 8.6: A sketch of the introduction of prefix layout notation on the grammar level, and the mapping back to original layout on the parse tree level.

1. The user inputs a grammar, including the definitions of lexical and layout syntax. He then extends the grammar with syntax for variables that range over the syntactic categories (non-terminals) in the grammar (Figure 8.4).

2. The Add ASF Syntax tool generates a fresh context-free production for each lexical production, and replaces each lexical variable by a context-free variable. The idea is to generate a context-free grammar that mimics the exact structure of the original lexical grammar. Figure 8.7 contains an example. The tool follows the following steps:

   - The left-hand sides of all lexical productions `L -> R` are prefixed with a name and surrounded by brackets. The result is `r "(" L ")" -> R`. The name, `r`, is automatically generated from the original non-terminal name, by replacing all uppercase characters by lowercase characters.
   - All occurrences of any lexical non-terminal `N` are renamed by wrapping it with a parameterized symbol to obtain `LEXCONS[[N]]`.
   - For every wrapped non-terminal `N` a production is added: `LEXCONS[[N]] -> N`.
   - For every character class `C` that was wrapped by `LEXCONS` a new production is added: `CHARACTER -> LEXCONS[[C]]`.
   - The resulting productions are put in a `context-free syntax` section. Note that the effect of this is that the **LAYOUT** will be inserted in the left-hand sides of the generated productions.
   - For every lexical variable that ranges over a character class C, its right-hand side is changed to `LEXVAR[[C]]`, and a production is added: `LEXVAR[[C]] -> CHARACTER`.

```
context-free syntax
  "layout"  "(" "/*"
                (LEXCONS[[~[\*]]] | LEXCONS[[Star]])*
                "*/"                ")"     -> LEXCONS[[LAYOUT]]
  "star"    "(" LEXCONS[[[\*]      ]] ")" -> LEXCONS[[Star  ]]
  "layout"  "(" LEXCONS[[[\t\n\r\ ]]] ")" -> LEXCONS[[LAYOUT]]
context-free syntax
  CHARACTER          -> LEXCONS[[~[]       ]]
  CHARACTER          -> LEXCONS[[[\*]      ]]
  CHARACTER          -> LEXCONS[[[\t\n\r\ ]]]
  LEXCONS[[LAYOUT]] -> LAYOUT
  LEXCONS[[Star   ]] -> Star
variables
  "L"[0-9]*        -> LEXCONS[[LAYOUT]]
  "Contents"[0-9]* -> (LEXCONS[[ ~[\*]]]) | LEXCONS[[Star]] )*
context-free syntax
  LEXVAR[[[\t\n\r\ ]]] -> CHARACTER
  LEXVAR[[~[\*]]]      -> CHARACTER
```

Figure 8.7: The syntax defined in Figure 8.4 is embedded in a prefix notation, preserving all non-terminals by embedding them in the parameterized non-terminal `LEXCONS`.

- Every lexical variable that does not range over a character class, but over a non-terminal `N`, is changed to range over `LEXCONS[[N]]`
- The resulting variable productions are put in a context-free variables section.

3. The resulting syntax definition can be used to generate a parser for ASF equations that use the prefix syntax.

4. The counterpart of step 2. The resulting parse trees are translated to remove all syntactic sugar that was introduced by the "Add ASF Syntax tool". This is done by the "ASF Normalizer" tool. It implements exactly the inverse of the operations that have been applied in step *2*, in order to obtain parse trees over lexical productions. It throws away the **LAYOUT** trees, removes the prefix name and the brackets, restores all original non-terminal names by removing the `LEXCONS` and `LEXVARS` wrappers. Finally, it replaces parse trees over the `CHARACTER` non-terminal by the actual characters these trees represent. The resulting parse trees correspond exactly to parse trees over the original lexical productions that occurred in the user defined syntax.

5. A normal rewriter can now be generated which matches terms that have been parsed directly using the original grammar with the patterns that have been produced via the route described above.

An important difference with concrete syntax is that all characters will have visible notations instead of invisible ones. This syntax is defined by the generic `CHARACTER`

```
lexical syntax
  [A-Za-z0-9\-] -> Dashed
  [A-Za-z0-9\_] -> Underscored
lexical variables
  "Common" -> [A-Za-z0-9]
context-free syntax
  translate(Dashed) -> Underscored
equations
[A] translate(dashed(Common)) = underscored(Common)
[B] translate(dashed(\-))     = underscored(\_)
```

Figure 8.8: An example ASF+SDF specification that employs lexical constructor functions and sub-typing of character classes.

non-terminal. For example, a newline will be written as `\n`. As a result the invisible characters can be used again to decompose a pattern into parts, which was not possible in the concrete syntax notation (see Section 8.6.2). The ASF normalizer tool takes care of replacing the visible notation with the actual characters (Step 4).

In the current first order type system of ASF+SDF, fully structured lexical syntax is very restrictive. It does not allow to transport characters from one class into the other. For example, the first occurrence in meta variable *Common* in Figure 8.8 is not allowed since its type is `[A-Za-z0-9]`, and not `[A-Za-z0-9\-]`. The "Add ASF Syntax" tool circumvents this limitation by changing the non-terminals that lexical variables range over. If a lexical variable ranges over a character class, a new non-terminal is introduced (see Step 2), and this non-terminal is accepted by `CHARACTER`. As a result, lexical variables that range over character classes directly, are accepted at all places where `CHARACTER` is accepted. Now a lexical variable can be used to transport single characters from one place to the other regardless of the character class they belong to. As a result, syntax safety is no longer guaranteed, but at least the structure down the individual characters is isomorphic to the original characters. Now, we will design a simple type system that can statically verify syntax safety for the individual characters too.

8.5 Type checking for syntax safety

We relaxed the generated ASF+SDF grammars such that a variables ranging over character classes where accepted by every character class. This freedom must now be slightly restricted to guarantee syntax safety.

Previous implementations of ASF+SDF, that employed flattened tokens, did not guarantee syntax safety on the lexical level. Checking syntax safety would have required a general parsing algorithm at rewriting time to recover the structure of lexicals, which would have introduced both an efficiency bottleneck and unwanted tangling of parsing and rewriting functionality.

8.5.1 Type checking

We will use a partial ordering on character classes. When a character class A is equal to or less than another class B, we will accept variables that range over A at locations where B occurs. Any other derivations are type incorrect. Any A that is not less than or equal to B will not be accepted at B locations. Since character classes can be represented by *sets* of characters, we can use the subset relation to define the partial ordering: $A \leq B \Leftrightarrow \text{chars}(A) \subseteq \text{chars}(B)$, where $\text{chars}(C)$ produces the set of characters that a character class C represents. In other words, A is a subtype of B when all characters of A are in B.

Now, the example in Figure 8.8 is type correct, since `[A-Za-z0-9]` is a subset of both `[A-Za-z0-9\-]` and `[A-Za-z0-9\_]`. The sub-typing on character classes ensures syntax safety, and still allows for carrying characters from one class to another. When translating between character classes, the programmer will have to be just precise enough to satisfy the sub-typing relation.

This type checking can be done by the ASF normalizer. For all characters and variables that range over character classes, the above sub typing relation should be checked. Any violation should be reported as an error. It detects character class violations in closed patterns, and check sub typing violations for variables that range over character classes. Since the structure of the lexical trees is specific down to the character level, only very localized and efficient character class inclusion checks are necessary.

8.5.2 Matching

The matching algorithm must also take the sub-typing relation into account. It should guarantee, at rewriting time, that never a character is bound to a variable that ranges over a class that does not contain the character. For the example in Figure 8.8, this means that equation `[A]` should not match with `translate(-)`, because - is not in the `[A-Za-z0-9]` class that *`Common`* ranges over.

We implement this test by automatically introducing a condition for every character class variable. The condition will dynamically check the if a character is an element of the character class. This is another feature of the ASF normalizer. If all equations guarantee that all variable bindings are correct in this above manner, then all ASF+SDF programs can be statically guaranteed to be syntax safe.

8.6 Ignoring layout

After dealing with structured lexicals and syntax safety, our second requirement is to be able to distinguish between ignoring layout and utilizing layout.

8.6.1 Run time environment

For ignoring layout during term rewriting, but preserving it anyhow, the solution is to automatically introduce *meta variables* in the parse trees of matching and constructing

```
context-free syntax
  "_" "_"       -> MyNonTerminal {layout-bracket}
variables
  "Exp"         -> Expression
  "Stat"[0-9]* -> Statement
lexical variables
  "L"[0-9]*     -> LAYOUT
equations
[A] if (!Exp) Stat1 else Stat2 =
    if ( Exp) Stat2 else Stat1

[B] if (!Exp) L1 Stat1 L2 else L3 Stat2 =
    if (Exp ) L3 Stat2 L2 else L1 Stat1

[C] if (Exp) layout(\n) Stat =
    if (Exp) layout(\ ) Stat

[D] _ layout(\t) _ = _ layout(\ ) layout(\ ) _
```

Figure 8.9: Examples of differentiating between ignoring and utilizing layout.

patterns. The previous solution (Chapter 7) extended the matching machinery to know the difference between layout and normal parse trees. This is not necessary anymore. We will have meta variables at the points where the programmer indicated to ignore layout.

These layout variables can be either introduced automatically, or put there by the programmer. She can now *utilize* layout like any other information, *consuming* it by matching layout trees and *producing* it by constructing layout trees. *Propagating* layout is done by matching layout variables, and using them to construct new source code patterns. *Ignoring* layout is taken care of by automatically introduced variables.

Note that equality testing modulo layout is still done by the procedure used in Chapter 7. This algorithm still knows about the difference between layout parse trees and other parse trees. What remains to be designed is a notation that corresponds to the above described representation, and a compiler to map this notation to the representation.

8.6.2 Syntax

In order to distinguish between layout that must be ignored from layout that is to be matched or constructed, we must introduce a syntactic notation. This notation will be used to separate the interesting locations of layout from the uninteresting ones. We propose to use the following rules for separating ignored layout from utilized layout, which are demonstrated in Figure 8.9:

- Layout in concrete syntax is to be ignored. For example, equation `[A]` contains only spaces. It swaps the branches of any negated C conditional regardless of

what layout it has.

- Meta variables ranging over layout will not be ignored. For example, equation `[B]` mimics `[A]` and also transports some of the layout. Note that the spaces around each layout variable are ignored, `L1` matches *all* layout between the closing bracket and the statement body.
- Layout expressed using *lexical constructors* will not be ignored. For example, equation `[C]` matches only on C conditionals that have a *single* newline between the closing bracket and the statement body. Any other layout of the conditional is ignored.
- Layout wrapped by *layout-bracket* productions will not be ignored. A production with the `layout-bracket` attribute can be used to express equations directly over layout without any context. For example, equation `[D]` will rewrite all tabs to two spaces.

The first rule allows for backward compatibility. It is also a convenient and minimal notation when not interested in layout. The second and third rules are for convenience, since a user that uses lexical constructors or uses layout variables is implicitly interested in it. The fourth rule is necessary for completeness. If the above three implicit convenience rules do not apply, the user can always introduce an explicit notation for not ignoring layout. She does this by defining a production with the `layout-bracket` attribute, and using this notation to identify layout of interest.

8.6.3 Compilation

The compilation of the above notation is straightforward and will be located in the ASF normalizer. All patterns in a specification can be divided into either matching patterns, or constructing patterns. For example, the left-hand side of an equation is a matching pattern, while the right-hand side is a constructing pattern.

- In all matching patterns, all layout trees that contain only concrete syntax, and no layout variables, lexical constructors, or layout brackets, will be replaced by a fresh variable ranging over **LAYOUT**.
- Similarly, in all constructing patterns we will replace layout trees that contain only concrete syntax by a default parse tree of a space. Or, for higher fidelity, we can reuse layout variables introduced in matching patterns using any kind of heuristics.
- Matching and constructing patterns that contain layout variables, layout prefix notation, or layout brackets will remain. Any layout in between any of those classes of trees is removed.
- Applications of layout bracket production are removed from the parse trees before rewriting. Any "bracketed" tree is replaced by the child of the bracketed tree. This is analogous to normal bracket productions for expression grammars [32].

To summarize, we generate fresh meta variables where the layout is ignored by the programmer, and use the layout patterns that the programmer specified otherwise.

8.7 Summary

This concludes the description of first-class layout. Firstly, lexical syntax is now fully structured in ASF+SDF. We introduced a prefix notation for constructing and deconstructing lexical trees. To obtain full syntax safety, the type system and matching algorithms of ASF+SDF were extended to accept sub typing between character classes. Secondly, the difference between ignoring and utilizing layout is now made explicit by the programmer, using layout variables, lexical constructors or layout brackets. The key design decision was to make *all* information that is present in parse trees available to the programmer, on demand. We can now apply standard term rewriting idioms for analyzing the C source code of our case study.

8.8 Case study revisited

As described in Section 8.2, we have a medium sized C system to analyze. Each function should document its input and output parameters using comments. A prototype tool for extracting the input and output parameters for each function from the actual code is available [54]. It recognizes an output parameter if somewhere in the (interprocedural) control-flow of a function a new value could be stored at the address that the parameter points to. The other parameters default to input parameters according to the tool. There are three questions we would like to see answered by this case study:

Q1 Can we effectively extract information from the comments using lexical constructor functions in ASF+SDF?

Q2 Are the source code comments in this particular system up-to-date with respect to coding conventions and the actual state of the code?

Q3 Is the automated extraction of input and output parameters from the source code correct?

8.8.1 Extracting information from the comments

We first defined an SDF definition of the comment convention. By inspecting a number of example functions we reverse engineered the syntax. It is depicted in Figure 8.10. In this definition we add an alternative grammar for comments, generated by the nonterminal `IO-Comment`, that is more precise than the general C comment. Because this more specialized grammar obviously overlaps with general C comments, we use the SDF attribute {`prefer`} to choose for the more specialized comment whenever possible (see Chapter 3).

An ASF specification will now accumulate the following information, using a traversal function (see Chapter 6):

- Which functions have `IO-Comments` between their header and their body, which have comments not recognized as `IO-Comments`, and which have no comment at all (Figure 8.11).
- If an IO-Comment is found between a function, lists of input, output and in/out parameters.

The division into three classes of functions allows us to evaluate the quality of the syntax definition for `IO-Comments`. When many functions are commented, but the comments can not be recognized as `IO-Comments` the definition must be too precise. We went through a number of grammar adaptations before the number of unrecognized comments dropped below 1% of the total number of functions. We then adapted the comments that were left unrecognized manually to enforce that they obey the layout convention. Figure 8.11 shows how the extraction is implemented. The traversal function `divide-functions` visits all C function definitions. Each equation matches a class of functions. The first matches functions that have `IO-Comments` between the declarator and the functions body. The second equation matches functions that have normal comments at the same location. The last equation stores the remaining functions. The three equations each update a store that contains three sets of function names.

Then we mined the `IO-Comment` trees for more specific information. By traversing the trees and using pattern matching we extracted more basic facts. For example, an input parameter can be identified by using the lexical constructor function for the `Inputs` and `Params` non-terminals we defined (Figure 8.10). We applied the same traversal scheme used before to divide functions into classes, to divide the parameters of each function into three classes.

8.8.2 Comparing the comments with extracted facts

We show both the information extracted by the ASF+SDF specification, and the data from the aforementioned reverse engineering tool in Table 8.1. The "Commented" column displays information that was extracted from the source code comments, while the "Coded" column displays information that was extracted from the actual code. The upper part of the table summarizes the basic facts that have been extracted. It appears that 45% of the parameters in the studied component have been commented.

We have obtained two independent sources of the same information. The opportunity arises to check the comments that have been put in manually, with the data that was extracted from the source code. The comments might be wrong, as well as the reverse engineering tool, or our comment extractor. By comparing the two sets of data, a relatively small number of inconsistencies will appear that need to be checked and fixed manually.

```
lexical syntax
  IO-Comment -> LAYOUT       {prefer}

lexical syntax
  "/*" {WS "*"}* Definition+ "*/" -> IO-Comment

  Inputs          -> Definition
  Outputs         -> Definition
  InOuts          -> Definition
  Returns         -> Definition
  Purpose         -> Definition
  Notes           -> Definition

  "Input(s)"  WS  ":" WS Params   -> Inputs
  "Output(s)" WS  ":" WS Params   -> Outputs
  "InOut(s)"  WS  ":" WS Params   -> InOuts
  "Returns"   WS  ":" Description -> Returns
  "Purpose"   WS  ":" Description -> Purpose
  "Notes"     WS  ":" Description -> Notes

  Param Line ([\ ]*[\t][\t] Param Line)*  -> Params
  Param {WS "*"}* -> Params

  [A-Za-z\_][A-Za-z0-9\_]*        -> Param
  "<none>"                        -> Param

  WS "*" [\t][\t][\t] Description -> Line
  (Word | WS)* "*"                -> Line

  [\t\ \n]+       -> WS
  ~[\*\ \t\n]+    -> Word
  [\*]            -> Star

  Star            -> Part
  Word            -> Part
  WS              -> Part
  Part+           -> Description

lexical restrictions
  Part  -/- [\/]
  Word  -/- ~[\*\t\ \n]
  WS    -/- [\t\ ]
```

Figure 8.10: Syntax definition of a comment convention. An `IO-Comment` consists of a list of definitions. Each definition starts with a certain keyword, and then a list of parameters with a description. The syntax uses tab characters to separate which words are parameters and which are words describing a parameter.

```
context-free syntax
  Specifier* Declarator Declaration*
    "{" Declaration* Statement* "}" -> FunctionDefinition

equations
[store-recognized-io-comments]
  divide-functions(
  Specifiers Declarator
  L1 layout(IO-Comment) L2
  { Decls Stats },
  Store)
  =
  store-elem(RecognizedComments,get-id(Declarator),Store)

[store-not-recognized-comments]
  divide-functions(
  Specifiers Declarator
  L1 layout(Comment) L2
  { Decls Stats },
  Store)
  =
  store-elem(UnRecognizedComments,get-id(Declarator),Store)

[default-store-uncommented-functions]
  divide-functions(
  Specifiers Declarator { Decls Stats },
  Store)
  =
  store-elem(NotCommented,get-id(Declarator),Store)
```

Figure 8.11: The grammar production that defines C function definitions, and a function **divide-functions**, that divides all function into three classes: properly commented, commented, and not commented. The **store-elem** function, which is not shown here, stores elements in named sets. **get-id** retrieves the name of a function from a declarator.

Basic facts	**Commented**	**Coded**	**Ratio (%)**
Lines of code	—	20994	—
Functions	92	173	53%
. . . . . Void functions	18	21	85%
. . . . . Compared functions	74	152	49%
Input parameters	124	304	41%
Output parameters	95	190	50%
All parameters	219	494	45%

Raw inconsistencies	**Number**	**Ratio (%)**	**of basic facts**
Not coded inputs	21	17%	commented inputs
Not commented inputs	5	2%	coded inputs
Not coded outputs	6	6%	commented outputs
Not commented outputs	25	12%	coded outputs

Raw summary	**Number**	**Ratio (%)**	**of basic facts**
Inconsistencies	57	25%	commented parameters
Inconsistent parameters	33	14%	commented parameters
Inconsistent functions	24	32%	commented funcs

Refined inconsistencies	**Number**	**Ratio (%)**	**of raw inconsistencies**
Commented inputs as outputs	3	60%	not commented inputs
Code has added inputs	2	40%	not commented inputs
Commented outputs as inputs	21	84%	not commented outputs
Code has added outputs	4	16%	not commented outputs
Commented inputs as outputs	3	50%	not coded outputs
Code has less outputs	3	50%	not coded outputs
Com. outputs as inputs	21	100%	not coded inputs
Code has less inputs	0	0%	not coded inputs

Refined summary	**Number**	**Ratio (%)**	**of raw inconsistencies**
Input versus output params	24	73%	inconsistent parameters
. . . . . per function	20	83%	inconsistent funcs
Added or removed params	9	27%	inconsistent parameters
. . . . . per function	3	15%	inconsistent funcs
Functions with both issues	1	2%	inconsistent funcs

Table 8.1: Automatically extracted and correlated facts from the code and comments of a large software system show that possibly 32% of the functions have comments that are inconsistent. For 73% of the wrongly commented parameters the cause is that they swapped role from input to output or vice versa. The other 27% can be attributed to evolution of either the code or the comments.

We use a relational calculator to compute the differences between the two data sets [101]. The second part of Table 8.1 summarizes the differences between commented and coded facts. For example, we have 21 parameters commented as input parameters, that do not appear to be input parameters according to the extracted facts from the code. In total, there are 57 of such inconsistencies caused by 33 parameters used by 24 different functions.

The above information does not give a clue about the cause of these inconsistencies. A further analysis using relational calculus offers some insight. The last part of Table 8.1 summarizes the results. The first block denotes what happened to parameters that appear to be coded, but not commented. The second block denotes parameters that are coded, but not commented. The final block summarizes the measurements per parameter and per function. By intersecting the different sets we compute that 24 of the parameters have been assigned the wrong role in the comments. They either should have been outputs and are said to be inputs (21), or vice versa (3). The other errors (9) are due to newly added, renamed or removed parameters which was not reflected in the comments.

We first assumed that if both the comments and the source code agree on the role of a parameter, they are correct. This absolves us from checking $219 - 33 = 186$ parameters. Then, we checked the inconsistencies manually by browsing the source code. We have checked all 24 parameters that were reported inconsistent. More than half of these errors lead to a single cause in the system: one very big recursive function. The comments declare several input parameters which are not guaranteed by all execution paths of this function not be output parameters. It is very hard to check manually why this is the case. Every function that calls this complex function, and passes a pointer parameter, inherits this complexity.

There were some relatively easy to understand errors too. For example, parameter names that have been removed, added or changed without updating the comments. One interesting case is a utility function that is called with a boolean flag to guide the control-flow. If the flag is set to false, the function is guaranteed not the use a certain parameter as output. This is something the reverse engineering tool does not take into account. The use of a boolean flag to guide control flow can indicate that the programmer could also have split the function into several separate functions.

Note that we did not arrive immediately at the results of Table 8.1. First we had twice as much inconsistencies. We started at the top of the list to check them manually, and concluded that some of the extracted facts were wrong. We removed some minor bugs from the extraction tool, and arrived at the current results in six cycles. All inconsistencies have now been checked manually. Considering that we spent about an hour on 24 out of 494 parameters, we have saved about 20 hours of highly error-prone labor by automating the process.

The conclusion is that the quality of the comments is surprisingly high in this software system. Although the comments have never been validated before, still most of the parameters have been commented correctly ($100\% - 14\% = 86\%$). Many of the functions that have comments are correct ($100\% - 32\% = 68\%$), and these comments aid enormously in code understanding. Moreover, they have aided in improving the reverse engineering tool that we used to compute facts from the code. From this case study we learn that neither information from source code comments, nor auto-

matically extracted "facts" from source code can be trusted. Still, they can both be valuable sources of information: instead of having to check 219 parameters, we have now checked only 33 and automated the rest of the checking process.

8.8.3 Case study summary

The answer to **Q1**, can we efficiently extract information from the source code comments using ASF+SDF, can be answered positively. The larger effort was in defining the syntax of the convention. The extraction using traversal and lexical constructors functions was trivial after that. The comments are not up-to-date with respect to the source code, namely 32% is wrong. This answers the second question, **Q2**. The automated extraction tool went through a number of improvements, which is an answer to **Q3**.

8.9 Discussion

ASF+SDF language design

Contrary to the lightweight experiments described in Chapter 7, in this chapter we have rigorously changed the ASF+SDF language and its implementation. In my opinion, this was necessary for obtaining a conceptually clean result. We can now add the following properties to the description of ASF+SDF:

- All structure and type information defined in SDF is used and available in ASF, including the full structure of lexical syntax, whitespace and source code comments.
- All ASF+SDF specifications are guaranteed to be fully syntax safe.

However, the syntax safety notion is not strong enough. We defined it to be: the input and output of an ASF term rewriting engine conforms to *an* SDF syntax definition. In particular syntax definitions in SDF are modular and can thus be combined to form complex constellations. The syntax safety concept does not separate input from output languages. The notion of *completeness* of a transformation with respect to a certain domain and range language seems important, and requires further investigation.

Another issue is the limitation of a single well known non-terminal for layout: **LAYOUT**. This implies that languages that are combined in a modular fashion must share this non-terminal. ASF+SDF should allow several layout non-terminals, such that languages can be composed without unnecessary clashes. For our case study we had to change the layout definition of the fact data structure from an otherwise reusable library module, only to prevent ambiguities with the C language conventions.

ASF+SDF normalization

The introduction of an ASF normalizer in the architecture offers possibilities for changing ASF programs before they are compiled or executed in a back-end neutral fashion.

For example, it removes the syntactic sugar of constructor functions, and adds equation conditions to check character class inclusion.

Other instrumentation of ASF term rewriting systems can be easily plugged in at this point. For example, origin tracking [70] can be implemented by adding conditions. Origin tracking techniques could also be applied to automate implicit propagation of layout nodes from the left-hand side to the right-hand side of equations.

Related work

Fully informative parse trees, or extended abstract syntax trees are currently the most popular vehicle for high-fidelity transformations [165, 169, 107, 13]. Some systems take the approach of source code markup [123, 142, 165]. With these techniques syntax trees are not the principle vehicle for storing information, but rather the source code itself. We will shortly describe related approaches. A general observation is that static syntax safety is a feature that seems to be unique to ASF+SDF.

A common idiom in the TXL language is to store intermediate analysis results in the source text directly using markup [123]. Parsing may be used to obtain the markup, but syntax trees are not used as the vehicle for transporting the resulting information. The close relation to the original source text that is remained by this design choice also offers the opportunity for high-fidelity transformations. The concept of source code *factors* is important. Factors are special kind of markup designed to directly log origin information of past transformations into the source code. For example, C preprocessor expansion can be stored using source code factors, such that later the original code from before macro expansion can be recovered. The techniques explained in this chapter can be applied to systems that use source code markup, to make them syntax safe, and to let the programmer ignore factors that he is not interested in.

Scaffolding [142] is a method that combines source code markup with the parse tree approach. A grammar is extended with syntactical constructs for markup. That allows the programmer to separate analysis and transformation stages within a complex re-engineering application that still operates on parse trees. Layout information is an example of information that could be stored in a scaffold. The fully informative parse trees that we use might be seen as abstract syntax trees *implicitly* scaffolded with layout information. As opposed to normal scaffolds, which are not hidden from the programmer's view.

Lämmel [107] describes how tree *annotations* can be used to store scaffolds without exposing this information directly to the structure of the abstract syntax tree. We do not employ annotations for storing layout because there is no principal difference between characters in layout trees from characters in other trees. This unification simplifies our term rewriting back-ends significantly. ASF+SDF does provide an interface for storing annotations on parse trees, but this is not used for layout. Rather for locally storing intermediate results of analyses. Lämmel proposes rigorous separation of concerns by removing the annotation propagation aspect from the matching and constructing patterns to separate declarations of annotation propagation rules. Aspect oriented term rewriting [103, 94] may provide an alternative solution for separating high-fidelity aspects from the functional effect of transformation.

In [165] the authors describe how high-fidelity C++ transformation systems can be

generated from high-level descriptions in the YATL language. This system effectively uses the fully informative parse trees, scaffolds, and source factors that were described earlier. In a first stage the source code is marked up, in a second stage this marked up code is fully parsed and then transformed. The YATL language is compiled to Stratego programs that transform the parse trees of marked up code. This compiler takes a number of big steps towards high-fidelity C++ transformations. For example, it deals with the C preprocessor problems, automatically propagates layout and source code comments during transformations, and learns how to pretty print newly introduced code from the surrounding code fragments. We do not provide such automation, but rather the possibility for declarative access to the layout information in a language independent manner. We consider heuristics for automated propagation of layout an orthogonal issue that may be dealt with in the future by the ASF normalizer. Also, conservative pretty printing [64] is outside the scope of the functionality provided by this chapter. Our functionality regarding layout is language independent, fully typed and syntax safe. As such ASF+SDF may provide a syntax safe back-end for the YATL language.

The CobolX system [169] uses a Stratego language feature called *overlays* to let the user abstract from layout details, while still having an abstract syntax tree that contains layout nodes underneath. Overlays are an extension of signatures. As a result, the two modes of operation with CobolX are either full exposure to the complex parse trees, or full abstraction from all details. We provide an interface for arbitrary ignoring or utilizing layout nodes in any position.

One exceptional approach for high-fidelity transformations is taken by the authors of the Haskell refactoring tool HaRe [122]. HaRe reuses an existing front-end for the Haskell language that produces abstract syntax trees with a lot of type information, but not the information needed for high-fidelity transformations. To solve this problem, an API was designed for transforming Haskell abstract syntax trees that, as a side-effect, also transforms the *token stream* of the original program.

8.10 Conclusions

The complexity of high-fidelity transformation is witnessed by the research effort that has now been spent on it. It is not only a trivial matter that is only investigated to have industrial partners adopt long standing source code transformation techniques. In this chapter we have taken a few steps towards a better support of high-fidelity transformations by fulfilling two requirements: syntax safety and full accessibility to syntactical information about the source code.

We have introduced “first class layout”, offering the programmer expressive notation for matching, construction, and ignoring layout. The resulting language allows language independent and syntax safe analysis and transformations of layout. All the other features that are normally available in term rewriting can be applied to layout.

We have applied “first class layout” to extract facts from source code comments in the C language, to conclude that their quality was high, but not 100% correct. Apparently, source code comments can be used to test reverse engineering tools, by providing an alternate independent source of the same information.

CHAPTER 9

A Generator of Efficient Strongly Typed Abstract Syntax Trees in Java

Abstract syntax trees are a very common data structure in language related tools. For example compilers, interpreters, documentation generators, and syntax-directed editors use them extensively to extract, transform, store and produce information that is key to their functionality.

We present a Java back-end for ApiGen, a tool that generates implementations of abstract syntax trees. The generated code is characterized by strong typing combined with a generic interface and maximal sub-term sharing for memory efficiency and fast equality checking. The goal of this tool is to obtain safe and more efficient programming interfaces for abstract syntax trees.

The contribution of this work is the combination of generating a strongly typed data-structure with maximal sub-term sharing in Java. Practical experience shows that this approach is beneficial for extremely large as well as smaller data types. [1]

9.1 Introduction

The technique described in this chapter aims at supporting the engineering of Java tools that process tree-like data structures. We target for example compilers, program analyzers, program transformers and structured document processors. A very important data structure in the above applications is a tree that represents the program or document to be analyzed and transformed. The design, implementation and use of such a tree data structure is usually not trivial.

A Java source code transformation tool is a good example. The parser should return an abstract syntax tree (AST) that contains enough information such that a transformation can be expressed in a concise manner. The AST is preferably strongly typed to

[1] This chapter was published in a special issue on Language definitions and tool generation in IEE Proceedings – Software in 2005. It is co-authored by Mark van den Brand and Pierre-Etienne Moreau.

distinguish between the separate aspects of the language. This allows the compiler to statically detect programming errors in the tool as much as possible. A certain amount of redundancy can be expected in such a fully informative representation. To be able to make this manageable in terms of memory usage the programmer must take care in designing his AST data structure in an efficient manner.

ApiGen [62] is a tool that generates automatically implementations of abstract syntax trees in C. It takes a concise definition of an abstract data type and generates C code for abstract syntax trees that is strongly typed and uses maximal sub-term sharing for memory efficiency and fast equality checking. The key idea of ApiGen is that a full-featured and optimized implementation of an AST data structure can be generated automatically, and with a very understandable and type-safe interface.

We have extended the ApiGen tool to generate AST classes for Java. The strongly typed nature of Java gives added functionality as compared to C. For example, using inheritance we can offer a generic interface that is still type-safe. There are trade-offs that govern an efficient and practical design. The problem is how to implement maximal sub-term sharing for a highly heterogeneous data type in an efficient and type-safe manner, and at the same time provide a generic programming interface. In this chapter we demonstrate the design of the generated code, and that this approach leads to practical and efficient ASTs in Java. Note that we do not intend to discuss the design of the code generator, this is outside the scope of this chapter.

9.1.1 Overview

We continue the introduction with our major case study, and descriptions of maximal sub-term sharing and the process of generating code from data type definitions. Related work is discussed here too. The core of this chapter is divided over the following sections:

Section 9.2: Two-tier interface of the generated AST classes.

Section 9.3: Generic first tier, the ATerm data structure.

Section 9.4: A factory for maximal sub-term sharing: `SharedObjectFactory`.

Section 9.5: Generated second tier, the AST classes.

Figure 9.5 on page 184 summarizes the above sections and can be used as an illustration to each of them. In Section 9.6 we validate our design in terms of efficiency by benchmarking. In Section 9.7 we describe results of case-studies and applications of ApiGen, before we conclude in Section 9.8.

9.1.2 Case-study: the JTom compiler

As a case-study for our work, we introduce JTom [127]. It is a pattern matching compiler, that adds the *match construct* to C, Java and Eiffel. The construct is translated to normal instructions in the host language, such that a normal compiler can be used to complete the compilation process. The general layout of the compiler is shown in Figure 9.1. The specifics of compiling the match construct are outside the scope of this

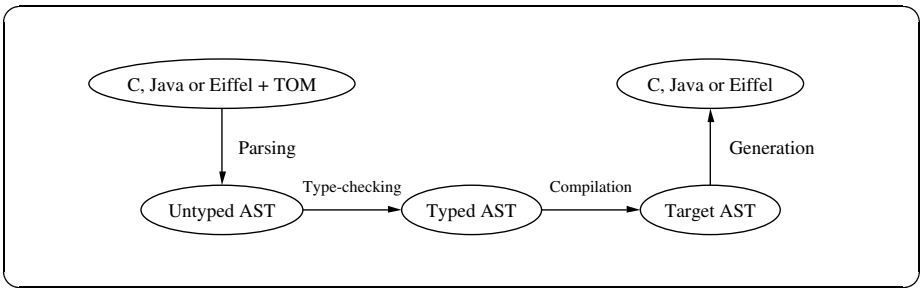

Figure 9.1: General layout of the JTom compiler.

chapter. It is only relevant to know that ASTs are used extensively in the design of the JTom compiler, so it promises to be a good case-study for ApiGen.

9.1.3 Maximal sub-term sharing

In the fields of functional programming and term rewriting the technique of maximal sub-term sharing, which is frequently called hash-consing, has proved its benefits [77, 33], however not in all cases [5]. The run-time systems of these paradigms also manipulate tree-shaped data structures. The nature of their computational mechanisms usually lead to significant redundancy in object creation.

Maximal sub-term sharing ensures that only one instance of any sub-term exists in memory. If the same node is constructed twice, a pointer to the previously constructed node is returned. The effect is that in many cases the memory requirements of term rewriting systems and functional programs diminish significantly. Another beneficial consequence is that equality of sub-terms is reduced to pointer equality: no traversal of the tree is needed. If the data or the computational process introduce a certain amount of redundancy, then maximal sub-term sharing pays off significantly. These claims have been substantiated in the literature and in several implementations of programming languages, e.g., [77].

Our contribution adds maximal sub-term sharing as a tool in the kit of the Java programmer. It is not hidden anymore inside the run-time systems of functional programming languages. We apply it to big data structures using a code generator for heterogeneously typed abstract syntax trees. These two properties make our work different from other systems that use maximal sub-term sharing.

9.1.4 Generating code from data type definitions

A data type definition describes in a concise manner exactly how a tree-like data structure should be constructed. It contains types, and constructors. Constructors define the alternatives for a certain type by their name and the names and types of their children. An example of such a definition is in Figure 9.2. Well-known formalisms for data type definitions are for example XML DTD and Schemas [81], and ASDL [168].

```
datatype Expressions
  Bool ::= true
         | false
         | eq(lhs:Expr, rhs:Expr)
  Expr ::= id(value:str)
         | nat(value:int)
         | add(lhs:Expr, rhs:Expr)
         | mul(lhs:Expr, rhs:Expr)
```

Figure 9.2: An example data type definition for an expression language.

As witnessed by the existence of numerous code generators, e.g., [62, 108, 168, 78, 151], such concise descriptions can be used to generate implementations of tree data structures in any programming language. An important aspect is that if the target language has a strong enough typing mechanism, the types of the data type definition can be reflected somehow in the generated code.

Note that a heterogeneously typed AST representation of a language is important. An AST format for a medium-sized to big language contains several kinds of nodes. Each node should have an interface that is made specific for the kind of node. This allows for static well-formedness checking by the Java compiler, preventing the most trivial programming errors. It also leads to code on a higher level of abstraction.

As an example, suppose an AST of a Pascal program is modeled using a single class `Node` that just has an array of references to other `Nodes`. The Java code will only implicitly reflect the structure of a Pascal program, it is hidden in the dynamic structure of the `Nodes`. With a fully typed representation, different node types such as declarations, statements and expressions would be easily identifiable in the Java code.

The classes of an AST can be instrumented with all kinds of practical features such as serialization, the Visitor design pattern and annotations. Annotations are the ability to decorate AST nodes with other objects. The more features offered by the AST format, the more beneficial a generative approach for implementing the data structure will be.

9.1.5 Related work

The following systems are closely related to the functionality of ApiGen:

ASDL [168, 82] is targeted at making compiler design a less tedious and error prone activity. It was designed to support tools in different programming languages working on the same intermediate program representation. For example, there are implementations for C, C++, Java, Standard ML, and Haskell.

ApiGen for C [62] is a predecessor of ApiGen for Java, but written by different authors. One of the important features is a connection with a parser generator. A syntax definition is translated to a data type definition which defines the parse trees

that a parser produces. ApiGen can then generate code that can read in parse trees and manipulate them directly. In fact, our instantiation of ApiGen also supports this automatically, because we use the same data type definition language.

The implementation of maximal sub-term sharing in ApiGen for C is based on type-unsafe casting. The internal representation of every generated type is just a shared ATerm, i.e. `typedef ATerm Bool;`. In Java, we implemented a more type-safe approach, which also allows more specialization and optimization.

JJForester [108] is a code generator for Java that generates Java code directly from syntax definitions. It generates approximately the same interfaces (Composite design pattern) as we do. Unlike JJForester, ApiGen does not depend on any particular parser generator. By introducing an intermediate data type definition language, any syntax definition that can be translated to this language can be used as a front-end to ApiGen.

JJForester was the first generator to support JJTraveler [112] as an implementation of the Visitor design pattern. We have copied this functionality in ApiGen directly because of the powerful features JJTraveler offers (see also Section 9.5). Note that JJForester does not generate an implementation that supports maximal sharing.

Pizza [131] adds algebraic data types to Java. An algebraic data type is also a data type definition. Pizza adds much more features to Java that do not relate to the topic of this chapter. In that sense, ApiGen targets a more focused problem domain and can be used as a more lightweight approach. Also, Pizza does not support maximal sub-term sharing.

Java Tree Builder [134] and JastAdd [84] are also highly related tools. They generate implementations of abstract syntax trees in combination with syntax definitions. The generated classes also directly support the Visitor design pattern.

All and all, the idea of generating source code from data type definitions is a well-known technique in the compiler construction community. We have extended that idea and constructed a generator that optimizes the generated code on memory efficiency without loosing speed. Our generated code is characterized by strong typing combined with a generic interface and maximal sub-term sharing.

9.2 Generated interface

Our intent is to generate class hierarchies from input descriptions such as shown in Figure 9.2. We propose a class hierarchy in two layers. The upper layer describes generic functionality that all tree constructors should have. This upper layer could be a simple interface definition, but better even a class that actually implements common functionality. There are two benefits of having this abstract layer:

1. It allows for reusable generic algorithms to be written in a type-safe manner.
2. It prevents code duplication in the generated code.

```
public abstract class Bool extends ATermAppl { ... }
package bool;
  public class True extends Bool { ... }
  public class False extends Bool { ... }
  public class Eq extends Bool { ... }

public abstract class Expr extends ATermAppl { ... }
package expr;
  public class Id extends Expr { ... }
  public class Nat extends Expr { ... }
  public class Add extends Expr { ... }
  public class Mul extends Expr { ... }
```

Figure 9.3: The generated Composite design sub-types a generic tree class `ATermAppl`.

The second layer is generated from the data type definition at hand. Figure 9.3 depicts the class hierarchy that is generated from the definition show in the introduction (Figure 9.2). The Composite design pattern is used [75]. Every type is represented by an abstract class and every constructor of that type inherits from this abstract class. The type classes specialize some generic tree model `ATermAppl` which will be explained in Section 9.3. The constructor classes specialize the type classes again with even more specific functionality.

The interface of the generated classes uses as much information from the data type definition as possible. We generate an identification predicate for every constructor as well as setters and getters for every argument of a constructor. We also generate a so-called possession predicate for every argument of a constructor to be able to determine if a certain object has a certain argument.

Figure 9.4 shows a part of the implementation of the `Bool` abstract class and the `Eq` constructor as an example. The abstract type `Bool` supports all functionality provided by its subclasses. This allows the programmer to abstract from the constructor type whenever possible. Note that because this code is generated, we do not really introduce a fragile base class problem here. We assume that every change in the implementation of the AST classes inevitably leads to regeneration of the entire class hierarchy.

The class for the `Eq` constructor has been put into a package named `bool`. For every type, a package is generated that contains the classes of its constructors. Consequently, the same constructor name can be reused for a different type in a data type definition.

9.3 Generic interface

We reuse an existing and well-known implementation of maximally shared trees: the ATerm library. It serves as the base implementation of the generated data structures. By doing so we hope to minimize the number of generated lines of code, profit from

```
public abstract class Bool extends ATermAppl {
  public boolean isTrue()      { return false; }
  public boolean isFalse()     { return false; }
  public boolean isEq()        { return false; }
  public boolean hasLhs()      { return false; }
  public boolean hasRhs()      { return false; }
  public Expr getLhs()         { throw new GetException(...); }
  public Expr getRhs()         { throw new GetException(...); }
  public Bool setLhs(Expr lhs) { throw new SetException(...); }
  public Bool setRhs(Expr rhs) { throw new SetException(...); }
}

package bool;
public class Eq extends Bool {
  public boolean isEq()      { return true; }
  public boolean hasLhs()    { return true; }
  public boolean hasRhs()    { return true; }
  public Expr getLhs()       { return (Expr) getArgument(0); }
  public Expr getRhs()       { return (Expr) getArgument(1); }
  public Bool setLhs(Expr e) { return (Bool) setArgument(e,0);}
  public Bool setRhs(Expr e) { return (Bool) setArgument(e,1);}
}
```

Figure 9.4: The generated predicates setters and getters for the `Bool` type and the `Eq` constructor.

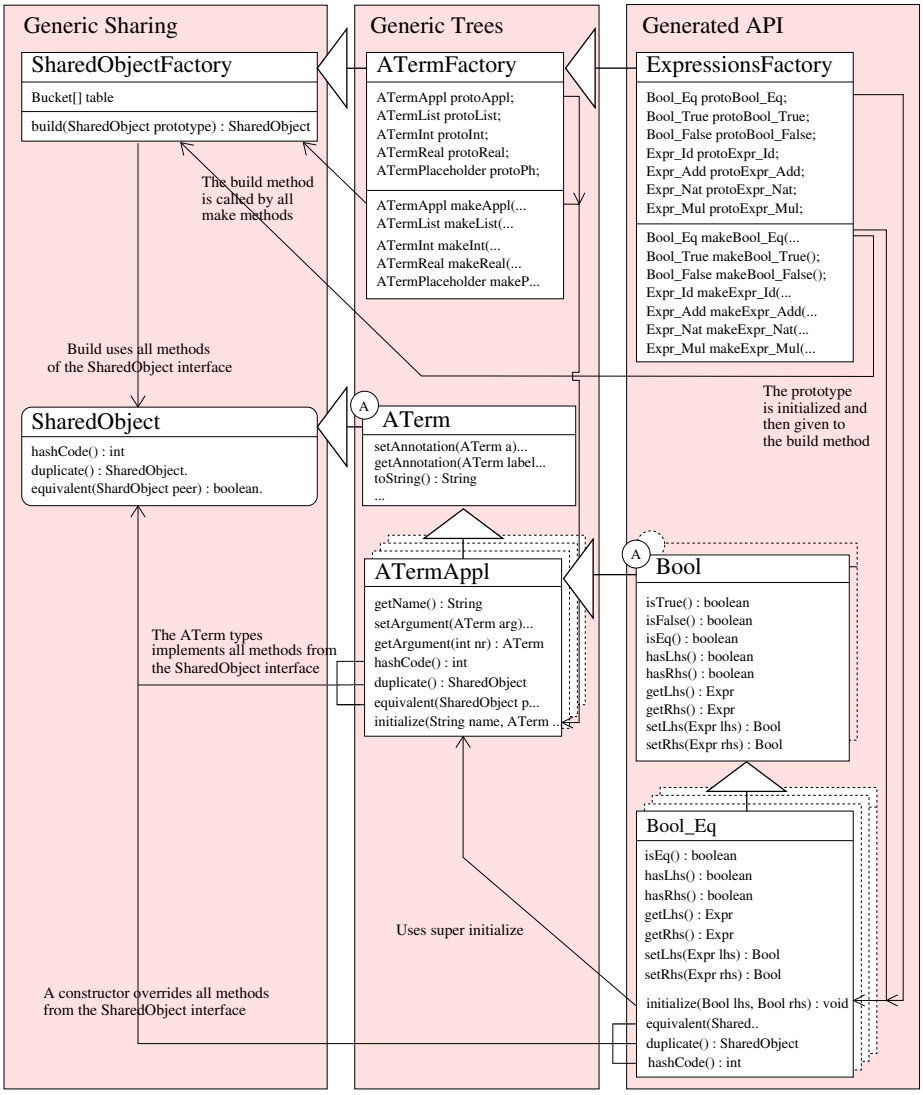

Figure 9.5: A diagram of the complete ApiGen architecture.

```
public abstract class ATerm {
  ...
  public ATerm     setAnnotation(ATerm label, ATerm anno);
  public ATerm     getAnnotation(ATerm label);
}

public class ATermAppl extends ATerm {
  ...
  public String    getName();
  public ATermList getArguments();
  public ATerm     getArgument(int i);
  public ATermAppl setArgument(ATerm arg, int i);
}
```

Figure 9.6: A significant part of the public methods of `ATerm` and `ATermAppl`.

the efficiency of the existing implementation and effortlessly support the ATerm exchange formats. It also immediately provides the generic programming interface for developing reusable algorithms.

The ATerm data structure implements maximal sub-term sharing. However, this implementation can not be reused for the generated tier by using inheritance. Why this can not be done will become apparent in the next section.

The ATerm library offers five types of AST nodes: function application, lists, integers, reals and placeholders. In this presentation we concentrate on function application, implemented in the class `ATermAppl`. The abstract superclass `ATerm` implements basic functionality for all term types. Most importantly, every `ATerm` can be decorated with so-called annotations. We refer to [31] for further details concerning ATerms.

The first version of our case-study, JTom, was written without the help of ApiGen. ATerms were used as a mono-typed implementation of all AST nodes. There were about 160 different kinds of AST nodes in the JTom compiler. This initial version was written quickly, but after extending the language with more features the maintainability of the compiler deteriorated. Adding new features became harder with the growing number of constructors. By not using strict typing mechanisms of Java there was little static checking of the AST nodes. Obviously, this can lead to long debugging sessions in order to find trivial errors.

9.4 Maximal sub-term sharing in Java

Before we can continue discussing the generated classes, we must first pick a design for implementing maximal sub-term sharing. The key feature of our generator is that it generates strongly typed implementations of ASTs. To implement maximal sub-term sharing for all of these types we should generate a factory that can build objects of the correct types.

9.4.1 The Factory design pattern

The implementation of maximal sub-term sharing is always based on an administration of existing objects. In object-oriented programming a well-known design pattern can be used to encapsulate such an administration: a Factory [75].

The efficient implementation of this Factory is a key factor of success for maximal sharing. The most frequent operation is obviously looking up a certain object in the administration. Hash-consing [5] is a technique that optimizes exactly this. For each object created, or about to be created, a hash code is computed. This hash code is used as an index in a hash table where the references to the actual objects with that hash code are stored. In Java, the use of so-called weak references in the hash table is essential to ensure that unused objects can be garbage collected by the virtual machine.

The ATerm library contains a specialized factory for creating maximally shared ATerms: the `ATermFactory`.

9.4.2 Shared Object Factory

The design of the original `ATermFactory` does not allow extension with new types of shared objects. In order to deal with any type of objects a more abstract factory that can create any type of objects must be constructed. By refactoring the `ATermFactory` we extracted a more generic component called the `SharedObjectFactory`. This class implements hash consing for maximal sharing, nothing more. It can be used to implement maximal sharing for any kind of objects. The design patterns used are AbstractFactory and Prototype. An implementation of this factory is sketched in Figure 9.7.

A prototype is an object that is allocated once, and used in different situations many times until it is necessary to allocate another instance, for which a prototype offers a method to duplicate itself. The Prototype design allows a Factory to abstract from the type of object it is building [75] because the actual construction is delegated to the prototype. In our case, Prototype is also motivated by efficiency considerations. One prototype object can be reused many times, without the need for object allocation, and when duplication is necessary the object has all private fields available to implement the copying of references and values as efficiently as possible.

The `SharedObject` interface contains a `duplicate` method[2], an `equivalent` method to implement equivalence, and a `hashCode` method which returns a hash code (Figure 9.7). The Prototype design pattern also has an initialize method that has different arguments for every type of shared-object. So it can not be included in a Java interface. This method is used to update the fields of the prototype instance each time just before it is given to the `build` method.

For a sound implementation we must assume the following properties of any implementation of the `SharedObject` interface:

- `duplicate` always returns an exact clone of the object, with the exact same type.

[2] We do not use the `clone()` method from `Object` because our `duplicate` method should return a `SharedObject`, not an `Object`.

```
public interface SharedObject {
  int          hashCode();
  SharedObject duplicate();
  boolean      equivalent(SharedObject peer);
  // void      initialize(...); (changes with each type)
}
public class SharedObjectFactory {
  ...
  public SharedObject build(SharedObject prototype) {
    Bucket bucket = getHashBucket(prototype.hashCode());
    while (bucket.hasNext()) {
      SharedObject found = bucket.next();
      if (prototype.equivalent(found)) {
        return found;
      }
    }
    SharedObject fresh = prototype.duplicate();
    bucket.add(fresh);
    return fresh;
  }
}
```

Figure 9.7: A sketch of the essential functionality of `SharedObjectFactory`.

- `equivalent` implements an equivalence relation, and particularly makes sure that two objects of different types are never equivalent.
- `hashCode` always returns the same hash code for equal objects.

Any deviation from the above will most probably lead to class cast exceptions at runtime. The following guidelines are important for implementing the `SharedObject` interface efficiently:

- Memorize the `hashCode` in a private field.
- `duplicate` needs only a shallow cloning, because once a SharedObject is created it will never change.
- Analogously, `equivalent` can be implemented in a shallow manner. All fields that are `SharedObject` just need to have equal references.
- The implementation of the `initialize` method is pivotal for efficiency. It should not allocate any new objects. Focus on copying the field references in the most direct way possible.

Using the `SharedObjectFactory` as a base implementation, the `ATermFactory` is now extensible with new types of terms by constructing new implementations of the `SharedObject` interface, and adding their corresponding prototype objects. The next step is to generate such extensions automatically from a data type definition.

```
package bool;
public class Eq extends Bool implements SharedObject {
  ...
  public SharedObject duplicate() {
    Eq.clone = new Eq();
    clone.initialize(lhs, rhs);
    return clone;
  }
  public boolean equivalent(SharedObject peer) {
    return (peer instanceof Eq) && super.equivalent(peer);
  }
  protected void initialize(Bool lhs, Bool rhs) {
    super.initialize("Bool_Eq", new ATerm[] {lhs, rhs});
  }
}
```

Figure 9.8: The implementation of the `SharedObject` interface for the `Eq` constructor.

9.5 The generated implementation

The complete ApiGen architecture including the generated API for our running example is depicted in Figure 9.5. Two main tasks must be fulfilled by the code generator:

- Generating the Composite design for each type in the definition, by extending `ATermAppl`, and implementing the `SharedObject` interface differently for each class.
- Extending the `ATermFactory` with a new private prototype, and a new `make` method for each constructor in the definition.

9.5.1 ATerm extension

Figure 9.8 shows how the generic `ATermAppl` class is extended to implement an `Eq` constructor of type `Bool`. It is essential that it overrides all methods of `ATermAppl` of the `SharedObject` interface, except the computation of the hash code. This reuse is beneficial since computing the hash code is perhaps the most complex operation.

Remember how every `ATermAppl` has a name and some arguments (Figure 9.6). We model the `Eq` node of type `Bool` by instantiating an `ATermAppl` with name called "Bool_Eq". The two arguments of the operator can naturally be stored as the arguments of the `ATermAppl`. This is how a generic tree representation is reused to implement a specific type of node.

9.5.2 Extending the factory

The specialized make methods are essential in order to let the user be able to abstract from the ATerm layer. An example generated `make` method is shown in Fig-

```
class ExpressionFactory extends ATermFactory {
  private bool.Eq protoBool_Eq;
  public ExpressionFactory() {
    protoBoolEq = new bool.Eq();
  }
  public bool.Eq makeBool_Eq(Expr lhs, Expr rhs) {
    protoBool_Eq.initialize(lhs, rhs);
    return (bool.Eq) build(protoBool_Eq);
  }
}
```

Figure 9.9: Extending the factory with a new constructor `Eq`.

ure 9.9. After initializing a prototype that was allocated once in the constructor method, the `build` method from `SharedObjectFactory` is called. The downcast to `bool.Eq` is safe only because `build` is guaranteed to return an object of the same type. This guarantee is provided by the restrictions we have imposed on the implementation of any `SharedObject`.

Note that due to the `initialize` method, the already tight coupling between factory and constructor class is intensified. This method has a different signature for each constructor class, and the factory must know about it precisely. This again motivates the generation of such factories, preventing manual co-evolution between these classes.

9.5.3 Specializing the ATermAppl interface

Recall the interface of `ATermAppl` from Figure 9.6. There are some type-unsafe methods in this class that need to be dealt with in the generated sub-classes. We do want to reuse these methods because they offer a generic interface for dealing with ASTs. However, in order to implement type-safety and clear error messaging they must be specialized.

For example, in the generated `Bool_Eq` class we override `setArgument` as shown in Figure 9.10. The code checks for arguments that do not exist and the type validity of each argument number. The type of the arguments can be different, but in the `Bool_Eq` example both arguments have type `Expr`. Analogously, `getArgument` should be overridden to provide more specific error messages than the generic method can. If called incorrectly, the generic methods would lead to a `ClassCastException` at a later time. The specialized implementations can throw more meaningful exceptions immediately when the methods are called.

Apart from type-safety considerations, there is also some opportunity for optimization by specialization. As a simple but effective optimization, we specialize the `hashFunction` method of `ATermAppl` because now we know the number of arguments of the constructor. The `hashCode` method is a very frequently called method, so saving a loop test at run-time can cause significant speed-ups. For a typical benchmark that focuses on many object creations the gain is around 10%.

```
public ATermAppl setArgument(ATerm arg, int i) {
  switch (i) {
    case 0: if (arg instanceof Expr) {
               return factory.makeBool_Eq((Expr) arg,
                                          (Expr) getArgument(1));
            } else {
              throw new IllegalArgumentException("...");
            }
    case 1: ...
    default: throw new IllegalArgumentException("..." + i);
  }
}
```

Figure 9.10: The generic `ATermAppl` implementation must be specialized to obtain type-safety.

A more intrinsic optimization of `hashCode` analyzes the types of the children for every argument to see whether the chance of father and child having the same `hashCode` is rather big. If that chance is high and we have deeply nested structures, then a lookup in the hash table could easily degenerate to a linear search.

So, if a constructor is recursive, we slightly specialize the `hashCode` to prevent hashing collisions. We make the recursive arguments more significant in the hash code computation than other arguments. Note that this is not a direct optimization in speed, but it indirectly makes the hash-table lookup an order of magnitude faster for these special cases.

This optimization makes most sense in the application areas of symbolic computation, automated proof systems, and model checking. In these areas one can find such deeply nested recursive structures representing for example lists, natural numbers or propositional formulas.

9.5.4 Extra generated functionality

In the introduction we mentioned the benefits of generating implementations. One of them is the ability of weaving in all kinds of practical features that are otherwise cumbersome to implement.

Serialization. The ATerm library offers serialization of ATerms as strings and as a shared textual representation. So, by inheritance this functionality is open to the user of the generated classes. However, objects of type `ATermAppl` are constructed by the `ATermFactory` while reading in the serialized term. From this generic ATerm representation a typed representation must be constructed. We generate a specialized top-down recursive binding algorithm in every factory. It parses a serialized ATerm, and builds the corresponding object hierarchy, but only if it fits the defined data type.

The Visitor design pattern is the preferred way of implementing traversal over object structures. Every class implements a certain interface (e.g., `Visitable`) allowing a `Visitor` to be applied to all nodes in a certain traversal order. This design pattern prevents the pollution of every class with a new method for one particular aspect of a compilation process, the entire aspect can be separated out in a `Visitor` class. JJTraveler [163] extends the Visitor design pattern by generalizing the visiting order. We generate the implementation of the `Visitable` interface in every generated constructor class and some convenience classes to support generic tree traversal with JJTraveler.

Pattern matching is an algorithmic aspect of tree processing tools. Without a pattern matching tool, a programmer usually constructs a sequence of nested `if` or `switch` statements to discriminate between a number of patterns. Pattern matching can be automated using a pattern language and a corresponding interpreter or a compiler that generates the nested `if` and `switch` statements automatically.

As mentioned in the introduction, our largest case study JTom [127] is such a pattern matching compiler. One key feature of JTom is that it can be instantiated for any data structure. As an added feature, ApiGen can instantiate the JTom compiler such that the programmer can use our generated data structures, and match complex patterns in a type-safe and declarative manner.

9.6 Performance measurements

Maximal sub-term sharing does not always pay off, since its success is governed by several trade-offs and overheads [5]. We have run benchmarks, which have been used earlier in [33], for validating our design in terms of efficiency. We consider both runtime efficiency and memory usage of the generated classes important issues. To be able to analyze the effect of some design decisions we try to answer the following questions:

1. How does maximal sub-term sharing affect performance and memory usage?
2. Does having a generic `SharedObjectFactory` introduce an overhead?
3. What is the effect of the generated layer on the performance?
4. Do the specializations of hash functions have any effect on performance?

9.6.1 Benchmarks

We have considered three benchmarks which are based on the normalization of expressions $2^n \bmod 17$, with $18 < n < 20$, where the natural numbers involved are Peano integers. These benchmarks have been first presented in [33]. They are characterized by a large number of transformations on large numbers of AST nodes. Their simplicity allows them to be easily implemented in different kinds of languages using different kinds of data structures.

Benchmarks (in seconds)	(1) New ATerms without sharing	(2) Old ATerms with sharing	(2) New ATerm with sharing	(4) ApiGen without hash functions	(5) ApiGen with hash functions
evalsym(18)	*7.2*	5.8	6.9	**5.7**	**5.7**
evalsym(19)	*14.3*	11.4	13.8	11.5	**11.3**
evalsym(20)	*28.7*	22.7	27.7	22.9	**22.5**
evalexp(18)	*11.8*	**6.7**	7.4	7.1	7.1
evalexp(19)	*23.2*	**13.7**	14.8	14.4	14.0
evalexp(20)	*46.5*	**27.5**	29.4	28.6	27.8
evaltree(18)	*16.0*	6.7	7.8	**4.8**	**4.8**
evaltree(19)	*30.8*	13.4	15.6	9.7	**9.5**
evaltree(20)	-	26.6	31.1	19.4	**19.0**

Table 9.1: The `evalsym`, `evalexp`, and `evaltree` benchmarks for five different implementations of AST classes in Java. We obtained these figure by running our benchmarks on a Pentium III laptop with 512Mb, running WindowsXP.

- The `evalsym` benchmark is CPU intensive, but does not use a lot of object allocation. For this benchmark, the use of maximal sub-term sharing does not improve the memory usage, but does not slow down the efficiency either.
- The `evalexp` benchmark uses a lot of object allocation.
- The `evaltree` benchmark also uses a lot of object allocation, but with a lower amount of redundancy. Even now, the use of maximal sub-term sharing allows us to keep the memory usage at an acceptable level without reducing the run-time efficiency.

In Table 9.1 the rows show that three benchmarks are run for three different sizes of input. The columns compare five implementations of these three benchmarks. All benchmarks were written in Java.

Column 1: for this experiment, we use modified implementations of the `SharedObjectFactory` and the ATerm library where the maximal sub-term sharing mechanism has been deactivated. This experiment is used to measure the impact of maximal sharing.

Column 2: for this experiment, we use a previous implementation of the `ATermFactory` with a specialized version of maximal sub-term sharing (i.e. not using the `SharedObjectFactory`). This experiment is used to measure the efficiency cost of introducing the reusable `SharedObjectFactory`.

Column 3: this corresponds to the current version of the ATerm library, where maximal sharing is provided by the `SharedObjectFactory`. This experiment is

used as a reference to compare with the generated strongly typed classes.

Column 4: for this experiment, we use a modified version of ApiGen where specialized hash functions are not generated. This experiment is used to see if the generation of specialized hash functions has any effect on performance.

Column 5: for this experiment, we use the version of ApiGen presented in this chapter, where specialized hash functions are generated.

In a previous comparison between several rewrite systems [33], the interest of using maximal sub-term sharing was clearly established. In this chapter, we demonstrate that maximal sub-term sharing can be equally beneficial in a Java environment. More importantly, our results show that the performance of maximal sub-term sharing can be improved when each term is also strongly typed and specialized:

- Column 1 indicates that the approach without maximal sharing leads to the slowest implementation.

  As mentioned previously, the `evalsym` benchmark does not use a lot of object allocation. In this case, the improvement is due to the fact that equality between nodes reduces to pointer equality.

  On the other side, the `evalexp` and the `evaltree` benchmarks use a lot of object allocation. Columns 2, 3, 4 and 5 clearly show that maximal sharing is highly interesting in this case. The last result given in Column 1 indicates that for bigger examples ($n \geq 20$), the computation can not be completed with 512Mb of memory.

  To conclude this first analysis, the results certify, in the Java setting, the previous results on maximal sub-term sharing from [33].

- When comparing Column 2 and Column 3, the `ATermFactory` with and without the `SharedObjectFactory`, we notice that the previous version of the `ATermFactory` was faster than the current one, but not significantly. This is the slowdown we expected from introducing the `SharedObjectFactory`.

- The difference between the untyped ATerm library (Column 3) and generated classes (Column 4) shows that by specializing the AST nodes into different types we gain efficiency.

- Introducing specialized hash functions (from Column 4 to 5) we can see that the generation of specialized hash functions improves the efficiency a little bit. However, this improves the efficiency just enough to make the benchmarks run more efficiently than a program which use the original implementation of the ATerm library (Column 2). The negative effect of introducing a more generic and maintainable architecture has been totally negated by the effects of specialization using types.

The effects on memory usage are depicted in Figures 9.11, 9.12 and 9.13. The figures show that without redundancy the overhead of maximal sub-term sharing is constant.

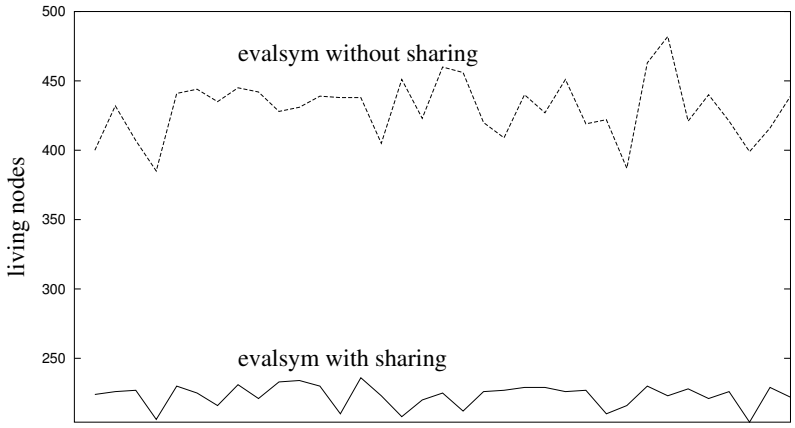

Figure 9.11: A comparison of the memory usage of the evalsym benchmark with and without maximal sub-term sharing.

This can be expected because the administration of existing objects allocates some space. However, in the presence of some redundancy maximal sub-term sharing can save an order of magnitude of memory.

9.6.2 Quantitative results in the JTom compiler

It is also interesting to have an idea of performance and memory usage in a more realistic application. The effect of maximal sub-term sharing in the JTom compiler is shown in Figure 9.14. There is a significant improvement with respect to memory usage. These measurements have been obtained by replacing the generated factory temporarily by a factory that does not implement maximal sub-term sharing. We also had to replace the implementation of the equality operator by a new implementation that traverses the complete ASTs to determine the equality of two trees.

While measuring the run-time performance of the compiler we measured significant differences between the versions with and without maximal sub-term sharing, but these results need to be interpreted carefully. The design of the compiler has been influenced by the use of maximal sub-term sharing. In particular, it allows us to forget about the size of the AST while designing the compiler. We can store all relevant information inside the ASTs without compromising memory consumption limitations. Our experiences indicate that in general maximal sub-term sharing allows a compiler designer to concentrate on the clarity of his data and algorithms rather than on efficiency considerations.

The effect of introducing the generated layer of types in the JTom compiler could not be measured effectively. The reason is that the current version is no longer comparable (in terms of functionality) with the previous version based on the untyped ATerm library. The details of the compilation changed too much. Although the functionality has changed a lot, we did not observe any performance penalty.

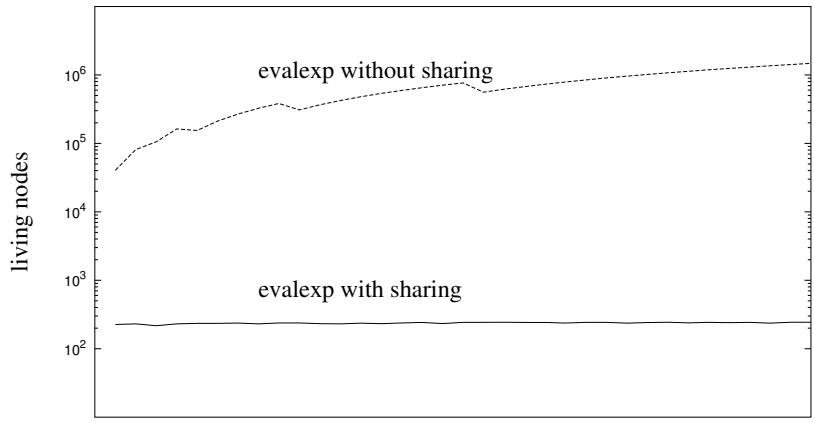

Figure 9.12: A comparison of the memory usage of the evalexp benchmark with and without maximal sub-term sharing.

9.6.3 Benchmarking conclusions

To summarize the above:

- A heterogeneously typed representation of AST nodes is faster than a mono-typed representation, because more information is statically available.
- Specializing hash-functions improves the efficiency a little bit in most cases, and enormously for some deeply recursive data structures where the hash-function would have degenerated to a constant otherwise.
- The design of `SharedObjectFactory` based on AbstractFactory and Prototype introduces an insignificant overhead as compared to generating a completely specialized less abstract Factory.

We conclude that compared to untyped ASTs that implement maximal sub-term sharing we have gained a lot of functionality and type-safety without introducing an efficiency bottleneck. Compared to a non-sharing implementation of AST classes one can expect significant improvements in memory consumption, in the presence of redundant object creation.

9.7 Experience

ApiGen for Java was used to implement several Java tools that process tree-like data structures. The following are the two largest applications:

- The GUI of an integrated development environment.
- The JTom compiler.

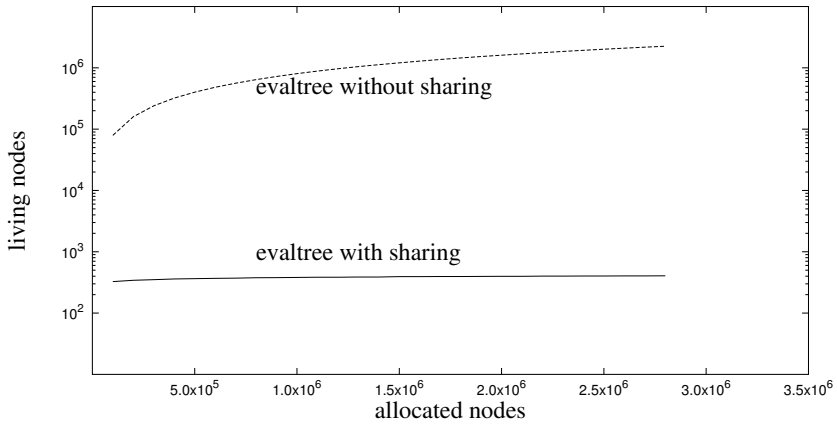

Figure 9.13: A comparison of the memory usage of the evaltree benchmarks with and without maximal sub-term sharing.

9.7.1 The GUI of an integrated development environment

The ASF+SDF Meta-Environment [28] is an IDE which supports the development of ASF+SDF specifications. The GUI is written in Java with Swing. It is completely separated from the underlying language components, and communicates only via serialization of objects that are generated using ApiGen APIs.

Three data structures are involved. An error data structure is used to for displaying and manipulating error messages that are produced by different components in the IDE. A configuration format is used to store and change configuration parameters such as menu definitions, colors, etc. The largest structure is a graph format, which is used for visualizing all kinds of system and user-defined data.

ApiGen for Java is used to generate the APIs of these three data structures. The graph API consists of 14 types with 49 constructors that have a total of 73 children (Figure 9.15). ApiGen generates 64 classes (14 + 49 + a factory), adding up 7133 lines of code. The amount of hand-written code that uses Swing to display the data structure is 1171 lines. It actually uses only 32 out of 73 generated getters, 1 setter out of 73, 4 possession predicates, 10 identity predicates, and 4 generated make methods of this generated API. Note that all 73 make methods are used by the factory for the deserialization algorithm which is called by the GUI once. The graph data is especially redundant at the leaf level, where every edge definition refers to two node names that can be shared. Also, node attributes are atomic values that are shared in significant amounts.

The error and configuration data types are much smaller, and so is the user-code that implements functionality on them. Almost all generated getters are used in their application, half of the predicates, no setters and no make methods. The reason is that the GUI is mainly a consumer of data, not a producer. The data is produced by tools written in C or ASF+SDF, that use the C APIs which have been generated from the same data type definitions. So, these ApiGen definitions effectively serve as contracts

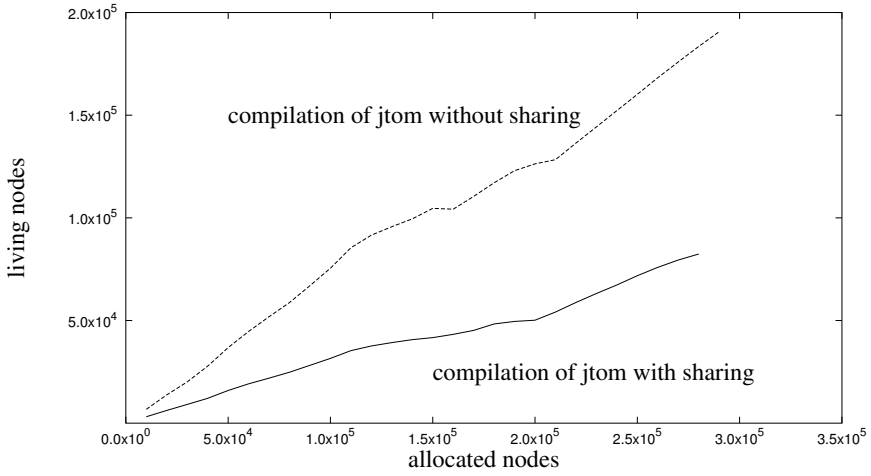

Figure 9.14: Impact of maximal sub-term sharing on the JTom compiler.

```
datatype Tree
  Graph ::= graph(nodes:NodeList,
                  edges:EdgeList,
                  attributes:AttributeList)

  Node ::= node(id:NodeId, attributes:AttributeList)

  Edge ::= edge(from:NodeId,to:NodeId,attributes:AttributeList)

  Attribute ::= bounding-box(first:Point,second:Point)
         | color(color:Color)
         | curve-points(points:Polygon)
         | ...
```

Figure 9.15: A part of the data type definition for graphs, which is 55 LOC in total.

between producers and consumers of data.

9.7.2 JTom based on ApiGen

The ASTs used in JTom have 165 different constructors. We defined a data type for these constructors and generated a typed representation using ApiGen.

There are 30 types in this definition, e.g., Symbol, Type, Name, Term, Declaration, Expression, Instruction. By using these class names in the Java code it has become more easily visible in which part of the compiler architecture they belong. For example, the “Instructions” are only introduced in the back-end, while you will find much references to “Declaration” in the front-end of the compiler. As a result, the code has become more self documenting. Also, by reverse engineering the AST constructors to a typed definition, we found a few minor flaws in the compiler and we clarified some

hard parts of the code.

ApiGen generates 32.544 lines of Java code for this data type. Obviously, the automation ApiGen offers is beneficial in terms of cost price in this case. Implementing such an optimized and typed representation of this type of AST would not only be hard, but also a boring and expensive job. Of the generated code 100% of the getters, make functions and identity predicates are used in the compiler. None of the possession predicates and setters are used. Note that application of class pruning tools such as JAX [148] would help to reduce the byte-code size by removing the code for the setters and the possession predicates.

The JTom compiler contains a number of generic algorithms for term traversal and origin tracking. These algorithms already used the ATerm interface, but now they are checked statically and dynamically for type errors. The generated specializations enforce that all ASTs that are constructed are well formed with respect to the original data type definition.

9.8 Conclusions

We presented a powerful approach, ApiGen for Java, to generate classes for ASTs based on abstract data type descriptions. These classes have a two-tier interface. The generic ATerm layer allows reusability, the specific generated layer introduces type-safety and meaningful method names.

We conclude that compared to mono-typed ASTs that implement maximal sub-term sharing we have gained a lot of functionality and type-safety, and improved efficiency. Secondly, compared to a non-sharing implementation of AST classes one can expect significant improvements in memory consumption, in the presence of redundant object creation.

To be able to offer maximal sub-term sharing in Java we have introduced a reusable `SharedObjectFactory`. Based on the AbstractFactory and Prototype design patterns, it allows us to generate strongly typed maximally shared class hierarchies with little effort. The class can be reused in different contexts that require object sharing.

The generated classes are instrumented with practical features such as a generic programming layer, serialization, the Visitor design pattern, and pattern matching. We demonstrated their use by discussing the JTom compiler, and some other smaller examples.

Part IV

CHAPTER 10

Conclusions

In this chapter we zoom out and return to the general subject of analysis and transformation of source code. We first revisit the research questions from the introduction. We then summarize the software that was developed in the context of this thesis. We conclude with a reflective note on the main topic of this thesis.

10.1 Research questions

Table 10.1 repeats the research questions that were formulated in Chapter 1. For each question we summarize the conclusions, make some recommendations, and identify directions for further improvement.

10.1.1 How can disambiguations of context-free grammars be defined and implemented effectively?

Conclusions

Starting from the theme "disambiguation is a separate concern", we have obtained two answers to this question:

- For certain grammatical idioms we can provide declarative disambiguation rules. The less information these rules need, the earlier they can be implemented in the parsing architecture, and the faster the resulting parsers are (Chapter 3).
- A completely general and feasible approach is to filter parse forests, provided they are compactly represented and traversed without revisiting (Chapter 4).

Furthermore, based on ample experience with the design of disambiguation filters, we formulate the following recommendations:

- There is no solution to be found in the search for a meaningful *generic* and *language independent* filter of a parse forest. Filters based on counting arguments, such as the multi-set filter [157], or the infamous injection count (shortest derivation) filter are low fidelity heuristics. Such filters, although they reduce the parse

#	Research questions	Chapter	Publication
1	How can disambiguations of context-free grammars be defined and implemented effectively?	3, 4	[46, 40]
2	How to improve the conciseness of meta programs?	5, 6	[39, 38]
3	How to improve the fidelity of meta programs?	7,8	[50]
4	How to improve the interaction of meta programs with their environment?	2, 9	[45, 44]

Table 10.1: Research questions in this thesis.

forest, lower the fidelity of the entire parsing architecture. This observation is based on countless examples provided by users of early versions of ASF+SDF where a disambiguation would filter the "wrong" derivation.

- Scannerless parsing needs specific disambiguation filters that are applied early in the parsing architecture to obtain feasible scannerless parser implementations.
- Term rewriting serves nicely as a method for implementing disambiguation filters.
- It is not always necessary to resolve ambiguity at all, as long as the entire analysis and transformation pipeline can deal with the parse forest representation.

Future work

Static analysis. The major drawback of generalized parsing as opposed to deterministic parsing technology is the lack of static guarantees: a generated parser may or may not produce multiple trees. Although ambiguity of context-free grammars is undecidable in general, we suspect that many ambiguities are easy to find automatically. To make SGLR parsing acceptable to a wider audience, tools could be developed that recognize frequently occurring patterns of unnecessary ambiguity.

Filter design. Disambiguation filters may be hard to design. Syntactic ambiguity borders the syntactic and semantic domains of programming language implementations. It is an artifact of context-free grammars, but filtering is always governed by language semantics considerations. For example, the priority of the multiplication over the addition operator of C does not change the syntax of the language, but it has severe implications for the interpretation of any C expression. This gap between syntax and semantic must be bridged by the designer of any disambiguation filter.

We noticed this gap when ambiguity clusters appeared in parse forests at "unexpected locations", higher in derivation trees and not at all localized near the cause of the ambiguity. We may document a number of disambiguation filter design patterns to help guide the programmer bridge this gap. On the other hand, more automation offered by ASF+SDF for defining disambiguation filters may be designed.

More generic disambiguation notions. More grammatical idioms may be identified that give rise to declarative disambiguation filters. For example, the current set of disambiguation constructs in SDF is not powerful enough to deal with instances of the offside rule [115]. This is such a frequently occurring design pattern in programming languages, that a generic disambiguation notation may be designed to facilitate the definition of languages with an offside rule.

10.1.2 How to improve the conciseness of meta programs?

Regarding the conciseness of meta programs we identified three aspects (Chapter 1):

- Traversal of parse trees,
- Managing context information,
- Expressing reusable functions.

Firstly, traversal functions have been added to ASF+SDF to deal with the traversal aspect (Chapter 6). Secondly, with a minor extension of ASF the programmer now has the ability to store and retrieve parse tree annotations to deal with the context information aspect. Finally, we have investigated how the type system of ASF+SDF can be extended to allow parametric polymorphism, such that reusable functions are more easily expressible (Chapter 5).

Conclusions

We can draw the following conclusions:

- By using function attributes, a language may offer generic tree traversal primitives without introducing the need for higher order functions as a programming language feature. Limiting the types of traversal to "transformers" and/or "accumulators" is key to obtain a sound type system.
- Traversal functions significantly improve the conciseness of source code analyses and transformations. Instead of having to write programs that grow with the size of a language, many transformations can be programmed in a few lines.
- Traversal functions can be implemented efficiently. Unnecessarily visiting nodes that a manual traversal might have skipped does not impose a noticeable bottleneck.
- The lightweight extension to ASF+SDF of parse tree annotations helps to separate data and control flow.

Informally, we note that traversal functions also support the distribution of context information. The reason is that a traversal may carry non-local information up and down a tree effortlessly. Another advantage is that the fidelity of transformations is generally higher when traversal functions are used. The reason is that no unnecessary node visits are made that may have unwanted side-effects.

Future work

Notation. The chosen notation for traversal functions is cumbersome. While a traversal function is in fact polymorphic in its first argument, we must repeat its definition for every type it applies to. The reason is that we are limited to the first order type system of ASF+SDF, as described in Chapter 5. In fact Chapter 5 provides us with a solution to introduce type parameters and disambiguate the application of polymorphic functions using type checking. It is worthwhile investigating whether we can reduce the notational overhead of traversal functions by applying this solution.

Traversal orders. The chosen set of traversal orders that are provided by the traversal attributes have remained a topic of discussion. The main missing orders are *right-to-left* visiting, and *fixed point* application at a certain level in a tree. However, these concepts naturally fit into the already existing scheme. Fixed point is an alternative to break and continue, and right-to-left is very similar to the bottom-up and top-down primitives. Although no such feature has been requested yet, we might consider adding them, since there is no practical coding idiom to simulate their behavior.

Data threading. Although traversal functions and parse tree annotations alleviate the context information issue, they do not solve the amount of explicit threading of parameters through function applications. Especially transformations that map source code to collections of facts appear to suffer from repetitive code when they pass around environments of already accumulated facts. In a procedural or object-oriented language this would be solved by global or field variables, but in a pure term rewriting formalism this is not feasible. We may apply solutions that have been found in the functional programming arena to this problem, such as "implicit parameters" [121].

Type checker. Based on the result in Chapter 5 we can define a type system for ASF+SDF that does allow parametric polymorphic functions. The idea is to limit polymorphic behavior to prefix function syntax. Improvements we expect from deploying such a type system are:

- Less distance between function definition and function declaration,
- Less syntactic overhead for defining traversal functions,
- The ability to provide *generic* functions without explicit binding of type parameters,
- The ability of ASF+SDF to provide more precise error messages.

10.1.3 How to improve the fidelity of meta programs?

Conclusions

- A prerequisite for high-fidelity is high-resolution of both the data structures that represent source code, and the algorithms that operate on them (Chapter 8).

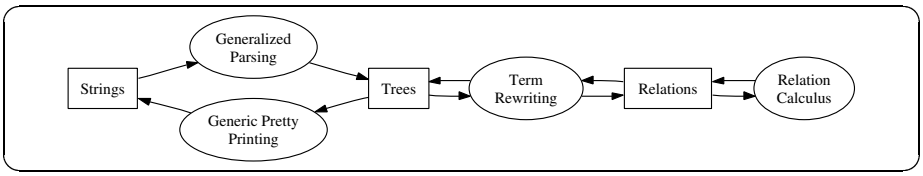

Figure 10.1: Four technologies in the Meta-Environment.

- Parse trees of source code with characters as leafs are a maximally high-resolution data structure. When the parse trees contain all characters of the original source code, including keywords, whitespace and source code comments, then they are also high-fidelity (Chapter 7).
- Such fully informative parse trees can be processed in an efficient manner to facilitate industrial sized transformation projects (Chapter 7).
- "First class layout" makes it possible to automatically process source code comments in order to obtain valuable information about a sofware system (Chapter 8).

Future work

Preprocessing. Several solutions have been proposed that improve the resolution and fidelity of transformation systems when they are applied to preprocessed languages such as C and C++ [165, 123]. Since the C preprocessor is reused for other languages than C, the question remains whether a high-fidelity language independent architecture can be constructed that deals with the problems introduced by the C preprocessor, or possibly with any preprocessor. One particular question is if fully informative parse trees can be reused as a central data structure in such an architecture.

Compilation. All experiments carried out with respect to high-fidelity have been done using the ASF interpreter. A straightforward adaptation of the ASF compiler would make this functionality available for compiled specifications. The question is whether interesting optimizations are possible. The implementation of parse tree equality is an obvious candidate for optimization. For example, parse trees that are constructed after reading in the source code, and never appear inside the resulting source code, need not to be treated as high-fidelity parse trees. Such a static analysis may dramatically improve the run-time behavior of high-fidelity meta programs.

10.1.4 How to improve the interaction of meta programs with their environment?

Conclusions

We have studied the basic prerequisites for such interaction: data integration and coordination. We have presented a generic architecture for hosting meta programming

tools in, and a Java code generation tool to obtain typed interfaces. We now have the following recommendations:

- By the use of API generation, we enforce a typed contract between two components that share certain data. This is an essential feature for managing the evolution of a complex component based systems.
- Although it comes with an initial investment, explicitly allowing *heterogeneity* of programming languages in a single system ensures both the openness to foreign tools, and the continuity of the system itself. The main benefit is that we can incrementally refit components without changing their interface to the rest of the system.

Future work

Java back-end for ASF+SDF. We have written a prototype Java back-end for ASF+SDF, using ApiGen for Java and the TOM system [127]. A full implementation of an ApiGen based back-end should improve on the previous implementations of the compiler in terms of interaction. The idea is that the code ApiGen generates is optimized toward human readability. A compiler back-end using such APIs would also generate readable code, such that compiled ASF+SDF programs can be used as libraries directly instead of commandline tools.

Pretty printing. Figure 10.1 repeats the architecture of the ASF+SDF Meta-Environment from the introduction. In this thesis we have studied the parsing and term rewriting components, and how they interact. The pretty printing and relational calculus components have remained untouched. Recently, advancements have been made on the interaction between rewriting and pretty printing [41]. However, that connection needs to be restudied with *high-fidelity* as a requirement.

Relational calculus. The other bridge that needs studying is the connection between term rewriting and relational calculus. As a first important step, we have standardized the location abstraction between ASF and RScript. However, the current collaboration between the two tiers is still done using files and a collection of commandline tools. A tighter, type safe connection, with fewer overhead for the meta programmer must be designed. The main problem to overcome is the snapshot/update problem, for which it seems reasonable to obtain inspiration from the database community.

Connections to other formalisms. Finally, the standardization and interaction of the data structures within the ASF+SDF Meta-Environment is not enough. In order to harvest data and algorithms from other foreign tools, a number of import/export filters need to be constructed. For example, grammars written in Yacc [92] can be imported to SDF. Also, the relations used in RScript may be exported to GXL [170], or any other standardized data format such that other fact analysis tooling can be easily applied. The design of such import/export filters should not be underestimated, since the semantics of each formalism may be unclear and thus offer only a weak foundation for a sound translation.

10.2 Discussion: meta programming paradigms

In this thesis we have put term rewriting as a central paradigm for analysis and transformation of source code. We have concluded that the language constructs it provides, together with the constructs we have added, are a valid collection of features that are well suited for tackling the meta programming application domain.

A different philosophy is to start from a more globally accepted programming paradigm, such as object-oriented programming. The method is to carry selected features of term rewriting and functional programming over to this paradigm. Instances of such features are algebraic data types (Chapter 9), generic traversal (JJTraveler [112]), pattern matching (JTom [127]), and concrete syntax (JavaBorg [53]).

The first philosophy is beneficial because it reuses many language features that are readily available, the second philosophy is beneficial because of the ease of integration with other software. The point is that not the paradigm as a whole, but rather some particular features it offers make it apt in the meta programming domain. These features may be implemented as native language constructs, as library interfaces, or as preprocessors. From the perspective of meta programming, it is more fundamental to know the domain specific language constructs that are needed, than to know the preferred choice of programming paradigm to host them in.

10.3 Software

The empirically oriented research projects in software engineering often lead to the development of substantial amounts of software. This software is not always easily identifiable in scientific publications. Such software sometimes has more one than goal: Firstly, it is created by the researcher to test and validate new solutions to research questions. Secondly, to provide these solutions to a broader audience, as immediately usable tools. Thirdly, some research software serves as an infrastructure, or laboratory, for other research. The ASF+SDF Meta-Environment, including the software developed in the context of this thesis, has all the above goals.

We will, briefly, list the results of the design, implementation and maintenance effort that was carried out in the context of this thesis. This description mentions features and design only. Figures in terms of code volume or time spent are not provided, since such measures are not only hard to obtain, but their interpretation is also highly controversial.
These are the three categories we concentrated on:

Meta-Environment: generic or cross cutting functionality,

SDF: parsing and disambiguation data structures and algorithms,

ASF: analysis and transformation data structures and algorithms.

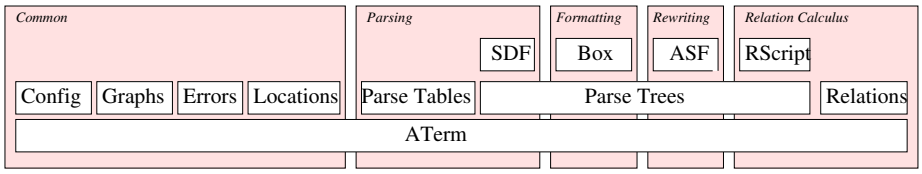

Figure 10.2: Three layers, and five columns of data structures in the Meta-Environment.

10.3.1 Meta-Environment

A number of tasks have been carried out that improve the separation of concerns, while maintaining consistency and uniformity in the design of the Meta-Environment. Note that the results in this section are obtained by close teamwork of several people within the Generic Language Technology project.

Build environment. We have separated the system into separate source code packages, that can be build, tested and deployed individually. The packages have explicit dependencies and are recombined for convenience at deployment time [65]. We standardized on the use of GNU tools: autoconf, automake, libtool, gmake, gcc. A daily build system was deployed to ensure continuous quality assurance. One of the beneficial results of the above exercises is that several institutes and companies now eclectically download and reuse slices of the ASF+SDF Meta-Environment (Chapter 2).

Stratification of data structures. Using ApiGen to generate data structures (Chapter 9), we have reimplemented all data structures in the Meta-Environment and divided them over three conceptual layers and five topical columns (Figure 10.2). The design is stratified such that each column is consistently designed and implemented. Note that each layer was designed to hide the lower layer in a completely type safe interface. After refactoring all of the existing code, the readability and maintainability of the algorithms that use these data structures was improved radically [62]. Note that the relational calculus column, and the formatting column are depicted here only for completeness.

Generic IDE. We have migrated the Meta-Environment from a hard coded ASF+SDF IDE to a generic programming environment (Chapter 2). The ToolBus is always used as a generic platform for heterogeneous distributed programming. On top of that platform we now offer a set of generic programming environment tools, which we specialize and bind much later to obtain the ASF+SDF Meta-Environment. This means that a number of important functionalities have been refactored to be either completely anonymous or highly parameterized. We created generic tools for file I/O, in-memory storage (state management), (structure) editing, error management and visualization, *locations* as a general abstraction for referencing to areas in files, configuration management, including configurable menus and button bars, visualization of graphs (in-

cluding import structures). Finally we integrated TIDE [132], an existing generic debugging platform into the Meta-Environment.

10.3.2 SDF

More specific to the subject of this thesis we have adapted the implementation of SDF. This work was supervised by and done together with with Mark van den Brand.

Redesign of SGLR filters. For backward compatibility with the original ASF+SDF Meta-Environment (Chapter 1), SGLR contained several heuristics based disambiguation filters that were aimed at optimizing the rapid development of language descriptions. We intensively studied the characteristic behavior of these filters and then switched them off, since their heuristic nature severely lowers the fidelity of the parsing architecture.

The implementation of the remaining filters was redesigned. An extreme separation of the disambiguation filters from the parsing algorithm proved to be beneficial for both clarity of code and efficiency. Due to the nondeterministic nature of scannerless generalized parsing it appears to be better to wait for the final parse forest and filter it, than to filter partial parse forests that later not survive anyway (Chapter 3).

Parse forests. We have experimented with different encodings of parse forests as maximally shared ATerms. Note that the parse forests data structure cross cuts several topical columns in the design of ASF+SDF (Figure 10.2), so a change in this format means updating many components. The design trade-off is: high-resolution (fully informative) versus low memory footprint. We decided on an ATerm representation containing full derivation trees down the character level. We do flatten the applications of productions that produce *lists* to plain ATerm lists (Chapter 7 and Chapter 8). The reason is that in ASF list concatenation is an *associative operator*, thus all elements are on the same level from a semantics point of view.

Position information. We added a separate tool that annotates parse trees with position information. Each annotated sub-tree now has a unique identity. For the annotation we use the *location* abstraction discussed before that is standardized across the Meta-Environment.

Annotating all sub-trees with their location is an enabling feature for a lot of functionality in the Meta-Environment:

- Unique identity of source code artifacts is a primary requirement in source code analyses, cross-linking, and visualization,
- A generic syntax highlighting tool traverses any parse tree and creates a mapping of locations to categories of font parameters. This can be given as input to any editor that can manipulate font attributes by position information (e.g., Emacs, GVim, JEdit),
- Easy production of error and warning messages that contain references to the causes of an error.

- Origin tracking, to be able to find the original code that seeded the existence of new code after a source code transformation, is trivially implemented by forwarding position annotations during term rewriting.

Visualization. We developed a parse forest visualization tool by mapping parse trees to the generic graph representation data structure. This tool aides enormously in the analysis of ambiguity, and the design of disambiguation filters.

10.3.3 ASF

Apart from improving the architecture of the ASF implementation, we have extended and adapted ASF in several ways. The goals were to test and validate several research projects, service the application area of meta programming and to improve the general flexibility of ASF+SDF. This has been a mostly personal project, supervised by Paul Klint and Mark van den Brand.

Architecture. The goal is to improve the decomposition of tools and algorithms needed to parse and finally compile or interpret ASF specifications. The implementation of ASF as a separate formalism from SDF is somewhat controversial because of the close cooperation. In the original system and the first versions of the renovated Meta-Environment, there was a completely tight coupling on the implementation level between ASF and SDF. We separated the implementation of ASF as a collection of independent components. In particular we introduced the ASF normalizer tool, which manipulates ASF specifications in a back-end neutral fashion (Chapters 8).

Renovation. We have renovated the implementation of the ASF compiler and the ASF interpreter. The goals were to make the implementation of the ASF compiler independent of the SDF formalism, to improve the flexibility of the ASF interpreter to facilitate easy experimentation with new language features (Chapters 4, 6, 7, 8), and finally to be able to handle the new parse tree format necessary for high-fidelity transformations (Chapters 7, 8).

Optimization. We have improved the efficiency of compiled ASF specifications by introducing optimizations for *recursive functions*, and *list matching*. Recursive applications of functions on lists are a very common coding idiom in ASF. The introduction of these optimizations have led to an average 15% performance gain.

New features. Thirdly, we have added a number of features and language constructs to ASF. Among those are syntax and semantics for *unit testing* ASF programs, a new syntax and semantics for conditions that raises their intentionality, the implementation of "rewriting with layout" (Chapter 7), traversal functions (Chapter 6), the addressability of ambiguity clusters (Chapter 4) and "first class layout" (Chapter 8).

Built-in architecture. An important change in the design of ASF was the addition of an architecture for *built-in functions*. Before, ASF was a completely pure language, now it has a *fixed* set of co-routines that can be called from any ASF program. This design has solved several bottlenecks in the application to the meta programming paradigm. For example, reading in several input source code files and producing several output source code files is not an issue any more. The following *fixed* set of built-ins have been added:

- Setting and getting of parse tree annotations, including *position information* in terms of locations,
- Calling parsing and unparsing during rewriting,
- Posix functionality: file I/O, pipes and other system calls,
- Reflection: lifting user-defined syntax to the parse tree level, and back,
- Debugging: a generic interface between ASF and the TIDE generic debugging system, which can be used to rapidly implement interactive debuggers for domain specific languages [26].

Bibliography

[1] A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 96–105. ACM Press, 1988.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] R. op den Akker, B. Melichar, and J. Tarhio. Attribute Evaluation and Parsing. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 187–214. Springer-Verlag, 1991.

[4] H. Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 1–15, Berlin Heidelberg New York, 1991. Springer Verlag.

[5] A.W. Appel and M.J.R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

[6] B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, April 1995.

[7] J. Aycock. Why Bison is becoming extinct. *ACM Crossroads*, Xrds-7.5, 2002.

[8] J. Aycock and R.N. Horspool. Faster generalized LR parsing. In S. Jähnichen, editor, *CC'99*, volume 1575 of *LNCS*, pages 32–46. Springer-Verlag, 1999.

[9] J. Aycock and R.N. Horspool. Directly-executable earley parsing. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 299–243, Genova, Italy, 2001. Springer-Verlag.

[10] J. Aycock and R.N. Horspool. Schrödinger's token. *Software, Practice & Experience*, 31:803–814, 2001.

[11] J. W. Backus, J. H. Wegstein, A. van Wijngaarden, M. Woodger, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Report on the algorithm language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[12] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Cambridge University Press, 1990.

[13] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[14] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[15] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[16] M. Bezem, J.W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, 2003.

[17] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th annual meeting on Association for Computational Linguistics*, pages 143–151, Morristown, NJ, USA, 1989. Association for Computational Linguistics.

[18] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer-Verlag, 2001.

[19] F. Bonsu. Graphic generation language: Automatic code generation from design. Master's thesis, Universiteit van Amsterdam, 1995.

[20] P. Borovanský. *Le controle de la réécriture : étude et implantation d'un formalisme de stratégies.* PhD thesis, Université Henri Poincaré, October 1998.

[21] P. Borovanský, S. Jamoussi, P.-E. Moreau, and Ch. Ringeissen. Handling ELAN Rewrite Programs via an Exchange Format. In *Proc. of [98]*, 1998.

[22] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[23] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *WRLA*, volume 15 of *ENTCS*. Elsevier Sciences, 1998.

[24] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *1st Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Sciences, 1996.

[25] M.G.J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[26] M.G.J. van den Brand, B. Cornelissen, P.A. Olivier, and J.J Vinju. TIDE: a generic debugging framework. In J. Boyland and G. Hedin, editors, *Language Design Tools and Applications*, June 2005.

[27] M.G.J van den Brand and M. de Jonge. Pretty-printing within the ASF+SDF Meta-Environment: a generic approach. Technical Report SEN-R9904, CWI, 1999.

[28] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[29] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[30] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[31] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[32] M.G.J. van den Brand and P. Klint. *ASF+SDF User Manual Release 1.5*, 2004. `http://www.cwi.nl/projects/MetaEnv`.

[33] M.G.J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC '99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[34] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[35] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation: an annotated bibliography. *ACM Software Engineering Notes*, 22(1):42–57, January 1997.

[36] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications, WRLA 98*, 1998.

[37] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.

[38] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term Rewriting with Type-safe Traversal Functions. In B. Gramlich and S. Lucas, editors, *Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[39] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):152–190, 2003.

[40] M.G.J. van den Brand, S. Klusener, L. Moonen, and J.J. Vinju. Generalized Parsing and Term Rewriting - Semantics Directed Disambiguation. In Barret Bryant and João Saraiva, editors, *Third Workshop on Language Descriptions Tools and Applications*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[41] M.G.J. van den Brand, A.T. Kooiker, N.P. Veerman, and J.J. Vinju. Context-sensitive formatting – extended abstract. In *International Conference on Software Maintenance*, 2005. accepted for publication.

[42] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and implementation of a new ASF+SDF meta-environment. In M.P.A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, 1997. Springer/British Computer Society.

[43] M.G.J. van den Brand, P.-E. Moreau, and C. Ringeissen. The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In M.G.J van den Brand and R. Lämmel, editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65, Grenoble (France), April 2002. Electronic Notes in Theoretical Computer Science.

[44] M.G.J. van den Brand, P.-E. Moreau, and J.J. Vinju. Environments for Term Rewriting Engines for Free! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.

[45] M.G.J. van den Brand, P.-E. Moreau, and J.J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software*, 152,2:70–78, April 2005.

[46] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. Nigel Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

[47] M.G.J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 144, Washington, DC, USA, 1997. IEEE Computer Society.

[48] M.G.J. van den Brand, A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy system. In *CSMR*, pages 11–20, 1998.

[49] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36:209–266, 2000.

[50] M.G.J. van den Brand and J.J. Vinju. Rewriting with layout. In Claude Kirchner and Nachum Dershowitz, editors, *Proceedings of RULE2000*, 2000.

[51] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[52] M. Bravenboer, R. Vermaas, J.J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Generative Programming and Component Engineering (GPCE)*, 2005. to appear.

[53] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[54] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, 2005. to appear.

[55] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press, 1997.

[56] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[57] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[58] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences, 1996.

[59] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[60] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[61] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., sixth edition, 1994.

[62] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, 2004.

[63] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[64] M. de Jonge. Pretty-printing for software reengineering. In *Proceedings: International Conference on Software Maintenance (ICSM 2002)*, pages 550–559. IEEE Computer Society Press, October 2002.

[65] M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, April 2002.

[66] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Parigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[67] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[68] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[69] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[70] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[71] A. van Deursen and L. Moonen. Type inference for COBOL systems. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proc. 5th Working Conf. on Reverse Engineering*, pages 220–230. IEEE Computer Society, 1998.

[72] K.-G. Doh and P.D. Mosses. Composing programming languages by combining action-semantics modules. In M.G.J. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44, 2001.

[73] A. Felty. A logic programming approach to implementing higher-order term rewriting. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming (ELP '91)*, volume 596 of *Lecture Notes in Artifial Intelligence*, pages 135–158. Springer-Verlag, 1992.

[74] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[75] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[76] J.F. Gimpel. *Algorithms in SNOBOL4*. John Wiley & Sons, 1976.

[77] J. Goubault. HimML: Standard ML with fast sets and maps. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.

[78] J. Grosch. Ast – a generator for abstract syntax trees. Technical Report 15, GMD Karlsruhe, 1992.

[79] Object Management Group. *MDA – Model Driven Architecture*. `http://www.omg.org/mda`.

[80] Object Management Group. *UML – Unified Modelling Language*. `http://www.uml.org`.

[81] XML Schema Working Group. *W3C XML Schema*. Available at: `http://www.w3c.org/XML/Schema`.

[82] D.R. Hanson. Early Experience with ASDL in lcc. *Software - Practice and Experience*, 29(3):417–435, 1999.

[83] G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Lund University, 1992.

[84] G. Hedin and E. Magnusson. JastAdd - a Java-based system for implementing frontends. In M.G.J. van den Brand and D. Parigot, editors, *Proc. LDTA'01*, volume 44-2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[85] J. Heering. Implementing higher-order algebraic specifications. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 141–157. University of Pennsylvania, Philadelphia, 1992. Published as Technical Report MS-CIS-92-86.

[86] J. Heering. Second-order term rewriting specification of static semantics: An exercise. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping*, volume 5 of *AMAST Series in Computing*, pages 295–305. World Scientific, 1996.

[87] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[88] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1350, 1990.

[89] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell. A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[90] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[91] M. Johnson. The computational complexity of glr parsing. In M. Tomita, editor, *Generalized LR Parsing*, pages 35–42. Kluwer, Boston, 1991.

[92] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[93] M. de Jonge and J. Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *LNCS*, pages 85–99, Erfurt, Germany, 2001. Springer-Verlag.

[94] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In H. Cirstea and N. Marti-Oliet, editors, *RULE 2005*, 2005. to appear.

[95] D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[96] U. Kastens, P. Pfahler, and M. T. Jung. The eli system. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 294–297, London, UK, 1998. Springer-Verlag.

[97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[98] C. Kirchner and H. Kirchner, editors. *Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), September 1998. Elsevier.

[99] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[100] P. Klint. Is strategic programming a viable paradigm? In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57/2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[101] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[102] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[103] P. Klint, T. van der Storm, and J.J. Vinju. Term rewriting meets aspect-oriented programming. Technical Report SEN-E0421, CWI, 2004.

[104] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

[105] J.W. Klop. Term rewriting systems. In D. Gabbay, S.Abramski, and T. Maibaum, editors, *Handbook of Logic and Computer Science*, volume 1. Oxford University Press, 1992.

[106] D.E. Knuth. *The Art of Computer Programming, Volume 1*. Addison-Wesley, 1968.

[107] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proc. Source Code Analysis and Manipulation (SCAM'03)*. IEEE Computer Society Press, September 2003. 10 pages.

[108] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M.G.J. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[109] R. Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, 2003.

[110] R. Lämmel and C. Verhoef. Semi-Automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[111] R. Lämmel and C. Verhoef. VS COBOL II grammar[1], 2001.

[112] R. Lämmel and J. Visser. Typed combinators for generic traversal. In *PADL 2002: Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2002.

[1] `http://www.cs.vu.nl/grammars/browsable/vs-cobol-ii/`

[113] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Published as Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

[114] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M.G.J. van den Brand and D. Parigot, editors, *Proc. LDTA'01*, volume 44-2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[115] P.J. Landin. The next 700 programming languages. In *CACM*, volume 9, pages 157–165, March 1966.

[116] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *LNCS*, pages 255–269. Springer-Verlag, 1974.

[117] S.B. Lassen, P.D. Mosses, and D.A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In *Proc. 3rd International Workshop on Action Semantics*, volume NS-00-6 of *Notes Series*, pages 19–36. BRICS, 2000.

[118] B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.

[119] M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *Proceedings of the 2nd international conference on Software engineering*, pages 350–357. IEEE Computer Society Press, 1976.

[120] M. E. Lesk and E. Schmidt. *LEX — A lexical analyzer generator*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[121] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–118, Jan 2000.

[122] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. In J. Boyland and G. Hedin, editors, *Fifth workshop on language descriptions tools and applications (LDTA)*, April 2005.

[123] A.J. Malton, K.A. Schneider, J.R. Cordy, T.R. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *IWPC '01: Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'01)*, page 127, Washington, DC, USA, 2001. IEEE Computer Society.

[124] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Mass., 1966.

[125] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[126] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, 2001.

[127] P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

[128] P.D. Mosses. Action semantics and ASF+SDF: System demonstration. In *Language Design Tools and Applications*, volume 65-3 of *ENTCS*, 2002.

[129] P.D. Mosses and J. Iversen. Constructive action semantics for core ML. In P. Klint, editor, *Special issue on Language definitions and tool generation*, volume 152,2, pages 79–98. IEE Proceedings - Software, April 2005.

[130] R. Nozohoor-Farshi. Handling of ill-designed grammars in tomita's parsing algorithm. In *Proceedings of the International Parsing Workshop*, pages 182–192, 1989.

[131] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[132] P. A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, 2000.

[133] J. Paakki. Attribute grammar paradigms: a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[134] J. Palsberg, K. Tao, and W. Wang. Java tree builder. Available at `http://www.cs.purdue.edu/jtb`. Purdue University, Indiana, U.S.A.

[135] T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software – Practice & Experience*, 7(25):789–810, 1995.

[136] H.A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer-Verlag, 1990.

[137] J. van de Pol. JITty: a Rewriter with Strategy Annotations. In S. Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *LNCS*, pages 367–370. Springer-Verlag, 2002.

[138] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[139] D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, 1989.

[140] D.J. Salomon and G.V. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Dept. of Computer Science, University of Manitoba, 1995.

[141] E. Scott, A. Johnstone, and S.S. Hussain. Tomita-Style Generalised LR Parsers. Technical Report TR-00-12, Royal Holloway, University of London, Computer Science Dept., 2000.

[142] A. Sellink and C. Verhoef. Scaffolding for software renovation. In *Proceedings of the Conference on Software Maintenance and Reengineering*, page 161. IEEE Computer Society, 2000.

[143] M.P.A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In *Proceedings of Conference on Maintenance and Reengineering (CSMR'99)*, pages 72–82, Amsterdam, March 1999.

[144] M.P.A. Sellink and C. Verhoef. Native patterns. In M.R. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society Press, 1998.

[145] A. A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

[146] Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[147] F. Tip and T.B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10:5–55, January 2001.

[148] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.

[149] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[150] D.A. Turner. *SASL Language Manual*. University of Kent, Canterbury, 1979.

[151] C. van Reeuwijk. Rapid and Robust Compiler Construction Using Template-Based Metacompilation. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 247–261. Springer-Verlag, May 2003.

[152] J. van Wijngaarden and E. Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University., May 2003.

[153] N.P. Veerman. Revitalizing modifiability of legacy assets. In M.G.J. van den Brand, G. Canfora, and T. Gymóthy, editors, *7th European Conference on Software Maintenance and Reengineering*, pages 19–29. IEEE Computer Society Press, 2003.

[154] J.J. Vinju. A type driven approach to concrete meta programming. Technical Report SEN-E0507, CWI, 2005.

[155] J.J. Vinju. Type-driven automatic quotation of concrete object code in meta programs. In *Rapid Integration of Software Engineering techniques*, Lecture Notes in Computer Science. Springer-Verlag, September 2005. to appear.

[156] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[157] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[158] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming*, pages 86–104, Ponte de Lima, July 2000. Published as Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

[159] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[160] E. Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57/2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[161] E. Visser. Meta-programming with concrete object syntax. In D. Batory and C. Consel, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*. Springer-Verlag, 2002.

[162] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[163] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

[164] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Notices*, 24(7):131–145, 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[165] D.G. Waddington and Bin Yoa. High-fidelity C/C++ code transformation. In J. Boyland and G. Hedin, editors, *Fifth workshop on language descriptions tools and applications (LDTA)*, April 2005.

[166] T.A. Wagner and S.L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 31–43. ACM Press, 1997.

[167] L. Wall. *Practical Extraction and Report Language*. O' Reilly. `http://www.perl.com/pub/doc/manual/html/pod/perl.html`.

[168] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

[169] H. Westra. Configurable transformations for high-quality automatic program improvement. CobolX: a case study. Master's thesis, Utrecht University, February 2002.

[170] A. Winter, B. Kullbach, and V. Riediger. An overview of the gxl graph exchange language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

[171] H. Zaadnoordijk. Source code transformations using the new ASF+SDF meta-environment. Master's thesis, University of Amsterdam, Programming Research Group, 2001.

[172] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–19, Vancouver, Canada, October 2004. Springer.

CHAPTER 11

Samenvatting

De titel van dit proefschrift is "Analysis and Transformation of Source Code by Parsing and Rewriting", ofwel "Het analyseren en aanpassen van broncode met behulp van ontleden en termherschrijven".

Er is een grote hoeveelheid broncode van computerprogramma's die aangepast moet worden aan voortdurend veranderende eisen en omstandigheden. Het is voor bedrijven vaak aantrekkelijker om aanpassingen aan te brengen doen in de broncode waarin al veel geld is geïnvesteerd dan om helemaal opnieuw te beginnen. Om de kosten van dergelijke aanpassingen te verlagen, kan men proberen het analyseren en aanpassen van broncode te automatiseren met behulp van de computer zelf. Dit vormt het centrale thema van mijn onderzoek.

Het gereedschap dat ik hiervoor gebruik is de programmeertaal ASF+SDF. De algemene vraag is in hoeverre deze taal inderdaad geschikt is voor het uitvoeren van analyses en aanpassingen in broncode. Het resultaat van dit onderzoek is een nieuwe versie van ASF+SDF waarin een aantal functionele aspecten en kwaliteitsaspecten zijn verbeterd. Ten eerste is het probleem van ambiguïteit in contextvrije grammatica's aangepakt. Ten tweede kunnen analyse en aanpassing beknopter worden gedefinieerd door de introductie van zogenaamde Traversal Functions. Ten derde is de precisie verbeterd, door nu alle aspecten van broncode te kunnen analyseren en aanpassen inclusief het commentaar in broncode. Ten vierde zijn de mogelijkheden tot samenwerken met andere softwarecomponenten uitgebreid.

11.1 Inleiding

De bovengenoemde automatisering van taken van programmeurs gaat met behulp van "metaprogrammeren". Dit is het schrijven van broncode die andere broncode manipuleert. Het creëren van metaprogramma's is absoluut geen sinecure. Dat komt niet alleen door de technische problemen die daarbij overwonnen moeten worden. Vooral de grote hoeveelheid details die in ogenschouw moet worden genomen om de kwaliteit van het eindproduct, het analyserapport of de aangepaste broncode, te waarborgen

is daar debet aan. Bovendien is het essentieel dat de investering in het ontwikkelen van dergelijke metaprogramma's opweegt tegen de kosten van het gewoon met de hand uitvoeren van de gewenste taken.

Zonder verder in te gaan op een kosten- en batenanalyse, probeer ik in dit proefschrift het proces van het ontwikkelen van nieuwe metaprogramma's te stroomlijnen door gereedschappen daarvoor te ontwikkelen of aan te passen. Deze gereedschappen worden steeds beoordeeld op hun effectiviteit en vooral ook op de kwaliteit van het eindproduct waaraan ze bijdragen. De onderzoeksmethode is empirisch:

- Stel een nieuwe eis of tekortkoming vast,
- Ontwikkel een nieuw gereedschap of pas een bestaand gereedschap aan,
- Gebruik het nieuwe gereedschap om een taak te automatiseren en voer die taak vervolgens uit op een (industriële) verzameling broncode,
- Beoordeel het resultaat op bepaalde kwaliteitsaspecten en vergelijk de aanpak met concurrerende of anderszins gerelateerde technieken,
- Formuleer een conclusie.

Het basisgereedschap is al aanwezig. Ik gebruik de programmeertaal ASF+SDF. Deze taal leent zich, in principe, voor het ontleden en vervolgens analyseren of aanpassen van broncode. In een aantal industriële projecten is ASF+SDF toegepast om metaprogramma's te construeren. Er zijn echter een aantal haken en ogen die de toepasbaarheid beperken. De reden hiervoor is dat ASF+SDF oorspronkelijk niet ontworpen was met het doel om broncode in bestaande talen te analyseren of aan te passen maar juist om te experimenteren met nieuwe programmeertalen. Deze verschuiving in het toepassingsgebied leidt onvermijdelijk tot een ander eisenpakket.

ASF+SDF bevat twee geavanceerde functies die gebruikt kunnen worden bij het analyseren en aanpassen van broncode: ontleden en termherschrijven. Het doel van het ontleden van broncode is om te achterhalen wat de structuur ervan is, net zoals dat het doel is bij het ontleden van Nederlandse zinnen. Het ontleden is vaak de eerste stap bij het systematisch onderzoeken van de betekenis van broncode.

Het tweede gereedschap dat ASF+SDF aanbiedt is termherschrijven. Dat is een techniek waarmee gestructureerde informatie kan worden gemanipuleerd. Een *term* is een structuur die getekend kan worden als een boom. Het *herschrijven* gebeurt door middel van het herkennen van patronen in een dergelijke boom, en die door andere patronen te vervangen. De structuur die het resultaat is van ontleden kan gezien worden als een term. Zo kan termherschrijven gebruikt worden om broncode te manipuleren, namelijk door de bijbehorende structuren te herschrijven.

Tenslotte wordt ASF+SDF ondersteund door een interactieve omgeving: de Meta-Environment. Onder de Meta-Environment verstaan we de user-interface en andere software componenten die het ontwikkelen en gebruiken van ASF+SDF programma's ondersteunen.

#	Onderzoeksvragen	Hoofdstuk	Publicatie
1	Hoe kan men de disambiguatie van contextvrije talen op effectieve wijze definiëren en implementeren?	3, 4	[46, 40]
2	Hoe kunnen metaprogramma's beknopter worden geformuleerd?	5, 6	[39, 38]
3	Hoe kan de precisie van metaprogramma's verbeterd worden?	7,8	[50]
4	Hoe kan de samenwerking tussen metaprogramma's en hun omgeving verbeterd worden?	2, 9	[45, 44]

Tabel 11.1: Onderzoeksvragen in dit proefschrift.

11.2 Onderzoeksvragen

In tabel 11.1 staan de onderzoeksvragen opgesomd met verwijzingen naar de relevante hoofdstukken en bijbehorende publicaties. Ik ga kort in op elke onderzoeksvraag.

Hoe kan men de disambiguatie van contextvrije talen op effectieve wijze definiëren en implementeren? Hoe groter de uitdrukkingskracht van een programmeertaal is, des te lastiger het is om een éénduidige betekenis te achterhalen tijdens het ontleden van broncode. Er ontbreekt allerlei achtergrondinformatie die nodig is om te kunnen kiezen. In dat geval levert ASF+SDF meerdere structuren af, omdat geen éénduidige keuze gemaakt kan worden. We spreken in dit geval van *ambiguïteit*. De vraag is op welke manier we de missende achtergrondinformatie effectief kunnen formaliseren, opdat we ASF+SDF wel een éénduidige en bovendien correcte keuze kunnen laten maken.

Op verschillende manieren probeer ik in dit proefschrift het concept van *disambiguatiefilters* toe te passen. Een dergelijk filter is een manier om achtergrondinformatie te kunnen gebruiken bij de keuze tussen verschillende structuren. Bij elk filter kun je de vraag stellen waar de informatie vandaan komt (de formalisering van bepaalde feiten) en vervolgens hoe die informatie tot een correct en efficiënt filter leidt (de implementatie). Voor verschillende klassen van ambiguïteiten worden in dit proefschrift verschillende methodes voorgesteld en met succes toegepast.

Hoe kunnen metaprogramma's beknopter worden geformuleerd? De begrippen "context informatie" en "volgorde van het uitvoeren van berekeningen" komen niet direct tot uiting in termherschrijfsystemen. Juist bij metaprogramma's zijn context informatie en de precieze controle van de volgorde bij het uitvoeren van aanpassingen belangrijk. Het verspreiden van belangrijke informatie naar de punten in het ASF+SDF programma waar deze nodig is, moet daarom gesimuleerd worden. Ook de controle op de volgorde van toepassing van herschrijfregels moet gesimuleerd worden. Hierdoor worden ASF+SDF programma's soms onnodig ingewikkeld en onnodig lang. Ik stel

een nieuw taalconcept voor dat "Traversal Functions" heet, waarmee beide problemen kunnen worden aangepakt. Traversal Functions automatiseren het doorlopen van ingewikkelde boomstructuren in allerlei verschillende volgordes, en kunnen daarbij gemakkelijk contextinformatie propageren.

Hoe kan de precisie van metaprogramma's verbeterd worden? Tijdens het uitvoeren van ASF+SDF programma's verdwijnt sommige informatie uit de originele broncode. De reden is dat het *oplossend vermogen* niet groot genoeg is en daardoor laat ook de *precisie* te wensen over. In ASF+SDF is het namelijk niet mogelijk om commentaar, dat secundair aan het programma in de broncode aanwezig is, te bewaren of überhaupt aan te spreken tijdens het termherschrijven. Dat commentaar is meestal bedoeld voor de lezer van de broncode om ingewikkelde formuleringen toe te lichten. Het verlies van dergelijke documentatie zou desastreus kunnen zijn voor het voortbestaan van de broncode. In dit proefschrift worden oplossingen aangedragen om commentaar tijdens analyses en aanpassingen aan te spreken en te behouden.

Communiceren met de buitenwereld. Een van de vele uitdagingen in de ICT is nog steeds het combineren van verschillende technieken en het op elkaar aansluiten van verschillende systemen. Zo ook kan de ASF+SDF taal zelf maar moeizaam samenwerken met andere systemen. Op twee manieren probeer ik hiervoor een oplossing te bieden. Ten eerste probeer ik de taal ASF+SDF onafhankelijk te maken van de Meta-Environment opdat zowel de Meta-Environment als ASF+SDF los van elkaar en in meerdere omstandigheden toepasbaar zijn (zie hoofdstuk 2). Ten tweede heb ik gewerkt aan een efficiënte communicatielijn richting de populaire programmeertaal Java (zie hoofdstuk 9).

11.3 Conclusie

Dit proefschrift bevat onderzoeksresultaten ten behoeve van het automatiseren van taken van programmeurs. Het resultaat is een bundel praktisch toepasbare gereedschappen. Ondanks het feit dat ASF+SDF is gebruikt als "laboratorium" voor dit onderzoek, zijn de lessen die geleerd zijn toepasbaar in andere systemen voor metaprogrammeren.

De methode die gebruikt is, is gebaseerd op het praktisch toepassen en evalueren van die nieuwe ontwikkelde technieken. Mede daardoor heeft de software van de ASF+SDF Meta-Environment gedurende het onderzoek een behoorlijke ontwikkeling doorgemaakt. Mijn bijdrage daaraan staat beschreven in sectie 10.3.

Alhoewel metaprogrammeren nog steeds een specialisme is, verwacht ik dat de beschikbaarheid van praktisch en precies gereedschap metaprogrammeren voor een breder publiek van programmeurs en softwarebedrijven toegankelijk zal maken.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of

Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between*

System Components. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19