CWI Tracts

Managing Editors

Editorial Board

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

CWI Tract 42

Essays on concepts, formalisms, and tools

editid by
P.R.J. Asveld
A. Nijholt

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

1980 Mathematics Subject Classification: 68-02, 68-03, 68B05, 68C01, 68C05, 68C20, 68C30, 68C40, 68D22, 68D25, 68D35, 68Fxx, 68F05, 68F10, 68F20, 68F25, 68G15, 68G99.
1983 CR Classification Scheme: D.1.2, D.1.4, D.2.2, D.3.1-4, F.0, F.1.1-3, F.2.2, F.3.2-3, F.4.0-3, G.1.0, I.1.1-2, I.2.3, K.2.
ISBN 90 6196 326 5
NUGI-code: 811

Essays on Concepts, Formalisms, and Tools

A Collection of Papers Dedicated
to Leo A.M. Verbeek

PREFACE

The papers in this book are dedicated to Leo A.M. Verbeek, professor of (Theoretical) Computer Science at Delft University of Technology (1968-1974) and at Twente University of Technology (1974-1987). Each of the authors has had the privilege to spend part of his scientific life in the stimulating atmosphere created by Leo Verbeek. Students, Ph.D. students, assistants, and colleagues have had the opportunity to benefit from his attitude and integrity with respect to teaching, research, and human relations.

Unfortunately, not everyone who has been associated with Leo Verbeek could contribute to this volume. They could, however, attend the symposium held on the occasion of his retirement on October 16, 1987 at Twente University. Some of the papers in this book have been presented at this symposium.

The editors are indebted to the Centre for Mathematics and Computer Science at Amsterdam, and in particular to its Publication Department, for the timely and fine technical realization of this volume.

August 1987

Peter R.J. Asveld, Enschede, The Netherlands

Anton Nijholt, Brussels, Belgium

CONTENTS

INTRODUCTION

Peter R.J. Asveld

Department of Computer Science, Twente University
P.O. Box 217, 7500 AE Enschede, The Netherlands

Anton Nijholt

Faculty of Sciences, Free University of Brussels
Pleinlaan 2, 1050 Brussels, Belgium

Formal Approaches

Although the papers in this book cover a broad range of topics, they possess a common theme: there is no area in Computer Science which does not benefit from a formal approach to its methods and concepts. Formal approaches in Computer Science require the separation of an abstract model from a concrete application and from particular implementation issues. This separation enables computer scientists to study and prove properties of the model and to share or borrow other work using the same or similar models.

The following aspects of modeling a subject matter can be distinguished:

- select or construct a model (or theory) and give an account of the domain to be modeled (i.e., its concepts and processes) in terms of the model; assess its adequateness
- modify — i.e., extend, restrict, or take a different level of abstraction — the model to grasp as much as necessary, in the best possible manner
- study the properties of the model and its modifications in order to obtain insight in its (descriptive) power and its limitations; this insight is useful for assessing its adequateness and it provides insight about the subject matter
- once a formal description of a process — in terms of a particular model — has been obtained, then it can be used to build programs that facilitate the building or that allow the generation of (parts of) programs for the handling of these processes by a computer

Concern for precision leads to formalization. Formal descriptions can be viewed as completely formal objects that can be studied, having

representations which can be manipulated.

This book contains a collection of papers in which these different aspects of modeling subject matters can be recognized. Most papers in this volume deal with "artificial" situations. Their subject matters are human-defined or human-constructed languages and systems. The authors introduce and study formalisms, show how a subject matter can be modeled, or discuss the building and usefulness of tools for the generation of programs that facilitate the writing or processing of user programs. Ultimately, the introduction and study of the formalisms that are discussed in these papers have been inspired by practical considerations. Practical considerations may lead to intriguing theoretical problems. In some contributions to this book the authors concentrate on these theoretical problems and they accept that no foreseeable practical application of the results of their investigations can be given.

Each paper in this book will be discussed in some detail below. It is useful to introduce this book with some short remarks about each paper. Nijholt's paper is in fact a historical survey of attempts to go from intuitive methods, through a process of abstraction, refining and borrowing from other fields, to model-based methods in the area of (artificial) language description and manipulation. This paper is followed by a series of papers devoted to topics in some subfields of (Theoretical) Computer Science, or to topics and approaches which illustrate the aspects of modeling subject matters that have been mentioned above. As mentioned earlier, most papers deal with the modeling of "artificial" situations, i.e., what is studied and modeled are human-defined or human-constructed systems. Hoenkamp's paper is an exception, since it is concerned with an attempt to develop a realistic model for aspects of a human activity, viz. human reasoning. Engelfriet, Vogler, Hogendorp, and Asveld study properties of transducers and rewriting systems which have been introduced as models for describing aspects of languages and their processing by programs. Op den Akker studies and tries to formalize issues that rise when context-free grammars are used to model programming languages. It is an example of a theoretical paper based on observations on the world of compiler construction. Fokkinga takes concepts that can be recognized in actual programming languages and maps them into equivalent ideas in a mathematical formalism — the lambda calculus — to study these concepts. On the other hand the work of Kuper deals with some fundamental aspects of the lambda calculus itself. Sommerhalder presents in his contribution a summary of unification algorithms which are relevant in modeling the implementation of (logic) programming languages. Filé's paper addresses some decidability questions about properties of a formalism introduced to model aspects of inference making. Finally, the contributions of Alblas & Faase and of van Hulzen discuss the development of tools which generate parts of programs. There are, however, important differences. The work reported

by van Hulzen fits in the tradition of offering programmers an environment which facilitates the construction of software. Alblas & Faase's work is a contribution to the development of formalisms with which one describes the process of converting a completed program into an appropriate sequence of machine instructions. The ultimate aim of these formalisms is to obtain efficient programs that convert language specifications into a compiler.

We conclude this introduction to this collection of papers with a short exposition of each of the contributions.

Summaries of the Papers

The first contribution to this book, *From Mechanical to Theoretical – Aspects of the Origins of Theoretical Computer Science* by Anton Nijholt, is a survey paper on those aspects of early Computer Science from which Formal Language Theory and Theoretical Computer Science emerged. The emphasis in this paper is on the early attempts to formalize the description of programming languages and to delegate the conversion from program to machine instructions to the computer. Observations on the relation between BNF and Chomsky's phrase structure grammars are followed by a presentation of Knuth's attempts to characterize the generative power of BNF and his generalization of Iron's method of syntax-directed translation to attribute grammars. The paper continues with some views on the early development of Theoretical Computer Science and the developments of its subfields. The emphasis is on formal language theory and the relation with its domains of application and it is argued that the approaches in this subfield of Theoretical Computer Science have set an example for the other and more recent subfields. The author's exposition is concluded with some contemplative remarks on the interaction between theory and practice in Computer Science.

The next paper, *Generating Strings with Hypergraph Grammars*, by Joost Engelfriet investigates the string-generating power of context-free hypergraph grammars. (Hyper)graph grammars constitute an obvious generalization of string grammars. Formal definitions of context-free hypergraph grammars and their string languages are presented and well-known families of string and tree grammars are viewed as hypergraph grammars. In addition some useful relationships with the set of dependency graphs of derivation trees associated with attribute grammars are established. The author presents characterizations of the string languages generated by the context-free hypergraph grammars in terms of tree-to-tree string transducers (deterministic tree-walking transducers) and 2-way deterministic finite state transducers.

In *Modular Tree Transducers*, Heiko Vogler defines operations on trees with the help of tree transducers. Often, in practical applications,

these operations are naturally defined in a structural recursive and modular way. Vogler gives some examples of these operations and argues that existing formal models such as (generalized) syntax-directed translation schemes, attribute grammars and top-down tree transducers do not reflect both the recursive and modular aspects of these tree operations. Therefore he introduces a new formal device: the *modular* tree transducer. After illustrating the adequateness of the model, its relationships with other, already existing, models are investigated and some formal properties are established.

Jan Anne Hogendorp generalizes in *Nonterminal Separating Macro Grammars* some structural definitions, originally introduced for context-free grammars, to macro grammars. Then he establishes a few characterization results for these macro grammars which are inspired by and similar to corresponding known results for context-free grammars.

Peter Asveld summarizes in *Complexity Aspects of Iterated Rewriting — A Survey* a number of results with respect to the complexity of the membership problem of some quite abstract grammar models. Originally, these abstract grammars have been defined as a generalization of some rewriting systems introduced in developmental biology where they serve as a model to study filamentous growth. Because these abstract grammar models are so general, decidability of and complexity bounds on the membership problem are of primary concern.

Rieks op den Akker's paper *On Covers and Left-Corner Parses* takes the reader to the area of parsing theory and transformations on context-free grammars. Often the objective of transforming a grammar is to obtain properties which make the grammar more amenable to certain parsing methods. However, there may be reasons to retain the syntax of the original grammar. Hence, after parsing its result should be given in terms of the original syntax description. These ideas have been modeled with the concept of *cover*. Op den Akker introduces a transformation with some desirable properties which allows the definition of a cover between the transformed and the original grammar. In addition, the transformation has the property that it yields an $LL(k)$ grammar if and only if the original grammar is $LC(k)$. The traditional cover concept can be viewed as expressing a semantically useful but nevertheless syntactic similarity relation between context-free grammars. In the final section of the paper the question is raised how to generalize this relation to one between attribute grammars. This would allow, for example, attributed variants of transformations for left factoring, for the elimination of left recursion, and for transforming one class of deterministically parsable attributed grammars to another class of deterministically parsable attributed grammars. Existing approaches are discussed and suggestions are presented.

In the contribution *Programming Language Concepts — The Lambda Calculus Approach* Maarten Fokkinga shows the benefits of expressing programming language concepts in the framework of Church's lambda calculus. Once expressed in this formalism, the properties of a programming language concept can be studied without reference to a particular programming language. Obviously, such a study may prove useful for the design of new programming languages and for a correct understanding of present programming languages. Fokkinga illustrates the significance of the lambda calculus for this study by expressing a variety of programming language concepts in this calculus or its extensions. Readers who are not familiar with the lambda calculus do not have to worry since it is introduced as a simple programming language with a clear syntax and semantics. Once this has been done, syntactic and semantic abstractions of various programming language constructs are added to this language. This approach allows the introduction and discussion of various useful principles for programming language design. Much attention is paid to a description of typing. Various theories are discussed, but the emphasis is on SVP-typing, the author's own approach to this problem.

In the next paper the lambda calculus is not used to study programming language concepts, but it becomes object of study in itself. *A Representation Principle for Sets and Functions* by Jan Kuper is a study based on the observation that in the literature on models of the lambda calculus selfapplication for functions is considered to be quite normal, whereas selfmembership for sets is considered to be undesirable. Intuitively, this distinction is strange. In order to study this distinction two views on sets and functions are introduced. One view considers them as intuitive objects, the other view considers them as mathematical objects. The consequences of these different views for models of the lambda calculus and for the relationship between such models and set theory are investigated.

Unification is a well-known problem in algebra and logic. Its practical importance increased enormously since the introduction of Prolog and logic programming in general. In order to obtain efficient implementations fast string unification algorithms are necessary. In *Unification — An Overview* Ruud Sommerhalder presents formal results on the decidability and the complexity of unification. In addition the problem to generate unification algorithms for various equational theories is discussed.

In the paper of Gilberto Filé, *The Relation Between Two Patterns with Comparable Languages*, properties of *patterns* are studied. Patterns are strings consisting of terminals and variables. They may be converted into terminal strings by substituting terminal strings to the variables. Patterns have been introduced in the context of (inductive) inference making. However, the paper is not concerned with this particular application. It studies the formalism by considering a rather

natural problem: if we consider two arbitrary patterns, is it decidable whether the language generated from one pattern includes the other language? This question is given a detailed treatment in which several restricted variations of the problem are distinguished.

Henk Alblas and Frans Faase write about *Attributed Abstract Program Trees*. Traditionally, attribute grammars can be viewed as an extension of context-free grammars. The grammar symbols are augmented with attributes and the grammar rules have associated attribute evaluation rules. For a given derivation tree the values of the attribute instances at the nodes of the tree can be computed using the attribute evaluation rules. Alblas and Faase consider attributes for abstract program trees. In these trees information which is redundant for further phases in the compilation process is deleted, allowing a more compact and simplified representation. In the synthesis phase of the compilation this representation has to be translated into the instructions of the target machine. The ultimate goal of the authors is to specify this translation by a stepwise application of tree transformations. Their paper concentrates on the initial phase of their research: the introduction of a formalism for the specification of the structure and the attributes of abstract program trees.

Hans van Hulzen's paper on *Program Generation through Symbolic Processing* is on the use of a computer algebra system as a facility to assist in the construction of programs for numerical purposes. Such an application requires a symbolic-numeric interface to transport information from the symbol processing environment to the numeric processing environment. After a discussion on the functioning of computer algebra systems and their rich variety of output features, illustrated with the REDUCE system, some approaches to the development of symbolic-numeric interfaces are presented. Van Hulzen mentions packages and tools to construct programs using output produced by computer algebra systems. Special attention is paid to Barbara Gates' work on the code GENeration and TRANslation package GENTRAN which allows REDUCE (or MACSYMA) users to generate complete and efficient programs for numerical purposes. The author reports on his present work which aims at offering REDUCE users integrated facilities for (arithmetic) code optimization and program generation.

In *Non-Monotonic Reasoning in Man and Machine* Edward Hoenkamp presents a fundamental discussion on an area in the foundations of Artificial Intelligence, namely that of non-monotonic reasoning. Humans are able to reason with incomplete and vague information. They have default assumptions about the domain and unless evidence to the contrary is presented they are unaware of these assumptions when they draw conclusions. When new information comes available and tacit assumptions are contradicted then certain beliefs have to be revised and earlier conclusions based on these beliefs have to be discarded. Hoenkamp surveys the approaches that have been taken to

model these aspects of human or "default" reasoning. Traditional systems of logic can not be used. When new truths, i.e. new axioms, are added to a system then there is no reason to retract earlier conclusions. It is this non-monotonic aspect of default reasoning which is difficult to formalize. Some approaches that have been taken are McCarthy's method of circumscription, the introduction of default rules in the traditional logics, and the introduction of so-called meta-devices. A well-known example of a meta-device is Doyle's Truth Maintenance System (TMS), a system that supports non-monotonic reasoning by detecting inconsistencies and by resolving them by altering beliefs, i.e. retracting premises. By keeping track of the justifications of conclusions TMS can maintain a consistent database of beliefs.

In the present paper the pros and cons of these logical approaches are discussed. One of the shortcomings that is mentioned is that the logical approaches allow the derivation of differents sets of beliefs but that there is no explanation why people prefer one set over the others. In order to study this phenomenon the author turns to a set of well-documented psychological experiments in which the participants are hoaxed, i.e, they are deceived about the true nature of the setting of the experiment. After the experiment they are "dehoaxed", i.e. an attempt is made to convince them that they have been hoaxed. Hence people are asked to change their beliefs. One of these experiments is taken to develop a model, based on Doyle's TMS, with which the changes from one state of belief to another can be shown. This model is then extended to allow the handling of degrees of belief and changes in the degrees of belief during the experiment.

From Mechanical to Theoretical – Aspects of the Origins of Theoretical Computer Science

Anton Nijholt

Faculty of Sciences, Free University Brussels
Pleinlaan 2, 1050 Brussels, Belgium

Some aspects of the prehistory and the background of Theoretical Computer Science are discussed. We consider the introduction of notations to describe dynamic processes, the change to the algorithmic specification of problems and the attempts to develop programs to make programming easier. The impact of Chomsky's theory of generative grammar, its reception and its relationship with the BNF-description are discussed. Observations on formal language theory and its development into Theoretical Computer Science conclude this (sketchy) survey.

1. Viewing Programs as Data

Notations for Computations

During the Second World War, after having studied the work of the logicians Frege, Hilbert, and Carnap, Konrad Zuse started to develop an extension of the propositional and predicate calculus for the description of problems for a digital computer. The dynamic process of computation which needs to be described requires that a notation should be given for an assignment operation. Since Zuse's "plancalculus" had to be mathematically exact, a notation $z+1 = z$ with the intended meaning: *"The new value of z is obtained by adding one to its old value"*, could not be used. Therefore the notation $z+1 \Rightarrow z$ was introduced. Knuth and Trabb Pardo [34] remark that such an operation had never been used before and they mention that the systematic use of assignments distinguishes computer-science thinking from mathematical thinking. Another distinction is constituted by the formal description of the control mechanism for a computation. In mathematics, even in proofs, this is done informally. In the plancalculus the idea of structured data was incorporated. Moreover, Zuse used to state the mathematical relations between the variables in his programs, in this way giving the start to a theory of program correctness. Zuse's ideas were hardly published and only in the seventies, when interest in the history of computers and computing started to develop, his writings received attention. One of the first example programs written in this language dealt with the checking of the well-formedness of Boolean expressions. Instead of Zuse's theoretical and logic-based approach the

more pragmatical approach came to dominate the development of programming languages and programming theory.

In 1945 John von Neumann wrote his "Draft Report" on the EDVAC. In this proposal each 32-bit word was either a number or an instruction word. In an instruction word the specific operation was denoted by a group of adjacent bits. In this way there were instructions for, among others, addition, multiplication, the transfer of the contents of memory locations to registers, test instructions and jump instructions. A program had to consist of a sequence of instruction words in binary form. In a separate memorandum von Neumann wrote a sorting program to test whether this set of instructions would be adequate for the control of a nontrivial computation. Von Neumann did not write the program in binary notation. Instead he used a private notation which came close to a symbolic code. That is, instead of presenting instruction words by 32 bits, they were presented with a few suggestive words, mostly in a one-to-one correspondence with a decomposition (e.g., in operands and operators) of the binary instruction words.

Most of the computers constructed after the war were patterned after EDVAC's design and were programmed in *machine code*, i.e., with binary coded instructions which operate on the contents of memory locations and on the registers or accumulators of the computer. The coding of a problem with such instructions is a difficult task with a high chance of errors. Therefore *symbolic* or *mnemonic code* was developed, and once a program was completed it was translated into machine language. This translation was done by humans. The next step was to have this translation done by the computer itself and to use (almost) conventional mathematical notation and arithmetic expressions in these symbolic codes. H. Rutishauser in Switzerland and the logician Haskell B. Curry in the United States were among the first to consider and program this problem.

Machine-oriented symbolic code uses the symbolic rather than the actual bit-addresses of the memory locations and also the operations to be performed are given symbolic (mnemonic) names. In a simple symbolic code there is a one-to-one correspondence with the actual hardware operations. Despite this improvement programming remains a tedious task. Therefore libraries of short programs for standard operations and frequently occurring (numerical) computations were developed. Hence,

> *"All that the programmer has to do is to punch the address in which the routine is stored into his main programme."*

(cf. [6], p.77). Goldstine [23], giving an account of the situation at the Institute for Advanced Study in Princeton, remarks:

> *"One of the first developments in automatic programming was introduced in the fall of 1949 on the EDSAC, where the*

> *conversion from the symbolic form to the machine one was done by the computer itself. . . . . We did not work on what are now called higher-level languages. Attention instead was focussed on developing of libraries of programs (routines, subroutines) that could be used repeatedly to save the labour of rewriting them many times."*

Subroutines had names and such a name can be considered as a *macro-instruction* which stands for a set of machine instructions. The task of an "automatic coder" consists of translating the instructions of the symbolic code into machine instructions. This coder or *assembler* takes care of the assignation of machine addresses to the operand names, machine operations to the operator names, and the proper treatment of the macro-instructions. Once this frame of mind has been accepted it becomes clear that it is possible to introduce operations, instructions, and control structures in a symbolic code. This further reduces the necessity of knowledge of the machine code and makes it possible that users which are only familiar with their own problems can use the computer. Especially the need of having a way to use a notation closer to the customary way of writing mathematical formulae started the departure from symbolic codes to programming languages. One of the first computers which had this possibility was the M.A.D.M. computer of Manchester University (Great Britain). With their notation it was possible to describe the numerical calculations (addition, subtraction, and multiplication) and the organization of the calculations into an automatic process. For the latter 13 English words were used. An example of a "numerical calculation" in this description is $+x+y+z+a+b \rightarrow c$. In [6] it is explained that subroutines could be evoked by writing the word *subroutine* followed by a number describing which subroutine is meant.

> *"By an extension of this technique it would be possible to call for the particular subroutine by name .... This has not yet been done as the gain in convenience would be too small to warrant the trouble."*

Algebraic Compilers and Formula Translators

The more complicated "automatic coders" which were now needed were called *compilers*. A compiler was not only able to convert a simple assignment statement with an expression, like, e.g., $a := b+c\times d$ into a sequence of instructions of an assembly-like code but it also converted computational control structures and other programming constructs into appropriate sequences of machine instructions. The building of these compilers provided another view on the use of a computer. Until then most of the applications had to do with the computation of numerical results. Hence, both the input and the output of a program consists of numbers. A compiler, however, is a program which takes as

input a program and generates from it another program. Grace M. Hopper was aware of this viewpoint when she wrote one of the first compilers. In 1949 Francis (Betty) Holberton had already written a program which generated another program. In an interview Hopper once remarked:

> *"Everyone's forgotten that Betty wrote the first program that wrote a program, a sort/merge generator. Because she had been ahead of me, I had a good deal more nerve in going ahead to build the A-0 compiler."*

In the same interview (cf. [41]) Hopper recalls another experience which had an eye-opening effect on the Harvard staff. An insurance company came to Harvard to run a problem on the numerically oriented MARK I computer using digits to represent alphabetical characters. Hopper: *"That opened up a new perspective none of us had ever thought of."* Hopper's A-0 compiler was built for the UNIVAC computer and it was completed in 1952. The compiler was written in the following way:

> *"There sat that big beautiful machine whose job was to copy things and do addition. So I thought, why not let the computer do it. That's why I sat down and wrote that first compiler. It was very stupid. What I did was watch myself write a program and make the computer do what I did. That's why it is a single pass compiler, ..."*

The first programming codes were very close to the machine and symbolic codes of the machines for which they were used. Also in Germany and Switzerland, most notably by K. Zuse and H. Rutishauser, the idea of automatic program construction ("Automatische Rechenplanfertigung") was conceived. In March 1951 Rutishauser lectured on this subject at a meeting of the West-German GAMM (Gesellschaft für Angewandte Mathematik und Mechanik).

The IBM 701 computer could be programmed with "Speed Code", developed by John Backus. For the BINAC and UNIVAC computers a "Short Code" was used. An interpretive routine processed each instruction and then the necessary actions were performed. Later these machines used the A-0 compiler which did not interpret but instead composed a machine code program from the scanned instructions. The A-0 compiler handled a code which hardly differed from the machine code. Codes which allowed mathematical notation for formulae were handled by so-called *algebraic compilers*. During the period 1951-1957 various of these compilers for specific machines were developed. Among the earliest were the Autocode compiler of A.E. Glennie of the Royal Armaments Research Establishment in England, written in 1952, and the Whirlwind compiler written in 1953 by J.H. Laning and N. Zierler of MIT (Massachusetts Institute of Technology).

A milestone was the building of the first FORTRAN (FORmula TRANslator) compiler. The work started in early 1954. The emphasis was not on the design of a language but on the production of a compiler for the IBM 704 computer. This machine was considered to be so powerful that only a few of them would be constructed. In [51] one of the designers, John Backus, remembers that in the beginning:

> *"We certainly had no idea that languages almost identical to the one we were working on would be used for more than one IBM computer, not to mention those of other manufacturers."*

Hopper once remarked that programmers felt insulted when their programs were treated as if they were data. The early programmers were sceptical about obtaining efficient programs by writing in a "high-level" language. Hand-coded programs would run faster and would need less memory. Their ingenuity could not be matched by a machine. Therefore, in the case of FORTRAN, the efforts were directed towards the construction of an efficient translator rather than towards the design of a well-structured language. FORTRAN remained close to the 704's machine code and a compiler was constructed that produced code which could compete in speed with that of experienced programmers. Moreover, it was soon recognized that any loss of efficiency was compensated by an increase in the programmer's productivity and a reduction of the training required for programmers.

FORTRAN allowed the writing of expressions in the statements of a program. The programmer should be informed what form of the expressions is expected by the FORTRAN compiler. Below is an example of the "syntax" specification of (mathematical) expressions as it appears in the original FORTRAN Manual. It is borrowed from a partial reprint of this manual in [51].

Formal Rules for Forming Expressions. By repeated use of the following rules, all permissible expressions may be derived.

- Any fixed point (floating point) constant, variable, or subscripted variable is an expression of the same mode. Thus 3 and I are fixed point expressions, and $ALPHA$ and $A(I,J,K)$ are floating point expressions.
- If $SOMEF$ is some function of n variables, and if $E, F, \ldots, H$ are a set of n expressions of the correct modes for $SOMEF$, then $SOMEF(E,F,\ldots,H)$ is an expression of the same mode as $SOMEF$.
- If E is an expression, and if its first character is not + or −, then $+E$ and $-E$ are expressions of the same mode as E. Thus $-A$ is an expression, but $+-A$ is not.
- If E is an expression then (E) is an expression of the same mode as E. Thus (A), $((A))$, $(((A)))$, etc. are expressions.
- If E and F are expressions of the same mode, and if the first character of F is not + or −, then $E+F$, $E-F$, $E \times F$, and E/F

are expressions of the same mode. Thus $A - + B$ and $A / + B$ are not expressions. The characters +, −, ×, and / denote addition, subtraction, multiplication and division.

It is interesting to note the amount of detail in this specification and, moreover, that in fact this "syntax" is presented as a generative system ("... *by repetitive use of* ..."). Moreover, the specification is such that the "syntactic" rules take care of the modes of the expressions. On the other hand, without associated rules of precedence for the operators this set of formal rules conceived as a generative system yields ambiguous expressions. In order to analyze these expressions and translate them into an assembly-like language J. Backus and I. Ziller developed a technique which inserted parentheses in the expressions. By Sheridan [47] the validity of their method was shown.

The programming language ALGOL, which was developed a few years later, was not designed with a specific machine in mind. ALGOL grew from attempts from the West-German GAMM and the ACM (Association for Computing Machinery) of the U.S.A. to obtain a standard programming language. Unlike FORTRAN, which was an "Automatic Coding System" for the IBM 704, ALGOL was a *language*, it had a *grammar* and an attempt was made to have a clear distinction between *syntax* and *semantics*. In 1959, at a UNESCO conference in Paris, Backus presented the work of a committee on the design and the description of this language; cf. [1]. It had its syntax described by formal rules, which became known as the *Backus Normal Form* description of the ALGOrithmic Language ALGOL 60. In a preliminary report of 1958 on this "International Algebraic Language" the notation did hardly differ from that of FORTRAN. At this Paris conference, in other sessions, work was presented on discovery procedures for phrase structure grammars and in a session on mechanical translation V.H. Yngve presented the MIT programming language COMIT which was intended to be used for mechanical translation purposes. In a footnote of Yngve's paper it is mentioned that "*Some of the features of the notation used by N. Chomsky in his theory of grammar has been incorporated.*"

2. Language as a Mathematical Object

Mathematics and Grammar

In the 19th and 20th century attempts to formalize mathematical proofs led to the introduction of formal theories and formal languages of logic. In 1879 Gottlob Frege introduced his "*Begriffschrift*, a formula language, modeled upon that of arithmetic, for pure thought" in order to unify and extend existing notations and the use of formal language for reasoning in fields such as arithmetic, geometry and chemistry. Members of the Wiener Kreis studied formal languages of logic.

Church, Post and Turing introduced and studied symbol manipulating formalisms. In 1943 E.L. Post recognized that the customary proof systems can be considered as *rewriting systems*, that is, systems that formalize the rewriting of strings of symbols in order to obtain new strings. Post introduced a formalism consisting of an axiom and a finite set of productions (rules of inference). Similar systems had already been studied by the Norwegian logician A. Thue in 1914. Another logician, Y. Bar-Hillel, became one of the main representatives of the field of machine translation. Instead of using statistical and cryptological methods he suggested the use of (structural) linguistic methods. In 1951 Bar-Hillel wrote:

> *"A considerable body of descriptive data about the languages of the world has been amassed in recent years, but so far no operational syntax of any natural language exists with a sizeable degree of completeness, and the necessity of providing such a syntax has apparently not been recognized by linguists."*

With this "operational syntax" it should be possible to analyze the sentences of a natural language. This analysis should form the basis of the translation. Instead of having a mere word-for-word translation this analysis should lead to a *phrase-for-phrase* or *sentence-for-sentence* translation. Bar-Hillel discussed these matters with R. Carnap and N. Chomsky since 1951. A first approach, using ideas of the Polish logician Ajdukiewicz, to the "mechanical" determination of the syntactic structure of sentences was given by Bar-Hillel in 1953; cf. [4]. During the same years Noam Chomsky was concerned with the question what part of linguistics could be made purely formal without reference to semantics. In 1953 Chomsky introduced an axiom system for syntactic analysis; cf. [7].

Although it was not the prime interest of the money supplying agencies part of the research on machine translation was devoted to theoretical issues related to word and sentence analyzing problems. At MIT linguistics was classified as a "communication science" and therefore it obtained more financial support from the military than at other universities. Since 1955 Chomsky was assigned to a research project, headed by V. Yngve, on machine translation in the Research Laboratory of Electronics at MIT. One of the results of this project was the earlier mentioned COMIT programming language. Most of the linguists on the project were not much interested in these applied problems and spent their time on general linguistic problems. In 1955 Chomsky finished a manuscript called *The Logical Structure of Linguistic Theory*. A "sketchy and informal" version of this manuscript was used as course notes of an undergraduate course at MIT and it was published under the name *Syntactic Structures* [9]. This book inaugurated a revolution in linguistics by considering a grammar as a generative system. That is, a finite device that can produce all and only the sentences of the

language. It should be done in such a way that this production reveals our competence of constructing sentences. Hence, contrary to the prevailing viewpoints in linguistics, which were influenced by behavioristic psychology, Chomsky introduced again, in the tradition of nineteenth-century linguists (e.g., von Humboldt), a mentalistic interpretation of language into linguistics. Other nineteenth-century linguists (e.g., de Saussure and de Courtenay) had already given thought to the use of mathematics for linguistics. However, due to Chomsky a mathematically oriented mode of thinking was introduced into linguistics. His first publication did not appear in a linguistic journal but in *Journal of Symbolic Logic*, and in the early discussions on generative grammars his work was compared with the specification methods for well-formed mathematical formulae. In [14] it is remarked that:

> *"In fact, a real understanding of how a language can (in Humboldt's words) "make infinite use of finite means" has developed only within the last thirty years, in the course of studies in the foundations of mathematics."*

After the Second World War the introduction of formal models in the different branches of science was widespread. Technological and mathematical approaches to the study of human behavior started to flourish and it was thought that natural sciences could be extended to describe and explain phenomena of human mind and cognition. The pursuit of a precise formulation of the notion of grammar can be illustrated by the emphasis which laid by many authors in the early nineteen fifties on their mathematical approach. This is reflected in the titles of their publications by using the words "logical syntax" (also used by Carnap), "model", "axiomatic syntax", "syntactic calculus", "quasi-arithmetical notation", etc. During these years formal models were sought for the method of constituent analysis. Initiating work on this topic had been performed by Wells [50] and Harris [27]. They have been considering "linear" schemes (in contrast to hierarchic) from which sentences can be obtained by substitution of elements which have the correct distribution. Chomsky [9], however, introduced the following model:

> *"Customarily, linguistic description on the syntactic level is formulated in terms of constituent analysis (parsing). We now ask what form of grammar is presupposed by description of this sort. As a simple example of the new form for grammars associated with constituent analysis, consider the following:*

(13) (i) Sentence → NP + VP
(ii) NP → T + N
(iii) VP → Verb + NP
(iv) T → the
(v) N → man, ball, . . .
(vi) Verb → hit, took, . . .

Suppose that we interpret each rule $X \rightarrow Y$ *as the instruction "rewrite* X *as* Y*". We shall call (14) a derivation of the sentence "the man hit the ball", where the numbers at the right of each line of the derivation refer to the rule of the "grammar" (13) used in constructing that line from the preceding line.*

(14) Sentence

NP + VP	(i)
T + N + VP	(ii)
T + N + Verb + NP	(iii)
the + N + Verb + NP	(iv)
the + man + Verb + NP	(v)
the + man + hit + NP	(vi)
the + man + hit + T + N	(ii)
the + man + hit + the + N	(iv)
the + man + hit + the + ball	(v)

... We can represent the derivation (14) in an obvious way by means of the following diagram:" [cf. Figure 1.]

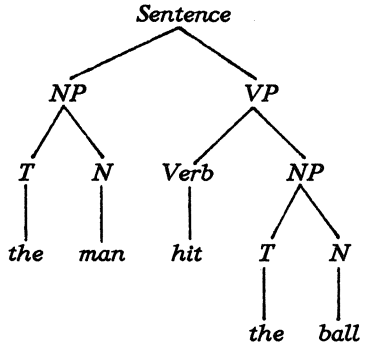

Figure 1. Representation of derivation (14).

It is worth noting that unlike immediate constituent analysis a generative grammar as used in Chomsky's example predicts the grammatical sentences. The grammar is a finite and explicit characterization of the grammatical sentences. In the tree the dominance and precedence of the constituents which constitute the sentence is shown. This dominance and precedence can give a formal account of ambiguity of sentences. Another aspect to be mentioned is recursion. The rule $NC \rightarrow NC\ Conj\ NC$ is an example of a recursive rule. It can be applied recursively without a limit to the number of applications. Recursion is of interest for the description of embedded sentences. In the example $X \rightarrow Y$ has been interpreted as "rewrite X as Y", where X should be treated as a single name or symbol. In *Syntactic Structures* it is also allowed that the rules have the form $xAy \rightarrow xwy$,

where A is a single symbol or name and x, w, and y are strings of symbols or single names. Hence, in the context of x and y it is allowed to rewrite A to w.

The Chomsky Hierarchy

In [11] a mathematical investigation of classes of formal grammars and languages is presented. Here, a language is a set of strings of finite length over a (terminal) alphabet. A grammar consists of a vocabulary V which is subdivided into two disjoint sets, the terminal alphabet Σ and the nonterminal alphabet N, and a finite set of rewrite rules. Alphabet N contains a distinguished symbol, the so-called "Sentence-symbol", mostly denoted by S or by "Sentence". Let a be a symbol in Σ; x, and y, and w be words over V and let A and B be symbols in N. A grammar is said to be *unrestricted* (type 0) if its rules are of the form $x \rightarrow y$, *context-sensitive* (type 1) if its rules are of the form $xAy \rightarrow xwy$ (w is non-empty), *context-free* (type 2) if its rules are of the form $A \rightarrow w$, and *finite state* (type 3) if its rules are of the form $A \rightarrow a$ or $A \rightarrow aB$. Hence, by imposing restrictions on the forms of the rewrite rules different classes of grammars are induced. Starting from the sentence-symbol we can repeatedly apply the rewrite rules. The language which is generated by the grammar consists of the strings of terminal symbols which can be obtained with this process. Languages generated by finite state, context-free, context-sensitive, and type 0 grammars are called finite state, context-free, context-sensitive, and type 0 languages, respectively. It can be shown that the induced hierarchy of families of languages is proper.

In Chomsky's paper the names "context-sensitive" and "context-free" were not yet used. The first occurrence of the name "context-free" in the literature appears in [12]. Finite state grammars had been presented before as finite state diagrams (finite state Markov processes) in communication theory. The adjective "regular" had been used by S.C. Kleene to denote certain sets of strings (regular events). These sets turned out to be equivalent to the finite state languages defined by the Markov processes. Since then finite state languages have also been called *regular* languages. Later it became clear that they could be defined with the above given type 3 restriction. Due to Post, unrestricted grammars (semi-Thue systems) were already a well-known formalism in Logic. The language which is generated with an unrestricted grammar is also called a *recursively enumerable* language. The family of languages which have the property that for each string it can be decided whether or not it is in the language are called the *recursive* languages. This family is properly situated between the families of context-sensitive and recursively enumerable languages. A grammatical characterization of this family is not available.

3. Computer Science Has Its Eye on Grammar

The ALGOL 60 Report

In May 1960 the ALGOL 60 report was published (cf. [37,38]), followed by a flood of papers, letters to the editor, etc., on the ALGOL definition and on compiling ALGOL. However, none of these authors refers to Chomsky's phrase structure grammars for describing languages. ACM decided to use ALGOL as the language for communicating algorithms and authors were invited to present algorithms in ALGOL 60.† An *ALGOL Bulletin* was set up and made part of ACM's newsletter *SIGPLAN Notices*. In 1970, due to "financial reasons" it was again separated from this newsletter.

The ALGOL 60 Report presents the "defining" language of ALGOL. It is expected to be the basic reference and guide for compiler builders. In the (Revised) ALGOL 60 Report the formalism for syntactic description is explained as follows. The syntax is described with the help of metalinguistic formulae. Their interpretation is explained by the following example in which we use two formulae.

<unsigned integer> ::= <digit> | <unsigned integer> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Sequences of characters enclosed in the brackets < and > represent metalinguistic variables whose values are sequences of symbols. Hence, in the first formula we have two metalinguistic variables, <unsigned integer> and <digit>. In the second formula there is only one, viz. <digit>. The marks ::= and | (the latter with the meaning of "or") are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself. Hence, the marks 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 denote themselves. Juxtaposition of these latter marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formulae above give a (recursive) rule for the formation of values of the variable <unsigned integer> and a rule for values of the variable <digit>. Two kinds of expressions in ALGOL 60 are

<expression> ::= <arithmetic expression> | <Boolean expression>

Other metalinguistic variables obtain similar formulae. For example,

<arithmetic expression> ::= <simple arithmetic expression> |
<if clause> <simple arithmetic expression> ELSE <arithmetic expression>

and in the same style a set of values of <simple arithmetic expression> are

† *"All contributions will be refereed both by human beings and by an ALGOL compiler."* (From the ACM Algorithms Policy).

<adding operator> ::= + | −

<multiplying operator> ::= × | /

<primary> ::= <unsigned number>|<variable>|(<arithmetic expression>)

<factor> ::= <primary> | <factor>↑<primary>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<simple arithmetic expression> ::= <term> | <adding operator> <term> |
<simple arithmetic expression> <adding operator> <term>

In the previous paragraph only the context-free part of the definition of ALGOL 60 has been considered. In the ALGOL 60 Report a similar definition is followed by sections with examples and considerations on semantics, and with constraints on, e.g., the types (modes) of the constituents of the expressions. For example,

a. An arithmetic expression is a rule for computing a numerical value. This value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression.
   - a.1 The actual numerical value is obvious in the case of numbers.
   - a.2 For variables it is the current value (assigned last in the dynamic sense).
   - a.3 For arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other two kinds.

b. The constituents of arithmetic expressions must be of types "real" or "integer". The operators +, −, and × have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be "integer" if both of the operands are of "integer" type, otherwise "real".

c. The sequence of operations within one expression is generally from left to right, with the following additional rules:
   - c.1 According to the syntax given the following rules of precedence hold:
     first: ↑, second: × /, third: + −
   - c.2 The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations.

We have omitted the "semantics" of / and ↑, and a discussion on the interpretation of "real" numbers and variables. This added English text on semantics and constraints will be helpful for a compiler builder but it is not complete and precise. Interesting are the rules of precedence between operators. They are "*according to the syntax given*". Hence, the syntax does not only define the "well-formed" sequences of

symbols which form an arithmetic expression, but the way the syntax is organized determines the semantic interpretation of the expressions. If a is the name of a variable with current value 2 then the answer to the question whether $a+4\times3$ will be interpreted as an expression with numerical value 18 or 14 will depend on the underlying syntax of this expression. This dependence can be explained with the help of syntax trees.

Formal Description of Formulae and Languages

In [5] a short survey is given of the early work on the formal description ("rules of spelling") of arithmetic and Boolean formulae with and without parentheses. Most of this work is done by logicians between 1930 and 1950. Later, algorithms were designed which checked the well-formedness of these formulae and which were able to evaluate them. One of the earliest algorithms for evaluating arithmetic formulae is due to H. Rutishauser. The algorithm was made suitable for a sequentially working process by C. Böhm in 1952. In one of the first FORTRAN compilers similar ideas were used. Before evaluation a preprocessor inserted parentheses in the formulae in order to make them fully parenthesized.

The recognition that arithmetic and logical expressions could be parsed and converted into assembly-like instructions led to the concept of a high-level programming language. In [51] the designer of the JOVIAL language recalls that an article on expression analysis (cf. [52]) was quite a revelation to them. It was one of the significant things which made them decide to develop a high-level language for programming the future U.S. Air Force's air defense systems:

> "..., *but the idea of being able to understand and parse complex expressions in itself was of sufficient interest to motivate our efforts.*"

Roughly summarizing, the following "mechanisms" can be distinguished:

- rules of well-formedness
- algorithms which check well-formedness
- formalisms which generate the well-formed formulae
- algorithms which reveal the way a well-formed formula has been generated by the formalism
- algorithms which evaluate expressions to a numerical or Boolean value.

The BNF description of ALGOL 60 is merely a set of rules of well-formedness. The ALGOL reports do not provide the notation and the terminology to *derive*, *produce*, or *generate* the well-formed programs. However, there is the intention to link well-formed programs to conceptual structures by means of meta-linguistic formulae.

Moreover, there is the underlying assumption which says that the associated semantics is "syntax-directed", that is, derivable from the conceptual structure of the program. If the BNF description is interpreted as a context-free grammar, then it allows the generation of the well-formed sequences of symbols which constitute ALGOL 60 programs. Checking well-formedness of computer programs has become known as checking whether a program is syntactically correct. Analyzing the program, in the sense of revealing the structural description, has become known as *parsing*. The algorithms for the evaluation of expressions have developed into compiling algorithms. These algorithms presuppose a parsing algorithm which reveals the structure. From this structure code can be produced which will be used to do the actual evaluation of the expression.

BNF versus Context-Free

"Is your Chomsky really necessary?" (F.G. Duncan, in [48], p.298).

Historical notes on BNF and some fighting about its introduction can be found in [51]. The fighting is done in a paper by P. Naur, comments on Naur's view by F.L. Bauer and K. Samelson, and in the transcript of a question and answer session. Both Bauer and Backus mention that the use of the notation came from similar notation in mathematical logic. Backus remarks:

> *"As to where the idea came from — it came from a class that I took from Martin Davis. ... , talking about the work of Emil Post and the idea of a production. It was only in trying to describe ALGOL 58 that I realized that there was trouble about syntax description. It was obvious that Post's productions were just the thing, and I hastily adapted them to that use."*

(p.162 in [51]). Also in [51] J.E. Sammet discusses the syntax description of COBOL in relation to BNF.

> *"Unfortunately, because on one hand we called this a notation, and on the other because it was a metalanguage quite different from that proposed by Backus in his paper, it became very fashionable and quite common to say (at least orally if not in writing) in 1959-1961 that COBOL had no formal definition. I think anyone who looks will indeed recognize that the syntax of COBOL was (and still is) defined in just a formal way as ALGOL 60; ... I would venture to guess that more languages are defined today using some variation of the COBOL metalanguage than are actually defined today using (even a variation of) BNF."*

The BNF description of ALGOL was not appreciated by an IBM representative who suggested, after working through the description of

ALGOL 60, to add a new entry to Webster's Dictionary, **Algolagnia**: "*The finding of pleasure in inflicting or suffering pain.*"† In 1964 Knuth suggested to use the name **Backus Naur Form** in order to honor P. Naur's work as editor of the ALGOL Report.

In the August issue of *Comm. ACM* Gorn [24] discussed some basic terminology of *mechanical languages* and their processors. However, "BNF", "context-free language" or "formal language" are words which are not used. In December, however, Gorn [25] remarks that

> "*The specification restrictions implicit in Backus normal form place the languages so specified in the class of "Phrase Structure Languages".*"

In the reprinted version of this paper (in: *Readings in Automatic Language Processing*. D.G. Hays (Ed.), American Elsevier, 1966) this citation is immediately followed by: "*more specifically, they are the "context-free languages"*", being the only change in the text of the paper. In *Comm. ACM* **5** (1962) at p.62 we find a Research Summary reported by S. Gorn, October 1961, which is titled "Theory of Mechanical Languages" and which mentions research into the relationship between Chomsky's phrase structure languages and the languages specified with Backus normal form. And on p.185 of the same volume we find an interesting discussion in the "Letters to the Editor" section between Knuth and Gorn. Knuth starts his letter by remarking that he is interested in Gorn's papers

> "*... primarily because I have been doing a bit of research in my spare time considering various implications of "Backus normal form".*"

Then Knuth continues with a discussion on the generative power of BNF:

> "*... the class of strings ab, aabb, aaabbb, etc., can be represented in Backus notation, ... the class of strings abc, aabbcc, aaabbbccc, etc., cannot be represented in Backus notation.*"

And, in the tradition of Fermat, Knuth concludes with:

> "*... (I have constructed formal proofs of these facts.) The whole subject is quite fascinating.*"

At that time it had already been shown that in Chomsky's formalism the language *aba*, *aabbaa*, *aaabbbaaa*, etc., is not context-free (cf. [45]) and in 1960 Bar-Hillel and others (cf. [3]) had introduced tools for proving such negative results. Ginsburg and Rice [21] (the paper was received in February 1961) further discuss the relationship between BNF and phrase structure. Here we see the formal statement

† Similar remarks have been registered at the reception of the Operation Manual of the ENIAC and the ALGOL 68 Report.

that

> *"The defining scheme for ALGOL turns out to be equivalent to one of the several schemes described by Chomsky in his attempt to analyze the syntax of natural languages."*

Twenty years later Ginsburg remarks (cf. [22])

> *"That observation opened the flood gates for formal language theory."*

In two papers Floyd [19,20] showed, using Bar-Hillel's technique, that programming languages are not necessarily context-free and that it is undecidable whether a context-free grammar is unambiguous. The latter problem became interesting when it turned out that the initial BNF description of ALGOL was ambiguous. However, it became clear that considerable parts of programming languages could be defined with context-free grammars. Research concentrated on this class of grammars and languages, and more general formalisms sometimes were obtained as generalizations based on the context-free grammars.

The various names which were used for the type 2 grammars and languages (e.g., (context-free) constituent structure grammar, (simple) phrase structure grammar, push-down store grammar, ALGOL-like grammar, BNF description, context-free grammar) sometimes gave rise to confusion during these early years. Examples of this confusion can be found in discussions included in the proceedings of a Working Conference on *Mechanical Language Structures* (cf. *Comm. ACM*, February 1964). See also the proceedings of the IFIP Working Conference on *Formal Languages: Description Languages for Computer Programming* (cf. [48]) held in New York in September 1964.

4. The Impact of Compiler Construction

Beyond a Context-Free Description

In the ALGOL 60 report the syntax of the language was expressed formally by means of BNF. Natural (English) language was used to express the semantics. Because of the use of BNF rules other, similarly defined, languages have been called ALGOL-like languages. However, ALGOL is not ALGOL-like. Its BNF rules define a superset of the ALGOL language and only by satisfying some restrictions, expressed verbally in the defining report, the ALGOL language is obtained from the production rules. The same observation can be made for other programming languages, i.e, additional conditions consisting of context-sensitive dependencies have to be satisfied. Checking of these dependencies can be done during parsing or in a subsequent pass of the compiler which is concerned with the semantic interpretation. Hence, the (context-free) parser accepts a superset of the programming language and auxiliary information is used to reject the incorrect programs. Restrictions which filter out the syntactically correct programs from a

language which is otherwise described with a context-free grammar have become known as *contextual constraints*. Sometimes these constraints are referred to as *static semantics*. See e.g. [35], where Koster explains that static semantics is "... *syntax expressed verbally because of impossibility to treat it in a formal way.*" That is, with a syntax formalism more powerful than a context-free grammar or BNF this "semantics" could have been part of the formal syntax. Although the common user of a programming language can be presented a more understandable description of the language it is advantageous to have a formal description of the static semantics. It provides the compiler writer a guide for the implementation and in certain cases the user may find it necessary to have an understanding of the details of the language. Moreover, programs which automatically generate (parts of) compilers need a formal description of their input.

ALGOL 60 was introduced and subsequently studied as a language with a distinction between syntax and semantics. In the theoretically oriented research first interest focused on syntactical questions and on more powerful formalisms which could define a more complete syntax of a language. Moreover, formalisms were introduced which lend themselves to the description of the translation from high-level programming language to machine or assembly language. In a later stage researchers started to think about defining semantics independently from the compilation process.

In general the attempts to automate the production of those parts of a compiler which explicitly deal with the translation are based on certain enrichments of context-free grammars. There are obvious reasons why in compiling theory the concept of context-free grammar never has been abandoned. Context-free grammars give comprehensible descriptions of languages and they are easy to handle. A context-free grammar is a rigorous mathematical object and therefore it has well-defined properties. It is decidable whether an arbitrary string is part of the language of a context-free grammar and there exist methods for automatically constructing parsers from a context-free grammar. On the other hand, context-free grammars do have some deficiencies. The syntax specification can sometimes lead to rather long lists of productions, it is not possible to accommodate the above-mentioned contextual constraints and, last but not least, in compiler construction we are interested in the translation from the programming language to an intermediate language or to, ultimately, some form of assembly code. Therefore a formalism which accommodates these tasks is desirable. Various generalizations of context-free grammars and BNF have been introduced addressing one or more of these deficiencies. Some of these generalizations are introduced from the point of view of being able to generate or accept a more powerful class of languages, without considering the possibility of efficient parsing and translation methods. Explicit use of contextual constraints can be found in formalisms

which maintain the BNF-syntax specification and augment it with predicates. Ledgard [36] gives an example of the specification of PL/1 with a formal notation called Production Systems. In his notation context-sensitive requirements such as the compatibility between the declaration of an identifier and its uses and the correspondence between actual and formal parameters are described by including "predicates" in the productions which should be satisfied in order to obtain legal strings. Similar descriptions have been given for the semantic rules of BASIC and ALGOL 60.

Consider now the second reason why context-free grammars are not satisfactory for the description of programming languages. Although not presented in a completely formal way, E.T. Irons [31] explicitly defined the problem of translating from source text through the intermediate level of a syntax tree to the semantics (meaning). A possible solution was given, that is,

> "..., *a translation using the description can be effected by fitting already discovered syntactic units (starting with the syntactic units which are the basic symbols of the language) into the syntactic structure to produce a new set of larger syntactic units, and assign meanings to these new units according to the meanings of the original units.*"

The aim of this approach was to produce an ALGOL 60 compiler. It is generally assumed that Irons' paper started the research on syntax-directed compiling. Irons' ideas amount to defining the semantics by associating meanings to each nonterminal symbol of the grammar and associating *semantic rules* to each production. These rules define the meaning of the nonterminal symbol in the left-hand side as a function of the meanings of the symbols in the right-hand side. This can be considered as an application of Frege's principle of assigning meaning to composed constructs.

A first approach to a formalization of Irons' ideas has led to the introduction of *syntax-directed translation schemes*. These schemes define string-to-string translations by means of "lock-stepped" derivations in two related context-free grammars. The translation string can be considered as the meaning of the original sentence. Consider a context-free grammar rule, say $A \rightarrow aBcD$ where A, B and D are nonterminal symbols and a and c are terminal symbols. A *simple* syntax directed translation scheme (simple SDTS) has rules of the form, say $A \rightarrow aBcD, pBqDr$ where p, q and r are called translation symbols. This rule can be viewed as consisting of two rules, a context-free source rule $A \rightarrow aBcD$ and an associated context-free target rule $A \rightarrow pBqDr$. The idea is that when a sentence w is generated with the source rules its translation is obtained by simultaneously rewriting the associated target rules. Hence, if we start with (S,S), where S is the start symbol, then a translation (w,w') is obtained,

where w' is a sequence of translation symbols. From a more practical point of view string w' can be considered as a sequence of semantic routine calls for evaluating the semantic rules of the productions and make certain checks when necessary. The result of a routine call can be a piece of code or text in the target language.

So far the simple SDTS is a definition of a string-valued translation with a possible practical interpretation. From the parsing point of view the recognition of (parts of) the rule $A \rightarrow aBcD$ during context-free parsing invokes the routines represented by p, q and r. The above-given quotation suggests a rule of the form $A \rightarrow aBcD, BDr$ where the only routine r is called when the complete production $A \rightarrow aBcD$ has been recognized during parsing. The target rules of the SDTS determine the moment when the routines are invoked. The (parsing) properties of the context-free source grammar in combination with the form of the target rules determine whether an efficient translation process is possible. Before going to the next generalization it is useful to introduce yet another point of view on the translation process. The rule $A \rightarrow aBcD, pBqDr$ can be considered as the definition of the translation associated with a particular node in the parse tree with label A. In this view w' is the translation at the root S of the tree and at node A the translation is the string consisting of the symbol p, followed by the string which is the translation at node B (a direct descendant of A), followed by the symbol q, followed by the translation at node D (a direct descendant of A), followed by the symbol r. Hence, with the following self-explaining notation, the rule can be written as

$$A \rightarrow aBcD, \quad t(A) = p\, t(B)\, q\, t(D)\, r.$$

Now it is possible to introduce multiple translations at a node. For example,

$$A \rightarrow aBcD, \quad t_1(A) = p\, t_2(D), \quad t_2(A) = t_1(D)\, r\, t_2(B)$$

and at the root S of the tree we can obtain multiple translations of sentence w.

Instead of string-valued translations more general translations can be introduced. Moreover, it might be necessary to check constraints which have to be fulfilled at certain nodes of the parse tree. In this way each grammar rule, say $A \rightarrow aBcD$, is accompanied by a set of translation rules which determine the "translations" of A as a function of the "translations" of the symbols which appear in the right-hand side of the grammar rule. Instead of "translations" it is more appropriate to speak of *attributes* of A and their values. Instead of "translation rules" it is now more appropriate to speak of semantic or attribute (evaluation) rules. Values are assigned to the attributes of A by evaluating the rules which are associated with the grammar rule $A \rightarrow aBcD$. In [33] the next generalization is presented. In Knuth's attribute grammars each vocabulary symbol of the context-free

grammar has an associated finite set of attributes which describe the properties of that symbol. Each attribute has a not necessarily finite, fixed domain from which its values are taken. Attribute evaluation rules associated with the production rules of the grammar determine the values of the attributes. In the schemes above the meaning or translation at a node in the parse tree was given as a function of the meaning of its descendants. One may expect that in certain cases the context plays a role. In that case part of the information which determines the meaning at a node in a parse tree may come from outside its subtree. As a consequence, the "meaning" which is obtained from the subtree dominated by a node may depend on this context information.

In order to describe the latter situation Knuth distinguished between two types of attributes. If the attribute values are obtained from the values of the ancestor or from the siblings of the node in the parse tree then the attributes are called *inherited*. If they are obtained from the descendant nodes the attributes are called *synthesized*. Apart from the formal setting provided by Knuth, the main novelty of attribute grammars is the added feature to define the semantics "top-down" by the inherited attributes. Since the evaluation is not necessarily in a single direction the semantic rules of an attribute grammar can give rise to a circular definition. That is, it is not necessarily the case that for each parse tree of the grammar there exists an evaluation order which guarantees that the arguments of a semantic rule have already been evaluated when this rule has to be executed. When such an evaluation order exists the grammar is said to be well-defined or non-circular. There exist algorithms for deciding well-definedness. Once the (context-free) syntax tree has been constructed it is possible to evaluate the attributes associated with its nodes. Conditions for well-definedness have been developed which make it possible to evaluate the attributes in a fixed number of passes over the syntax tree. Interesting cases are those which permit attribute evaluation in a single left-to-right pass and those where the syntax analysis and the attribute evaluation can be done together in a single pass from left to right. Since in general the programming language will be a context-sensitive subset of the language generated by the underlying context-free grammar, semantic conditions on the productions must be satisfied by the values of the attributes in order to obtain a legal sentence or a program.

Attribute grammars are more directed towards the handling of semantics in the practical situation of compiler writing than towards the formal definition of semantics. Other attempts have been made to give complete and formal definitions of programming languages. The first aim to do so — to have a formal definition which can help in the construction of an implementation or which can be used as input to a compiler generating system — has already been discussed. The second aim is to provide a model in which the meaning of a program is defined. The model can be used to prove that programs satisfy claimed

properties. Most of the attempts started with the description of ALGOL 60. These attempts were invited by the success of its formal syntax definition. Markov algorithms were used by de Bakker [2]. Others used Church's lambda calculus or recursive functions. For one of ALGOL's successors, the language EULER, the semantics was defined by showing how the syntactic constructs should be translated to an informally described assembly code. In a formal setting this approach consists of the definition of an abstract machine and a mapping of the syntactic constructs of the language to the operations of this machine. The first language to be defined this way was LISP in 1960, by John McCarthy. In denotational semantics each syntactic construct is associated with a mathematical function which expresses its meaning. Hence, we have a mapping from a linguistic domain to a domain with well-understood mathematical concepts which model the semantics. The resulting meaning of a program is based on its inductive structure.

Automatic Production of Compilers

"We call the preparation of a grammar BNF programming, and the process of modifying it until acceptable, BNF debugging." (W.M. McKeeman, et al. *A Compiler Generator*. Prentice-Hall, 1970; p.183).

Every program has its own input language. Sometimes this language is simple, e.g., when the only input which is allowed is a list of numbers in a predefined format. Sometimes the input language is rich, e.g., when the input consists of a program which has to be checked on syntactic correctness or when the input consists of a compiler specification. The approach to compiler construction where a compiler specification is converted by a program into a compiler has been pursued since the early sixties when a prototype of such a system was developed for the ATLAS computer of the University of Manchester (Great Britain). The following enthusiastic review appeared in *Datamation* 7, May 1961, p.27:

> *"With ATLAS comes a new approach to symbolic programming. Dr. R.A. Brooker, of Manchester University, has devised a scheme in which any programming language can itself be defined. In effect, this scheme enables one to "teach" ATLAS any language one chooses, after which the computer can accept programs written in that language, it is a compiler of compilers."*

It is necessary to have a meta-language to describe a compiler for a specific language. The BNF or context-free grammar notation of a specific syntax can be considered as a meta-language. In its turn this meta-language can be described with a (simple) grammar. The input of a parser generator can consist of a specific set of BNF rules, that is, a sentence in this meta-language. A compiler writing system will require more than a set of BNF rules. Its input language can consist of sets of

BNF rules supplemented with semantic information. A formal attribute grammar notation can be considered as a meta-language in which the input of compiler writing systems is expressed. This notation can gradually evolve into a special purpose programming language suitable for writing compilers. Such a language is much less error-prone than different formalisms and notations for scanning, syntax, error and semantic analysis, and code generation. It should satisfy the condition that only straightforward transcriptions have to be done from the language designer's definition grammar to the description which will be input to the system.

Floyd [18] was among the first to recognize the necessity of creating a special description language for compilers. Obviously, the language was first used in the development of an ALGOL 60 compiler. A modified version of this language was used by Evans [15] and it became known as the "Floyd-Evans Production Language". Feldman [17] introduced the description of semantics in this language. His Formal Semantic Language (FSL) was the basis for a compiler-compiler:

> *"In the present form FSL itself can be considered a problem oriented computer language. The problem involved is the representation of meaning in computer languages."*

Often these compiler description languages are simple, e.g., without assignment and hardly any control structures. On the other hand some of them have grown to general system implementation languages with classical control structures and abstraction and extension mechanisms. Sometimes it is possible to recognize the original grammar formalism and intended parsing method in the language definition.

5. Towards Theoretical Computer Science

Formal Language Theory

"We live or die on the context-free languages." (S. Ginsburg, in [22], p.7).

The introduction of the Chomsky hierarchy led to a flood of papers on mathematical and, to a lesser degree, linguistic properties of its grammar and language classes. Especially machine characterizations of the various language classes were sought. Turing machines were known to be equivalent to type 0 grammars. By Chomsky [13] and by Evey [16] a pushdown automaton as a recognizing device for context-free languages was introduced. In the next subsection we discuss the introduction of pushdown stacks in computer science. There exist methods to convert a context-free grammar to an "equivalent" pushdown automaton and vice versa. In the early 1960s these conversions were not immediately clear. It was necessary to get used to the idea

that instructions of the automaton could be carried out without reading the input and, more importantly, that nondeterminism was an essential concept. Nondeterminism had been used before in the characterization of regular languages by finite automata; cf. [10]. By Rabin and Scott [43] it was shown that for these simple devices nondeterminism was not really necessary. Each nondeterministic device could be converted into an equivalent deterministic device. At that time researchers were not yet used to nondeterminism; cf. [26].

Machine characterizations of languages could be viewed as models of *parsers* for these languages. At first, parsing methods were not based on theory. The following quotation (cf. [30]) on the construction of the FLOW-MATIC compiler might be instructive.

> *"In order to quickly pick up the word — we didn't know anything about parsing algorithms at that point in time — and what happened was you picked up the verb, and then jumped to a subroutine which parsed that type of sentence. In order to do that quickly, and also to make it easy to manufacture that jump, the first and third letters of the verbs in FLOW-MATIC were unique."*

However, soon it became clear that in writing programs languages were involved and language became an object of study in computer science. When the relation between the syntax specification of ALGOL 60 and the context-free grammars was established and, moreover, E.T. Irons had shown how to use the syntax specification in the construction of an ALGOL compiler, computer scientists started to show interest in parsing methods.

Context-free grammars could be shown to be equivalent to (nondeterministic) pushdown automata. Suppose that we write a parsing program which uses the (nondeterministic) pushdown automaton in such a way that it tries all possible choices until a successful sequence of moves for an input string has been obtained (or it can be concluded that the input string is not in the language accepted by the automaton). It is not difficult to see that due to the nondeterminism the number of steps of the parser grows exponentially with the length of the input string. Methods which require exponential time are viewed as not acceptable. This might become clear from the table in Figure 2. Assume that each primitive step of a parser takes 1 microsecond. In the table examples are given of linear, polynomial and exponential functions which for each input length express the execution time.

A pushdown automaton which does not use nondeterminism is called a *deterministic* pushdown automaton. Languages which can be accepted with a deterministic pushdown automaton are called *deterministic* (context-free) *languages*. These languages constitute a proper subset of the context-free languages. A parsing method which would be based on a deterministic pushdown automaton requires linear time.

time function	length of the input n				
	10	20	30	40	50
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years

Figure 2. Polynomial and exponential time functions.

From the table it will be clear that such parsing methods are desirable. However, they can not work for all context-free languages.

Some (selective) backtrack parsing algorithms have been used in early compiler writing systems and for parsing natural language. Due to the exponential "blow-up" no widespread applications of these algorithms could be expected. Moreover, in the computer science area researchers had already started to devise practical algorithms for their programming languages. These algorithms were suitable for very restricted subclasses of the context-free grammars and they worked in linear time; cf. [39]. Even when such an algorithm can not handle all the syntactic constraints in the specification of a particular programming language, methods can be given to reject incorrect structures in an additional phase of the compiling process. In the early 1960s Robert W. Floyd devised some practical schemes and soon theoretical questions about the properties of the classes of grammars and languages for which the methods could be used were asked and studied. Rather than being a problem for practitioners in the computer science area the search for better algorithms for general context-free grammars became a concern for linguists working on natural language processing projects and for formal language theorists. Greibach [26] (p.71) comments on this situation:

> *"We were very much aware of the problem of exponential blow-up in the number of iterations (or paths), though we felt that this did not happen in "real" natural languages; I do not think we suspected that a polynomial parsing algorithm was possible."*

A polynomial time algorithm was already available but not recognized as such. This was Cocke's algorithm, first mentioned in [29] and used for parsing a context-free grammar for English developed at RAND Corporation.

Pushdown Stack Applications

In many early compilation methods the "last-in first-out" (LIFO) principle which governs the pushdown stack was implicitly used. The principle can be used to convert arithmetical expressions from a traditional infix notation to a more convenient Polish postfix notation (after the Polish logician J. Lukasiewicz) and to evaluate expressions presented in this form. In postfix form the operators occur in the order in which they are to be used. Therefore Polish postfix notation can be considered as an intermediate language between the source language and the assembly code. It is possible to convert the usual programming language constructs into a Polish postfix form. Statements in this notation can be easily translated into an assembly-like code. In order to understand the conversion to Polish postfix form an analogy with a simple railway network (see Figure 3) was introduced.

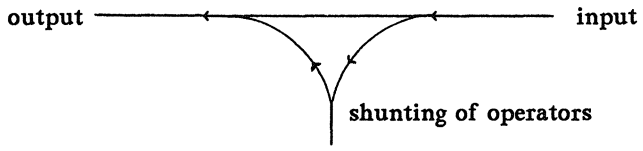

Figure 3. The railway analogy.

With this analogy it is easy to see how an infix expression, e.g., $a \times (b + c)$, is converted into the postfix expression $abc+\times$. The identifiers are directly moved from input to output and the operators are moved from input to output via the "siding" (the pushdown stack).

A slightly more complex example might be more instructive. Consider the expression $a - b + c \times d$. With the normal precedence rules we expect this to be evaluated as $(a - b) + (c \times d)$. The normal precedence rules are

↑ (raising to the power) highest precedence
× and / are of next highest precedence
+ and − are of lowest precedence

The relative precedences of the operators can be collected in a table. The Polish postfix form of $a - b + c \times d$ becomes $ab - cd \times +$. In order to realize this conversion the pushdown stack is used as follows. The string is read from left to right. Each operand is copied directly to the output. Each operator will be moved to the output via the stack. However, before stacking the precedence of the current operator is compared with that of the operator on top of the stack. If it has greater precedence, then the current operator is pushed on the stack. If it has lower or equal precedence then operators are popped from the stack and copied to the output until the stack is empty or a top operator has lower precedence. Then the current operator is pushed on stack. The process has become *table-driven*. We have an algorithm, based on an

input and output tape and a pushdown stack, and a table which controls the actions. If, e.g., we want to change the precedences of the operators only the table need to be revised.

Stack applications first appeared in the fifties. Scientists to which the idea has been attributed include, among others, W.L. van der Poel (1952), who proposed it to store subroutine return calls, A.W. Burks, D.W. Warren and J. Wright (1954), who used it to check and evaluate parenthesis-free notations of logical expressions, and A. Newell and J.C. Shaw (1957), who used it in the description of their Logic Theorist. The railway analogy appeared after a rather explicit introduction of the pushdown stack (or cellar, after the German word Keller) in parsing theory by Samelson and Bauer [44]. The principle was used in attempts to develop ALGOL 60 compilers and it was implemented in computer architectures, e.g., the Burroughs 5000 system issued in 1963, to allow the efficient compilation of ALGOL 60. At that time context-free languages where not widely known among computer scientists and pushdown automata were not yet introduced. The analogy has been attributed to E.W. Dijkstra who used it in the report *Making a translator for ALGOL 60*, first published in May 1961. Dijkstra's object machine performed its arithmetic with the help of a stack. Notice that in the *evaluation* of an expression in Polish postfix form the operands are pushed on the stack and operators are applied to the two topmost elements of the stack. In this way the stack can hold all temporary intermediate results. In order to realize the *translation* from an ALGOL 60 program to the object program, with the help of a stack, precedence rules were introduced by assigning priority numbers to the terminal symbols (**BEGIN**, **END**, **IF**, **THEN**, **ELSE**, :=, ×, +, etc.) of the ALGOL 60 grammar.

The formal and explicit introduction of the pushdown stack in mathematical linguistics was motivated by a particular kind of parsing method (cf. [40]) which grew out of reflections on a technique used by Ida Rhodes and others in the automatic translation from Russian to English. See [5] and [26] for further historical references on the pushdown principle.

Theoretical Computer Science

Although the definitions and the focus of interest are not at all independent of the notions of interest for natural languages, their grammars, and other possible applications (e.g., in computer science, developmental biology, psycholinguistics and pattern recognition), the properties of formal languages and grammars and the theory developed to study these systems are not necessarily directly relevant to the field in which the concepts being modeled play a role. In order to exist theory has to abstract away from practical details. Without abstraction and formalization no deep scientific results can be obtained. Formal methods are part of a theory or theory can be developed which

suits the methods. Therefore errors in methods can be avoided and more reliable systems can be created since there are means to show that a system meets the given specifications and, moreover, it may have become possible to automatically generate system parts from a formal description. In Computer Science a full formal analysis of non-trivial systems is not always possible. Only parts or aspects of a complete system can be looked at and errors have to be avoided by careful design. Investigation of limitations of formalisms helps in understanding the formalisms and whether or not they can be applied in what practical situations and at what cost. Insight will be gained from becoming acquainted with formal methods and concepts and this will improve the quality of the use of more ad hoc techniques. This is not only true for formal language theory. It holds as well for any theory which is developed to be applied to benefit practice.

Formal language theory is part of Theoretical Computer Science and Theoretical and Computational Linguistics. Theoretical Computer Science, a field of knowledge born in the mid-1960s, studies the fundamental concepts of computer science by theoretical tools. In this field formal models are provided to study and clarify concepts of computer science. The study of these models is done with theoretical tools borrowed from mathematics and logic and developed in the field itself. The study of these models and the development of theoretical tools to be used in this study result in a coherent framework unifying a body of practice. In models we refrain from looking at all practical details. By distinguishing between relevant and less relevant matters and by emphasizing certain points of view only the essential parts of the problem remain. Due to this abstraction of concrete situations meaningful theorems can be obtained which apply to many concrete situations and which otherwise would not be recognized or would be impossible to state. The framework and its theorems can help to understand practical situations and to manage the complexity of the design of practical systems. Moreover, the framework provides a means to communicate results and methods to others and to teach them to the students of the field.

The three classical subfields of Theoretical Computer Science are formal language theory, automata theory and computability theory. *Formal language theory* flourished after the introduction of the grammar concept in computer science. Generative linguistics and the design of programming languages such as ALGOL have been the two main sources from which formal language theory has been developed. Greibach [26] states that until 1964 formal language theory still could be considered part of (mathematical) linguistics. After 1964 formal language theory developed as a separate branch within several fields of knowledge. Formal language theory has been successful in the classification of grammar and language classes, either by properties of the grammar rules, by parsing properties or by complexity properties.

The study of such properties demonstrates the theoretical limitations of the formal systems. From these limitations their suitability as a model of, e.g., cognitive or linguistic concepts or as an abstract device whose implementation can be used in compiling a (programming) language, can be judged. *Automata theory* started much earlier than formal language theory. It was recognized as a research area in the midfifties, especially after *Automata Studies* appeared. This book, edited by C.E. Shannon and J. McCarthy, contained a collection of papers on different versions of Turing machines, automata to model brain activity and automata to describe the operation of electromechanical systems. The in- and output of automata can be considered as strings of symbols (sentences) from an in- and output language. Therefore the study of automata theory became closely related to that of formal language theory. *Computability theory* started in the early thirties as a subfield of logic. Its first components were recursive function theory and the Turing machine as a model of a "computer". Presently, incorporated in Theoretical Computer Science, it is concerned with the (theoretical) limitations of computer science. It shows what can and cannot be computed by establishing fundamental properties of recursive and recursively enumerable sets. In this field a body of theory has been developed to provide evidence in support of the Church-Turing Thesis. Cf. [32] for a sketch of the development of computability theory.

Especially when equivalences between recognizing and generating devices were established these subfields were linked together. The method of study in formal language theory has become exemplary for the the other subfields of Theoretical Computer Science. Many concepts in other subfields find their origins in formal language theory and often problems in these subfields can be reduced to problems in formal language theory. Because of practical needs other research areas came into existence. *Complexity theory* is the theoretical study of concepts which can be used to measure the effectiveness of algorithms and their application in order to find more efficient techniques for solving problems. The measures are in terms of the spending of computational resources (e.g., computing time and memory space) on specific machine models (e.g, Turing machines or Random Access Machines). While computability theory may yield the result that a particular problem is solvable or unsolvable, complexity theory may give the answer whether a possible solution is practically realizable. Cf. [28] for a sketch of the development of complexity theory. The *Theory of Semantics* is concerned with the development of formal systems for describing the meaning of programming language constructs. The main methods of semantic description are the so-called operational and the mathematical methods. In the operational approach each language construct is associated with a piece of behavior — i.e., the execution of a certain sequence of elementary actions — on an abstract machine. The

mathematical approaches are the axiomatic Floyd-Hoare approach and the functional or denotational approach of D.S. Scott and C. Strachey. In the latter approach mathematical functions are associated with the linguistic constructs of the programming language. Much of this theory is based on models of the lambda calculus provided by Scott [46]. Background knowledge of semantic theories can help the designer of a programming language to avoid ill-understood constructs. For a particular programming language a formal definition helps in the (automatic) implementation of the language and the theory can be used to develop valid proof rules for proving program correctness.

However, there are many other subfields of Computer Science which invite theoretical approaches. It is beyond the goals of this paper to survey these fields. We mention theories developed in support of relational database design, search and representation techniques in Artificial Intelligence, computational geometry, the description of parallel processes, etc. The approaches in these fields rely heavily on the results and the methods of the other, older and more extensively worked out subfields of Theoretical Computer Science. The origins of the subfields' tools and concepts can often be found in the same areas. In the case of formal language theory these areas are mentioned in the table of Figure 4.

Logic, Recursive Function Theory	Thue, Post, Carnap, Church, Turing, Kleene	1910–1955
Communication Theory, Cryptography, Switching Theory	Shannon	1935–1950
Neurophysiology	McCulloch, Pitts, Kleene	1940–1956
Linguistics	Chomsky	1950–
Machine Translation	Bar-Hillel, Yngve, Oettinger, Rhodes	1950–
Programming Language Specification, Compiler Construction	Backus, Naur, Irons, Floyd	1958–
Algebra	Chomsky, Schützenberger, Nivat, Ginsburg, Eilenberg	1963–
Developmental Biology	Lindenmayer	1968–

Figure 4. Origins of formal language theory.

Some of these origins can be characterized as the application of logic in attempts to formalize the *manipulation of symbols* in certain fields. Much of this work was done by logicians interested in more practical research areas.

It will be clear that formal mathematical methods play an important role in Theoretical Computer Science. The workers in this research area are assumed to maintain a mathematical integrity and its subfields use the paradigms of mathematics. When Compiler Construction was a new, and therefore important, topic in computer science much of the research dealt with syntax instead of semantics, for the simple reason that syntax could be formalized. This led to a concentration of research activity in a rather restricted area. This area has been extensively worked out, its results have been and still are applied in practice and many results have lasting value. Moreover, it has been an important, necessary and useful phase in the maturing of computer science and computer scientists. An important part of Computer Science's preoccupation is the manipulation of symbols and strings. Having become acquainted with the formal methods (and their limitations) which govern this manipulation is a sign of maturity. Research in this area has introduced fruitful and scientific attitudes and methodologies in Computer Science. Nevertheless, part of the interest in this area can be explained from the background of computer scientists. Workers in computer science used to be from an engineering/industrial or from a pure mathematics/logic background. This latter background and the association of computer science groups with mathematical departments makes it understandable that such an emerging science wants to earn respect by adhering to the paradigms of its environment and by concentrating on publishable research.

Wegner [49] distinguishes three phases of programming language development, corresponding roughly to the 1950s, 1960s and 1970s. These phases are *discovery* and *description* of concepts, *elaboration* and *analysis* of concepts, and *software technology*. They are characterized by an *empirical*, *mathematical* and an *engineering* approach, respectively. A similar global distinction in periods can be made for more topics of Computer Science. However, often this static tripartition does not do justice to the area. There is a continuous interaction between theory and practice. In this interaction the empirical, mathematical and engineering approach can often be recognized but not always and not always in that order. Therefore it is useful to add the following three observations to such a global view. Firstly, as in any scientific area, there is a development of theory as a means to advance our understanding of the basic concepts of the theory itself. This development is not necessarily irrelevant for practice. The theoretical framework can provide a common cultural background for the practitioners from which practical concepts and methodologies can emerge. Moreover, advances in technology may make it possible to use theoretical ideas which until then had to be discarded. Secondly, there is the continuous effort to grasp more aspects of a practical situation — in this case compiler construction — in a comprehensive theoretical framework. Finally, the theory receives impulses from new ideas and

concepts which are discovered in practical situations or are invoked by technological advances.

Practical problems are far from clean and clear. Research in computer science should also be motivated by practical technological considerations. It is difficult to discriminate in this practical research between concepts which really advance our understanding of computational processes and concepts which will have no lasting value. More fundamental research may provide the framework in which concepts can be judged and accepted or rejected. Computer Science has many commercial and military implications. Its funding of projects is often determined by short-term yield. Researchers are looking for fashionable research areas with a direct practical payoff and for which funding is easy and publications will be accepted. They are not necessarily motivated by the objective of obtaining deep results which advance our understanding of computational processes and their management. Neither are they motivated to leave behind a coherent body of methods and results before moving to the next fashionable field. This following of trends of fashion is not necessarily beneficial for the long-term development of computer science.

References

1. J.W. Backus: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. *Int. Conf. on Inform. Processing*, UNESCO Paris, 1959, 125-132.
2. J.W. de Bakker: *Formal Definition of Programming Languages, with an Application to the Definition of ALGOL 60*. Math. Cent. Tracts **16**, Mathematisch Centrum, Amsterdam, 1967.
3. Y. Bar-Hillel, M. Perles & E. Shamir: On formal properties of simple phrase structure grammars, *Z. Phonetik. Sprach. Komm.* **14** (1961) 143-179. Also: Tech. Rep. No. 4 (July 1960), Applied Logic Branch, The Hebrew University of Jerusalem.
4. Y. Bar-Hillel: *Language and Information. Selected Essays on Their Theory and Application*, Addison-Wesley, Reading, Mass., 1964.
5. F.L. Bauer: Historical remarks on compiler construction, *in*: F.L. Bauer & J. Eickel (Eds.): *Compiler Construction: An Advanced Course*, Lect. Notes Comp. Sci. **21** (1974) 603-621, Springer-Verlag, Berlin, Heidelberg, New York.
6. B.V. Bowden (Ed.): *Faster Than Thought*, Pitman, London, 1953.
7. N. Chomsky: Systems of syntactical analysis, *J. Symbolic Logic* **18** (1953) 242-256.
8. N. Chomsky: Three models for the description of language. *IRE Trans. in Inform. Theory*, **2** (1956) 113-124.

9. N. Chomsky: *Syntactic Structures*. Mouton, The Hague, 1957.

10. N. Chomsky & G.A. Miller: Finite state languages, *Inform. and Control* **1** (1958) 91-112.

11. N. Chomsky: On certain formal properties of grammars, *Inform. and Control* **2** (1959) 137-167.

12. N. Chomsky: A note on phrase structure grammars, *Inform. and Control* **2** (1959) 393-395.

13. N. Chomsky: Context-free grammars and pushdown storage, RLE Quart. Prog. Rept. No. 65, MIT, Cambridge, Mass., 1962.

14. N. Chomsky: *Aspects of the Theory of Syntax*. The MIT Press, Cambridge, Mass., 1965.

15. A. Evans Jr: An ALGOL 60 compiler, *Annual Review in Automatic Programming* **4** (1964) 87-124, Pergamon, Elmsford, N.Y.

16. R.J. Evey: The theory and application of pushdown machines, *in*: Mathematical Linguistics and Automatic Translation. Computation Lab. Rept. NSF-10, Harvard University, Cambridge, Mass., 1963.

17. J.A. Feldman: A formal semantics for computer languages and its application in a compiler-compiler, *Comm. Assoc. Comput. Mach.* **9** (1966) 3-9.

18. R.W. Floyd: A descriptive language for symbol manipulation, *J. Assoc. Comput. Mach.* **8** (1961) 579-584.

19. R.W. Floyd: On the nonexistence of a phrase structure grammar for ALGOL 60, *Comm. Assoc. Comput. Mach.* **5** (1962) 483-484.

20. R.W. Floyd: On ambiguity in phrase structure grammars, *Comm. Assoc. Comput. Mach.* **5** (1962) 526, 534.

21. S. Ginsburg & H. Rice: Two families of languages related to ALGOL, *J. Assoc. Comput. Mach.* **9** (1962) 350-371.

22. S. Ginsburg: Methods for specifying families of formal languages — Past, present, future, *in*: R.V. Book (Ed.): *Formal Language Theory. Perspectives and Open Problems*, Academic Press, 1980, pp. 1-22.

23. H.H. Goldstine: *The Computer from Pascal to von Neuman*. Princeton University Press, 1972.

24. S. Gorn: Some basic terminology connected with mechanical languages and their processors, *Comm. Assoc. Comput. Mach.* **4** (1961) 336-337.

25. S. Gorn: Specification languages for mechanical languages and their processors — A baker's dozen, *Comm. Assoc. Comput. Mach.* **4** (1961) 532-542.

26. S.A. Greibach: Formal languages: Origins and directions, *in*: 20th Annual IEEE Symposium on Foundations of Computer Science,

1979, 66-90.

27. Z. Harris: *Methods in Structural Linguistics*. University of Chicago Press, 1951.
28. J. Hartmanis: Observations about the development of Theoretical Computer Science, *in*: 20th Annual IEEE Symposium on Foundations of Computer Science, 1979, 224-233.
29. D.G. Hays: Automatic language-data processing, *in*: H. Borho (Ed.): *Computer Applications in the Behavioral Sciences*. Prentice-Hall, Englewood Cliffs, N.J., 1962.
30. G.M. Hopper: Keynote Address, *in*: [50], 7-20.
31. E.T. Irons: A syntax directed compiler for ALGOL 60, *Comm. Assoc. Comput. Mach.* **4** (1961) 51-55.
32. S.C. Kleene: Origins of recursive function theory, *in*: 20th Annual IEEE Symposium on Foundations of Computer Science, 1979, 371-382.
33. D.E. Knuth: Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145. Correction *in*: *Math. Systems Theory* **5** (1971) 95-96.
34. D.E. Knuth & L. Trabb Pardo: The early development of programming languages, *in*: *Encyclopedia of Computer Science and Technology*, Vol. 7, Dekker, New York, 1977, 419-493.
35. C.H.A. Koster: Two-level grammars, *in*: F.L. Bauer & J. Eickel (Eds.): *Compiler Construction: An Advanced Course*. Lect. Notes Comp. Sci. **21** (1974) 146-156, Springer-Verlag, Berlin, Heidelberg, New York.
36. H.F. Ledgard: Production systems: or Can we do better than BNF? *Comm. Assoc. Comput. Mach.* **17** (1974) 94-102.
37. P. Naur (Ed.): Report on the algorithmic language ALGOL 60, *Comm. Assoc. Comput. Mach.* **3** (1960) 299-314.
38. P. Naur (Ed.): Revised report on the algorithmic language ALGOL 60, *Comm. Assoc. Comput. Mach.* **6** (1963) 1-17.
39. A. Nijholt: *Deterministic Top-Down and Bottom-Up Parsing: Historical Notes and Bibliographies*. Mathematical Centre, Amsterdam, 1983.
40. A.G. Oettinger: Automatic syntactic analysis and the pushdown store, *in*: R. Jakobson (Ed.): *Structure of Language and its Mathematical Aspects*. Proc. of Symposia in Appl. Math., Vol. XII, Amer. Math. Soc., Providence, R.I., 1961, 104-129.
41. E. Pantages: They made the future in the past: Captain Grace Murray Hopper, *Data* **11** (1981) Nr. 1/2, February, 14-19.
42. E.L. Post: Formal reductions of the general combinatorial problem, *Amer. J. Math.* **65** (1943) 197-268.

43. M. Rabin & D.S. Scott: Finite automata and their decision problems, *IBM J. Res. Develop.* **3** (1959) 114-125.

44. K. Samelson & F.L. Bauer: Sequentielle Formelübersetzung. *Elektron. Rechenanlagen* **1** (1959) Vol.4. Also: Sequential formula translation, *Comm. Assoc. Comput. Mach.* **3** (1960) 76-83.

45. S. Scheinberg: Note on the Boolean properties of context-free languages, *Inform. and Control* **3** (1960) 372-375.

46. D.S. Scott: Outline of a mathematical theory of computation. Proc. 4th Annual Princeton Conference on Inf. Sciences and Systems, 1970, 169-176.

47. P.B. Sheridan: The arithmetic translator-compiler of the IBM FORTRAN automatic coding system, *Comm. Assoc. Comput. Mach.* **2** (1959) 9-21.

48. T.B. Steel Jr (Ed.): *Formal Languages: Description Languages for Computer Programming*. Proc. of the IFIP Working Conf. (held in New York 1964), North-Holland, Amsterdam, 1966.

49. P. Wegner: Programming languages: The first 25 years. *IEEE Trans. Comput.* **25** (1976) 1207-1225.

50. R.S. Wells: Immediate constituents, *Language* **23** (1947) 81-117.

51. R.L. Wexelblat (Ed.): *History of Programming Languages*. Academic Press, New York, 1981.

52. H. Wolpe: Algorithm for analyzing logical statements to produce truth function table, *Comm. Assoc. Comput. Mach.* **1** (1958) 4-13.

Generating Strings with Hypergraph Grammars

Joost Engelfriet

Department of Computer Science, University of Leiden
P.O. Box 9512, 2300 RA Leiden, The Netherlands

Context-free hypergraph grammars generate the same string languages as deterministic tree-walking transducers.

1. Introduction

A graph grammar generates a set of graphs, also called a graph language. To obtain an overview of the usefulness of graph grammars, see [6,10,11]. Since strings can be viewed as (chain-like) graphs, every string grammar can be viewed as a graph grammar in an obvious way. More importantly, every type of graph grammar may also be used as a type of string grammar: just restrict attention to those graph grammars that generate strings only. Thus the sentential forms of such a grammar may be arbitrary graphs, but the generated graphs are strings. In this paper we investigate the string-generating power of a particular type of graph grammar: the context-free hypergraph grammar, recently (re-)introduced in [5,19,24] (see [19] for historical remarks). In such a grammar the sentential forms are directed hypergraphs, of which the hyperedges are labeled by terminal and nonterminal symbols. One derivation step consists of replacing one hyperedge (labeled by a nonterminal) by a hypergraph, according to some production of the grammar. These grammars are of interest because (1) they generate a reasonably large class of (hyper)graph languages, and (2) the way they work is easy to understand and to visualize (a vital feature of graph grammars). They can be used, e.g., to model the top-down design of a relational database scheme [4].

We will characterize the string languages generated by context-free hypergraph grammars to be those generated by the tree-to-string transducers of [1], thus answering question (4) in the conclusion of [19]. These languages are also closely related to the dependency path languages of attribute grammars (see [12]). Intuitively, this characterization can be understood through the notion of derivation tree of a context-free hypergraph grammar; cf. [23]. In fact, the graph (in particular, string) generated in a derivation of the grammar, is distributed over the corresponding derivation tree in a "snake-like" manner, reminiscent both of the route taken on a derivation tree by a tree-walking automaton, and of the dependency graph of a derivation tree in an

attribute grammar. As a special case we characterize the string languages generated by linear hypergraph grammars to be those generated by 2-way finite state transducers.

2. Hypergraphs and Hypergraph Grammars

A directed hypergraph consists of a set of nodes and a set of (hyper)edges, just as an ordinary graph except that an edge is incident with any number of nodes rather than exactly two. The edges are directed in the sense that the nodes incident with a given edge are linearly ordered. Formally (cf. [5,24]), a (directed edge-labeled) *hypergraph* (or, shortly, *graph*) is a system $H = (V,E,\Sigma,nod,lab)$ where V is a finite set of nodes (or vertices), E is a finite set of (hyper)edges, Σ is an alphabet of edge labels, $nod: E \to V^*$ is the incidence function, and $lab: E \to \Sigma$ is the edge labeling function. Thus, *nod* maps every edge into a sequence of nodes (of any length). If $nod(e) = (v_1,\ldots,v_n)$, $n \geqslant 0$, then e is called an n-edge, v_i is also denoted by $nod(e,i)$, and e and v_i are said to be incident. Pictorially (cf. [24]), nodes are indicated by fat dots, as usual, and the edge e is indicated by a box containing $lab(e)$, with a line between e and v_i labeled by i, for each $1 \leqslant i \leqslant n$. These lines are called the "tentacles" of the hyperedge [19].

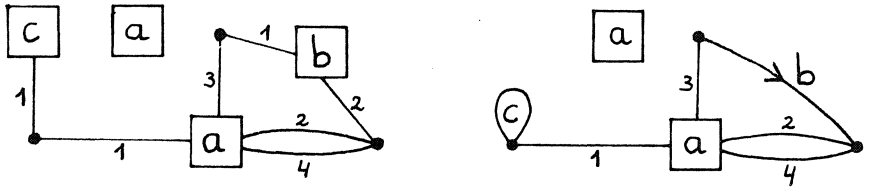

Figure 1.

As an example, the hypergraph in the left part of Figure 1 has (enumerated from left to right) $V = \{u,v,w\}$ and $E = \{e_1,e_2,e_3,e_4\}$, and it has $\Sigma = \{a,b,c\}$, $nod(e_1) = (u)$, $nod(e_2) = ()$, $nod(e_3) = (u,w,v,w)$, $nod(e_4) = (v,w)$, $lab(e_1) = c$, $lab(e_2) = lab(e_3) = a$, and $lab(e_4) = b$. To simplify comparison with ordinary directed graphs we will also draw a 2-edge e, with $nod(e) = (v_1,v_2)$, as an ordinary directed edge from v_1 to v_2, labeled by $lab(e)$, and we will also draw a 1-edge e, with $nod(e) = (v)$, as a "balloon", "tied" at v and labeled by $lab(e)$, as indicated in a picture of the same hypergraph in Figure 1, to the right. Note that the "balloons" can serve as node labels; thus each ordinary node- and edge-labeled directed graph can be viewed as a hypergraph in a natural way.

For a given hypergraph H, its components are denoted by V_H, E_H, Σ_H, nod_H, and lab_H, respectively (and the subscript H is dropped if it is clear from the context). For an alphabet Σ, the set of all hypergraphs H with $\Sigma_H = \Sigma$ is denoted by $HGR(\Sigma)$. A subset of $HGR(\Sigma)$ is called a (hyper)graph language.

Figure 2.

Since we will be interested in particular in strings, we now define the graphs that we use to model strings; cf. [19]. Let Σ be an alphabet, and let $w = \sigma_1 \cdots \sigma_k$ be a string over Σ with $k \geq 0$, $\sigma_i \in \Sigma$. Then the *string graph* corresponding to w is $gr(w) = (V,E,\Sigma,nod,lab)$ with $V = \{0,1,...,k\}$, $E = \{1,...,k\}$, $nod(i) = (i-1,i)$, and $lab(i) = \sigma_i$. Note that $gr(\lambda)$, where λ is the empty string, consists of one node and no edges. As an example, Figure 2 shows the string graph $gr(abaa)$. In what follows we will not always distinguish between a string w and the string graph $gr(w)$, and, similarly, between a string language L and the string graph language $gr(L) = \{gr(w) \mid w \in L\}$. It should be observed here that in graph languages we will, as usual, not distinguish between isomorphic graphs (where isomorphisms are defined in the obvious way). Thus, in $gr(w)$, the fact that the nodes and edges are integers is irrelevant.

To be able to discuss the application of grammatical productions to hypergraphs, we need four easy operations on hypergraphs.

(1) Removal of one edge. For $H \in HGR(\Sigma)$ and $e \in E_H$, $H-e$ denotes the hypergraph $(V_H, E_H-\{e\}, \Sigma, nod, lab)$ where nod and lab are the restriction to $E_H-\{e\}$ of nod_H and lab_H, respectively. Pictorially, one hyperedge is removed, by erasing the corresponding box with its tentacles.

(2) Disjoint union. Let $H,K \in HGR(\Sigma)$ be disjoint graphs, i.e., V_H, V_K and E_H, E_K are disjoint sets. Then

$$H+K = (V_H \cup V_K, E_H \cup E_K, \Sigma, nod_H \cup nod_K, lab_H \cup lab_K).$$

Pictorially, the pictures of H and K are put together into one picture, without interconnection.

(3) Identification of nodes. Let $H \in HGR(\Sigma)$ and let $R \subseteq V_H \times V_H$. Intuitively, we want to identify nodes u and v, for every pair $(u,v) \in R$. Let $\equiv_R$ denote the smallest equivalence relation on V_H containing R; for $v \in V_H$, let $[v]_R$ denote the equivalence class of v with respect to $\equiv_R$, and let $V_H/\equiv_R = \{[v]_R \mid v \in V_H\}$. Then

$$H/R = (V_H/\equiv_R, E_H, \Sigma, nod, lab_H)$$

where, for every n-edge $e \in E_H$,

$$nod(e) = ([nod(e,1)]_R, ..., [nod(e,n)]_R).$$

Note that H/R has the same edges as H. Pictorially, for each $(u,v) \in R$, nodes u and v are moved together (carefully) until they coincide.

(4) Gluing along an edge. Let $H,K \in HGR(\Sigma)$ be disjoint hypergraphs, and let $e \in E_H$ and $f \in E_K$ be n-edges for some $n \geqslant 0$. Define $R = \{(nod_H(e,i), nod_K(f,i)) \mid 1 \leqslant i \leqslant n\}$. Then

$$glue(H,e,K,f) = ((H-e)+(K-f))/R.$$

Intuitively, the graphs are glued together by pairwise identification of the nodes of e and f; the edges e and f themselves disappear. Put your fingertips together and think about it. An example is given in Figure 3; from left to right: H with e, K with f, and $glue(H,e,K,f)$, with edge labels omitted.

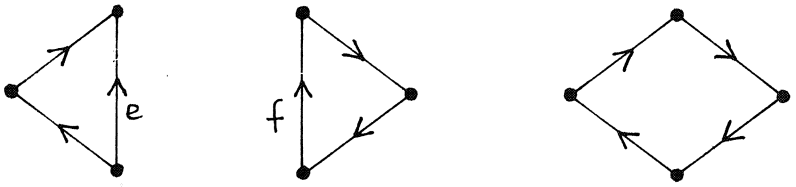

Figure 3.

We are now prepared for the definition of context-free hypergraph grammar.

Definition 1. A *context-free hypergraph grammar* (shortly *cfhg*) is a system $G = (\Sigma,\Delta,P,S)$ where Σ is an alphabet, $\Delta \subseteq \Sigma$ is the terminal alphabet (and $\Sigma-\Delta$ is the nonterminal alphabet), P is the finite set of productions, and $S \in \Sigma-\Delta$ is the initial nonterminal. Every production in P is of the form (e,H) where $H \in HGR(\Sigma)$ and $e \in E_H$ with $lab_H(e) \in \Sigma-\Delta$. □

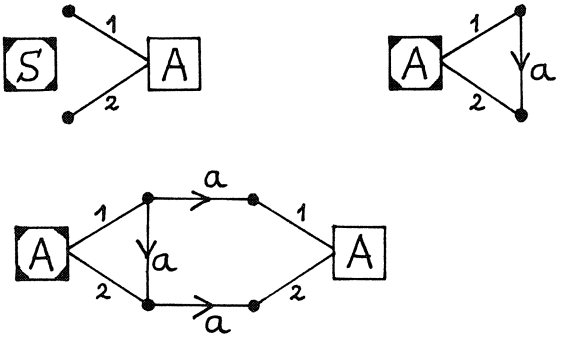

Figure 4.

Intuitively, the application of a production (e,H) consists of replacing an edge e by the hypergraph $H-e$. A picture of (e,H) is given by a picture of H, in which the box corresponding to e is decorated with black corners. As an example, Figure 4 shows the three

productions of a cfhg $G = (\Sigma, \Delta, P, S)$ with $\Sigma = \{S, A, a\}$ and $\Delta = \{a\}$. Terminology: A nonterminal edge is an edge e with $lab(e) \in \Sigma - \Delta$, and similarly for a terminal edge. For a production (e, H), e is called the left-hand side and H the right-hand side of the production.

Let $G = (\Sigma, \Delta, P, S)$ be a cfhg. Formally, application of a production $\pi = (e, H)$ of G is defined as follows. Let $K \in HGR(\Sigma)$; in case K is not disjoint with H, take an isomorphic copy of K that has this property. Then π is applicable to K at a nonterminal edge f of K if $lab_K(f) = lab_H(e)$ and f, e are both n-edges for some $n \geqslant 0$. The application of π to K at f results in the graph $K' = glue(H, e, K, f)$, or any graph isomorphic to K'; notation: $K \Longrightarrow K'$. As usual, the *language generated by* G is $L(G) = \{H \in HGR(\Delta) \mid \underline{S} \Longrightarrow^* H\}$ where $\underline{S}$ is the hypergraph without nodes and with one edge e such that $nod(e) = ()$ and $lab(e) = S$. A graph $H \in HGR(\Sigma)$ such that $\underline{S} \Longrightarrow^* H$ is called a sentential form of G. The class of all languages generated by context-free hypergraph grammars is denoted by CFHG. Moreover, the class of all string (graph) languages generated by cfhg's is denoted by STR(CFHG). Thus

$$\text{STR(CFHG)} = \{L \in \text{CFHG} \mid L \subseteq gr(\Delta^*) \text{ for some alphabet } \Delta\}.$$

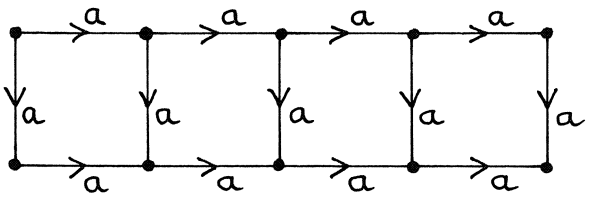

Figure 5.

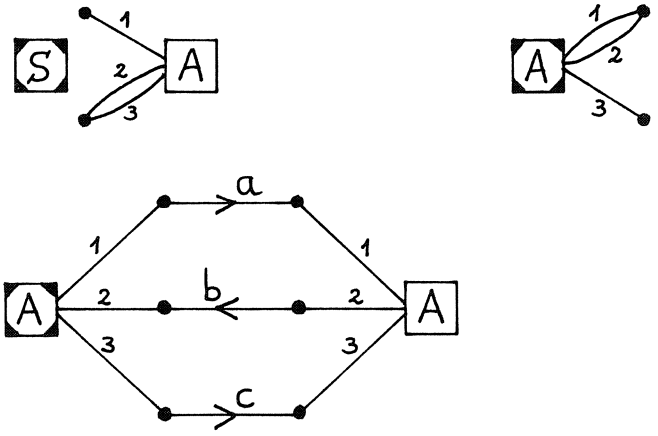

Figure 6.

As an example, for the grammar G of Figure 4, $L(G)$ consists of all "ladders" of the form given in Figure 5. As another example, the grammar G of Figure 6 generates the language $gr(L)$ with $L = \{a^n b^n c^n \mid n \geq 0\}$; thus, identifying L and $gr(L)$, $L \in \mathrm{STR(CFHG)}$. Using the productions of G in the way suggested by Figure 6, one can see that G generates the string $a^n b^n c^n$ in a "snake-like" fashion, as shown in Figure 7 for $n = 4$. As a final example, very similar to the previous one, consider the cfhg G of Figure 8, with 6 productions ($x = a$ or $x = b$). G generates all strings $\$w\$w\$$ where w is an odd-length palindrome over the alphabet $\{a,b\}$. This time, the way the productions are drawn suggests that the strings are generated as chains rather than snakes.

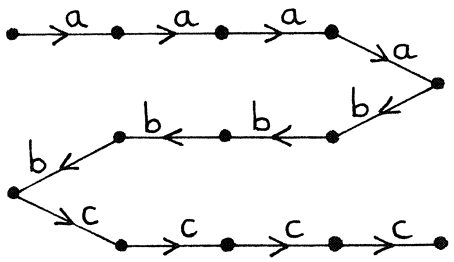

Figure 7.

As the reader may have noticed in Figures 1 and 6, different tentacles of a hyperedge may lead to the same node. However, as suggested by [9], this phenomenon can always be avoided in cfhg's for nonterminal edges (not for terminal edges of course). To formulate this as a result, we need some terminology. An edge e in a hypergraph H is *loop-free* if the nodes in $nod_H(e)$ are all different. A cfhg G is *loop-free* if, for every production (e,H) of G, all nonterminal edges of H are loop-free. Thus, the cfhg's of Figures 4 and 8 are loop-free, but the one of Figure 6 is not. We now state the "loop-free lemma".

Theorem 2. *For every cfhg G there is a loop-free cfhg G' such that* $L(G') = L(G)$. □

Remarks. (1) This result is similar to the removal of λ-productions from a context-free grammar. (2) In [19,24] every production in a cfhg should have a loop-free left-hand side; in [5] arbitrary left-hand sides are allowed. □

Loop-free cfhg's are more attractive than arbitrary cfhg's because the way they work is much easier to visualize: when computing $glue(H,e,K,f) = ((H-e)+(K-f))/R$, in the application of a production (e,H), both e and f are loop-free, and hence no other nodes than those indicated by R are identified (i.e., $\equiv_R = R \cup R^{-1}$). This means that nodes of a sentential form can never be identified in a later stage of the derivation, and, consequently, the "terminal part" of the

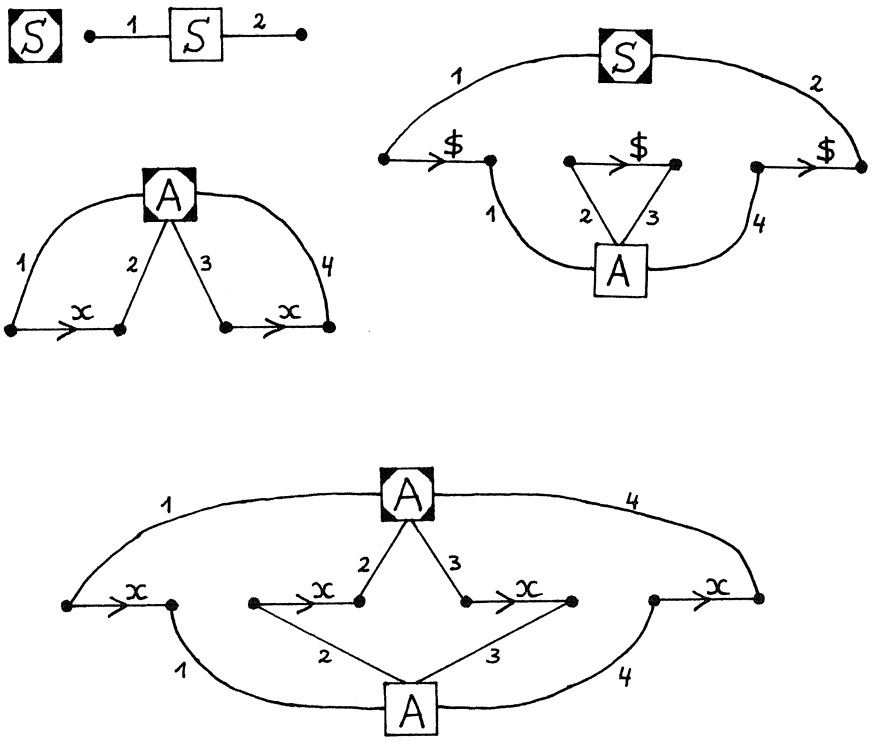

Figure 8.

sentential form (obtained by removing all nonterminal edges) is a subgraph of the generated graph. It also means that for every production (e,H), applied in a derivation, the terminal part of H is a subgraph of the generated graph.

In this sense, the loop-free lemma may be viewed as a way of showing the power of the attractive formalism of loop-free cfhg's. Thus, whenever we will construct cfhg's that are not loop-free, we will say that this is possible "due to the loop-free lemma".

3. Known Formalisms Viewed as Hypergraph Grammars

To become more familiar with cfhg's, we consider in this section some well-known string and tree grammars that can be viewed as context-free hypergraph grammars. Also, cfhg's are very suitable to generate the dependency graph language of an attribute grammar, as shown in [7].

Figure 9 contains a cfhg generating the string language a^*b. It clearly corresponds to a regular (string) grammar with productions $S \rightarrow A$, $A \rightarrow aA$, and $A \rightarrow b$. Note that, in general, all nonterminal edges (except S) are 1-edges. Regular string grammars can be

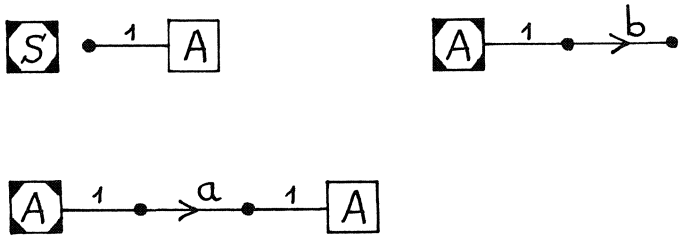

Figure 9.

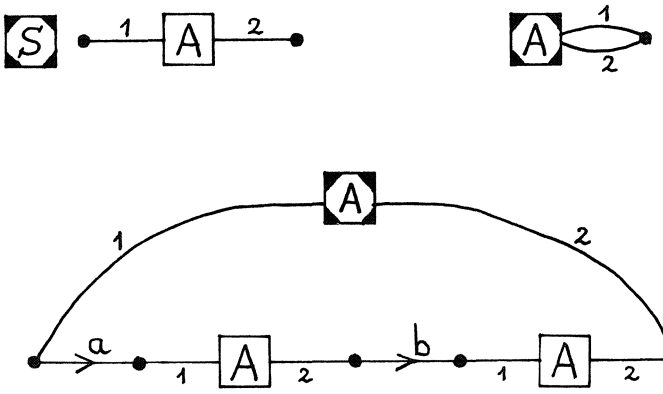

Figure 10.

generalized to context-free string grammars and to regular tree grammars. Figure 10 contains a cfhg generating the string language of all well-formed parenthesis expressions (where a is the left- and b the right-parenthesis); it corresponds to the context-free grammar with productions $S \to A$, $A \to aAbA$, $A \to \lambda$. Note that λ-productions can be simulated "due to the loop-free lemma". From this example it should be clear that all context-free grammars can be viewed as cfhg's (where all nonterminal edges, except S, are 2-edges). Hence STR(CFHG) contains all context-free (string) languages (properly of course, see Figures 6 and 8). Figure 11 contains a cfhg that generates all ordered binary trees. Internal nodes of the trees are labeled a, and leaves are labeled b; the order is indicated by edge labels f and h (standing for left and right, respectively). This cfhg corresponds to the regular tree grammar (cf. [17,7]) with productions $S \to A$, $A \to a(A,A)$, $A \to b$.

For the reader familiar with context-free tree grammars (cf., e.g., [15]) we note that they can also easily be simulated by cfhg's, as long as they are noncopying and nondeleting. For a copying, nondeleting, (IO) context-free tree grammar G it is possible to construct a cfhg G' that generates DOAGs (directed ordered acyclic graphs) which, when

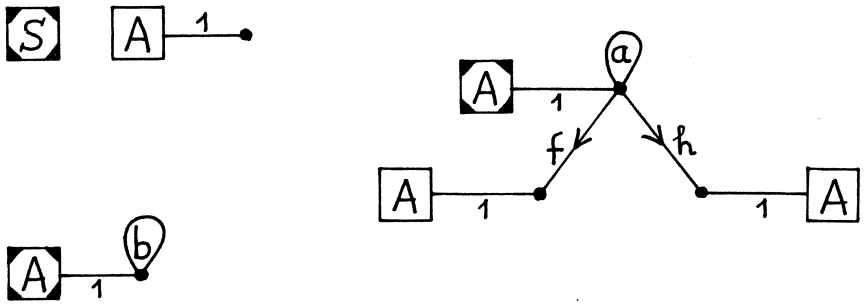

Figure 11.

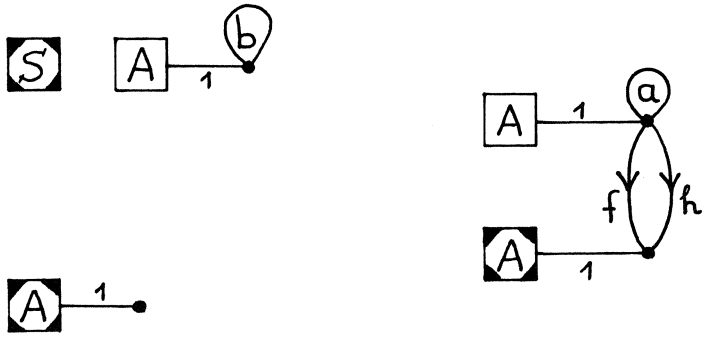

Figure 12.

unfolded, give the trees generated by G; thus $L(G) = \text{unfold}(L(G'))$. In this sense the cfhg of Figure 12 simulates the context-free tree grammar with productions $S \to A(b)$, $A(x) \to A(a(x,x))$, $A(x) \to x$, generating all full binary trees.

In general, a graph grammar can also be used to generate a transduction, i.e., a relation between graphs: if (H,K) is in the relation, then the grammar generates the disjoint union $H+K$, and in some way marks H and K, to distinguish the input graph H from the output graph K. Thus, one may investigate how the top-down tree transducer (see, e.g., [17]) and even the macro tree transducer (see, e.g., [16]) can be simulated by cfhg's.

We now turn to the dependency graphs of attribute grammars (AGs). An attribute grammar [22] associates a "dependency graph" with each production and each derivation tree of a given context-free (string) grammar. The set of all dependency graphs of derivation trees forms the *dependency graph language* defined by the AG. We assume the reader to be familiar with attribute grammars; see, e.g., [22,2,12]. Let DEP-AG denote the class of all dependency graph languages of AGs. We will assume here that the nodes of dependency graphs (corresponding to attributes) are not labeled, but the edges

(corresponding to dependencies between attributes) are; in particular, each edge of the dependency graph of a production is given a unique label; see [12]. Now the following result is straightforward to show; see Section 16.8 of [7].

Theorem 3. DEP-AG $\subseteq$ CFHG. □

In fact, each dependency graph of a production of the AG corresponds to a production of the simulating cfhg, in a straightforward way. The nonterminals of the underlying context-free grammar of the AG are also the nonterminals of the cfhg; each nonterminal has a tentacle to each of its attributes. An example should make this clear. Figure 13 contains the dependency graphs of an AG, corresponding to the productions $S \to A$, $A \to AA$, $A \to a$ of the underlying context-free grammar. The nonterminal A has an inherited attribute α and a synthesized attribute β, and the nonterminal S has attribute β only. The dependency edges are given arbitrary unique labels a to f. Figure 14 shows the cfhg that generates the dependency graph language of this AG.

Remarks 4. (1) Suppose that, in Figure 13, the edge labeled e is not present. Then the dependency graph language is clearly a string language. Let STR(DEP-AG) denote the class of string languages in DEP-AG. Then STR(DEP-AG) $\subseteq$ STR(CFHG), by Theorem 3.

(2) It is shown in [13] that NLC graph grammars can also be used to generate dependency graph languages. However, cfhg's do this in a more natural way.

(3) It should be clear from the example that the translation of an AG into an equivalent cfhg can be realized in deterministic logarithmic space. This implies that lower bounds carry over from dependency graph languages of AGs to languages in CFHG. As an example, it is immediate from [21] that there is no polynomial time algorithm to decide, for a given cfhg G, whether all graphs in $L(G)$ are acyclic. The same holds, e.g., for "planar" and "bipartite" instead of "acyclic" (These properties are decidable for cfhg's, by the elegant result of [8]).

(4) Some edges in dependency graphs of an AG are often known to be "passing" edges: the nodes (i.e., attributes) connected by such an edge are meant to have the same value. Thus, it is meaningful to define a variation of dependency graphs in which these nodes are identified. Due to the loop-free lemma the new dependency graph language can still be generated by a cfhg. If, e.g., the edge labeled e in Figure 13 is a passing edge, then one just identifies the incident nodes in the corresponding production in Figure 14 (removing the edge). □

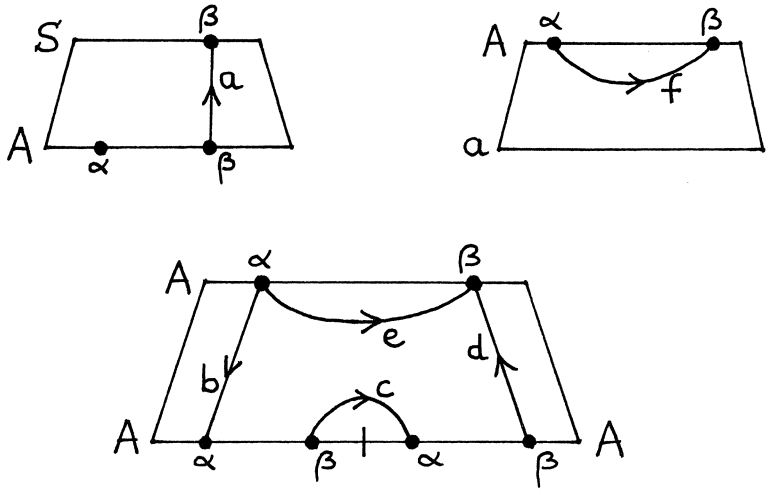

Figure 13.

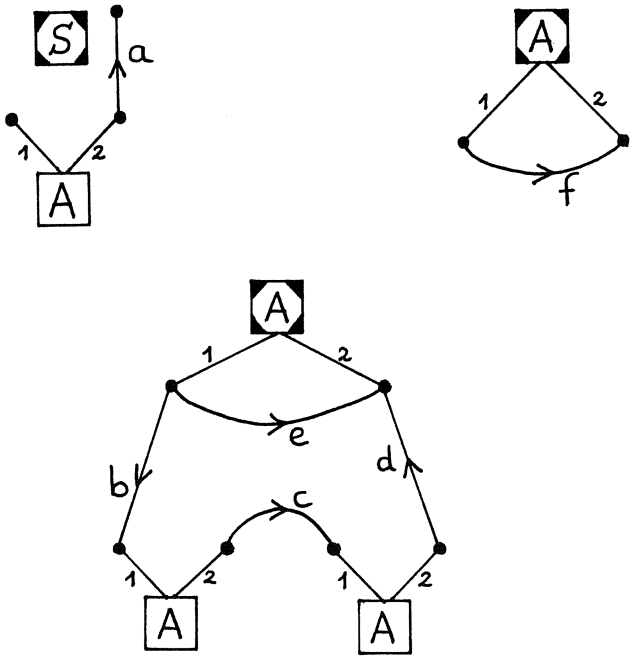

Figure 14.

4. The String Generating Power of Hypergraph Grammars

Up to now we did not say anything new. In this section we turn to a new result: we characterize STR(CFHG) as a class of string languages known in formal language theory, in particular the theory of tree transducers and attribute grammars. It is the class OUT(DTWT) of output languages of the deterministic tree-walking transducers of [1]; see also [14,12]. These transducers walk on the derivation trees of a context-free grammar, and translate them into strings. The class OUT(DTWT) is equal to the class $yT_{fc}(\mathrm{REC})$ of yields of images of the regular (or recognizable) tree languages under finite-copying top-down tree transducers; see Corollary 4.11 of [14], where OUT(DTWT) is denoted DCT(REC). From this and the proof of Theorem 5.7 of [12], it follows that OUT(DTWT) $\subseteq$ HOM(STR(DEP-AG)) where HOM denotes the class of homomorphisms (on strings). Thus, to show that OUT(DTWT) $\subseteq$ STR(CFHG), it suffices, by Theorem 3 of the previous section (cf. Remarks 4(1)), to prove the following lemma.

Lemma 5. STR(CFHG) *is closed under (string) homomorphisms.*

Proof: Let $G = (\Sigma, \Delta, P, S)$ be a cfhg that generates a string (graph) language, and let $h: \Delta^* \to \Omega^*$ be a string homomorphism. We have to show that $h(L(G)) \in \mathrm{CFHG}$. A cfhg $G' = ((\Sigma - \Delta) \cup \Omega, \Omega, P', S)$ generating $h(L(G))$ is constructed from G by changing every right-hand side H of a production of G as follows: every terminal edge e of H is replaced by $gr(h(lab_H(e)))$. More precisely, let $lab_H(e) = a$ and $nod_H(e) = (u,v)$. If $h(a) = b_1 \cdots b_k$ $(b_i \in \Omega)$ with $k \geqslant 1$, then $k-1$ "new" nodes $w_1, \ldots, w_{k-1}$ are added to H, and e is replaced by k "new" edges $e_1, \ldots, e_k$ with $lab(e_i) = b_i$ and $nod(e_i) = (w_{i-1}, w_i)$ for $1 \leqslant i \leqslant k$ (where $w_0 = u$ and $w_k = v$). If $h(a) = \lambda$, then e is dropped from H and the nodes u and v are identified (which is possible due to the loop-free lemma). □

It remains to show that STR(CFHG) $\subseteq$ OUT(DTWT). We prove this by a direct simulation of a string-generating cfhg by a tree-walking transducer. A *deterministic tree-walking transducer* (abbreviated *dtwt*) is an automaton with a finite control, an input tree, and an output string. The input trees are all derivation trees of a given context-free grammar. At any moment of time the automaton is at a certain node of the input tree. Depending on the state of its finite control and the label of the node, it changes state, outputs a string to the output tape, and either stays at the node or moves to the father or a specific son of the node. The automaton starts in its initial state at the root of the input tree, and halts whenever it reaches a final state. In this way it translates the input tree into an output string. The output language of the automaton is the set of all output strings obtained in this way. OUT(DTWT) denotes the class of all such output languages. For more details see [1], or [14] (where the dtwt is called a dct-transducer).

Lemma 6. STR(CFHG) $\subseteq$ OUT(DTWT).

Proof (sketch): Let $G = (\Sigma, \Delta, P, S)$ be a loop-free cfhg generating a string language. To better understand the idea of the proof we first assume that G satisfies the following two restrictions:
(1) There is a unique production $\pi_{in} = (e, H)$ in P with $lab_H(e) = S$. Moreover, $H - e$ consists of a 2-edge, i.e., $V_H = \{u, v\}$, $E_H = \{e, e'\}$, $nod_H(e) = ()$, and $nod_H(e') = (u, v)$. Furthermore, the first [last] node of every string generated by G is u [v, respectively].
(2) Each node of the right-hand side of a production in P is incident with at most one nonterminal edge.

The cfhg of Figure 10 satisfies (1) but not (2), and the cfhg of Figure 8 satisfies both restrictions (and is loop-free). In [19] STR(CFHG) is defined in such a way that (1) is always satisfied.

The dtwt M to be constructed walks on the derivation trees of a context-free grammar G' obtained directly from G as follows (in fact, these trees should also be viewed as derivation trees of G). The nonterminals of G' are the productions of G, and G' has no terminals. Its initial nonterminal is π_{in}, see (1) above. G' contains all productions $\pi_0 \rightarrow \pi_1 \pi_2 \cdots \pi_k$ with $\pi_i = (e_i, H_i) \in P$ such that $H_0 - e_0$ contains precisely k nonterminal edges, and, for $1 \leqslant i \leqslant k$, the i-th nonterminal edge has the same label as e_i and both are n-edges for some $n \geqslant 0$ (assuming that these k nonterminal edges are given some fixed but arbitrary order). It should be clear that every derivation tree t of G' determines a graph $H(t)$ in $L(G)$, obtained by taking the disjoint union of all terminal parts of (right-hand sides of) productions of G that occur as labels of nodes of t, and identifying nodes as follows: if production $\pi_0 \rightarrow \pi_1 \pi_2 \cdots \pi_k$ of G' occurs in t and f_i is the i-th nonterminal edge of $H_0 - e_0$, then $nod(f_i, j)$ should be identified with $nod(e_i, j)$, for all $1 \leqslant j \leqslant n$, $1 \leqslant i \leqslant k$. This is the key to understanding how the dtwt M can walk through t, producing $H(t)$ on its output tape. When M is at a node x of t, labeled $\pi = (e, H)$, then M also keeps track in its finite control of a node of H; in other words, M is also "at a node of H", and consequently also "at a node of $H(t)$". M starts at the root of t, labeled π_{in}, and at the node u of H_{in} (see (1) above); it halts when it returns to the root, at node v of H_{in}. Now suppose that M is at node x of t, labeled $\pi = (e, H)$, and at node u of H. Then M behaves as follows.

(i) If H has a terminal edge f with $nod_H(f) = (u, v)$, then M "moves" to node v of H, remains at node x of t, and outputs $lab_H(f)$.

(ii) Otherwise, if H has a nonterminal edge $f \neq e$ incident with u, f is the i-th nonterminal edge of $H - e$, and $u = nod_H(f, j)$, then M moves to the i-th son of x in t, labeled, say, by $\pi_i = (e_i, H_i)$, and moves to $nod(e_i, j)$ in H_i (without producing output).

(iii) Otherwise u is incident with e. In that case M moves to the father y of x. Suppose that $u = nod_H(e, j)$, that x is the i-th son of

y, and that y is labeled by $\pi_0 = (e_0, H_0)$ in t. Then M moves to $nod(f_i, j)$ in H_0, where f_i is the i-th nonterminal edge of $H_0 - e_0$. M does not produce output.

This ends the description of M. In the general case, the fact that a node may be incident with more than one nonterminal edge makes it impossible for M to choose the correct edge deterministically. However, G can first be changed in such a way that its new nonterminals are of the form (X,p) where X is an old nonterminal and p is a partial function from $\{1,...,n\}$ to itself for some $n \geq 0$. If (X,p) labels an n-edge e, in some sentential form of G, and $nod(e) = (u_1,...,u_n)$, then $p(i) = j$ means that (X,p) generates the substring of $H(t)$ from node u_i to node u_j (viewing $u_1,...,u_n$ as nodes of $H(t)$ too). From this information M can easily see which nonterminal edge to take, in case of doubt. A slight extension of the information allows M to find the start and end of $H(t)$. Note that this kind of information is analogous to the i/s-graphs (in attribute grammars) that model the dependency paths in the dependency graph of a derivation subtree. □

Our main result follows from Lemmas 5 and 6.

Theorem 7. STR(CFHG) = OUT(DTWT)
and STR(CFHG) = HOM(STR(DEP-AG)). □

To illustrate this result, it is easy to see from the cfhg's of Figures 8 and 10 (and Lemma 5) that, for every context-free language L, a cfhg can be constructed generating the language $\{ww \mid w \in L\}$. Of course this language can also be generated by a dtwt that walks on the derivation trees of a context-free grammar for L: the dtwt just walks twice through the tree in a depth-first left-to-right fashion. The language can also easily be defined by a 2-pass attribute grammar (and a homomorphism).

Quite a lot is known about OUT(DTWT); see, e.g., [14]. For instance, it is a full AFL containing Parikh languages only. The hierarchy result for STR(CFHG) in Theorem 4.4 of [19] can also be understood from a similar hierarchy result for OUT(DTWT) (in Theorems 3.2.5 and 4.9 of [14]): roughly speaking, if nonterminal edges have at most $2k$ tentacles, then the dtwt is at most k-crossing.

As an interesting special case we consider the linear cfhg's (studied in [25] and, as finite graph automata, in [20]). A cfhg is *linear* if every right-hand side of a production contains at most two nonterminal edges (The cfhg's in Figures 4, 6, 8, 9, and 12 are linear). Let LIN-CFHG denote the class of languages generated by linear cfhg's. Clearly, in the linear case, the derivation trees of the context-free grammar on which the dtwt works are not branching. Thus, we may view the dtwt as a 2-way deterministic finite state transducer with strings as input and output; see, e.g., [14]. Let OUT(2DGSM) denote the class of output languages of such transducers.

Theorem 8. STR(LIN-CFHG) = OUT(2DGSM). □

Also about OUT(2DGSM) quite a lot is known. As an example, we obtain the fact that there is a string language in CFHG that is not in LIN-CFHG. In fact there exists even a context-free language that is not in OUT(2DGSM); see, e.g., [18]. We also note that linear cfhg's are related to parallel rewriting: one nonterminal edge can grow pieces of graph at different places of the sentential form simultaneously (as in Figure 8). For strings there is a formal relationship between cfhg's and ET0L systems (a well-known type of parallel rewriting systems; see, e.g., [3, 14]): OUT(2DGSM) = $ET0L_{FIN}$, the class of ET0L languages of finite index; see Corollary 4.11 of [14] where OUT(2DGSM) is denoted DCS(REG).

References

1. A.V. Aho & J.D. Ullman: Translations on a context free grammar, *Inform. and Control* **19** (1971) 439-475.
2. H. Alblas: A characterization of attribute evaluation in passes, *Acta Inform.* **16** (1981) 427-464.
3. P.R.J. Asveld: Controlled iteration grammars and full hyper-AFL's, *Inform. and Control* **34** (1977) 248-269.
4. C. Batini & A. D'Atri: Relational data base design using refinement rules, *R.A.I.R.O. Inform. Théor.* **17** (1983) 97-119.
5. M. Bauderon & B. Courcelle: Graph expressions and graph rewritings (1986), Report I-8623, University of Bordeaux 1, France.
6. V. Claus, H. Ehrig & G. Rozenberg (Eds.): *Graph-Grammars and Their Application to Computer Science and Biology*, Lect. Notes in Comp. Sci. **73** (1979), Springer-Verlag, Berlin, Heidelberg, New York.
7. B. Courcelle: Equivalences and transformations of regular systems - Applications to recursive program schemes and grammars, *Theoret. Comput. Sci.* **42** (1986) 1-122.
8. B. Courcelle: Recognizability and second-order definability for sets of finite graphs (1986), Report I-8634, University of Bordeaux 1, France.
9. B. Courcelle: personal communication.
10. H. Ehrig, M. Nagl & G. Rozenberg (Eds.): *Graph-Grammars and Their Application to Computer Science*, Lect. Notes in Comp. Sci. **153** (1983), Springer-Verlag, Berlin, Heidelberg, New York.
11. H. Ehrig, M. Nagl & G. Rozenberg (Eds.): *Proc. Third Workshop on Graph-Grammars and Their Applications to Computer Science* (1986), Warrenton, Va.

12. J. Engelfriet & G. Filé: Passes and paths of attribute grammars, *Inform. and Control* **49** (1981) 125-169.

13. J. Engelfriet, G. Leih & G. Rozenberg: Apex graph grammars and attribute grammars (1987), Report 87-04, University of Leiden, The Netherlands.

14. J. Engelfriet, G. Rozenberg & G. Slutzki: Tree transducers, L systems, and two-way machines, *J. Comput. System Sci.* **20** (1980) 150-202.

15. J. Engelfriet & E.M. Schmidt: IO and OI, *J. Comput. System Sci.* **15** (1977) 328-353 and *J. Comput. System Sci.* **16** (1978) 67-99.

16. J. Engelfriet & H. Vogler: Macro tree transducers, *J. Comput. System Sci.* **31** (1985) 71-146.

17. F. Gécseg & M. Steinby: *Tree automata* (1984), Akadémiai Kiadó, Budapest.

18. S.A. Greibach: One-way finite visit automata, *Theoret. Comput. Sci.* **6** (1978) 175-221.

19. A. Habel & H.-J. Kreowski: Some structural aspects of hypergraph languages generated by hyperedge replacement, *in:* Proc STACS '87, Lect. Notes in Comp. Sci. **247** (1987) 207-219, Springer-Verlag, Berlin, Heidelberg, New York.

20. D. Janssens & G. Rozenberg: Hypergraph systems and their extensions, *R.A.I.R.O. Inform. Théor.* **17** (1983) 163-196.

21. M. Jazayeri, W.F. Ogden & W.C. Rounds: The intrinsically exponential complexity of the circularity problem for attribute grammars, *Comm. Assoc. Comp. Mach.* **18** (1975) 697-706.

22. D.E. Knuth: Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145; Correction *Math. Systems Theory* **5** (1971) 95-96.

23. H.-J. Kreowski: Rule trees represent derivations in edge replacement systems, *in:* G. Rozenberg & A. Salomaa (Eds.): *The Book of L* (1986), Springer-Verlag, Berlin, Heidelberg, New York.

24. U. Montanari & F. Rossi: An efficient algorithm for the solution of hierarchical networks of constraints (1987), University of Pisa, Italy.

25. T. Pavlidis: Linear and context-free graph grammars, *J. Assoc. Comp. Mach.* **19** (1972) 11-23.

Modular Tree Transducers

Heiko Vogler

Lehrstuhl für Informatik II, R.W.T.H. Aachen
Büchel 29-31, D-5100 Aachen, F.R.G.

In this article a new class of transducing devices, called modular tree transducers, is introduced and their relationship to (compositions of) macro tree transducers is studied. Modular tree transducers are term rewriting systems which define operations on trees in a structural recursive and modular way. The class of defined operations is closed under composition where the resulting transducers have in general more modules than the transducers started with. Modular tree transducers with one module correspond to macro tree transducers; however, every composition of macro tree transducers can be simulated by a modular tree transducer with just two modules. On the other hand "calling restricted" modular tree transducers characterize this composition in the sense that the number of modules corresponds to the number of composed macro tree transducers.

1. Introduction

In theoretical computer science one often is confronted with the task of defining operations on tree-structured objects. If these arise in practical applications, then frequently it is possible to specify them in a structural recursive way. Then the definition of such an operation f has the form of a case analysis on the (finitely many) different structures of the actual values of one *particular* argument position. For every structure t of this so-called recursion argument, an equation eq is entered into the case analysis; it specifies the result of f if the actual value of the recursion argument has the structure t. In general eq does not provide the final result immediately but only gives an approximation of it: besides the application of basic functions the right-hand side of eq may contain the operation f itself with the important restriction that f must be applied to one of the substructures of the recursion argument. Since we are interested in operations on trees only, we regard the basic functions as symbols with rank; thus the right-hand sides of equations are just trees.

Let us look at an example in which operations on binary trees are defined in a structural way. We only consider binary trees which either have the form of a "right-growing comb" or the form of a "left-growing comb", and in which the inner nodes are labeled by *cons*; cf. Figure 1(a) and (b). Note that the combs of Figure 1 may be viewed as representations of the lists $(a\ B\ C)$ and $((((\ )\ c)\ b)\ A)$, respectively, where a,b,c,A,B and C are atoms. (Clearly, the comb in

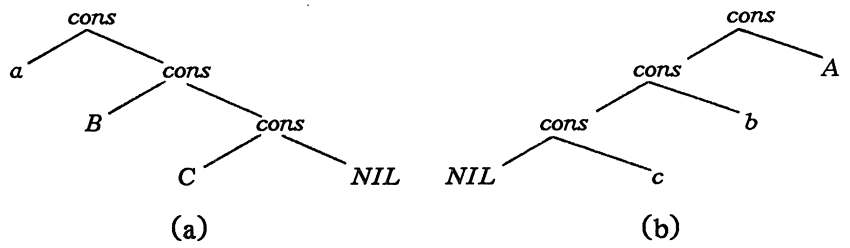

Figure 1. (a) right-growing comb with leaves a, B, C and NIL, (b) left-growing comb with leaves NIL, c, b and A.

Figure 1(b) is not the standard representation of the list $((((\,)\,c\,)\,b\,)\,A\,)$, but we have chosen this one because of technical convenience). Now we want to define the unary operation *mirror* which mirrors every right-growing comb at a vertical line and turns it into a left-growing comb; cf. Figure 2(a).

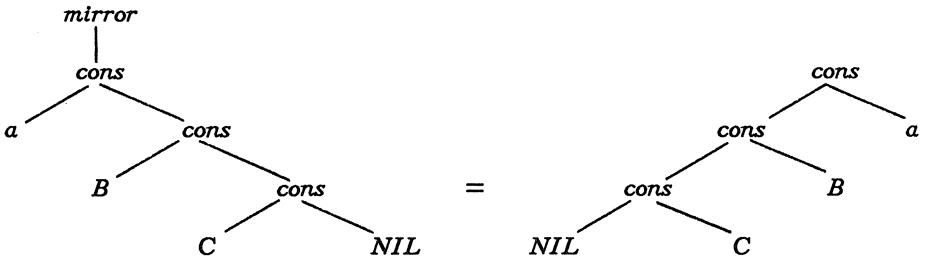

Figure 2. (a) Application of *mirror* to $cons\,(a,cons\,(B,cons\,(C,NIL\,)))$.

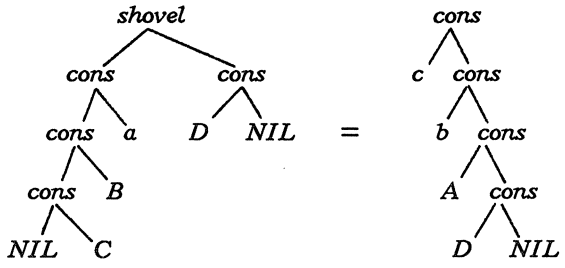

Figure 2. (b) Application of *shovel* to $cons\,(cons\,(cons\,(NIL,C\,),B\,),a\,)$ and $cons\,(D,NIL\,)$.

This partial operation can be specified by the equations

$$mirror\,(NIL\,) = NIL$$

$$mirror\,(cons\,(x_1,x_2)) = cons\,(mirror\,(x_2),x_1).$$

Here the two possible structures of the recursion argument are NIL and $cons\,(x_1,x_2)$, where x_1 and x_2 are variables that represent the sub-

structures of the recursion argument. Note that in the right-hand side *mirror* is applied to a substructure (viz. to x_2) of the actual value of the recursion argument. Note also that *cons* and *NIL* are both constructors for the values of the recursion argument and basic function symbols.

Another operation is *shovel* which has two arguments. As first argument it takes a left-growing comb, and its second argument is a right-growing comb. Now the operation shovels the leaves from its first argument onto its second argument, and simultaneously, it modifies them according to some table; cf. Figure 2(b). In our example a unary operation *table* replaces capital letters by the corresponding small letters and vice versa. The operation *shovel* can be defined by the following equations

$$shovel(NIL,y) = y$$

$$shovel(cons(x_1,x_2),y) = shovel(x_1,cons(table(x_2),y)).$$

The first argument of this operation is the recursion argument, and the second one serves as a kind of "accumulator". Note that the value of the accumulator depends on the output of the operation *table*; we say that *table* occurs nested in the accumulating parameter of *shovel*. But also observe that *shovel* is still defined in a structural recursive way. It is clear how to define *table*.

In this situation we want to design the unary operation *reverse* which takes as argument a right-growing comb and produces also a right-growing comb of the same height, but in the resulting tree the order of the leaves is reversed, and capital letters and small letters are interchanged; cf. Figure 3. One natural way of defining *reverse* would be to compose *mirror* and *shovel* as follows:

$$reverse(x) = shovel(mirror(x),NIL).$$

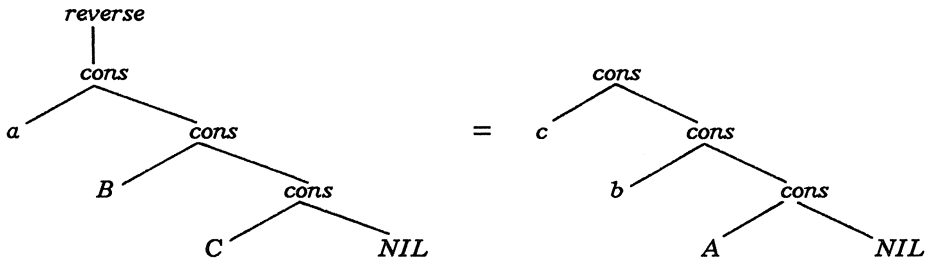

Figure 3. Application of *reverse* to $cons(a,cons(B,cons(C,NIL)))$.

Clearly, this equation does not obey any more the principle of a structural recursive definition: the recursion argument position of *shovel* in the right-hand side does *not* consist of a substructure of the recursion argument x of the left-hand side, but is computed by another opera-

tion. On the other hand, *reverse* is defined now in a very natural way, and hence, any design method for operations on trees should offer the feature of such a modular definition: the value of the recursion argument of one module is computed by another module.

Well, until now we discussed in an informal way the method of defining operations on trees and we did not give any syntax or semantics definition of the metalanguage in which the definitions are written down. What about *formal* metalanguages that also comprise computation models? Clearly, the simplest formalization of the structural recursive definition method are top-down tree transducers [12,13,3]. They allow the specification of unary operations by using simultaneous structural recursion. Equivalent concepts are generalized syntax-directed translation schemes [1] and attribute grammars with synthesized attributes only [11].

However, an operation like *shovel* cannot be specified by a top-down tree transducer, because the definition relies on the concept of accumulating parameter in which nested operations may occur. A formal model for the structural recursive definition method with the possibility of handling accumulating parameters is the macro tree transducer [4,2,6] (In [2] they are called primitive recursive schemes with parameters). Another formalization of this extended definition method are attribute grammars [11] which are slightly less powerful than macro tree transducers [5].

Are there also formalizations which reflect the feature of modularity? Clearly, one could just take lambda calculus and that is it. But we are interested in "the weakest" metalanguage that realizes our definition method. To answer the question under this aspect, let us examine what modularity means for tree transducers. Assume that there are transducers M_1 and M_2 that perform the operations *mirror* and *shovel*, respectively. Then we can define *reverse* as

$$comp(\tau(M_1),\tau(M_2),NIL)$$

where $\tau(M_i)$ denotes the operation induced by M_i, NIL is the unary operation that maps every argument to NIL, and $comp(f,g_1,g_2)$ denotes in an obvious way the composition of f with g_1 and g_2. Thus, if one wishes to specify operations in a modular way, then the class of operations induced by the used metalanguage should be closed under composition. And this requirement excludes macro tree transducers and also attribute grammars from the list of candidates: neither of them is closed under composition [5,6].

In this article we propose a formal computation model which realizes the method of defining operations on trees in a structural recursive *and* modular way: the modular tree transducer. Just as top-down tree transducers and macro tree transducers, modular tree transducers are (linear and non-overlapping) term rewriting systems in

which the operations and equations are realized by states and by rewrite rules, respectively. Actually, modular tree transducers are derived in a straightforward way from macro tree transducers by adding one more building rule for right-hand sides of rewrite rules: this new building rule reflects the modular principle. To every state q of a modular tree transducer a natural number is associated which is called the level of q. Intuitively, states of the same level together with the corresponding rewrite rules constitute one module. An n-modular tree transducer has n modules. Macro tree transducers as they have been defined in [6] are exactly the 1-modular tree transducers in which the initial state (i.e., the main operation) has one argument. Hence, macro tree transducers induce unary operations only.

Here we start the investigation of modular tree transducers by concentrating on two subjects: the closure under composition and the relation to compositions of macro tree transducers. This article consists of six sections. In Section 2 some general notations and notions are fixed. In Section 3 the basic model of modular tree transducer is introduced and an example is provided to illustrate the new device. In Section 4 the closure under composition is shown (Theorem 9), and it is proved that 2-modular tree transducers are more powerful than the composition closure of macro tree transducers (Theorem 11). In Section 5 "calling restricted" modular tree transducers are introduced which are also closed under composition (Theorem 15). The equivalence of "calling restricted" n-modular tree transducers and the n-fold composition of 1-modular tree transducers is proved inductively (Theorem 17). Finally, in Section 6 connections to other transducing devices are mentioned.

2. Preliminaries

We recall some notations and notions which will be used in this paper. In general some knowledge about special term rewriting systems such as top-down tree transducer (as presented in [3]) or macro tree transducer [4,6] would be helpful. Nevertheless, the paper is self-contained.

2.1. General notations

For every $n \geqslant 0$, the set $\{1,\ldots,n\}$ is abbreviated by $[n]$; hence, $[0]$ is the empty set. The end of definitions, lemmas, theorems etc. is indicated by $\square$. The elements in the sets $X = \{x_1, x_2, \cdots\}$ and $Y = \{y_1, y_2, \cdots\}$ are used as substitution variables of term rewriting systems. For $n \geqslant 0$, $X_n = \{x_1,\ldots,x_n\}$ and $Y_n = \{y_1,\ldots,y_n\}$. For two sets A and B, $A \subseteq B$ means that A is included in B; we use $A \subset B$ to denote strict inclusion.

For the substitution of strings into strings we use the following abbreviation. Let v be a string, let U and U' be arbitrary sets of strings, and let ϕ be a mapping from U into U'. If for every two

different elements u_1 and u_2 of U, u_1 and u_2 are not overlapping in v, then $v[u/\phi(u), u \in U]$ denotes the string obtained from v by replacing every occurrence of $u \in U$ by $\phi(u)$. If $U = \{u_1,...,u_n\}$, then we abbreviate this substitution by $v[u_i/\phi(u_i), i \in [n]]$.

2.2. Composition of relations

Let A be an arbitrary set. For $k \geqslant 1$, a $(k+1)$-ary relation R over A is a subset of A^{k+1}. A $(k+1)$-ary relation over A in which for every $a_1,...,a_k \in A$, there is at most one $b \in A$ such that $(a_1,...,a_k,b) \in R$ is called k-ary operation over A. Let R_0 be an $(r+1)$-ary relation over A for some $r \geqslant 1$, and for some $k \geqslant 1$, let R_i be a $(k+1)$-ary relation over A for every $i \in [r]$. The composition of R_0 with $R_1,...,R_r$, denoted by $comp(R_0, R_1,..., R_r)$, is the $(k+1)$-ary relation

$$\{(a_1,...,a_k,a) \mid \forall i \in [r]: \exists b_i: (a_1,...,a_k,b_i) \in R_i, \text{ and } (b_1,...,b_r,a) \in R_0\}.$$

Let REL_1 en REL_2 be two classes of relations over A. Then $COMP(REL_1, REL_2)$ denotes the class of relations $comp(R_0, R_1,..., R_r)$ for $R_0 \in REL_1$, and $R_1,..., R_r \in REL_2$ and appropriate r. If REL_1 and REL_2 are classes of binary relations, then $COMP(REL_1, REL_2)$ is also denoted by $REL_2 \circ REL_1$. For $n \geqslant 0$ and a class REL of relations, $COMP_n(REL)$ is the class of relations defined inductively as follows:

$$COMP_0(REL) = REL, \text{ and}$$
$$COMP_{n+1}(REL) = COMP(REL, COMP_n(REL)).$$

$COMP(REL)$ denotes the union of $COMP_n(REL)$ for every $n \geqslant 0$. If REL is a class of binary relations, then for every $n \geqslant 0$, $COMP_n(REL)$ is also denoted by REL^{n+1}.

2.3. Ranked alphabets and trees

A ranked alphabet Σ is a finite set in which to every symbol a unique number is associated, viz. its rank. The rank of a symbol is sometimes indicated as a superscript. E.g. $\sigma^{(2)}$ means that σ has rank 2.

Let Σ be a ranked alphabet. The set of (labeled) trees over Σ is denoted by T_Σ. A tree t in T_Σ is denoted by $\sigma(t_1,...,t_k)$ where the root of t is labeled by $\sigma^{(k)} \in \Sigma$ and $t_1,...,t_k$ are the immediate subtrees of t. If $k = 0$, then t is denoted by σ. The height of a tree is defined as usual inductively over the structure of the tree: (i) for $\sigma \in \Sigma$ of rank 0, $height(\sigma) = 1$, (ii) for $\sigma \in \Sigma$ of rank $k \geqslant 1$ and $t_1,...,t_k \in T_\Sigma$, $height(\sigma(t_1,...,t_k)) = 1 + \max\{height(t_i) \mid i \in [k]\}$. If Σ contains only symbols with rank 0 or 1, then trees over Σ are also denoted in the usual way as strings.

Let A be an arbitrary set. Then $T_\Sigma(A)$ denotes the set $T_{\Sigma \cup A}$ where the elements of A are viewed as symbols of rank 0. Any subset of T_Σ is called a tree language and the class of recognizable tree languages is denoted by $RECOG$.

3. Basic Model and Example

In this section we give the formal definition of the concept of modular tree transducer and of the class of operations on trees induced by them. The definition is illustrated by an example that describes the operation *reverse* on binary trees as discussed in the introduction.

Definition 1. Let $n \geqslant 1$. An *n-modular tree transducer* M is a tuple $((Q, level), \Sigma, q^{in}, R)$ where

- Q is the ranked alphabet of states (Every state has rank at least 1.) and $level : Q \to [n]$ is a mapping,
- Σ is the ranked alphabet of terminal (or input and output) symbols (Q and Σ are disjoint.),
- $q^{in} \in Q$ is the initial state with $level(q^{in}) = 1$,
- R is a finite set of productions of the form

$$q(\sigma(x_1,\ldots,x_m), y_1,\ldots,y_r) \to \zeta \qquad (*)$$

where $q \in Q$ with rank $r+1$ $(r \geqslant 0)$, $\sigma \in \Sigma$ with rank m $(m \geqslant 0)$, and some $\zeta \in RHS(Q, \Sigma, j, m, r)$ where $j = level(q)$ (Recall that $x_1,\ldots,x_m$ and $y_1,\ldots,y_r$ are substitution variables).

$RHS(Q, \Sigma, j, m, r)$ is the smallest subset RHS of $T_{Q \cup \Sigma}(X_m \cup Y_r)$ such that the following conditions are satisfied.
(i) Y_r is a subset of RHS,
(ii) if $\delta \in \Sigma$ with rank k $(k \geqslant 0)$ and $\zeta_1,\ldots,\zeta_k$ are elements of RHS, then $\delta(\zeta_1,\ldots,\zeta_k) \in RHS$,
(iii) if $p \in Q$ with rank $k+1$ $(k \geqslant 0)$ and $level(p) = j$, $x_i \in X_m$, and $\zeta_1,\ldots,\zeta_k \in RHS$, then $p(x_i, \zeta_1,\ldots,\zeta_k) \in RHS$,
(iv) if $p \in Q$ with rank k $(k \geqslant 1)$ and $level(p) > j$, and $\zeta_1,\ldots,\zeta_k \in RHS$, then $p(\zeta_1,\ldots,\zeta_k) \in RHS$. □

Remarks. (a) A rule like (*) is also called q-rule or, more specific, a (q,σ)-rule. (b) For every $j \leqslant n$, the set of q-rules with $level(q) = j$ form the module with level number j. (c) The first argument of a state is also called its recursion argument; the other arguments are referred to as accumulating parameters. □

A *modular tree transducer* is an n-modular tree transducer for some $n \geqslant 0$. Note that the previous definition only deals with the nondeterministic version of modular tree transducers. Before defining the total deterministic version, we insert a few easy examples of possible right-hand sides, because the inductive definition seems to be a bit involved.

Example 2. Let $Q = \{v^{(3)}, p^{(1)}, q^{(1)}, r^{(2)}, s^{(3)}, t^{(1)}\}$ be a set of states, where the superscripts indicate the ranks, and let $level(v) = level(p) = level(q) = level(r) = 1$ and $level(s) = level(t) = 2$. Let $\Sigma = \{a^{(0)}, b^{(0)}, \sigma^{(1)}, \delta^{(2)}, \gamma^{(3)}\}$. One possible left-hand side is
$v(\gamma(x_1, x_2, x_3), y_1, y_2)$ and the following trees are in $RHS(Q, \Sigma, 1, 3, 2)$:
1. $\delta(a, b)$,

2. $\delta(p(x_1),\sigma(q(x_2)))$,
3. $\delta(y_1,r(x_1,\sigma(q(x_2))))$,
4. $r(x_1,\sigma(s(t(y_2),a,q(x_1))))$. □

Definition 3. Let M be an n-modular tree transducer.
(1) M is *unary* if q^{in} has rank 1.
(2) M is *total deterministic* if for every $q \in Q$ and every $\sigma \in \Sigma$, there is exactly one (q,σ)-rule in R. □

Actually, macro tree transducers are precisely the unary 1-modular tree transducers, i.e., unary modular tree transducers with one module. Note that in 1-modular tree transducers, the building rule (iv) of the set of right-hand sides is never applicable, because there are no states with a level greater than 1. Indeed, rule (iv) mirrors the modular principle which is not realized by macro tree transducers. Top-down tree transducers [12,13] are macro tree transducers in which every state has rank 1. Thus, in Example 2, the terms 1-3 [terms 1 and 2] are possible right-hand sides of productions of macro tree transducers [of top-down tree transducers, respectively].

The translation induced by a modular tree transducer is defined by means of a derivation relation.

Definition 4. Let $M = ((Q,level),\Sigma,q^{in},R)$ be an n-modular tree transducer and let q^{in} have rank k for some $k \geqslant 1$.
(1) The *derivation relation* of M, denoted by $\Longrightarrow_M$, is the binary relation on $T_{Q\cup\Sigma}$ defined as follows. For $\xi_1,\xi_2 \in T_{Q\cup\Sigma}$, $\xi_1 \Longrightarrow_M \xi_2$ if and only if
there is a $\xi \in T_{Q\cup\Sigma}(\{z\})$ and z occurs exactly once in ξ,
there is a production $q(\sigma(x_1,\ldots,x_m),y_1,\ldots,y_r)\to\zeta$ in R,
there are $s_1,\ldots,s_m \in T_{Q\cup\Sigma}$ and $t_1,\ldots,t_r \in T_{Q\cup\Sigma}$ such that

$$\xi_1 = \xi[z/q(\sigma(s_1,\ldots,s_m),t_1,\ldots,t_r)], \text{ and}$$
$$\xi_2 = \xi[z/\zeta'] \text{ and } \zeta' = \zeta[x_i/s_i,\ i \in [m];\ y_j/t_j,\ j \in [r]].$$

(2) The *translation* induced by M, denoted by $\tau(M)$, is the $(k+1)$-ary relation $\{(s_1,\ldots,s_k,t) \in T_\Sigma^{k+1} \mid q^{in}(s_1,\ldots,s_k) \Longrightarrow_M^* t\}$ where as usual $\Longrightarrow_M^*$ denotes the reflexive and transitive closure of $\Longrightarrow_M$. □

Note that the rank of the initial state determines the arity of the induced translation. The class of translations induced by n-modular tree transducers is denoted by n-*ModT*; *ModT* denotes the union of the classes n-*ModT* for every $n \geqslant 1$. If the involved transducers are unary or total deterministic, then n-*ModT* is indexed by *un* or prefixed by D_t, respectively. E.g., $D_t\,n$-$ModT_{un}$ denotes the class of translations induced by total deterministic unary n-modular tree transducers. Let $D_t MT$ denote the class of translations induced by total deterministic macro tree transducers. Thus, D_t 1-$ModT_{un} = D_t MT$. Note that $D_t MT$ is a class of mappings (i.e., total functions) with one argument; cf. Section 3.3 of [6].

Claim 5. *The relations in D_t ModT are total operations.* □

Observation 6. *For every $n \geq 1$, D_t n-ModT $\subseteq$ D_t $(n+1)$-ModT.* □

We illustrate the above definitions by means of an example: the unary operation *reverse* as it was discussed in the introduction, is formulated as a modular tree transducer. The example shows that the present definition of modular tree transducer does not so much reflect the paradigms of a comfortable specification language. Rather it should serve as an appropriate starting point for theoretical studies and comparisons with other existing tree transducers. Nevertheless, the next example indicates how comfort can be gained without increasing the power of the underlying formalism.

Example 7. Here the unary operation *reverse* on binary trees is realized as a 2-modular tree transducer M. As repetition: *reverse* takes a right-growing comb as argument and produces a right-growing comb of the same height but with reversed order of leaves. Simultaneously, capital letters are turned to lower case and vice versa; cf. Figure 3 for an example of the application of *reverse*. As set of involved leaves we use $LEAVES = \{A,B,C,\ldots,a,b,c,\ldots\}$. From the introduction we first recall the equations that define the partial operations *mirror*, *shovel*, and *reverse*; cf. Figure 2 for an illustration of the meanings of the operations *mirror* and *shovel*.

SPEC1:

(1a) $mirror(NIL) = NIL$

(1b) $mirror(cons(x_1,x_2)) = cons(mirror(x_2),x_1)$

(2a) $shovel(NIL,y) = y$

(2b) $shovel(cons(x_1,x_2),y) = shovel(x_1,cons(table(x_2),y))$

(3) $reverse(x) = shovel(mirror(x),NIL)$

(4) $table(z) = z'$ for every $z \in LEAVES$
where z' is obtained from z by replacing capital letters by lower case letters and vice versa.

From this specification we develop the transducer M by eliminating step by step the "illegal" syntactic constructs. We only show the changes. The "free" occurrence of x_1 in the right-hand side of equation (1b) is not allowed in modular tree transducers. This is simulated by introducing an additional unary function *id* that just computes the identity, and by inserting *id* above x_1. Also some equations have to be added to realize *id*.

SPEC2:

(1b) $mirror(cons(x_1,x_2)) = cons(mirror(x_2),id(x_1))$

(5a) $id(z) = z$ for every $z \in LEAVES$

(5b) $id(NIL) = NIL$

(5c) $id(cons(x_1,x_2)) = cons(id(x_1),id(x_2))$.

In the next step equation (3) is compiled into two equations in order to get rid of "non-reading equations". For this purpose *mirror* is unfolded one step.

SPEC3:

(3a) $reverse(NIL) = NIL$
(3b) $reverse(cons(x_1,x_2)) = shovel(cons(mirror(x_2),id(x_1)),NIL)$.

Actually, SPEC3 can be turned immediately into the desired transducer $M = ((Q,level),\Sigma,reverse,R)$ as follows:

- $Q = \{reverse^{(1)},mirror^{(1)},shovel^{(2)},table^{(1)},id^{(1)}\}$ where the superscripts indicate the ranks,
  $level(reverse) = level(mirror) = level(id) = 1$ and
  $level(shovel) = level(table) = 2$,
- $\Sigma = LEAVES^{(0)} \cup \{cons^{(2)},NIL^{(0)}\}$,
- R contains the following rules

(1a) $mirror(NIL) \to NIL$
(1b) $mirror(cons(x_1,x_2)) \to cons(mirror(x_2),id(x_1))$

(2a) $shovel(NIL,y) \to y$
(2b) $shovel(cons(x_1,x_2),y) \to shovel(x_1,cons(table(x_2),y))$

(3a) $reverse(NIL) \to NIL$
(3b) $reverse(cons(x_1,x_2)) \to shovel(cons(mirror(x_2),id(x_1)),NIL)$

(4) $table(z) \to z'$ for every $z \in LEAVES$

(5a) $id(z) \to z$ for every $z \in LEAVES$
(5b) $id(NIL) \to NIL$
(5c) $id(cons(x_1,x_2)) \to cons(id(x_1),id(x_2))$.

This completes the construction of the modular tree transducer for *reverse*. Intuitively, the rules (3a), (3b), (1a), (1b) and (5a) form the module of level number 1, and the rules (2a), (2b) and (4) constitute the module with level number 2. Note that M is deterministic, but not total deterministic, because M only accepts input trees that have the form of right-growing combs. We finish this example by computing the application of *reverse* to the comb

$$t = cons(a,cons(B,cons(C,NIL))).$$

The numbers at the beginning of the lines indicate the applied rule.

$reverse(cons(a,cons(B,cons(C,NIL))))$
(3b) $\Rightarrow shovel(cons(mirror(cons(B,cons(C,NIL))),id(a)),NIL)$
(5a) $\Rightarrow shovel(cons(mirror(cons(B,cons(C,NIL))),a),NIL)$
(1b,5a) $\Rightarrow^2 shovel(cons(cons(mirror(cons(C,NIL)),B),a),NIL)$

(1b,5a) $\Rightarrow^2 shovel(cons(cons(cons(mirror(NIL),C),B),a),NIL)$
(1a) $\Rightarrow shovel(cons(cons(cons(NIL,C),B),a),NIL)$
(Note that at this point of the derivation, the first argument of *shovel* is the result of the application of *mirror* to t).
(2b,4) $\Rightarrow^2 shovel(cons(cons(NIL,C),B),cons(A,NIL))$
(2b,4) $\Rightarrow^2 shovel(cons(NIL,C),cons(b,cons(A,NIL)))$
(2b,4) $\Rightarrow^2 shovel(NIL,cons(c,cons(b,cons(A,NIL))))$
(2a) $\Rightarrow cons(c,cons(b,cons(A,NIL)))$. □

4. Composition of Modular Tree Transducers

In this section we prove the closure of total deterministic modular tree transducers under composition; cf. Theorem 9. By means of an example, we give an impression of the possible growth rate in the relationship between input and output trees of modular tree transducers (Example 10). Together Theorem 9 and Example 10 prove that (in the total deterministic case) 2-modular tree transducers are more powerful than the composition closure of macro tree transducers; cf. Theorem 11.

The proof of the composition closure of total deterministic modular tree transducers is prepared in the next lemma.

Lemma 8. *For every* $n,m \geqslant 1$,

$$COMP(D_t n\text{-}ModT, D_t m\text{-}ModT) \subseteq D_t max\text{-}ModT$$

where $max = \max\{n+1,m\}$. *In particular,*

$$D_t m\text{-}ModT_{un} \circ D_t n\text{-}ModT_{un} \subseteq D_t max\text{-}ModT_{un}.$$

Proof: Let $n,m \geqslant 1$ and let $\tau \in COMP(D_t n\text{-}ModT, D_t m\text{-}ModT)$ be a $(k+1)$-ary operation over T_Σ for some $k \geqslant 0$ and some ranked alphabet Σ. According to the definition of *COMP*, there is a total deterministic n-modular tree transducer $M_0 = ((Q_0, level_0), \Sigma, q^{in,0}, R_0)$ and $q^{in,0}$ has rank r for some $r \geqslant 1$, and for every $i \in [r]$, there is a total deterministic m-modular tree transducer $M_i = ((Q_i, level_i), \Sigma, q^{in,i}, R_i)$ such that $\tau = comp(\tau(M_0), \tau(M_1), \ldots, \tau(M_r))$ and $q^{in,i}$ has rank $k+1$. Without loss of generality we can assume that the involved sets of states are disjoint.

Construct the *max*-modular tree transducer $M = ((Q, level), \Sigma, q^{in}, R)$ with $max = \max\{n+1,m\}$ as follows.

- $Q = \bigcup \{Q_i \mid 0 \leqslant i \leqslant r\} \cup \{q^{in}\}$ and q^{in} has rank $k+1$ and $level(q^{in}) = 1$; for every $q \in Q_i$ with $i \geqslant 1$, $level(q) = level_i(q)$; for $q \in Q_0$, $level(q) = level_0(q)+1$ (Thus in particular, $level(q^{in,0}) = 2$).
- R contains $\bigcup \{R_i \mid 0 \leqslant i \leqslant r\}$ and, for every $\sigma \in \Sigma_j$ with $j \geqslant 0$, if for every $i \in [r]$ the rule $q^{in,i}(\sigma(x_1,\ldots,x_j), y_1,\ldots,y_k) \to \zeta_i$ is in R_i, then $q^{in}(\sigma(x_1,\ldots,x_j), y_1,\ldots,y_k) \to q^{in,0}(\zeta_1,\ldots,\zeta_r)$ is in R.

Note that M is total deterministic. We skip the formal proof of the correctness of the construction. □

Theorem 9. $COMP(D_t\,ModT) \subseteq D_t\,ModT$.

Proof: The statement of the theorem is an immediate consequence of: (*) for every $n \geqslant 0$, $COMP_n(D_t\,ModT) \subseteq D_t\,ModT$. The proof of (*) is an easy induction on n using Lemma 8 and Observation 6. □

The next example gives an impression of the possible growth rate in the relationship between input and output trees of 2-modular tree transducers. To be more precise, define the mapping $exp: \mathbb{N} \to \mathbb{N}$ ($\mathbb{N}$ is the set of non-negative integers.) inductively on the first argument: $exp(0,k) = k$ and $exp(n+1,k) = 2^r$ with $r = exp(n,k)$. Then define the unary mapping *super-exp* by $super\text{-}exp(k) = exp(k,1)$. We construct a 2-modular tree transducer for which the growth rate between input and output trees is described by the mapping *super-exp*.

Example 10. Let $\Sigma = \{\sigma^{(1)}, \alpha^{(0)}\}$ be a ranked alphabet; trees over Σ will be written in the obvious way as strings. The mapping $coding: \mathbb{N} \to T_\Sigma$ codes non-negative integers as monadic trees over Σ, i.e., $coding(k) = \sigma^k \alpha$. Then define

$$coding(super\text{-}exp) = \{(coding(k), coding(super\text{-}exp(k))) \mid k \geqslant 0\}.$$

We construct a total deterministic unary 2-modular tree transducer $M = ((Q, level), \Sigma, q, R)$ such that $\tau(M) = coding(super\text{-}exp)$.

- $Q = \{q^{(0)}, exp^{(2)}\}$ where superscripts indicate ranks, and $level(q) = 1$ and $level(exp) = 2$,
- R contains the following four rules

$$q(\sigma x) \to exp(q(x), \alpha)$$
$$q(\alpha) \to \sigma\alpha$$
$$exp(\sigma x, y) \to exp(x, exp(x, y))$$
$$exp(\alpha, y) \to \sigma y.$$

This completes the construction. It is easy to show that for every $k \geqslant 0$, $q(\sigma^k \alpha) \Longrightarrow^k exp(...exp(\sigma\alpha)..., \alpha)$ with k occurrences of exp and $k+1$ occurrences of α. Another easy induction yields the statement: for every $m \geqslant 0$, $exp(\sigma^m \alpha, y) \Longrightarrow^* \sigma^r y$ with $r = 2^m$. Together this proves that $\tau(M) = coding(super\text{-}exp)$. □

In [6] (see Theorem 3.24) it is shown that macro tree transducers can perform at most an exponential growth rate between input and output trees. More precisely, for every macro tree transducer M there is a constant c such that if $(s,t) \in \tau(M)$, then the height of t is bounded by $exp(1, c.height(s))$. Clearly, if (s,t) is an element of the translation induced by the n-fold composition of macro tree transducers, then $height(t) \leqslant exp(n, c'.height(s))$ for some constant c' (that depends on the involved transducers). An immediate consequence of this growth-rate property of macro tree transducers, the previous example, and Lemma 8 is the fact that $D_t\,2\text{-}ModT_{un}$ strictly includes the class of operations that are induced by the composition closure of macro tree transducers.

Theorem 11. $\bigcup \{D_t MT^n \mid n \geqslant 1\} \subset D_t 2\text{-}ModT_{un}$.

Proof: By Lemma 8, $D_t 2\text{-}ModT_{un} \circ D_t MT \subseteq D_t 2\text{-}ModT_{un}$. Hence, by induction, $\bigcup \{D_t MT^n \mid n \geqslant 1\} \subseteq D_t 2\text{-}ModT_{un}$. The strictness of this inclusion follows now immediately from Theorem 3.24 of [6] and from Example 10. □

5. Characterization of Compositions of Macro Tree Transducers

In this section we introduce "calling restricted" modular tree transducers and prove that [unary] "calling restricted" n-modular tree transducers are as powerful as the n-fold composition of 1-modular tree transducers [of macro tree transducers, respectively].

Clearly, in view of Theorem 11, it is necessary to restrict modular tree transducers if one wishes to decompose them into macro tree transducers. Let us motivate the nature of the used calling restriction at an example. Let M be a 6-modular tree transducer and let r be a q-rule of M where q is a state of M with $level(q) = 2$ and $rank(q) = 2$. Then it is possible that the right-hand side of r has the form $p(t(y_1,\ldots),\ldots)$ where p and t are states each with rank 2, and $level(p) = 3$. Now the important point is that $level(t)$ may range between $level(q)$ and 6; in particular, it may be *higher* than the level of p. Thus in general it is possible that the value of the recursion argument of a state with level number k can be computed by states with level number equal to or greater than k. Actually, this feature makes modular tree transducers more powerful than compositions of macro tree transducers. In fact, if the computation of the value of the recursion argument only calls states with level number *less* than k, then every so-obtained "calling restricted" modular tree transducer can be decomposed into 1-modular tree transducers; cf. Lemma 16. Actually, every module of a "calling restricted" modular tree transducer is transformed into one 1-modular tree transducer. In particular, every "calling restricted" unary n-modular tree transducer can be simulated by the composition of n macro tree transducers.

The calling restriction is realized by requiring the existence of a mapping *"call"* from states to the set of involved level numbers such that in particular, the following holds: for every right-hand side of a rule, if the state t occurs in the recursion argument of the state p, then $level(p)$ must be *greater* than $call(t)$.

Definition 12. Let $n \geqslant 1$ and let $M = ((Q, level), \Sigma, q^{in}, R)$ be an n-modular tree transducer. M is *calling restricted* if there is a mapping $call: Q \to [n]$ such that the following holds.
(a) For every $q \in Q$, $level(q) \leqslant call(q)$.
(b) If $q(\sigma(x_1,\ldots,x_m), y_1,\ldots,y_r) \to \zeta$ is a rule of M then
— for every state p occurring in ζ, $call(p) \leqslant call(q)$, and
— if $p(\zeta_1,\ldots,\zeta_k)$ is a subtree of ζ, then for every state t occurring in ζ_1, $call(t) < level(p)$. □

A calling restricted modular tree transducer is a calling restricted n-modular tree transducer for some $n \geq 1$. The class of translations induced by calling restricted modular tree transducers is denoted by $ModT_{cr}$. This denotation is modified in the obvious way for n-modular, unary, and total deterministic transducers; in particular, $D_t n\text{-}ModT_{cr,un}$ denotes the class of translations induced by unary total deterministic calling restricted n-modular tree transducers. Clearly, for 1-modular tree transducers, there is no difference between the unrestricted and the calling restricted version, i.e., $1\text{-}ModT = 1\text{-}ModT_{cr}$.

Observation 13. (a) $D_t\, 1\text{-}ModT_{cr} = D_t\, 1\text{-}ModT$.
(b) *For every* $n \geq 1$, $D_t n\text{-}ModT_{cr} \subseteq D_t (n+1)\text{-}ModT_{cr}$. □

Before decomposing calling restricted modular tree transducers, we first show that these transducers are closed under composition too; cf. Theorem 9 for the corresponding result of the unrestricted version. The following preparing lemma is similar to Lemma 8, but now an additivity relation holds between the maximal levels of the involved transducers.

Lemma 14. *For every* $n,m \geq 1$,

$$COMP(D_t n\text{-}ModT_{cr}, D_t m\text{-}ModT_{cr}) \subseteq D_t (m+n)\text{-}ModT_{cr}.$$

In particular,

$$D_t m\text{-}ModT_{cr,un} \circ D_t n\text{-}ModT_{cr,un} \subseteq D_t (m+n)\text{-}ModT_{cr,un}.$$

Proof: The involved construction is literally the same as in the proof of Lemma 8 except for one important point: the levels of the states of M_0 are not just incremented by 1, but they have to be incremented by m, i.e., for every $q \in Q_0$, $level(q) = level_0(q) + m$.

Then it is possible to define the calling function for the resulting transducer M. For $0 \leq i \leq r$, let $call_i$ be the calling function of transducer M_i. Define $call : Q \to [m+n]$ for M as follows:
- for every $q \in Q_i$ with $i \in [m]$, $call(q) = call_i(q)$,
- for every $q \in Q_0$, $call(q) = call_0(q) + m$,
- $call(q^{in}) = n+m$.

It is easy to verify that this mapping fulfills the requirements of Definition 12. □

Theorem 15. $COMP(D_t ModT_{cr}) \subseteq D_t ModT_{cr}$. □

Now we decompose calling restricted n-modular tree transducers into n calling restricted 1-modular tree transducers. The decomposition proceeds by induction. Consider an $(n+1)$-modular tree transducer M with terminal alphabet Σ. Intuitively, M is turned into an n-modular tree transducer M_1 by splitting up the module with level number $n+1$; that is, we consider in right-hand sides of productions every state q with level $n+1$ as a new terminal symbol; the q-rules are deleted. Thus M_1 computes trees over Σ and the new terminal

symbols. Now M_1 is composed with a macro tree transducer (i.e., unary 1-modular tree transducer) M_2 which realizes the module that has been split up from M. For this purpose, first M_2 "activates" every new terminal q by replacing it by the state q^* with level number 1, and second, M_2 evaluates these states by means of the rules of M that have a state with level number $n+1$ in their left-hand side.

Lemma 16. *Let* $n \geqslant 1$.
(a) $D_t(n+1)\text{-}ModT_{cr} \subseteq D_t n\text{-}ModT_{cr} \circ D_t MT$,
(b) $D_t(n+1)\text{-}ModT_{cr,un} \subseteq D_t n\text{-}ModT_{cr,un} \circ D_t MT$.

Proof: Let $M = ((Q, level), \Sigma, q^{in}, R)$ be a calling restricted total deterministic $(n+1)$-modular tree transducer and let *call* be the involved calling function. Let $rank(q^{in}) = r$ for some $r \geqslant 1$. Define $Q[n+1] = \{q \mid q \in Q \text{ and } level(q) = n+1\}$.

Construct the calling restricted total deterministic n-modular tree transducer $M_1 = ((Q_1, level_1), \Sigma_1, q^{in,1}, R_1)$ as follows.
- $Q_1 = Q - Q[n+1]$ and for every $q \in Q_1$, $level_1(q) = level(q)$, and if $call(q) \leqslant n$, then $call_1(q) = call(q)$, if $call(q) = n+1$, then $call_1(q) = n$,
- $q^{in,1} = q^{in}$,
- $\Sigma_1 = \Sigma \cup Q[n+1]$ and ranks are carried over from $Q[n+1]$ to Σ_1,
- R_1 contains all (q, σ)-rules of R for which $level(q) \in [n]$.

Construct the unary total deterministic 1-modular tree transducer $M_2 = ((Q_2, level_2), \Sigma_2, *, R_2)$ as follows.
- $Q_2 = \{*\} \cup \{q^* \mid q \in Q[n+1]\}$ and every state has level number 1, $rank(*) = 1$ and ranks carry over from $Q[n+1]$ to Q_2,
- $\Sigma_2 = \Sigma \cup Q[n+1]$ (ranks carry over),
- R_2 contains the following rules.

(a) For every $\delta^{(k)} \in \Sigma$ with $k \geqslant 0$,
$*(\delta(x_1, \ldots, x_k)) \to \delta(*(x_1), \ldots, *(x_k))$ is in R_2.
(b) For every $q \in Q[n+1]$ with $rank(q) = k+1$ for some $k \geqslant 0$,
$*(q(x_1, \ldots, x_{k+1})) \to q^*(x_1, *(x_2), \ldots, *(x_{k+1}))$ is in R_2.
(c) If $q(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_r) \to \zeta$ is in R with $level(q) \in [n+1]$, then $q^*(\sigma(x_1, \ldots, x_m), y_1, \ldots, y_r) \to \zeta^*$ is in R_2, where ζ^* is obtained from ζ by replacing every state p by p^*.

This completes the construction. Note that the calling restriction on m guarantees that every actual value of the recursion argument of a state q^* is already evaluated before the point at which q is activated and replaced by q^*. Intuitively, it is clear that $\tau(M) = \tau(M_1) \circ \tau(M_2)$. The formal proof of the correctness of the construction is left to the reader. □

Now the main theorem of this article follows immediately: the characterization of calling restricted n-modular tree transducers by the n-fold composition of 1-modular tree transducers.

Theorem 17. (a) *For* $n \geqslant 0$, $D_t(n+1)\text{-}ModT_{cr} = COMP_n(D_t\,1\text{-}ModT)$.
(b) *For every* $n \geqslant 1$, $D_t n\text{-}ModT_{cr,un} = D_t MT^n$.

Proof: The inclusions in (a) and (b) can be proved by an easy induction on n using Lemma 14 and Lemma 16. □

From the viewpoint of defining operations on trees, the previous theorem says the following: the concept of calling restricted modular tree transducer is the appropriate metalanguage to construct new operations from existing ones that have been specified by macro tree transducers. Actually, the 2-modular tree transducer of Example 7 that realizes the operation *reverse*, is calling restricted.

6. Conclusion

In this section we mention some relations to other tree transducing devices that have been studied in [14]. Section 8 of [8] contains a list of various classes of tree transducers which are equivalent with respect to their transformational power. All of them obey the concept of structural recursion, and some of them use an additional storage. Here we can add another equivalent class to this list. That is, for every $n \geqslant 1$, the total deterministic versions of the following transducers are equivalent:

- n-fold composition of macro tree transducers,
- n-iterated pushdown tree transducers; cf. Definition 4.10 of [8],
- n-level tree transducers; cf. Definition 4.5 of [8],
- unary calling restricted n-modular tree transducers.

The equivalence of the composition of macro tree transducers and iterated pushdown tree transducers is shown in Theorem 8.12 of [7]. In Theorem 7.12 of [8], high-level tree transducers are characterized by iterated pushdown tree transducers. In Theorem 17 of the present paper, it is proved that composition of macro tree transducers induce the same class of unary tree operations as unary calling restricted n-modular tree transducers.

Since the present paper just starts the investigation of modular tree transducers, some important questions remained open. In particular, we claim that $D_t ModT$ coincides precisely with the class of primitive recursive operations on trees as defined in [10].

Acknowledgments. Again I am grateful to Joost Engelfriet for patiently considering my "mental outputs" and improving them by valuable remarks. Moreover, I would like to thank Klaus Indermark for helpful discussions about structural recursion.

References

1. A.V. Aho & J.D. Ullman: Translations on a context-free grammar, *Inform. and Control* **19** (1971) 429-475.
2. B. Courcelle & P. Franchi-Zannettacci: Attribute grammars and recursive program schemes I, II, *Theoret. Comput. Sci.* **17** (1982)

163-191, 235-257.

3. J. Engelfriet: Bottom-up and top-down tree transformations — a comparison, *Math. Systems Theory* **9** (1975) 198-231.
4. J. Engelfriet: Some open questions and recent results on tree transducers and tree languages, *in* R.V. Book (Ed.): *Formal Language Theory: Perspectives and Open Problems* (1980), Academic Press, New York.
5. J. Engelfriet: Tree transducers and syntax-directed semantics, TW-Memorandum No. 363 (1981), Twente University of Technology, Enschede, The Netherlands.
6. J. Engelfriet & H. Vogler: Macro tree transducers, *J. Comput. System Sci.* **31** (1985) 71-146.
7. J. Engelfriet & H. Vogler: Pushdown machines for the macro tree transducer, *Theoret. Comput. Sci.* **42** (1986) 251-368.
8. J. Engelfriet & H. Vogler: High-level tree transducers and iterated pushdown machines, Technical Report 85-12 (1985), University of Leiden, The Netherlands.
9. P. Henderson: *Functional Programming — Application and Implementation* (1980), Prentice-Hall, Englewood Cliffs, N.J.
10. U.L. Hupbach: Rekursive Funktionen in mehrsortigen Peano-Algebren, *Elektron. Informationsverarb. Kybernet.* **14** (1978) 491-506.
11. D.E. Knuth: Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145, Correction, *Math. Systems Theory* **5** (1971) 95-96.
12. W.C. Rounds: Mappings and grammars on trees, *Math. Systems Theory* **4** (1970) 257-287.
13. J.W. Thatcher: Generalized^2 sequential machine maps, *J. Comput. System Sci.* **4** (1970) 339-367.
14. H. Vogler: *Tree Transducers and Pushdown Machines* (1986), Doctoral Dissertation, Twente University of Technology, Enschede, The Netherlands.

Nonterminal Separating Macro Grammars

Jan Anne Hogendorp*

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

We extend the concept of nonterminal separating (or NTS) context-free grammar to nonterminal separating m-macro grammar where the mode of derivation m is equal to "unrestricted", "outside-in" or "inside-out". Then we show some (partial) characterization results for these NTS m-macro grammars.

1. Introduction

Macro grammars have been introduced in [6,7] as a way to describe context-dependent aspects of the syntax of programming languages. They are an extension of context-free grammars generating, for each mode of derivation, a family of languages in between the families of context-free languages and of context-sensitive languages. Though outside-in (or OI-) macro languages are able to describe correctly the declaration and use of program variables, they have the disadvantage of possessing an NP-complete membership problem. For IO-macro languages the problem is roughly as complex as for context-free languages [1]; so it can be solved deterministically in polynomial time or in space $\log^2 n$. But IO-macro grammars seem to be less suitable for modeling the declaration of program variables.

Without considering this complexity issue any further we investigate in this paper a way to restrict macro grammars. It is inspired by a restriction on context-free grammars, viz. by the nonterminal separating (or NTS) condition [3]. For context-free grammars this restriction results in deterministic languages that have "disjunct syntactic categories" [3,5]. The actual NTS condition requires that adding the reductions corresponding to the productions of a grammar does not extend its set of sentential forms. Or, equivalently, the set of sentential forms does not change when we apply the rules of the grammar in both directions.

In Section 2 we provide the necessary notions, elementary results and terminology on macro grammars and on context-free grammars that satisfy the NTS condition. Section 3 is devoted to the definition of NTS macro grammar and some of their properties as far as they extend

* The work of the author has been supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

the corresponding results on NTS context-free grammars. We restrict our attention to characterization results of the NTS property for m-macro grammars where m is a mode of derivation, i.e., m equals either "outside-in" (or OI), "inside-out" (or IO) or "unrestricted" (or UNR). Finally, Section 4 contains some concluding remarks.

2. Preliminaries

2.1. Macro Grammars

Macro grammars have been introduced by Fischer in [6,7] as an extension of context-free grammars. In essence, they differ from context-free grammars in possessing a ranked alphabet of nonterminal symbols and so macro grammars are a particular kind of term rewriting system.

A *ranked alphabet* Δ is a finite set of symbols each of which is provided with a natural number, called its rank. For $i \geqslant 0$, let Δ_i denote the subalphabet of Δ that consists of all symbols of rank i. Thus if $i \neq j$, then $\Delta_i \cap \Delta_j = \emptyset$.

Definition 2.1.1. Let Δ be a ranked alphabet and PC the set of punctuation characters (i.e., left and right parenthesis and comma symbol). The *set* $T(\Delta)$ *of terms* over Δ is the smallest set of strings over $\Delta \cup$ PC that satisfies

(i) $\Delta_0 \cup \{\lambda\} \subseteq T(\Delta)$; λ denotes the empty word,

(ii) if $t_1, t_2 \in T(\Delta)$, then $t_1 t_2 \in T(\Delta)$,

(iii) if $A \in \Delta_n$ and $t_1, \ldots, t_n \in T(\Delta)$, then $A(t_1, \ldots, t_n) \in T(\Delta)$. □

Formally, we ought to write $A()$ if $A \in \Delta_0$; in practice we will omit the parentheses in that case. However, the notation $A(t_1, \ldots, t_n)$ does not imply that $n > 0$.

Definition 2.1.2. A *macro grammar* G is a 5-tuple $G = (\Phi, \Sigma, X, P, S)$ where Φ is a ranked alphabet of *nonterminals*, Σ is an alphabet of *terminals*, X is a finite set of *variables* (Each terminal and variable has rank zero. The sets Φ, Σ and X are disjoint.), $S \in \Phi_0$ is the *start symbol*, and P is a finite set of *productions* or *rules* of the form $A(x_1, \ldots, x_n) \rightarrow t$ with $A \in \Phi_n$, $x_1, \ldots, x_n$ are mutually distinct elements of X, and t is a term over $\Sigma \cup \Phi \cup \{x_1, \ldots, x_n\}$. □

Sentential forms of a macro grammar are terms over $\Sigma \cup \Phi$. Some specific subsets of terms give rise to interesting special types of macro grammars and corresponding sets of sentential forms. Viz. the set $BT(\Sigma \cup \Phi)$ of *basic terms* over $\Sigma \cup \Phi$ is the subset of $T(\Sigma \cup \Phi)$ of terms in which no $A \in \Phi$ appears in the argument list of another symbol of Φ (i.e., nonterminals are not nested). And the set $LBT(\Sigma \cup \Phi)$ of *linear basic terms* over $\Sigma \cup \Phi$ is the subset of $T(\Sigma \cup \Phi)$ of terms containing at most one nonterminal.

A production $A(x_1,...,x_n)\rightarrow t$ is called [*linear*] *basic* if t is a [linear] basic term. A macro grammar is [*linear*] *basic* if all its productions are [linear] basic. A production $A(x_1,...,x_n)\rightarrow t$ is called *argument preserving* if for each i $(1\leqslant i\leqslant n)$, t contains at least one occurrence of x_i, and it is called *non-duplicating* if t contains at most one occurrence of x_i for each i $(1\leqslant i\leqslant n)$.

In order to describe several modes of derivation for macro grammars we need the following concepts.

Definition 2.1.3. Let σ be a term over $\Sigma\cup\Phi$. τ is a *subterm* of σ if τ is a term over $\Sigma\cup\Phi$ and τ is a substring of σ.

A subterm τ of σ occurs at *top level* in σ if there exist subterms σ_1 and σ_2 such that $\sigma=\sigma_1\tau\sigma_2$. So τ does not appear within the argument list of some nonterminal in σ.

A term over $\Sigma\cup\Phi$ is called *expanded* if it contains no nonterminals together with its associated argument list, or equivalently, if it is a string over Σ. □

Using the productions of a macro grammar one can expand terms. As usual we distinguish three modes of derivation.
Unrestricted mode (*UNR*): An occurrence of a nonterminal together with its arguments can be expanded according to a production by replacing the nonterminal and its arguments by the right-hand side of that production in which the arguments have been substituted for the corresponding variables.
Inside-Out (*IO*): A nonterminal with its arguments is expanded only if its arguments are all expanded terms.
Outside-In (*OI*): A nonterminal with its arguments is expanded only if it occurs at top level.

Each of these modes of derivation gives rise to a derivation relation, formally defined as follows.

Definition 2.1.4. Let $G=(\Phi,\Sigma,X,P,S)$ be a macro grammer and let $\sigma,\tau\in T(\Sigma\cup\Phi)$. The relations $\Rightarrow_{UNR}$, $\Rightarrow_{IO}$ and $\Rightarrow_{OI}$ over $T(\Sigma\cup\Phi)$ are defined by
(1) $\sigma\Rightarrow_{UNR}\tau$ holds if σ contains a subterm of the form $A(t_1,...,t_n)$ where $A\in\Phi_n$ and $t_1,...,t_n\in T(\Sigma\cup\Phi)$, P contains a production $A(x_1,...,x_n)\rightarrow t$ and τ results from σ by substituting $A(t_1,...,t_n)$ by $t[t_1/x_1,...,t_n/x_n]$.
(2) $\sigma\Rightarrow_{IO}\tau$ holds in case $\sigma\Rightarrow_{UNR}\tau$ and all the arguments of the rewritten nonterminal are expanded terms.
(3) $\sigma\Rightarrow_{OI}\tau$ holds in case $\sigma\Rightarrow_{UNR}\tau$ and the subterm of σ which is rewritten occurs at top level in σ. □

Let $\Leftarrow_m$ be the converse of $\Rightarrow_m$, i.e., for all $\sigma,\tau\in T(\Sigma\cup\Phi)$, $\sigma\Leftarrow_m\tau$ holds if and only if $\tau\Rightarrow_m\sigma$. And let $\Leftrightarrow_m$ be the union of $\Rightarrow_m$ and $\Leftarrow_m$. The reflexive and transitive closures of $\Rightarrow_m$, $\Leftarrow_m$ and $\Leftrightarrow_m$ are denoted by $\Rightarrow_m^*$, $\Leftarrow_m^*$ and

$\Longleftrightarrow_m^*$, respectively. In case $\sigma \Longleftarrow_m^* \tau$ [$\sigma \Longleftarrow_m \tau$] we say that σ *reduces* [*directly*] *to* τ.

It is easy to see that $\Longleftrightarrow_m^*$ is a congruence relation. Obviously, it is an equivalence relation and the congruency follows from: $\sigma \Longleftrightarrow_m^* \tau$ and $\alpha \Longleftrightarrow_m^* \beta$ imply $\sigma\alpha \Longleftrightarrow_m^* \tau\beta$; for $m = UNR$ this is trivial and in the other cases it follows from the fact that concatenation does not cause any additional nesting.

Definition 2.1.5. Let G be a macro grammar and m a mode of derivation. An *m-macro grammar* is a pair (G,m), or simply denoted by G when m is known from the context. The *language* generated by an m-macro grammar $G = (\Phi,\Sigma,X,P,S)$ is defined by

$$L_m(G) = \{w \in \Sigma^* \mid S \Longrightarrow_m^* w\}.$$

By OI, IO and UNR we denote the family of languages generated by *OI*-, *IO*- and *UNR*-macro grammars, respectively. □

In [6] Fischer proved the equality OI = UNR, and the fact that IO and OI are incomparable.

In the sequel many of our results are restricted to macro grammars which possess the property that every term derived by the macro grammar has a derivation that ultimately yields a string over the terminal alphabet. These macro grammars are called admissible macro grammars [6]. This property is defined as follows.

Definition 2.1.6. A m-macro grammar $G = (\Phi,\Sigma,X,P,Z)$ with $Z \subseteq \Phi_0$ is *admissible* if either $\Phi = Z$ and $P = \emptyset$ or
(1) for each $A \in \Phi$, there exists a sentential form of G in which A occurs,
(2) for each $A \in \Phi_n$ $(n \geqslant 0)$ and each $\sigma_1,\ldots,\sigma_n \in \Sigma^*$ there exists a string w over Σ such that $A(\sigma_1,\ldots,\sigma_n) \Longrightarrow_m^* w$. □

In [6] it is shown that for each m-macro grammar there exists an equivalent admissible m-macro grammar. For $m = IO$ every (G,m) has an equivalent admissible subgrammar; for $m = OI$ the task to find such an admissible grammar is more elaborate.

Example 2.1.7. Let $L_0 \subseteq \{0,1\}^*$ the language containing exactly those words in which the number of 1's is equal to 2^n for some $n \geqslant 0$. L_0 is generated by the *OI*-macro grammar $G = (\Phi,\Sigma,X,P,S)$ with $\Phi = \Phi_0 \cup \Phi_1$, $\Phi_0 = \{S,A\}$, $\Phi_1 = \{B\}$, $X = \{x\}$, $\Sigma = \{0,1\}$ and P consists of the rules

$$\begin{aligned} S &\rightarrow B(A) \\ B(x) &\rightarrow B(xx) \mid x \\ A &\rightarrow 0A \mid A0 \mid 1 \end{aligned}$$

In [6] it has been shown that L_0 cannot be generated by any *IO*-macro grammar. □

2.2. The NTS Property for Context-Free Grammars

NTS or nonterminal separating grammars have been introduced by Boasson [3]. A context-free grammar possesses the NTS property if its set of sentential forms is invariant when we apply the rules in both directions, i.e., when we use apart from its productions the corresponding reductions too.

Let $G = (V,\Sigma,P,Z)$ be a context-free grammar with alphabet V, terminal alphabet Σ ($\Sigma \subseteq V$), set of productions P, and start set Z ($Z \subseteq V-\Sigma$). For each $\omega \in V^*$ we denote the set of words over Σ derivable from ω by G as

$$L(G,\omega) = \{w \in \Sigma^* | \omega \Longrightarrow^* w\}.$$

We call this set the language generated by G from ω. The language generated by G is

$$L(G) = \{w \in \Sigma^* | \exists S \in Z : S \Longrightarrow^* w\}.$$

The set of sentential forms generated by G from $\omega \in V^*$ is

$$\underline{L}(G,\omega) = \{\psi \in V^* | \omega \Longrightarrow^* \psi\}.$$

The relations $\Longleftarrow$, $\Longleftarrow^*$, $\Longleftrightarrow$ and $\Longleftrightarrow^*$ are defined in a way similar to §2.1; however, historically they were first defined for context-free grammars [3].

The set of words over V derivable from $\omega \in V^*$ by both productions and the corresponding reductions is

$$\underline{LR}(G,\omega) = \{\psi \in V^* | \omega \Longleftrightarrow^* \psi\}.$$

Definition 2.2.1. A context-free grammar $G = (V,\Sigma,P,Z)$ has the *NTS property* or is an *NTS grammar* if for all $A \in V-\Sigma$, $\underline{LR}(G,A) = \underline{L}(G,A)$. A language L is called an *NTS language* if there exists an NTS grammar that generates L. □

Proposition 2.2.2. [3,5]. *Let $G = (V,\Sigma,P,Z)$ be an NTS grammar. Then for all A and B in $V-\Sigma$, either $\underline{L}(G,A) \cap \underline{L}(G,B) = \emptyset$ or $\underline{L}(G,A) = \underline{L}(G,B)$ holds.* □

This property motivates the name of the concept defined in 2.2.1. However, the converse of 2.2.2 does not hold; e.g. $\{a^n b^n | n \geqslant 1\} \cup \{a^n b^{2n} | n \geqslant 1\}$ is not an NTS language [5], but it is easy to show that this language can be generated by a grammar that possesses "disjunct syntactic categories".

On the other hand NTS grammars can be characterized in the following way.

Theorem 2.2.3. [5,10]. *Let $G = (V,\Sigma,P,Z)$ be a context-free grammar. G has the NTS property if and only if for all $A,B \in V-\Sigma$ and for all $\alpha,\beta,u \in V^*$ the following implication holds:*

$$\text{if } A \Longrightarrow^* \alpha u \beta \text{ and } B \Longrightarrow^* u, \text{ then } A \Longrightarrow^* \alpha B \beta.$$

□

For further details of context-free NTS grammars and languages the reader is referred to [2,3,5,8,9,10].

3. The NTS Property for Macro Grammars

3.1. Definitions

We use the following notational conventions. Usually, $(\sigma_1,\ldots,\sigma_n)$ is abbreviated to $(\vec{\sigma}_{(n)})$. The subscript (n) is necessary to distinguish for example $A(\vec{x}_{(n)})$ and $B(\vec{x}_{(k)})$. Only if no confusion is possible we write $\vec{x}$. For $A \in \Phi$, $A(\vec{x})$ is the left-hand side of a production; so $A(\vec{x}) = A$ if $A \in \Phi_0$. In the sequel an m-macro grammar will have a finite set Z $(Z \subseteq \Phi_0)$ of initial symbols of rank 0 instead of a single initial symbol; cf. the definition of NTS context-free grammar.

Definition 3.1.1. Let $G = (\Phi,\Sigma,X,P,Z)$ be an m-macro grammar. Then the *language* generated by (G,m) is

$$L_m(G) = \{w \in \Sigma^* \mid \exists\, S \in Z : S \Longrightarrow_m^* w\},$$

and for each $t \in T(\Sigma \cup X \cup \Phi)$,

$$L_m(G,t) = \{w \in (\Sigma \cup X)^* \mid t \Longrightarrow_m^* w\},$$
$$\underline{L}_m(G,t) = \{\omega \in T(\Sigma \cup X \cup \Phi) \mid t \Longrightarrow_m^* \omega\},$$
$$\underline{LR}_m(G,t) = \{\omega \in T(\Sigma \cup X \cup \Phi) \mid t \Longleftrightarrow_m^* \omega\}.$$ □

We are now ready to define the nonterminal separating property for m-macro grammars.

Definition 3.1.2. An m-macro grammar $G = (\Phi,\Sigma,X,P,Z)$ has the *NTS property* or is an *NTS m-macro grammar* if for all $n \geqslant 0$, $A \in \Phi_n$, $\{x_1,\ldots,x_n\} \subseteq X$,

$$\underline{LR}_m(G,A(\vec{x})) = \underline{L}_m(G,A(\vec{x})).$$ □

Here we consider the variables $x_1,\ldots,x_n$ as members of a terminal alphabet Σ' with $\Sigma \subseteq \Sigma'$, according to Fischer [6]; cf. also [4].

Proposition 3.1.3. *Let $G = (\Phi,\Sigma,X,P,Z)$ be an NTS m-macro grammar. Then for all $n,k \geqslant 0$, $A \in \Phi_n$, $B \in \Phi_k$, $\{x_1,\ldots,x_n\} \subseteq X$, $\{x_1,\ldots,x_k\} \subseteq X$,*

$$\underline{L}_m(G,A(\vec{x}_{(n)})) \cap \underline{L}_m(G,B(\vec{x}_{(k)})) = \varnothing$$

or

$$\underline{L}_m(G,A(\vec{x}_{(n)})) = \underline{L}_m(G,B(\vec{x}_{(k)})).$$

Proof: Let ω be an element of $\underline{L}_m(G,A(\vec{x}_{(n)})) \cap \underline{L}_m(G,B(\vec{x}_{(k)}))$. Then $A(\vec{x}_{(n)}) \Longrightarrow_m^* \omega$ as well as $B(\vec{x}_{(k)}) \Longrightarrow_m^* \omega$ holds. This implies $A(\vec{x}_{(n)}) \Longleftrightarrow_m^* B(\vec{x}_{(k)})$. With the NTS property of G we get $A(\vec{x}_{(n)}) \Longrightarrow_m^* B(\vec{x}_{(k)})$ and $B(\vec{x}_{(k)}) \Longrightarrow_m^* A(\vec{x}_{(n)})$ which implies $\underline{L}_m(G,A(\vec{x}_{(n)})) = \underline{L}_m(G,B(\vec{x}_{(k)}))$. □

We see that NTS m-macro grammars have a similar "nonterminal separating property" as context-free grammars; cf. Proposition 2.2.2.

Example 3.1.4. Consider the linear basic macro grammar $G = (\Phi,\Sigma,X,P,Z)$ with $\Phi = \Phi_0 \cup \Phi_3$, $\Phi_0 = \{S\} = Z$, $\Phi_3 = \{A\}$, $X = \{x,y,z\}$,

$\Sigma = \{a,b,c,[,],\#\}$, and P consists of the productions

$$S \to A(\lambda,\lambda,\lambda)$$
$$A(x,y,z) \to A(ax,by,cz)$$
$$A(x,y,z) \to [x\#y\#z]$$

The language generated by G is $L(G) = \{[a^n \#b^n \#c^n] | n \geqslant 0\}$, and $\underline{L}(G,S) = \{S\} \cup \{A(a^n,b^n,c^n) | n \geqslant 0\} \cup L(G)$. Because $A(a^n,b^n,c^n)$, $(n \geqslant 1)$ only reduces to terms $A(a^k,b^k,c^k)$ with $0 \leqslant k < n$, and $[a^n \#b^n \#c^n]$ only reduces to $A(a^n,b^n,c^n)$, we have $\underline{L}(G,S) = \underline{LR}(G,S)$. A similar argument for $A(x,y,z)$ yields $\underline{L}(G,A(x,y,z)) = \underline{LR}(G,A(x,y,z))$; so G is an NTS macro grammar. □

We see also that in case $\Phi = \Phi_0$ and, consequently, G is a context-free grammar, Definition 3.1.2 corresponds to Definition 2.2.1 for context-free grammars.

3.2. Properties of NTS Macro Grammars

This section is devoted to some results which generalize Theorem 2.2.3 to m-macro grammars. To facilitate formulation and proofs we use the following notation.

Definition 3.2.1. Let $G = (\Phi,\Sigma,X,P,Z)$ be an m-macro grammar. Then G has *property* $\Pi(m)$ if for all $A \in \Phi_n$, $B \in \Phi_k$, $u,\alpha u \beta \in T(\Sigma \cup X \cup \Phi)$, with $\{x_1,\ldots,x_n\} \subseteq X$ and $\vec{\sigma}_{(k)} \in T^k(\Sigma \cup X \cup \Phi)$ the following implication holds

$$\text{if } A(\vec{x}_{(n)}) \Longrightarrow_m^* \alpha u \beta \text{ and } B(\vec{\sigma}_{(k)}) \Longrightarrow_m^* u,$$
$$\text{then } A(\vec{x}_{(n)}) \Longrightarrow_m^* \alpha B(\vec{\sigma}_{(k)})\beta.$$ □

First, we note that property $\Pi(m)$ is a natural extension of the property mentioned in Theorem 2.2.3 in the sense that if $\Phi = \Phi_0$, i.e., G is context-free, the two properties coincide. To establish Theorem 3.2.3 we need the following lemma.

Lemma 3.2.2. *Let G be an admissible m-macro grammar. Let $\omega,\psi \in T(\Sigma \cup X \cup \Phi)$. Then $\omega \Longrightarrow_{UNR} \psi$ implies $\omega \Longleftrightarrow_{OI}^* \psi$ as well as $\omega \Longleftrightarrow_{IO}^* \psi$. As a corollary we have $\omega \Longrightarrow_{UNR}^* \psi$ implies $\omega \Longleftrightarrow_m^* \psi$ for both $m = OI$ and IO.*

Proof: Let $\omega = \alpha A(\vec{\sigma})\beta$ with $A \in \Phi_n$, $n \geqslant 0$, $\vec{\sigma} \in T^n(\Sigma \cup X \cup \Phi)$ and $\psi = \alpha\delta(\vec{\sigma})\beta$. Then $\omega \Longrightarrow_{UNR} \psi$ using the rule $A(\vec{x}) \to \delta(\vec{x})$, $\delta(\vec{x}) \in T(\Sigma \cup X \cup \Phi)$.

$m = OI$. First we have $\alpha A(\vec{\sigma})\beta \Longrightarrow_{OI}^* \alpha' A(\vec{\sigma})\beta'$. This is the string obtained from ω such that every $A(\vec{\sigma})$ is on top level. Next we derive $\alpha' A(\vec{\sigma})\beta' \Longrightarrow_{OI}^* \alpha'\delta(\vec{\sigma})\beta'$. Now all new occurrences of $\delta(\vec{\sigma})$ are on top level; so we can write $\alpha'\delta(\vec{\sigma})\beta' \Longleftarrow_{OI}^* \alpha\delta(\vec{\sigma})\beta$.

$m = IO$. Similarly, using $A(\vec{\sigma}) \Longrightarrow_{IO}^* A(\vec{t})$, $A(\vec{t}) \Longrightarrow_{IO}^* \delta(\vec{t})$ and $\delta(\vec{t}) \Longleftarrow_{IO}^* \delta(\vec{\sigma})$, where $\vec{t} \in (\Sigma^*)^n$. □

Theorem 3.2.3. *Let G be an admissible m-macro grammar. Then (G,m) is an NTS m-macro grammar if and only if G has property $\Pi(m)$.*

Proof: First we prove the *if*-part. We have to show for G satisfying $\Pi(m)$ that for each $A \in \Phi_n$ $(n \geqslant 0)$,

$$\underline{L}_m(G, A(\vec{x})) = \underline{LR}_m(G, A(\vec{x})).$$

The inclusion from left to right $(\subseteq)$ is trivial. To establish the converse inclusion $(\supseteq)$, we ought to prove that $A(\vec{x}) \Longleftrightarrow_m^* t$ implies $A(\vec{x}) \Longrightarrow_m^* t$. This is done by induction on the length of $\Longleftrightarrow_m^*$.
Basic step $(p = 0)$: $A(\vec{x}) \Longleftrightarrow_m^0 t$ implies $A(\vec{x}) \Longrightarrow_m^* t$ trivially.
Induction step. As induction hypothesis we take: $A(\vec{x}) \Longleftrightarrow_m^p t$ implies $A(\vec{x}) \Longrightarrow_m^* t$.
Consider $A(\vec{x}) \Longleftrightarrow_m^{p+1} t$. We distinguish two cases:

Case 1. $A(\vec{x}) \Longleftrightarrow_m^p t' \Longrightarrow_m t$. Obvious.

Case 2. $A(\vec{x}_{(n)}) \Longleftrightarrow_m^p t' \Longleftarrow_m t$. Suppose $t \Longrightarrow_m t'$ by the derivation step $B(\vec{\sigma}_{(k)}) \Longrightarrow_m u$ and let $t = \alpha B(\vec{\sigma}_{(k)})\beta$, $t' = \alpha u \beta$ with $\alpha u \beta$, u, $B(\vec{\sigma}_{(k)}) \in T(\Sigma \cup X \cup \Phi)$. By the induction hypothesis we have $A(\vec{x}_{(n)}) \Longrightarrow_m^* t'$. Using $\Pi(m)$ on $A(\vec{x}_{(n)}) \Longrightarrow_m^* \alpha u \beta$ and $B(\vec{\sigma}_{(k)}) \Longrightarrow_m u$ we get $A(\vec{x}_{(n)}) \Longrightarrow_m^* \alpha B(\vec{\sigma}_{(k)})\beta = t$. This completes the induction and the proof of the second inclusion.

To prove the *only if*-part we need the following. Let G be an NTS m-macro grammar. Then for all $u, \alpha u \beta \in T(\Sigma \cup X \cup \Phi)$, $B \in \Phi_k$, $\vec{\sigma}_{(k)} \in T^k(\Sigma \cup X \cup \Phi)$,

$$B(\vec{\sigma}_{(k)}) \Longrightarrow_m^* u \text{ implies } \alpha B(\vec{\sigma}_{(k)})\beta \Longleftrightarrow_m^* \alpha u \beta.$$

It is easy to see that for $m = IO$ and $m = UNR$ this holds even without G being NTS and with $\Longrightarrow_m^*$ instead of $\Longleftrightarrow_m^*$. For $m = OI$ we obtain this implication as follows. If $B(\vec{\sigma}_{(k)}) \Longrightarrow_{OI}^* u$, then $B(\vec{\sigma}_{(k)}) \Longrightarrow_{UNR}^* u$ trivially; so $\alpha B(\vec{\sigma}_{(k)})\beta \Longrightarrow_{UNR}^* \alpha u \beta$ and by Lemma 3.2.2. we have $\alpha B(\vec{\sigma}_{(k)})\beta \Longleftrightarrow_{OI}^* \alpha u \beta$. (Note that because G is NTS, we now can even prove the stronger fact: $B(\vec{\sigma}_{(k)}) \Longrightarrow_{OI}^* u$ implies $\alpha B(\vec{\sigma}_{(k)})\beta \Longrightarrow_{OI}^* \alpha u \beta$).

Now, if $A(\vec{x}_{(n)}) \Longrightarrow_m^* \alpha u \beta$ and $B(\vec{\sigma}_{(k)}) \Longrightarrow_m^* u$, then we get $A(\vec{x}_{(n)}) \Longleftrightarrow_m^* \alpha B(\vec{\sigma}_{(k)})\beta$. Since (G, m) is NTS, we conclude with $A(\vec{x}_{(n)}) \Longrightarrow_m^* \alpha B(\vec{\sigma}_{(k)})\beta$. □

3.3. The Pre-NTS Property for Macro Grammars

Closely connected to the NTS property for context-free grammars is the pre-NTS property [3,5,9]; informally, the pre-NTS property equals the NTS property formulated for terminal strings only. It is still an open problem whether these two properties are equivalent for context-free grammars [3,5,9].

In this section we introduce and study the pre-NTS property for m-macro grammars.

Definition 3.3.1. Let $G = (\Phi, \Sigma, X, P, Z)$ be an m-macro grammar with $Z \subseteq \Phi_0$. Then (G, m) is *pre-NTS* or *has the pre-NTS property* if for all $A \in \Phi_n$ $(n \geqslant 0)$, and $\{x_1, \ldots, x_n\} \subseteq X$, $L_m(G, A(\vec{x})) = LR_m(G, A(\vec{x}))$ where $LR_m(G, A(\vec{x})) = \underline{LR}_m(G, A(\vec{x})) \cap (\Sigma \cup X)^*$. □

Definition 3.3.2. Let $G = (\Phi, \Sigma, X, P, Z)$ be an m-macro grammar with $Z \subseteq \Phi_0$. Then G has *property* $\pi(m)$ if for all $A \in \Phi_n$ $(n \geqslant 0)$, $B \in \Phi_k$, $u', \alpha u \beta \in (\Sigma \cup X)^*$, $\{x_1, \ldots, x_n\} \subseteq X$, and $\vec{\tau} \in T^k(\Sigma \cup X \cup \Phi)$, the following implication holds:

$$\text{if } A(\vec{x}) \Rightarrow_m^* \alpha u \beta,\ B(\vec{\tau}) \Rightarrow_m^* u \text{ and } B(\vec{\tau}) \Rightarrow_m^* u',$$
$$\text{then } A(\vec{x}) \Rightarrow_m^* \alpha u' \beta. \qquad \square$$

We want to prove the equivalence of Definition 3.3.1 and Definition 3.3.2. It turns out to be the easiest way to do this by introducing a second property $\rho(m)$ which is equivalent to both of them.

Definition 3.3.3. An m-macro grammar G has property $\rho(m)$ if for all $A \in \Phi_n$ $(n \geqslant 0)$, and $\{x_1, \ldots, x_n\} \subseteq X$, $t \in T(\Sigma \cup X \cup \Phi)$, $u, u' \in (\Sigma \cup X)^*$ the following implication holds:

$$\text{if } A(\vec{x}) \Rightarrow_m^* u,\ t \Rightarrow_m^* u, \text{ and } t \Rightarrow_m^* u', \text{ then } A(\vec{x}) \Rightarrow_m^* u'. \qquad \square$$

Theorem 3.3.4. *Let G be an admissible m-macro grammar. Then the following statements are equivalent:*

(1) *(G, m) is pre-NTS,*
(2) *G has property $\pi(m)$,*
(3) *G has property $\rho(m)$.*

Proof: (1) $\Rightarrow$ (2): Suppose there exist derivations $B(\vec{\tau}) \Rightarrow_m^* u$, $B(\vec{\tau}) \Rightarrow_m^* u'$ and $A(\vec{x}) \Rightarrow_m^* \alpha u \beta$ for u', $\alpha u \beta \in (\Sigma \cup X)^*$. Because $\alpha u \beta$ is a word over $\Sigma \cup X$ there is no distinction between the three modes of reduction from $\alpha u \beta$. Therefore we have $A(\vec{x}) \Rightarrow_m^* \alpha u \beta \Leftarrow_m^* \alpha B(\vec{\tau}) \beta$. Now in $\alpha B(\vec{\tau}) \beta$, $B(\vec{\tau})$ is on top level, so we continue with $\alpha B(\vec{\tau}) \beta \Rightarrow_m^* \alpha u' \beta$ which is a word over $\Sigma \cup X$. Thus $A(\vec{x}) \Leftrightarrow_m^* \alpha u' \beta$ and, as (G, m) is pre-NTS, $A(\vec{x}) \Rightarrow_m^* \alpha u' \beta$. Hence G has property $\pi(m)$.

(2) $\Rightarrow$ (3): Let $A(\vec{x}) \Rightarrow_m^* u$, $t \Rightarrow_m^* u$ and $t \Rightarrow_m^* u'$. Obviously, it is possible to write t as an unique sequence of terms, viz. $t = t_1 \ldots t_k$, such that no t_i is a concatenation of two or more terms. It is clear that in expanding some t_i, none of the other terms t_j is affected. So we can write u as $u_1 \ldots u_k$ and u' as $u_1' \ldots u_k'$ with $t_i \Rightarrow_m^* u_i$ and $t_i \Rightarrow_m^* u_i'$, respectively. Now we have for some i, $1 \leqslant i \leqslant k$ $A(\vec{x}) \Rightarrow_m^* u_1 \ldots u_i \ldots u_k$, $t_i \Rightarrow_m^* u_i$, $t_i \Rightarrow_m^* u_i'$, and with $\pi(m)$ we get $A(\vec{x}) \Rightarrow_m^* u_1 \ldots u_i' \ldots u_k$. We apply this argument to each u_i consecutively, which finally yields $A(\vec{x}) \Rightarrow_m^* u_1' \ldots u_k' = u'$ which is the desired result.

(3) $\Rightarrow$ (1): We have to show $LR_m(G, A(\vec{x})) \subseteq L_m(G, A(\vec{x}))$, which we do by induction on the number of reduction steps in $A(\vec{x}) \Leftrightarrow_m^* w$, with $w \in (\Sigma \cup X)^*$. We denote this by $\Leftrightarrow_m^{*n}$ which means that $\alpha \Leftrightarrow_m^{*n} \beta$ holds if and only if $\alpha \Leftrightarrow_m^* \beta$ in which n reduction steps have been used.

Basic step $(n = 0)$. $A(\vec{x}) \Leftrightarrow_m^{*0} w$ directly implies $A(\vec{x}) \Rightarrow_m^* w$.

Induction step. As induction hypothesis we have: $A(\vec{x}) \Leftrightarrow_m^{*n} w$ implies $A(\vec{x}) \Rightarrow_m^* w$. Let $A(\vec{x}) \Leftrightarrow_m^{*n+1} w$. To show that

$A(\vec{x}) \Rightarrow_m^* w$ we look at the last reduction step in $A(\vec{x}) \Leftrightarrow_m^{*n+1} w$. We write this as $A(\vec{x}) \Leftrightarrow_m^{*n} t \Leftarrow_m t' \Rightarrow_m^* w$. Because G is admissible there is a word $u \in (\Sigma \cup X)^*$ with $t \Rightarrow_m^* u$. Applying the induction hypothesis we get $A(\vec{x}) \Rightarrow_m^* u$, with $t' \Rightarrow_m^* u$, and $t' \Rightarrow_m^* w$ and property $\rho(m)$ this gives us $A(\vec{x}) \Rightarrow_m^* w$. □

4. Concluding Remarks

In the previous section we generalized some characterizations of NTS and pre-NTS context-free grammars to corresponding statements for (pre-) NTS m-macro grammars. On the other hand one wants results that are specific for NTS macro grammars in the sense that there is no analogue for context-free grammars. Or, in other words, results that are due to the fact that we deal with macro grammars rather than context-free grammars.

A first example of such a result shows that NTS "reduced macro grammars", i.e., admissible NTS macro grammars with no initial symbols in the right-hand sides of their productions, are argument-preserving.

Proposition 4.1. *Let $G = (\Phi, \Sigma, X, P, Z)$ be an admissible NTS m-macro grammar, with no elements of Z occurring in the right-hand side of any production. Then G is argument-preserving.*

Proof: Suppose we have a production rule $A(x_1,\ldots,x_n) \to t$ with $A \notin \Phi_0$, which is not argument-preserving, say x_i does not occur in t, $1 \leqslant i \leqslant n$. Suppose further that we have obtained a word $\omega \in T(\Sigma \cup \Phi)$ derived from some $S \in Z$ on which this rule is applicable. Writing ω as $\alpha A(\sigma_1,\ldots,\sigma_n)\beta$ we derive

$$\alpha t[\sigma_1/x_1,\ldots,\sigma_{i-1}/x_{i-1},\sigma_{i+1}/x_{i+1},\ldots,\sigma_n/x_n]\beta.$$

This last term however is, for instance, for some T in Z reducible to $\alpha A(\sigma_1,\ldots,\sigma_{i-1},T,\sigma_{i+1},\ldots,\sigma_n)\beta$, which we write as $\omega(T)$. So we have $S \Leftrightarrow_m^* \omega(T)$. Since G is NTS, we obtain $S \Rightarrow_m^* \omega(T)$. But no production rule can ever introduce a T from Z in a sentential form. Thus we cannot derive such a term $\omega(T)$ from S. □

The following statement is much more interesting. However, we are unable to prove it and therefore we formulate it as

Conjecture 4.2. *Each admissible NTS IO-macro grammar generates a basic macro language.* □

The first easy step in proving this conjecture, consists of the following observation.

Lemma 4.3. *Let G be an admissible NTS IO-macro grammar. Then for all $A \in \Phi$,*

$$\underline{L}_{UNR}(G, A(\vec{x})) = \underline{L}_{IO}(G, A(\vec{x})).$$

Proof: We only have to show $\underline{L}_{UNR}(G, A(\vec{x})) \subseteq \underline{L}_{IO}(G, A(\vec{x}))$, since the converse inclusion is trivial. Let $t \in T(\Sigma \cup X \cup \Phi)$ and $A(\vec{x}) \Rightarrow_{UNR}^* t$. Then we have by Lemma 3.2.2 $A(\vec{x}) \Leftrightarrow_{IO}^* t$, and

using the fact that (G,IO) is NTS, we obtain $A(\vec{x}) \Longrightarrow_{IO}^{*} t$. □

In order to complete the proof of Conjecture 4.2 it is sufficient to establish

Conjecture 4.4. *Let G be an NTS IO-macro grammar that contains a nested production*

$$A(\vec{x}) \to B(\vec{y}(\vec{x})) \qquad (*)$$

i.e., some entry of $\vec{y}$ contains a nonterminal symbol. If $\beta(\vec{x}) \in L_{UNR}(G, B(\vec{x}))$, then in the derivation $A(\vec{x}) \Longrightarrow_{IO}^{*} \beta(\vec{y}(\vec{x}))$ the rule $(*)$ has not been applied. □

Acknowledgment. I thank Peter Asveld for his helpful comments and for his aid during the preparation of the text.

References

1. P.R.J. Asveld: Time and space complexity of inside-out macro languages, *Internat. J. Comput. Math.* **10** (1981) 3-14.
2. J.M. Autebert, L. Boasson & G. Sénizergues: Langages de parenthèses, langages N.T.S. et homomorphismes inverses, *R.A.I.R.O. Inform. théor./Theor. Inform.* **18** (1984) 327-344.
3. L. Boasson: Dérivations et réductions dans les grammaires algébriques, *in* "7th International Colloquium on Automata Languages and Programming" Lect. Notes Comp. Sci. **85** (1980) 109-118, Springer-Verlag, Berlin - Heidelberg - New York.
4. J. Engelfriet, E.M. Schmidt & J. van Leeuwen: Stack machines and classes of non-nested macro languages, *J. Assoc. Comput. Mach.* 27 (1980) 96-117.
5. L. Boasson & G. Sénizergues: NTS languages are deterministic and congruential, *J. Comput. System Sci.* **31** (1985) 332-342.
6. M.J. Fischer: *Grammars with Macro-like Productions*, Ph.D. Thesis (1968), Harvard University, Cambridge, Mass.
7. M.J. Fischer: Grammars with macro-like productions, Proc. 9th Ann. IEEE Symp. on Switching and Automata Theory (1968) 131-142.
8. Ch. Frougny: Simple deterministic NTS languages, *Inform. Process. Lett.* **12** (1981) 174-178.
9. G. Sénizergues: A new class of C.F.L. for which the equivalence is decidable, *Inform. Process. Lett.* **13** (1981) 30-34.
10. G. Sénizergues: The equivalence and inclusion problems for NTS languages, *J. Comput. System Sci.* **31** (1985) 303-331.

Complexity Aspects of Iterated Rewriting - A Survey -

Peter R.J. Asveld

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

We present an overview of results on the complexity of the membership problem for families of languages generated by several types of generalized grammars. In particular, we consider generalized grammars based on iterated context-independent rewriting, i.e., grammars consisting of a finite number of (non)deterministic substitutions, and on iterated context-dependent rewriting, i.e., grammars composed of a finite number of transductions. We give some conditions on the classes of these substitutions and transductions that guarantee the solvability of this membership problem within certain time and space bounds. As consequences we obtain additional closure properties of some time- and space-bounded complexity classes.

1. Introduction

The concept of iteration grammar has been introduced as a generalization of a particular kind of parallel rewriting system, viz. ETOL-system [18], in order to extend some ad hoc combinatorial arguments to more general, structural proof techniques. For the origins and the early history of iteration grammars we refer to [2] and the references mentioned there.

In essence an iteration grammar is an ETOL-system in which the finite substitutions or tables have been replaced by arbitrary substitutions. So each table in an iteration grammar contains for each symbol a countable rather than a finite number of productions. Instead of ordinary substitutions one can also use deterministic substitutions which yields a generalization of EDTOL-systems, the so-called deterministic iteration grammars [6,7]. A deterministic substitution differs from an ordinary or nondeterministic substitution in the way it is applied to a string: instead of making a possibly different choice for each occurrence of the same symbol in a string we make a single choice in advance and then substitute this choice consistently for each occurrence of the symbol. Another different way to generalize EDTOL- or ETOL-systems consists of replacing the tables by transductions which yields an abstract context-dependent grammar model [5]. This latter approach extends some earlier generalizations in [9,22,16,4,17]. The major part of the research on these general grammar models is concerned with extending known results from L-system theory in the hope to obtain

general, algebraic or structural arguments rather than the combinatorial proofs usually applied in the original finite case.

In this paper we survey some results that have been obtained in this way with respect to the complexity of the membership problem for these types of grammars, i.e., given such a grammar G and a string x over the terminal alphabet of G, how much time and space does it take to decide whether $x \in L(G)$? As usual in this area, the answer highly depends on the properties of the underlying set of (non)deterministic substitutions and transductions. Or, in other words, making some appropriately chosen assumptions on the set of substitutions or transductions enables us to extend results from L-system theory to these abstract grammars. As a spin-off it becomes clear on which properties of the finite languages the original proof in L-system theory actually depends.

The remaining part of this paper is organized as follows. Some notions of formal language and complexity theory are recalled in Section 2. In Section 3 we consider controlled (non)deterministic iteration grammars and their membership problem; Section 4 is devoted to the complexity of this problem. Context-dependent grammars, i.e., grammars based on transductions are described in Section 5, where we also provide some conditions that imply the decidability of the corresponding membership problem. In Section 6 attention is focused on the complexity of this membership problem. Sections 4 and 6 also provide some interesting closure properties of some complexity classes. Finally, Section 7 contains some open problems and concluding remarks.

2. Preliminaries

We assume the reader to be familiar with the rudiments of formal language theory and of automaton-based complexity theory. For all unexplained concepts and notation we refer to standard texts like [10,11,18]. For each set X, $\mathbf{P}(X)$ denotes the power set of X. The empty word is denoted by λ.

Let K be a family of languages and let V be an alphabet. A (*nondeterministic*) *K-substitution* over V or n*K-substitution* is a mapping $\tau: V \to K \cap \mathbf{P}(V^*)$ extended to words over V by $\tau(\lambda) = \{\lambda\}$, $\tau(xy) = \tau(x)\tau(y)$ for each $x, y \in V^*$, and to languages L over V by $\tau(L) = \bigcup \{\tau(x) \mid x \in L\}$. Similarly, a *deterministic K-substitution* over V or d*K-substitution* is also a mapping $\tau: V \to K \cap \mathbf{P}(V^*)$ but now it is extended to words by $\tau(x) = \{h(x) \mid h$ is a homomorphism such that $h(\alpha) \in \tau(\alpha)$ for each α in $V\}$, and to languages L over V by $\tau(L) = \bigcup \{\tau(x) \mid x \in L\}$. Notice that applying a dK-substitution τ implies that each occurrence of a symbol α in a string ought to be substituted by the very same word from $\tau(\alpha)$. A dK- or nK-substitution τ over V is called λ-*free* if $\lambda \notin \tau(\alpha)$ for each α in V.

An *Abstract Family of Languages* or *AFL* is a family of languages different from $\{\emptyset\}$, which is closed under union, concatenation, Kleene +, λ-free homomorphism, inverse homomorphism and intersection with regular languages.

Frequently, we will tacitly assume that families of languages are closed under isomorphism ("renaming of symbols").

Let $g: \mathbb{N} \to \mathbb{N}$ be a monotonic function, i.e., $m \leqslant n$ implies $g(m) \leqslant g(n)$. For each such g, DSPACE(g) [NSPACE(g)] is the family of languages accepted by [non]deterministic two-way multi-tape Turing machines that use no more than $g(n)$ tape cells on any storage tape during a computation on an input of length n. And DTIME(g) [NTIME(g)] denotes the family of languages accepted by [non]-deterministic two-way multi-tape Turing machines that finish their [accepting] computations within time $g(n)$ on all inputs of length n. We use the following well-known abbreviations:

$$\mathbf{PSPACE} = \bigcup \{\mathrm{DSPACE}(n^d) \mid d \geqslant 1\} = \bigcup \{\mathrm{NSPACE}(n^d) \mid d \geqslant 1\},$$

$$\mathbf{P} = \bigcup \{\mathrm{DTIME}(n^d) \mid d \geqslant 1\}, \qquad \mathbf{NP} = \bigcup \{\mathrm{NTIME}(n^d) \mid d \geqslant 1\}.$$

A [λ-*free*] *nondeterministic generalized sequential machine with accepting states* or *NGSM* [λ*NGSM*] $\tau = (Q, \Delta_1, \Delta_2, \delta, q_0, Q_F)$ consists of a set of states Q with initial state q_0 ($q_0 \in Q$), a set Q_F of final states ($Q_F \subseteq Q$), an input alphabet Δ_1, an output alphabet Δ_2, and a function δ from $Q \times \Delta_1$ into the finite subsets of $Q \times \Delta_2^*$ [$Q \times \Delta_2^+$]. The function δ is extended from $Q \times \Delta_1^*$ into the finite subsets of $Q \times \Delta_2^*$ by

(i) $\delta(q, \lambda) = \{(q, \lambda)\}$,

(ii) $\delta(q, x\alpha) = \{(q', y) \mid y = y_1 y_2$ and for some $p \in Q$, $(p, y_1) \in \delta(q, x)$, and $(q', y_2) \in \delta(p, \alpha)\}$, where $q \in Q$, $\alpha \in \Delta_1$, $x \in \Delta_1^*$.

Each [λ-free] NGSM τ induces a transduction $\tau: \mathbf{P}(\Delta_1^*) \to \mathbf{P}(\Delta_2^*)$, called [$\lambda$-*free*] *NGSM mapping*, defined by $\tau(x) = \{y \mid (q, y) \in \delta(q_0, x)$ for some $q \in Q_F\}$ for each x in Δ_1^*, and $\tau(L) = \bigcup \{\tau(x) \mid x \in L\}$ for each language L over Δ_1.

A NGSM [λNGSM] τ is called *deterministic* or *DGSM* [λ*DGSM*] if δ is a function from $Q \times \Delta_1$ into $Q \times \Delta_2^*$ [$Q \times \Delta_2^+$]. By NGSM [λNGSM] we also denote the family of [λ-free] NGSM mappings, and similarly we use DGSM [λDGSM] in the deterministic case.

3. Iterated Context-Independent Rewriting

Iteration grammars have already been discussed in an informal way in Section 1. Now we recall the formal definition together with the controlled variant which clearly generalizes the concepts of controlled EDTOL- and ETOL-system studied in [1,2,6,7,8,13,14].

Definition 3.1. Let Γ and K be language families. A [*non*]*deterministic* K*-iteration grammar* or dK*-iteration* [nK*-iteration*] grammar

$G = (V,\Sigma,U,S)$ consists of an alphabet V, a terminal alphabet Σ ($\Sigma \subseteq V$), an initial symbol S ($S \in V$), and a finite set U of [non]deterministic K-substitutions over V. The language $L(G)$ generated by G is defined by $L(G) = U^*(S) \cap \Sigma^*$. A *$\Gamma$-controlled dK-iteration* [*nK-iteration*] *grammar* $(G;C) = (V,\Sigma,U,S,C)$ consists of a dK-iteration [nK-iteration] grammar (V,Σ,U,S) together with a *control language* $C \subseteq U^*$ with $C \in \Gamma$. The language generated by $(G;C)$ is defined by

$$L(G;C) = C(S) \cap \Sigma^* = (\bigcup \{\tau_p(...(\tau_1(S))...) | \tau_1...\tau_p \in C\}) \cap \Sigma^*.$$

The family of languages generated by dK-iteration [nK-iteration] grammars is denoted by $\eta(K)$ [$H(K)$]. And $\eta(\Gamma,K)$ [$H(\Gamma,K)$] denotes the family of languages generated by Γ-controlled dK-iteration [nK-iteration] grammars. A (Γ-controlled) iteration grammar is called *λ-free* when each of its substitutions is λ-free. □

In derivations of controlled (λ-free) iteration grammars we can use long control words to derive relatively short terminal strings. In this way we are able to simulate an erasing homomorphism. Basicly, this is the core in proving the following characterization of RE, the family of recursively enumerable languages.

Proposition 3.2. [1,2,6]. *If K and Γ are language families such that* $\{L \mid \text{card}(L) = 1\} \subseteq K \subseteq \text{RE}$, *and* $\{h(L) \mid L \in \Gamma$; *h is an arbitrary homomorphism* $\} = \text{RE}$, *then* $\eta(\Gamma,K) = H(\Gamma,K) = \text{RE}$. □

Thus in order to obtain recursive languages (of which we want to determine the complexity) at all, we ought to restrict the families K and Γ in some way. The conditions in the following lemma provide a first step to such a restriction.

Lemma 3.3. [3]. *Let K be a family which contains all alphabets.*
(1) *Let Γ be a family closed under finite substitution and intersection with regular languages, and let $(G;C) = (V,\Sigma,U,S,C)$ be a λ-free Γ-controlled dK-iteration* [*nK-iteration*] *grammar. Then there exists a λ-free Γ-controlled dK-iteration* [*nK-iteration*] *grammar $(G';C') = (V',\Sigma,U',S,C')$ such that $L(G';C') = L(G;C)$, and for each x in $L(G';C')$ there is a control word $\tau_1...\tau_p$ in C' such that $x \in \tau_p...\tau_1(S)$ and $p \leqslant 2|x|$.*
(2) *For each λ-free dK-iteration* [*nK-iteration*] *grammar $G = (V,\Sigma,U,S)$ there exists a λ-free dK-iteration* [*nK-iteration*] *grammar $G' = (V',\Sigma,U',S)$ such that $L(G') = L(G)$, and for each x in $L(G')$ there is a string $\tau_1...\tau_p$ over U' such that $x \in \tau_p...\tau_1(S)$ and $p \leqslant 2|x|$.* □

The proof of this lemma can be found in [3] and will not be repeated here; it extends an earlier result on controlled ETOL-systems from [8]. In Section 5 we meet a similar situation and a corresponding lemma of which the proof will be sketched.

Proposition 3.4. [5]. *Let Γ and K satisfy the assumptions of Lemma 3.3. If both Γ and K are λ-free subfamilies of the family of recursive languages, then each language in $H(\Gamma,K)$, $H(K)$, $\eta(\Gamma,K)$ and in*

$\eta(K)$ *is recursive.*

```
read x
if x ∉Σ+ then reject else
        for all u in C' with |u| ≤2|x| do
                if x ∈u(S) then accept fi
        od;
        reject
fi.
```

Figure 1.

Proof: Given a λ-free Γ-controlled [non]deterministic K-iteration grammar $(G;C)$, we first apply Lemma 3.3. Then the algorithm in Figure 1 determines whether a word x belongs to $L(G';C')$. Since after the execution of an **accept**- or **reject**-statement the algorithm is supposed to halt, termination is guaranteed for each input x. Hence $L(G';C')$ is recursive. In the uncontrolled case we replace "**for all** u in C'" by "**for all** u in U'^*" in Figure 1. □

Other conditions that guarantee the decidability of the membership problem for (controlled) λ-free iteration grammars can be found in [1,2].

4. Complexity Aspects of Iterated Context-Independent Rewriting

With respect to the complexity of the membership problem for (controlled) deterministic iteration grammars the following result is of principal interest. Its proof is too long to be given here; it consists of a straightforward generalization of the argument that EDTOL is included in NSPACE $(\log n)$ [12]; cf. [3] for details.

Let 1-NSPACE (g) be the family of languages accepted within space bound $g: \mathbb{N} \to \mathbb{N}$ by nondeterministic multi-tape Turing machines with a one-way read-only input tape.

Theorem 4.1. [3]. *Let* $g(n) \geqslant \log n$ *for all* $n \in \mathbb{N}$, *let* Γ *and* K *be families satisfying the assumptions of Lemma 3.3, and let* Γ *be closed under reversal.*
(1) *If* Γ *and* K *are included in* 1-NSPACE (g), *with* $g(2n) \leqslant c.g(n)$ *for some constant* c *and for all* $n \in \mathbb{N}$, *then* $\eta(\Gamma, K) \subseteq$ NSPACE (g).
(2) *If* $K \subseteq$ 1-NSPACE (g), *then* $\eta(K) \subseteq$ NSPACE (g). □

Since for functions $g: \mathbb{N} \to \mathbb{N}$ with $g(n) \geqslant n$ for all $n \in \mathbb{N}$, the family 1-NSPACE (g) equals NSPACE (g), this implies immediately

Theorem 4.2. [3]. *Let* $g(n) \geqslant n$ *for all* $n \in \mathbb{N}$, *and let* Γ *and* K *be families which satisfy the assumptions of Lemma 3.3.*
(1) *If* $g(2n) \leqslant c.g(n)$ *for some constant* c *and for all* $n \in \mathbb{N}$, *and if both* Γ *and* K *are included in* NSPACE (g), *then* $\eta(\Gamma, K) \subseteq$ NSPACE (g).
(2) *If* $K \subseteq$ NSPACE (g), *then* $\eta(K) \subseteq$ NSPACE (g). □

A similar results holds for $H(\Gamma, K)$ and $H(K)$; it has been mentioned implicitly in [20] and formulated in [3] in the following way.

Theorem 4.3. [20,3]. *Let $g(n) \geqslant n$ for all $n \in \mathbb{N}$, and let Γ and K be families which satisfy the conditions of Lemma 3.3.*
(1) *If $g(2n) \leqslant c.g(n)$ for some constant c and for all $n \in \mathbb{N}$, and if both Γ and K are included in* NSPACE(g), *then* $H(\Gamma, K) \subseteq$ NSPACE(g).
(2) $K \subseteq$ NSPACE(g) *implies* $H(K) \subseteq$ NSPACE(g). □

An inclusion analogous to 4.2(2) and 4.3(2) for DSPACE(g) originates from [20]; the proof is based on a divide-and-conquer technique to determine a derivation. The obvious extension to controlled iteration grammars is from [3].

Theorem 4.4. [20,3]. *Let $g(n) \geqslant n \log n$ for all $n \in \mathbb{N}$, and let Γ and K be families which satisfy the assumptions of Lemma 3.3.*
(1) *If $g(2n) \leqslant c.g(n)$ for some constant c and for all $n \in \mathbb{N}$, and if both Γ and K are included in* DSPACE(g), *then* $H(\Gamma, K) \subseteq$ DSPACE(g).
(2) *If* $K \subseteq$ DSPACE(g), *then* $H(K) \subseteq$ DSPACE(g). □

Van Leeuwen's proof of 4.4 can also applied to (controlled) deterministic iteration grammars; as observed in [3] the bound $g(n) \geqslant n \log n$ can in that case be replaced by $g(n) \geqslant n$ for all $n \in \mathbb{N}$.

Theorem 4.5. [3]. *Let $g(n) \geqslant n$ for all $n \in \mathbb{N}$, and let Γ and K be families which satisfy the conditions of Lemma 3.3.*
(1) *If $g(2n) \leqslant c.g(n)$ for some constant c and for all $n \in \mathbb{N}$, and if both Γ and K are included in* DSPACE(g), *then* $\eta(\Gamma, K) \subseteq$ DSPACE(g).
(2) $K \subseteq$ DSPACE(g) *implies* $\eta(K) \subseteq$ DSPACE(g). □

Taking K equal to DSPACE(g) or NSPACE(g) in 4.2(2) - 4.5(2) yields some interesting closure properties for these complexity classes. A language family K is closed under *iterated λ-free [non]deterministic substitution* if for each K-language L over V and each finite set U of λ-free [non]deterministic K-substitutions over V the language $U^*(L)$ belongs to K.

Theorem 4.6. [3]. *Let for all $n \in \mathbb{N}$, $g(n) \geqslant n$ and $g(2n) \leqslant c.g(n)$ for some constant c. Then* NSPACE(g) *and* DSPACE(g) *are AFL's closed under intersection and iterated λ-free deterministic substitution. Moreover,* NSPACE(g) *is closed under iterated λ-free nondeterministic substitution; this also applies to* DSPACE(g) *provided that $g(n) \geqslant n \log n$ for all $n \in \mathbb{N}$.* □

Corollary 4.7. [3]. *The following language families are AFL's closed under intersection and under iterated λ-free deterministic substitution:*

(1) **PSPACE**,

(2) NSPACE(n), *the family of nondeterministic context-sensitive languages,*

(3) DSPACE(n), *the family of deterministic context-sensitive languages,*

(4) NSPACE(n^2), *the family of two-way nondeterministic nonerasing stack automaton languages,*

(5) DSPACE($n \log n$), *the family of two-way deterministic nonerasing stack automaton languages.*

Moreover, the families under (1), (2), (4) *and* (5) *are also closed under iterated* λ*-free nondeterministic substitution.* □

We call a language family K closed under *controlled iterated* λ*-free* [*non*]*deterministic substitution* if $\eta(K,K) \subseteq K$ [$H(K,K) \subseteq K$]. To obtain in a similar way closure under controlled iterated λ-free substitution for these complexity classes fails; cf. Proposition 4.8, the proof of which is based on Proposition 3.2 and the fact that the Dyck set is in DSPACE ($\log n$).

A language family K is closed under *removal of right endmarker* if for each language Lc in K where $L \subseteq \Sigma^*$ for some alphabet Σ with $c \notin \Sigma$, the language L is in K too.

Proposition 4.8. [3]. *Let K be a family closed under removal of right endmarker. If* DSPACE ($\log n$) $\subseteq K \subset$ RE, *then K is not closed under controlled iterated λ-free (non)deterministic substitution. In particular this applies to each complexity class which includes* DSPACE ($\log n$). □

For the time complexity classes **P** and **NP** we have the familiar situation; viz. **NP** has "strong" closure properties, whereas **P** shares these properties if and only if **P** = **NP**.

Theorem 4.9. [3].
(1) **NP** *is an AFL closed under intersection and iterated* λ*-free (non)-deterministic substitution.*
(2) *Let C be either* DSPACE ($\log n$), NSPACE ($\log n$), *or* **P**. *Then the following propositions are equivalent.*

(a) $C = \mathbf{P} = \mathbf{NP}$.

(b) *C is closed under iterated λ-free nondeterministic substitution.*

(c) *C is closed under iterated λ-free deterministic substitution.*

(d) *C is closed under λ-free homomorphism.* □

5. Iterated Context-Dependent Rewriting

Central in our approach (cf. [5]) to introduce an abstract context-dependent grammar model is the notion of transduction.

Definition 5.1. Let V be an alphabet. A *transduction* τ over V is a function $\tau: V^* \to \mathbf{P}(V^*)$ extended to languages by $\tau: \mathbf{P}(V^*) \to \mathbf{P}(V^*)$ with $\tau(L) = \bigcup \{\tau(x) \mid x \in L\}$ for each language L over V.
Let f be an n-ary operation on languages. A family **T** of transductions is *closed* under (composition to the left with) f, if for all **T**-transductions $\tau_1, \ldots, \tau_n$ over some alphabet V, there exists a **T**-transduction τ over V such that $\tau(x) = f(\tau_1(x), \ldots, \tau_n(x))$ for all x

in V^*. □

In many proofs one wants to construct a new grammar G_N from an old one G_O by attaching a finite amount of information to the symbols of G_O. Then the transductions in G_N over this extended alphabet will be defined in terms of the old transductions of G_O using closure under isomorphism. Finally, we strip this additional information by applying an isomorphism in order to obtain words over the original alphabet. Therefore we make the following basic assumption.

Assumption 5.2. Henceforth **T** is a family of transductions that
(1) is closed under (composition to the left with) isomorphisms; cf. Definition 5.1,
(2) is closed under composition to the right with isomorphisms, i.e., for each **T**-transduction τ_1 over V_1 and each isomorphism $i: V \to V_1$ there exists a **T**-transduction τ over V such that $\tau(x) = \tau_1(i(x))$ for each x in V^*, and
(3) contains for each V the identity mapping over V. □

From 5.2 it follows that **T** also contains all isomorphisms. We are now ready for the main formal definition.

Definition 5.3. Let **T** be a family of transductions. A **T**-*grammar* $G = (V,\Sigma,U,S)$ consists of an alphabet V, a terminal alphabet Σ $(\Sigma \subseteq V)$, an initial symbol S $(S \in V)$, and a finite set U of **T**-transductions over V. The language $L(G)$ generated by G is defined by

$$L(G) = U^*(S) \cap \Sigma^* =$$
$$= (\cup \{\tau_p(...(\tau_1(S))...) | p \geqslant 0;\ \tau_i \in U,\ 1 \leqslant i \leqslant p\}) \cap \Sigma^*.$$

Let Γ be a family of languages. A Γ-*controlled* **T**-*grammar* $(G;C) = (V,\Sigma,U,S,C)$ is a **T**-grammar $(V,\Sigma,U,S,)$ provided with a control language $C \subseteq U^*$ from Γ. The language $L(G;C)$ generated by $(G;C)$ is defined by

$$L(G;C) = C(S) \cap \Sigma^* = (\cup \{\tau_p(...(\tau_1(S))...) | \tau_1 \cdots \tau_p \in C\}) \cap \Sigma^*.$$

L(**T**) [**L**(**T**;Γ), respectively] is the family of languages generated by [Γ-controlled] **T**-grammars, and **L**(**T**;m) [respectively **L**(**T**;Γ;m)] is the subfamily of languages generated by [Γ-controlled] **T**-grammars that possess at most m $(m \geqslant 1)$ **T**-transductions. □

Example 5.4. (1) Let HOM and FINSUB be the families of all homomorphisms and of all finite substitutions, respectively. Then **L**(HOM) = EDTOL and **L**(FINSUB) = ETOL. For Γ-controlled variations, see e.g. [1,2,6,8,13,14].
(2) Let dK-SUB [nK-SUB] denote the family of all [non]deterministic K-substitutions. Then **L**(dK-SUB) = $\eta(K)$ [respectively, **L**(nK-SUB) = $H(K)$] i.e., the family of languages generated by [non]deterministic K-iteration grammars.
(3) The language families **L**(**T**;Γ) and **L**(**T**) with **T** equal to λNGSM,

λDGSM, NGSM, and DGSM have been investigated in [9,22,16,4,17]; see [17] in particular, where e.g. the family of context-free languages is characterized by $\mathbf{L}(\mathbf{T})$ for a family $\mathbf{T}$ of restricted NGSM mappings. □

All the examples in 5.4 are transductions in the sense of Definition 5.1, and they all satisfy Assumption 5.2.

For these context-dependent abstract grammars we also need a decidable membership problem; cf. Proposition 5.7. Therefore we restrict our attention to so-called locally context-independent transductions [20,5] which enables us to establish an analogue of Lemma 3.3, viz. Lemma 5.6.

Definition 5.5. A transduction τ over some alphabet V is called *locally context-independent* if

(1) τ is *monotonic*, i.e., for each y in V^*, $y \in \tau(x)$ implies $|y| \geqslant |x|$.

(2) τ is context-independent in length-preserving applications, i.e., for all x_i, y_i in V^* with $|x_i| = |y_i|$ $(i = 1,2,3)$, $y_1y_2y_3 \in \tau(x_1x_2x_3)$ implies $y_1y_3y_2 \in \tau(x_1x_3x_2)$. □

Lemma 5.6. [5]. *Let* $\mathbf{T}$ *be a family of locally context-independent transductions, and let* $\mathbf{T}$ *contain for all alphabets* V *the length-preserving finite substitutions*

$$\tau(\alpha) = W \qquad \alpha \text{ in } V, \quad W \subseteq V.$$

(1) *Let* Γ *be a family closed under finite substitutions and under intersection with regular languages, and let* $(G;C) = (V,\Sigma,U,S,C)$ *be a* Γ*-controlled* $\mathbf{T}$*-grammar. Then we can effectively construct a* Γ*-controlled* $\mathbf{T}$*-grammar* $(G';C') = (V,\Sigma,U',S,C')$ *such that* $L(G';C') = L(G;C)$, *and for each* x *in* $L(G';C')$, *there is a control word* $\tau_1 \ldots \tau_p$ *in* C' *such that* $x \in \tau_p \ldots \tau_1(S)$ *and* $p \leqslant 2|x|$.

(2) *For each* $\mathbf{T}$*-grammar* $G = (V,\Sigma,U,S)$, *we can effectively construct a* $\mathbf{T}$*-grammar* $G' = (V,\Sigma,U',S)$ *such that* $L(G') = L(G)$, *and for each* x *in* $L(G')$, *there exists a word* $\tau_1 \ldots \tau_p$ *in* U'^* *such that* $x \in \tau_p \ldots \tau_1(S)$, *and* $p \leqslant 2|x|$.

Proof: (1) We add new control words to C such that the corresponding derivations possess the property that each length-preserving step in such a derivation is immediately followed by a length-increasing step.

If $V = \{\alpha_1,\ldots,\alpha_k\}$ for some $k \geqslant 1$, then we define $U' = U \cup \{[\tau,q] \mid \tau \in U, q \in Q\}$ with $Q = \{<X_1,\ldots,X_k> \mid X_i \subseteq V,\ 1 \leqslant i \leqslant k\}$, and $C' = \sigma(C)$ where $\sigma = (Q,U,U',\delta,q_0,Q_F)$ is an NGSM with $q_0 = <\{\alpha_1\},\ldots,\{\alpha_k\}>$, $Q_F = \{q_0\}$, while δ is defined by

$$\delta(<X_1,\ldots,X_k>,\tau) = \{(q_0,\tau) \mid <X_1,\ldots,X_k> = q_0\} \cup$$

$$\cup \{(<\tau(X_1) \cap V,\ldots,\tau(X_k) \cap V>,\lambda),(q_0,[\tau,<X_1,\ldots,X_k>])\}$$

(Notice that $C \subseteq C'$).

Next we indicate the way in which the new additional control words are obtained by means of σ from C, together with the effect of

these new control words. Consider an arbitrary derivation D according to $(G;C)$. At each step in D, determined by the application of some T-transduction τ, one of the following three cases applies (cf. the definition of δ):

Case (a): This application of τ is length-increasing.
The corresponding transition in σ is the identity transition: $(q_0,\tau)\in \delta(q_0,\tau)$. This case does not give rise to the addition of new control words.

Case (b): This application of τ is length-preserving and the next step in D will also be length-preserving.
The corresponding occurrence of τ in the control word is erased, and the length-preserving context-independent effect (cf. Definition 5.5) of τ is stored by means of changing the state of σ from $<X_1,...,X_k>$ to $<\tau(X_1)\cap V,...,\tau(X_k)\cap V>$.

Case (c): This application of τ is length-preserving but either the next step in D will be length-increasing, or this application of τ is the last step in D.
In the old control word we replace the corresponding occurrence of τ by $[\tau,<X_1,...,X_k>]$ where $<X_1,...,X_k>$ is the current state of σ in which the ultimate length-preserving effect of a consecutive sequence of erased transductions (cf. Case (b)) has been stored. This new transduction $[\tau,<X_1,...,X_k>]$ is a length-preserving finite substitution defined by

$$[\tau,<X_1,...,X_k>](\alpha_i)=\tau(X_i)\cap V \qquad \text{for each } i\ (1\leqslant i\leqslant k).$$

(2) Define $U'=U\cup\{\tau_u \mid u\in U^+\}$ with for each u in U^+, τ_u is the length-preserving substitution defined by

$$\tau_u(\alpha)=u(\alpha)\cap V \qquad \alpha \text{ in } V.$$

Then U' is finite, because there are only a finite number of length-preserving substitutions over V. □

The proof of the following result is almost identical to the one of Proposition 3.4.

Proposition 5.7. [5]. *Let* Γ *and* T *satisfy the assumptions of Lemma 5.6. If* Γ *is a subfamily of the family of recursive languages and if* T *is a subfamily of the family of recursive transductions, then each language in* $\mathbf{L}(\mathbf{T};\Gamma)$ *and in* $\mathbf{L}(\mathbf{T})$ *is recursive.* □

6. Complexity Aspects of Iterated Context-Dependent Rewriting

In this section we determine upper bounds for the space and time complexity of languages generated by (controlled) T-grammars; viz. Theorems 6.2, 6.5 and 6.8.

Throughout this section "function" means a monotone increasing function g over the natural numbers satisfying $g(n)\geqslant n$ for each

$n \in \mathbb{N}$.

Definition 6.1. Let for each function $g: \mathbb{N} \to \mathbb{N}$, DSPACETR($g$) [NSPACETR($g$), respectively] be the family of those transductions τ that satisfy
(1) τ is locally context-independent, and
(2) there exists a [non]deterministic algorithm that can decide a query "$y \in \tau(x)$?" for each x and y within space $g(|y|)$. □

Theorem 6.2. [5]. *Let g be a function.*
(1) *If* $\mathbf{T} \subseteq$ NSPACETR(g), *then* $\mathbf{L}(\mathbf{T}) \subseteq$ NSPACE(g).
(2) $\mathbf{L}$(NSPACETR(g)) = NSPACE(g).
(3) *Let Γ be a family closed under finite substitution and under intersection with regular languages. If* $\Gamma \subseteq$ NSPACE(g), $\mathbf{T} \subseteq$ NSPACETR(g), *and* $g(2n) \leqslant c.g(n)$ *for some constant c, then* $\mathbf{L}(\mathbf{T};\Gamma) \subseteq$ NSPACE(g).

Proof: (1) Consider the algorithm in Figure 2; remove the assignments in which the variable control is involved, and replace the last statement by **accept**.

```
read x;
control := λ;
if x ∉ Σ+ then reject else
      y := S;
      while y ≠ x and |y| ⩽ |x| do
            guess τ ∈ U;
            guess z ∈ V+ with |y| ⩽ |z| ⩽ |x|;
            if z ∈ τ(y) then control := control.τ;
                        y := z
                        else reject
            fi
      od
fi;
if control ∈ C then accept else reject fi.
```

Figure 2.

Then each step in this modified algorithm requires at most linear space, except the test "$z \in \tau(y)$" for which we need $g(|z|) \leqslant g(|x|)$ space. Thus for $|x| = n$, the total amount of space is $O(n + g(n)) = O(g(n))$.

(2) The inclusion $\mathbf{L}$(NSPACETR(g)) $\subseteq$ NSPACE(g) follows immediately from (1) by taking $\mathbf{T}$ = NSPACETR(g).

Conversely, let $L_0 \subseteq \Sigma^*$ be a language in NSPACE(g). Define $G = (V, \Sigma, \{\tau\}, S)$ with $V = \Sigma \cup \{S\}$, and τ is defined by

$$\tau(S) = L_0$$
$$\tau(w) = \{w\} \qquad \text{for each } w \text{ in } \Sigma^*.$$

Then $\tau \in$ NSPACETR(g), G is an NSPACETR(g)-grammar, $L(G) =$

L_0, and hence NSPACE $(g) \subseteq \mathbf{L}$ (NSPACETR (g)).

(3) Consider the algorithm of Figure 2. By Lemma 5.6 the last statement requires space $O(g(2n))$ which is $O(g(n))$ due to the assumption on g. So the total space needed to execute the algorithm is $O(n+g(n))+O(g(n)) = O(g(n))$; cf. the proof of (1). □

Corollary 6.3. [5]. $\mathbf{L}$ (NSPACETR (n)) = NSPACE (n). □

NSPACE (n) or, equivalently, the family of λ-free context-sensitive languages can be characterized by much simpler transductions than those used in 6.3; in 6.4 we combine results from [9,22,16,4,17] together with some simple properties.

Theorem 6.4.

$\mathbf{L}(\lambda$NGSM;REG$) = \mathbf{L}(\lambda$NGSM$) = \mathbf{L}(\lambda$NGSM$;1) =$
$\mathbf{L}(\lambda$DGSM;REG$) = \mathbf{L}(\lambda$DGSM$) = \mathbf{L}(\lambda$DGSM$;2) =$ NSPACE(n). □

Although this solves partially an open problem from [22], viz. $\mathbf{L}(\lambda$DGSM$;2) =$ NSPACE (n), the precise nature of $\mathbf{L}(\lambda$DGSM$;1)$ and an analogous characterization of DSPACE (n) are still unknown. However, it is easy to show that $\mathbf{L}(\lambda$DGSM$;1) \subseteq$ DSPACE (n).

For a deterministic counterpart of Theorem 6.2 we can generalize the proof of Theorem 5.2 in [21] straightforwardly. The details are left to the reader.

Theorem 6.5. *Let g be a function with $g(n) \geqslant n \log n$ for each $n \in \mathbb{N}$, and there exists a constant c such that $g(2n) \leqslant c.g(n)$ for each $n \in \mathbb{N}$. Let Γ be a family of languages closed under finite substitution and under intersection with regular languages.*
(1) *If* $\mathbf{T} \subseteq$ DSPACETR (g), *then* $\mathbf{L}(\mathbf{T}) \subseteq$ DSPACE (g).
(2) $\mathbf{L}$ (DSPACETR (g)) = DSPACE (g).
(3) *If* $\Gamma \subseteq$ DSPACE (g) *and* $\mathbf{T} \subseteq$ DSPACETR (g), *then* $\mathbf{L}(\mathbf{T};\Gamma) \subseteq$ DSPACE (g). □

Next we turn to time-bounded transductions and time-bounded complexity classes. Instead of a single bounding function we now consider a class of functions that is closed under certain operations. The following definition is a slight modification of a concept from [20].

Definition 6.6. A class $\mathbf{C}$ of functions is called *natural* if
(1) $\mathbf{C}$ contains the identity function $\lambda x.\ x$,
(2) for each f and g in $\mathbf{C}$, there is a monotone increasing function in $\mathbf{C}$ that majorizes $\lambda x.(f(x)+g(x))$,
(3) for each f and g in $\mathbf{C}$, there is a monotone increasing function in $\mathbf{C}$ that majorizes $\lambda x.(f(x)g(x))$, and
(4) for each f in $\mathbf{C}$, there is a monotone increasing function in $\mathbf{C}$ that majorizes $\lambda x.f(2x)$. □

Definition 6.7. Let for each class $\mathbf{C}$ of functions, NTIMETR $(\mathbf{C})$ be the family of those transductions τ that satisfy
(1) τ is locally context-independent, and

(2) there exists a nondeterministic algorithm that can decide a query "$y \in \tau(x)$?" within time $g_\tau(|y|)$ for some g_τ in **C**. □

For a class **C** of functions NTIME(**C**) is defined by NTIME(**C**) = $\bigcup \{\text{NTIME}(g) \mid g \in \mathbf{C}\}$. Let *poly* be the class of all polynomials over the natural numbers. Obviously, poly is a natural class.

Theorem 6.8. [5]. *Let* **C** *be a natural class of functions, and let* Γ *be a family of languages closed under finite substitution and under intersection with regular languages.*
(1) *If* $\mathbf{T} \subseteq \text{NTIMETR}(\mathbf{C})$, *then* $\mathbf{L}(\mathbf{T}) \subseteq \text{NTIME}(\mathbf{C})$.
(2) $\mathbf{L}(\text{NTIMETR}(\mathbf{C})) = \text{NTIME}(\mathbf{C})$.
(3) *If* $\Gamma \subseteq \text{NTIME}(\mathbf{C})$ *and* $\mathbf{T} \subseteq \text{NTIMETR}(\mathbf{C})$, *then* $\mathbf{L}(\mathbf{T};\Gamma) \subseteq \text{NTIME}(\mathbf{C})$.

Proof: The proof is similar to the one of Theorem 6.2. As an example we show (3). Let $(G;C) = (V,\Sigma,U,S,C)$ be a Γ-controlled **T**-grammar. Assume $U = \{\tau_1,...,\tau_m\}$, and for each i $(1 \leqslant i \leqslant m)$ a query "$z \in \tau(y)$?" can be resolved within time $g_i(|z|)$ for some g_i in **C**. Since **C** is natural there exists a function g in **C** that majorizes $\lambda x.(g_1(x)+...+g_m(x))$ and hence $g(x) \geqslant g_i(x)$ for each x and each i $(1 \leqslant i \leqslant m)$.

Consider the algorithm of Figure 2. By Lemma 5.6 it suffices to execute the body of the **while**-loop at most $2n$ times where n is the length of the input. All statements in this body require time $O(n)$ only, except the test "$z \in \tau(y)$?" which is $O(g(n))$. Therefore this **while**-loop can be executed in time at most $O(n.(n+g(n)))$. The preceding statements consume $O(n)$ time, while the last statement of the algorithm needs time $h_1(2n)$ for some h_1 in **C** (assuming that $C \in \text{NTIME}(h_1)$). As **C** is natural, $\lambda n.h_1(2n)$ is majorized by some h in **C**. Thus the total time to execute the algorithm is $O(n+n(n+g(n))+h(n))$. Since **C** is natural this is majorized by some function in **C**. Hence $L(G;C) \in \text{NTIME}(\mathbf{C})$. □

Corollary 6.9. L(NTIMETR(poly)) = **NP**. □

Theorem 6.8(2) and Corollary 6.9 are variations of results established by Van Leeuwen [20] for another rather abstract grammatical model.

In addition to Theorem 6.8 we remark that from the main result in [19] it follows that if **T** contains all (λ-free) finite substitutions, then the membership problem for $\mathbf{L}(\mathbf{T})$ is **NP**-hard.

We conclude this section with a counterpart of Theorem 4.6 with respect to closure under iterated locally context-independent time- or space-bounded transductions. We call a family K of languages closed under *iterated* **T**-*transductions* if for each language L in K with $L \subseteq V^*$ for some alphabet V, and each finite set U of **T**-transductions over V, the language $U^*(L)$ belongs to K.

Theorem 6.10. [5]. *Let g be a function such that there exists a constant c with $g(2n) \leqslant c.g(n)$ for each $n \in \mathbb{N}$.*
(1) *If $g(n) \geqslant n$ for each $n \in \mathbb{N}$, then* NSPACE(g) *is the smallest AFL closed under iterated locally context-independent nondeterministic g-space-bounded transductions. In particular this applies to*
– NSPACE(n), *the family of context-sensitive languages;*
– NSPACE(n^2), *the family of two-way nondeterministic nonerasing stack automaton languages;*
– **PSPACE**.
(2) *If $g(n) \geqslant n \log n$ for each $n \in \mathbb{N}$, then* DSPACE(g) *is the smallest AFL closed under iterated locally context-independent deterministic g-space-bounded transductions. In particular this applies to*
– DSPACE$(n \log n)$, *the family of two-way deterministic nonerasing stack automaton languages.*
(3) *If* **C** *is a natural class of functions, then* NTIME(**C**) *is the smallest AFL closed under iterated locally context-independent nondeterministic* **C***-time-bounded transductions. In particular this applies to* **NP**.

Proof: From 6.2(2), 6.5(2) and 6.8(2) closure under iterated **T**-transductions easily follows for **T** equal to NSPACETR(g), DSPACETR(g), and NTIMETR(**C**) respectively. Closure under iterated **T**-transductions implies closure under union, concatenation, Kleene + and λ-free homomorphism. The remaining two AFL-properties (closure under inverse homomorphism and intersection with regular languages) can be proved by standard automaton-theoretic constructions. Since each AFL closed under iterated **T**-transductions includes **L**(**T**), it is easy to see that **L**(**T**) is the smallest AFL closed under iterated **T**-transductions.

For the characterization of two-way nonerasing stack automaton languages in terms of complexity classes we refer to [11]. □

It is an open problem whether a similar proposition holds for DSPACE(n), the family of deterministic context-sensitive languages.

7. Concluding Remarks

We summarized some results on the complexity of the membership problem for (controlled) iteration grammars (Section 4) and for (controlled) grammars based on transductions (Section 6). In the former case these results are rather satisfactory; in the latter one we could extend the results of Section 4 only at the price of requiring local context-independency. Whether this is a serious restriction is still open. However, when we drop this condition we will probably need a rather different approach to solving the membership problem in the context-dependent case.

Apart from this general problem, a few more concrete open questions have already been mentioned; cf. the remarks after Theorems 6.4 and 6.10. Another long standing open problem in this area, is the

question whether DSPACE(n) is a hyper-AFL, i.e., whether it is an AFL closed under iterated λ-free nondeterministic substitution [21,3]. It is even unknown whether this question is equivalent to the classic LBA-problem, i.e., is DSPACE(n) = NSPACE(n)?. Thus, our knowledge in this respect is more restricted in case of linear space than it is in case of polynomial time; cf. Theorem 4.9.

Finally, we mention a different approach to the subject of this survey based on the notion of nondeterministic log-space-bounded reducibility; cf. [15]. Let for each language family K, NLOG(K) be the family of languages that are many-one reducible to a language in K by a reduction function computable nondeterministically in space $\log n$ by a Turing machine of which each computation is of polynomial length. As usual, LOG(K) is the class of languages many-one log-space reducible to languages in K. Then the following holds.

Proposition 7.1. [15].
(a) LOG(EDTOL) = NSPACE($\log n$),
(b) NLOG(EDTOL) = **NP**,
(c) LOG(ETOL) = **NP**,
(d) NLOG(ETOL) = **NP**. □

Let ONE be the language family of singleton sets, i.e., ONE = $\{L \mid \text{card}(L) = 1\}$. Then, e.g., η(ONE) = EDTOL.

Theorem 7.2. [15]. *Let Γ be a language family closed under reversal, finite substitution and intersection with regular languages. Then* $\eta(\Gamma, \text{ONE}) \subseteq \text{NLOG}(\Gamma)$. □

As corollaries one obtains the restricted version of Theorem 4.1 with K = ONE, the main result of [12], and implications of the form: if Γ satisfies the conditions of Theorem 7.2 and $\Gamma \subseteq$ LOG(CF), then $\eta(\Gamma, \text{ONE}) \subseteq$ LOG(CF) where CF is the family of context-free languages; cf. [15]. On the other hand it is still open whether η(EDTOL,ONE) $\subseteq$ **P**.

References

1. P.R.J. Asveld: Controlled iteration grammars and full hyper-AFL's, *Inform. and Control* **34** (1977) 248-269.
2. P.R.J. Asveld: Iterated Context-Independent Rewriting — An Algebraic Approach to Families of Languages, Doctoral Dissertation (1978), Twente University of Technology, Enschede, The Netherlands.
3. P.R.J. Asveld: Space-bounded complexity classes and iterated deterministic substitution, *Inform. and Control* **44** (1980) 282-299.
4. P.R.J. Asveld: On controlled iterated GSM mappings and related operations, *Rev. Roumaine Math. Pures Appl.* **XXV** (1980) 139-145.

5. P.R.J. Asveld: Abstract grammars based on transductions, Memorandum INF-85-13 (1985), Twente University of Technology, Enschede, The Netherlands.

6. P.R.J. Asveld & J. Engelfriet: Iterated deterministic substitution, *Acta Inform.* **8** (1977) 285-302.

7. P.R.J. Asveld & J. Engelfriet: Extended linear macro grammars, iteration grammars, and register programs, *Acta Inform.* **11** (1979) 259-285.

8. P.R.J. Asveld & J. van Leeuwen: Infinite chains of hyper-AFL's, TW-memorandum No. 99 (1975), Twente University of Technology, Enschede, The Netherlands.

9. A.C. Fleck: Formal languages and iterated functions with an application to pattern representations, Report No. 75-03 (1975), Department of Computer Science, The University of Iowa, Iowa City.

10. M.A. Harrison: *Introduction to Formal Language Theory* (1978), Addison-Wesley, Reading, Mass.

11. J.E. Hopcroft & J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation* (1979), Addison-Wesley, Reading, Mass.

12. N.D. Jones & S. Skyum: Recognition of deterministic ETOL languages in logarithmic space, *Inform. and Control* **35** (1977) 177-181.

13. K.-J. Lange: Context-free controlled ETOL systems, *in:* J. Díaz (Ed.): *Automata, Languages and Programming — 10th Colloquium*, Lect. Notes in Comp. Sci. **154** (1983) 723-733, Springer-Verlag, Berlin, Heidelberg, New York.

14. K.-J. Lange: Kontextfrei Kontrolierte ETOL-Systeme, Doctoral Dissertation (1983), University of Hamburg, F.R.G.

15. K.-J. Lange: L systems and NLOG-reductions, *in:* G. Rozenberg & A. Salomaa (Eds.): *The Book of L* (1985), Springer-Verlag, Berlin, Heidelberg, New York, pp. 245-252.

16. G. Paun: On the iteration of GSM mappings, *Rev. Roumaine Math. Pures Appl.* **XXIII** (1978) 921-937.

17. B. Rovan: A framework for studying grammars, *in:* J. Gruska & M. Chytil (Eds.): *Mathematical Foundation of Computer Science* 1981, Lect. Notes Comp. Sci. **118** (1981) 473-482, Springer-Verlag, Berlin, Heidelberg, New York.

18. G. Rozenberg & A. Salomaa: *The Mathematical Theory of L Systems* (1980), Academic Press, New York.

19. J. van Leeuwen: The membership question for ETOL languages is polynomially complete, *Inform. Process. Lett.* **3** (1975) 138-143.

20. J. van Leeuwen: Extremal properties of non-deterministic time-complexity classes, *in:* E. Gelenbe & D. Potier (Eds.): *International Computing Symposium* (1975), pp. 61-64, North-Holland, Amsterdam.

21. J. van Leeuwen: A study of complexity in hyper-algebraic families, *in:* A. Lindenmayer & G. Rozenberg (Eds.): *Automata, Languages, Development* (1976), pp. 323-333, North-Holland, Amsterdam.

22. D. Wood: Iterated a-NGSM maps and Γ systems, *Inform. and Control* **32** (1976) 1-26.

On Covers and Left-Corner Parses

Rieks op den Akker

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

A transformation is defined which is a modification of a classic transformation on context-free grammars. By means of this transformation, a proof is presented of the fact that any cycle-free context-free grammar can be left-to-left-corner covered by a non-left-recursive grammar. The proof method is based on the idea to transform the characteristic grammar associated with the simple syntax-directed translation scheme which defines the left-corner parse of the strings generated by the input grammar of the scheme. It is shown that the transformation yields an $LL(k)$ grammar if and only if it is applied to an $LC(k)$ grammar. Finally, some ideas are presented to extend the theory of covers to the semantical covering of attribute grammars.

1. Introduction

Let G_1 and G_2 be context-free grammars. G_1 and G_2 are called *weakly equivalent* if they generate the same context-free language. Let x denote a string in the language $L = L(G_1) = L(G_2)$. Weak equivalence of G_1 and G_2 does not imply that there is a structural similarity between the parse tree or parse trees of x with respect to G_1 and the parse tree or parse trees of x with respect to G_2. Here we will consider a stronger form of equivalence between context-free grammars. We say that G_2 *covers* G_1 if we can transform G_1 into the weakly equivalent grammar G_2 and we can systematically find the parse tree of each string x in L with respect to G_1 from the parse tree of x with respect to G_2.

A class Y of context-free grammars covers the class X of context-free grammars if there is a transformation defined on all grammars in X such that the transformed grammar is a grammar in class Y and covers the original grammar. If a class Y covers a class X then we can use a compiler writing system (CWS) based on a parsing method for grammars of class Y also for grammars in class X; see Figure 1.

First, we transform a grammar G from class X into the covering grammar G' and then built a parser for G'. The output of this parser is a parse tree with respect to G'. This tree is transformed into the corresponding parse tree with respect to the original grammar G. This last transformation makes it possible to define the semantics of x with respect to the semantical definition based on grammar G.

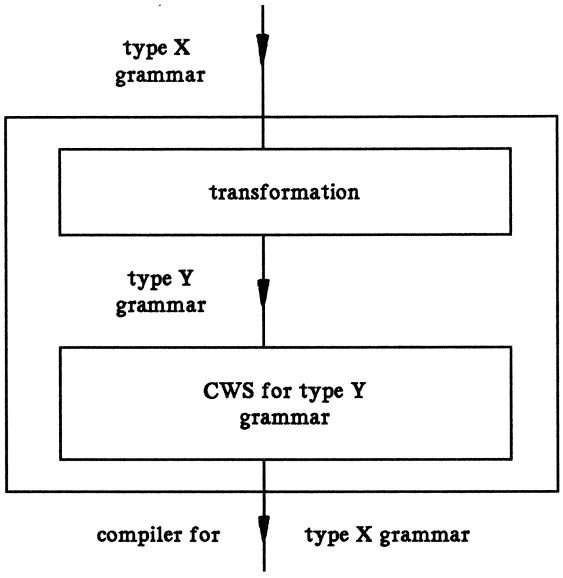

Figure 1.

For this reason the original grammar on which the syntax-directed semantics is based is called *semantic grammar* in [7] and the grammar obtained after transformation(s) is called the *parsing grammar*. The reason for using two grammars may be that the parsing grammar is not convenient for expressing semantics. Sometimes it is possible and also useful to do semantic actions or — in terms of the formalism of attribute grammars [5] — to evaluate attributes during the parsing of the input string. This allows for the use of semantic properties (attribute values) of the already analyzed part of the input string in making parsing decisions. Then it becomes practically interesting to adapt the transformations on context-free grammars in such a way that they preserve the semantics defined by the original (attribute) grammar.

We will define a transformation on context-free grammars. If this transformation is applied to a context-free grammar G, then the resulting grammar G' covers the original grammar G. If the transformation is applied to an $LC(k)$ grammar, then the resulting grammar is $LL(k)$. The transformation is a modification of the one given by Rosenkrantz and Lewis II [9]. The modification is due to the fact that our definition of the $LC(k)$ grammars is that of Soisalon-Soininen [11], which is a modification of the original definition of Rosenkrantz and Lewis II. The equivalence of this modified definition and the definition of $LC(k)$ grammars of Soisalon-Soininen and Ukkonen is proved in [2].

This paper is organized as follows. In Section 2 we give some preliminary definitions concerning parse relations and cover relations between classes of context-free grammars. In Section 3 we define the left-corner parse relation. In Section 4 we define a transformation on context-free grammars. In Section 5 we consider general cover properties of the transformation. Section 6 presents definitions of $LL(k)$ grammars and left-corner or $LC(k)$ grammars. In Section 7 we show that the transformation yields an $LL(k)$ grammar if and only if it is applied to an $LC(k)$ grammar. Finally, we present in Section 8 some ideas to extend the notion of cover to the semantical covering of attribute grammars.

2. Preliminaries on Covers

The first results on cover relations between context-free grammars were obtained by workers in the area of compiler writing. One of the first theoretical studies in this field is that of Gray and Harrison [6]. A general theory of covers has been developed by Nijholt [8].

We recall some preliminary notions from this theory of covers. Since we will only consider cover relations between grammars that have the same terminal alphabets, we do not need all notions in the general formulation presented by Nijholt.

$<w,G>$ denotes the degree of ambiguity of w with respect to G. Let G be a cfg (context-free grammar) and Δ_G a set of unique labeling symbols for the productions of G.

Definition 2.1. A relation $f_G \subseteq \Sigma^* \times \Delta_G^*$ is a *parse relation for* G if it satisfies the following conditions.

i. For each string $w \in L(G)$ there exists at least one element $(w,\pi) \in f_G$.

ii. For each $w \in \Sigma^*$, $|\{\pi \mid (w,\pi) \in f_G\}| \leqslant <w,G>$. □

Definition 2.2. A relation $f_G \subseteq \Sigma^* \times \Delta_G^*$ is a *proper parse relation for* G if it satisfies the following conditions.

i. If $(w,\pi) \in f_G$ and $(w',\pi) \in f_G$ then $w = w'$.

ii. For each $w \in \Sigma^*$, $|\{\pi \mid (w,\pi) \in f_G\}| = <w,G>$. □

Thus a proper parse relation for G is a parse relation for G.

Definition 2.3. Let $G = (N,\Sigma,P,S)$ and $G' = (N',\Sigma,P',S')$ be context-free grammars with labeling sets for the productions Δ_G and $\Delta_{G'}$. Let f_G and $f_{G'}$ be parse relations for G and G', respectively. A homomorphism $\phi : \Delta_{G'}^* \rightarrow \Delta_G^*$ is a *parse homomorphism* if $(w,\pi) \in f_{G'}$ implies $(w,\phi(\pi)) \in f_G$. □

Definition 2.4. A parse homomorphism is a *cover homomorphism* if for all $(w,\pi) \in f_G$, there exists $(w,\pi') \in f_{G'}$ such that $\phi(\pi') = \pi$. □

Definition 2.5. Let $G = (N,\Sigma,P,S)$ and $G' = (N',\Sigma,P',S')$ be cfgs. Let f_G and $f_{G'}$ be parse relations for G and G' respectively. Grammar G' *$f_{G'}$-to-f_G covers* G if there exists a cover homomorphism $\phi : \Delta_{G'}^* \rightarrow \Delta_G^*$. Notation: $G'[f_{G'}/f_G]G$. □

Two well-known parse relations are the following. The *left parse relation for* G is $l_G = \{(w,\pi) \mid S \Longrightarrow_l^{\pi} w\}$. The *right parse relation for* G is $r_G = \{(w,\pi^R) \mid S \Longrightarrow_r^{\pi} w\}$, in which π^R denotes the reverse of π.

The cover relation satisfies the *transitivity property*. Let f,g,h be parse relations for cfgs F,G,H, respectively. If $F[f/g]G$ with respect to cover homomorphism ϕ_1 and $G[g/h]H$ with respect to cover homomorphism ϕ_2 then $F[f/h]H$ with respect to cover homomorphism $\phi_2 \circ \phi_1$.

Most results on covers between cfgs concern left-to-left (or simply left), right-to-right (right), left-to-right and right-to-left-covers by grammars in some normal form, as for example Greibach Normal Form or Chomsky Normal Form, by grammars without ϵ-rules or by non-left-recursive grammars. All the results obtained upto 1980 can be found in Nijholt [8].

3. The Left-Corner Parse

Before we come to the definition of the left-corner parse of a string with respect to a given context-free grammar, we define a useful homomorphism. Let A be a set of symbols and $\Sigma \subseteq A$. The *Σ-erasing homomorphism on A*, $h_\Sigma : A^* \rightarrow A^*$ is defined by $h_\Sigma(a) = a$ if $a \notin \Sigma$ and $h_\Sigma(a) = \epsilon$ if $a \in \Sigma$. For a language L we define $h_\Sigma(L) = \{h_\Sigma(x) \mid x \in L\}$.

Let $G = (N,\Sigma,P,S)$ be a cfg, $|P| = m$, Δ a set $\{p_1,\ldots,p_m\}$ such that $\Sigma \cap \Delta = \emptyset$ and $\lambda_G : P \rightarrow \Delta$ a labeling function associating with each production in P a unique symbol in Δ. We will omit the subscript G and simply write λ instead of λ_G.

With each cfg G and label set Δ we associate the cfg $G_{lc} = (N,\Sigma \cup \Delta, P_{lc}, S)$ in which P_{lc} is defined as follows.

$$P_{lc} = \{A \rightarrow p_i \mid A \rightarrow \epsilon \text{ in } P \text{ and } \lambda(A \rightarrow \epsilon) = p_i\} \cup$$
$$\{A \rightarrow Xp_i\alpha \mid A \rightarrow X\alpha \text{ in } P \text{ and } \lambda(A \rightarrow X\alpha) = p_i\}.$$

Clearly, $h_\Delta(L(G_{lc})) = L(G)$.

We use the grammar G_{lc} in order to define the left-corner parse of a string $x \in L(G)$ with respect to G.

Definition 3.1. Let G be a cfg, $x \in L(G)$ and Δ, G_{lc} as defined above. $\pi \in \Delta^*$ is a *left-corner parse of x with respect to G* if there is a string $y \in L(G_{lc})$, such that $h_\Sigma(y) = \pi$ and $h_\Delta(y) = x$. □

The *left-corner parse relation for* G $-$ $\{(h_\Delta(y), h_\Sigma(y)) \mid y \in L(G_{lc})\}$ $-$ is the same as the one defined by a *simple syntax directed translation scheme* (SDTS) in [1] or [8]. In fact the grammar G_{lc} is the *characteristic grammar* [1] associated with the simple SDTS defining the left-corner parse relation. The left-corner parse relation is a *production directed parse relation* as defined by Nijholt [8].

Example 3.2. Let G be the cfg given by the productions in the left-most table of Figure 2. The right-most table shows the productions of the cfg G_{lc} associated with G and the production label set $\Delta = \{p_1, p_2, p_3, p_4\}$.

1.	$S \to a\,S\,a$	1.	$S \to a\,p_1 S\,a$
2.	$S \to A\,b$	2.	$S \to A\,p_2 b$
3.	$S \to c$	3.	$S \to c\,p_3$
4.	$A \to S$	4.	$A \to S\,p_4$

Figure 2. The productions of grammars G and G_{lc}.

Let $x_1 = ap_1ap_1cp_3p_4p_2baa$ and $x_2 = ap_1ap_1cp_3aap_4p_2b$. Since x_1 and x_2 are both sentences in $L(G_{lc})$ and $h_\Sigma(x_1) = h_\Sigma(x_2)$ although $h_\Delta(x_1) \neq h_\Delta(x_2)$, the left-corner parse relation is not proper for G. The sentences *aacbaa* and *aacaab* both have left-corner parse $p_1p_1p_3p_4p_2$ The derivation trees of the sentences $h_\Delta(x_1)$ and $h_\Delta(x_2)$ are shown in Figure 3. □

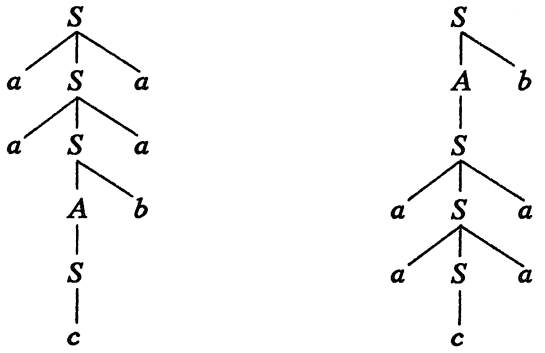

Figure 3. Derivation trees of *aacbaa* and *aacaab*.

4. The Transformation τ

We describe a transformation $-$ we call it τ $-$ which, when applied to a cfg yields a cfg that is equivalent to the original one. τ is a modification of a transformation described by Rosenkrantz and Lewis II in [9] for transforming an $LC(k)$ grammar into an $LL(k)$ grammar. Also Griffith and Petrick have used this transformation. The modification is because we prefer the definition of $LC(k)$ grammars by Soisalon-Soininen [11] which is equivalent to a slightly modified

version of the original $LC(k)$ definition in [9]. In Section 5 we show that transformation τ when applied to a cfg G yields a cfg $\tau(G)$ which left-to-left-corner covers G. In Section 7 we will proof that τ yields an $LL(k)$ grammar if and only if it is applied to an $LC(k)$ grammar. $LL(k)$ grammars and left-corner grammars are defined in Section 6.

As we already noticed in Section 2, most of the cover results concern left, right, left-to-right or right-to-left covers. The reason for this is simply that the parse relations involved in these covers correspond with the canonical left-most and right-most derivations in a cfg. For the left-corner parse relation there is not such a smooth canonical derivation. A left-corner parser (see for instance [1] or [9] for a description) jumps through the parse tree: the left-corner parsing method is a method which combines top-down and bottom-up recognition of parts of the parse tree. Proofs of theorems on left-corner grammars or related concepts tend to be long and tedious and are therefore mostly omitted.

Before we come to a description of τ we introduce some notation and one definition.

Let $G = (N,\Sigma,P,S)$ be a cfg. Context-free grammars are always assumed to be reduced, that is they do not contain useless symbols. We write ϵ for the empty string, V will denote $N \cup \Sigma$, V_ϵ will denote the set $V \cup \{\epsilon\}$ and Σ_ϵ the set $\Sigma \cup \{\epsilon\}$. If $\alpha \in V^*$ then $|\alpha|$ denotes the length of α. Furthermore, for an integer $k > 0$, $k{:}\alpha$ denotes α if $|\alpha| \leqslant k$ and $k{:}\alpha$ denotes the prefix of α of length k if $|\alpha| > k$ (notice that $k{:}\epsilon = \epsilon$ for any k). The *left-corner* of a production $A \to \alpha$ is the symbol $1{:}\alpha$.

Definition 4.1. We define the relation $\geqslant_{lc}^{G}$ with respect to a cfg G as follows:

i. $\geqslant_{lc}^{G} \subseteq N \times V_\epsilon$,

ii. $(X,Y) \in \geqslant_{lc}^{G}$ if and only if $X \to \alpha$ is a production of G and $Y = 1{:}\alpha$. □

We will write $X \geqslant_{lc} Y$ instead of $(X,Y) \in \geqslant_{lc}^{G}$. $\geqslant_{lc}^{+}$ will denote the transitive closure of $\geqslant_{lc}$.

Let $G = (N,\Sigma,P,S)$ be a context-free grammar and let $\overline{N}$ be the set $\{A \in N \mid A = S$ or there is a production in P of the form $B \to \alpha A \beta$, where $\alpha \neq \epsilon\}$. (Thus $A \in \overline{N}$ if A is the start symbol of G or A occurs in the right-hand side of a production of G of which it is not the left-corner). Let $\overline{N}$ be ordered: $\overline{N} = \{A_1, A_2, \ldots, A_n\}$. The transformed grammar $\tau(G)$ of G is the context-free grammar (N', Σ, P', S). N' is a superset of $\overline{N}$ and contains all symbols of the form $[A,Y]$, with $A \in \overline{N}$ and $Y \in V_\epsilon$, which appear in the productions of $\tau(G)$.

P' is defined as follows. Start with $P' = \emptyset$. P' will contain only those productions added to P' in one of the following three steps.

1. For all i, $1 \leqslant i \leqslant n$, for all $a \in \Sigma_\epsilon$ add to P' the production $A_i \to a[A_i, a]$ if $A_i \geqslant_{lc}^{+} a$.
2. For all $[A_i, Y]$, where $Y \in V_\epsilon$, which occur in the right-hand side of a production in P', for all productions in P of the form $B \to Y\beta$, where $\beta \in V^*$ such that $A_i \geqslant_{lc}^{+} B$, add the production $[A_i, Y] \to \beta[A_i, B]$ to P' if it is not already in P'.
3. Add $[A_i, A_i] \to \epsilon$ to P'.

Grammar $\tau(G)$ does not contain useless symbols.

We give two examples of the transformation.

Example 4.2. Consider the context-free grammar G given by the following productions.

1. $S \to S + T$
2. $S \to T$
3. $T \to T \times id$
4. $T \to id$

The symbols id, $+$ and $\times$ are terminal symbols. The transformed grammar G' has the following productions; cf. Figure 4. □

1'.	$S \to id\ [S,id]$	2'.	$[S,id] \to [S,T]$
3'.	$[S,T] \to [S,S]$	4'.	$[S,T] \to \times id\ [S,T]$
5'.	$[S,S] \to + T\ [S,S]$	6'.	$[S,S] \to \epsilon$
7'.	$T \to id\ [T,id]$	8'.	$[T,id] \to [T,T]$
9'.	$[T,T] \to \epsilon$	10'.	$[T,T] \to \times id\ [T,T]$

Figure 4.

Example 4.3. If we apply τ to the cfg G of Example 3.2, we obtain the grammar H given by the productions shown in Figure 5. Do not pay attention yet to the last column. We will later return to this example. □

1'	$S \to a[S,a]$	ϵ
2'	$S \to c[S,c]$	ϵ
3'	$[S,a] \to Sa[S,S]$	1
4'	$[S,c] \to [S,S]$	3
5'	$[S,S] \to [S,A]$	4
6'	$[S,A] \to b[S,S]$	2
7'	$[S,S] \to \epsilon$	ϵ

Figure 5.

5. General Properties of τ

In this section we will show that if transformation τ is applied to a cfg G we obtain a cfg G' which left-to-left-corner covers G. Therefore we have to show that there exists a homomorphism which maps left parses of a string $x \in L(G)$ with respect to G' onto left-corner parses

of x with respect to G. The method of proof is inspired by Soisalon-Soininen [10]. The diagram of Figure 6 shows the grammars involved in the proof and the steps we will take.

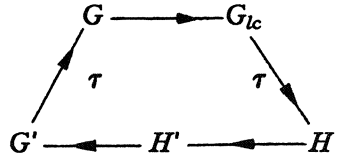

Figure 6.

We apply transformation τ to the grammar G_{lc} associated with G to obtain the cfg H. $L(H)$ should be $L(G_{lc})$. Then we define the grammar H' and a homomorphism ϕ_τ such that $L(H') = L(G)$ and moreover $H'[l/lc]G$ with respect to ϕ_τ. Finally, we construct the cfg G' from H' in such a way that $G' = \tau(G)$ and define a homomorphism ψ such that $G'[l/l]H'$. By the transitivity property of the cover relation we obtain the desired result.

We first show that τ preserves the language generated by the cfg to which it is applied.

Lemma 5.1. *Let $G = (N,\Sigma,P,S)$ be a context-free grammar. Then $L(G) = L(\tau(G))$.*

Proof: From the construction of $\tau(G)$ from G, it follows that for all $A \in \overline{N}$ (for the meaning of $\overline{N}$ see the transformation):

$$A \Rightarrow_l A_1\alpha_1 \Rightarrow_l A_2\alpha_2\alpha_1 \Rightarrow_l \cdots \Rightarrow_l A_{n-1}\alpha_{n-1}\cdots\alpha_2\alpha_1 \Rightarrow_l a\,\alpha_n\alpha_{n-1}\cdots\alpha_2\alpha_1, \quad (5.1.1)$$

$a \in \Sigma_\epsilon$, is a derivation in G if and only if

$$\begin{aligned} A &\Rightarrow_r a[A,a] \Rightarrow_r a\,\alpha_n[A,A_{n-1}] \Rightarrow_r a\,\alpha_n\alpha_{n-1}[A,A_{n-2}] \\ &\Rightarrow_r a\,\alpha_n\alpha_{n-1}\cdots\alpha_2\alpha_1[A,A] \\ &\Rightarrow_r a\,\alpha_n\alpha_{n-1}\cdots\alpha_2\alpha_1 \end{aligned} \quad (5.1.2)$$

is a derivation in $\tau(G)$. Notice that each symbol that occurs in $\alpha_n\cdots\alpha_2\alpha_1$, is either a terminal symbol or a symbol in $\overline{N}$.

Any derivation $A \Rightarrow_l^* x$ in G can be seen to be constructed from derivations of the form (5.1.1). And with these derivations there are corresponding derivations in $\tau(G)$ of the form (5.1.2). As a special case we have $S \Rightarrow^* x$ in G if and only if $S \Rightarrow^* x$ in $\tau(G)$. Thus $L(G) = L(\tau(G))$. So we can proof the Lemma by induction on this construction. We do not give the complete proof here.

As a basis for the induction consider the following observations. Let $A \in \overline{N}$ and let $A \to x$ be a production of G, with $x \in \Sigma^*$. First suppose that $x \neq \epsilon$. Let $x = ax'$ for some $a \in \Sigma$ and $x' \in \Sigma^*$. By the

definition of the transformation, it follows that in $\tau(G)$ there is a derivation $A \Longrightarrow a[A,a] \Longrightarrow ax'[A,A] \Longrightarrow ax' = x$. On the other hand this last derivation exists in $\tau(G)$ only if $A \rightarrow x$ is a production in G. Now suppose that $x = \epsilon$. Then in $\tau(G)$ we have the derivation $A \Longrightarrow [A,\epsilon] \Longrightarrow [A,A] \Longrightarrow \epsilon$. And also this last derivation only exists in $\tau(G)$ if $A \rightarrow \epsilon$ is a production in G. □

Let $G = (N,\Sigma,P,S)$ be a cfg, Δ a set of labeling symbols for the productions of G (λ denotes the corresponding labeling function), $G_{lc} = (N,\Sigma \cup \Delta,P_{lc},S)$ the grammar associated with G and Δ defined in Section 3 and H the grammar $\tau(G_{lc})$, i.e., the grammar obtained from G_{lc} by the transformation τ. $H = (N_H,\Sigma \cup \Delta,P_H,S)$. Furthermore, let $H' = (N_H \cup \Delta,\Sigma,P'_H,S)$, where $P'_H = P_H \cup \{p_i \rightarrow \epsilon \mid p_i \in \Delta\}$.

Lemma 5.2. *There is a leftmost derivation $A \Longrightarrow_l^{\pi} x$ for a terminal string x in H' if and only if there is a string y in $L(H)$ such that $h_\Delta(y) = x$, $h_\Sigma(y) = \pi_1\pi_2...\pi_n$ and $\pi'_1\pi_1\pi'_2\pi_2 \cdots \pi'_n\pi_n = \pi$, where $\pi'_1...\pi'_n$ is a left parse of y from A in H.*

Proof: By induction on the length of π. □

We now define a homomorphism ϕ_τ which should map left parses of a string x with respect to H' onto left-corner parses of x with respect to the original grammar $G = (N,\Sigma,P,S)$. Let Δ'_H be a set of labeling symbols for the productions in $P_{H'}$. Let λ' be a one-to-one labeling function from P' onto Δ'_H. Define the homomorphism ϕ_τ from Δ'^*_H to Δ^* as follows. For all productions in P_H of the form $[A,X] \rightarrow \beta[A,B]$, where $\beta \in (V \cup \Delta)^*$, introduced in step 2 of the transformation applied to G_{lc} we have: $\phi_\tau(\lambda'([A,X] \rightarrow \beta[A,B])) = \lambda(B \rightarrow X\beta)$. For all other elements q of Δ'_H: $\phi_\tau(q) = \epsilon$.

Lemma 5.3. *$H'[l/lc]G$ with respect to homomorphism ϕ_τ.*

Proof: By definition of a left-corner parse of a string $x \in L(G)$ with respect to G we have to show that $S \Longrightarrow_l^{\pi} x$ in H' if and only if $S \Longrightarrow^* y$ in G_{lc}, where $h_\Delta(y) = x$ and $h_\Sigma(y) = \phi_\tau(\pi)$.

By the previous Lemma there is a left parse π of $x \in L(H')$ if and only if there is a string y in $L(H)$ such that $h_\Delta(y) = x$ and $h_\Sigma(y) = \pi_1\pi_2...\pi_n$ and $\pi'_1\pi_1\pi'_2\pi_2 \cdots \pi'_n\pi_n = \pi$, where $\pi'_1...\pi'_n$ is a left parse of y in H. Since $L(H) = L(G_{lc})$ by Lemma 5.1, we are done if we could prove the following Claim.

Claim. $\phi_\tau(\pi) = \pi_1\pi_2...\pi_n$

Proof of the Claim: Recall that $\pi = \pi'_1\pi_1\pi'_2\pi_2 \cdots \pi'_n\pi_n$ and notice that $|\pi'_i| \geqslant 0$ and $|\pi_i| = 1$ for all i, $1 \leqslant i \leqslant n$. Since $\phi_\tau(\pi_i) = \epsilon$, we have to show that $\phi_\tau(\pi'_1\pi'_2...\pi'_n) = \pi_1...\pi_n$. If G does not contain ϵ-productions then simply $\phi_\tau(\pi'_i) = \pi_i$. Let $B \rightarrow \epsilon$ be an ϵ-production in P and let $\lambda(B \rightarrow \epsilon) = p$. Then $B \rightarrow p$ is a production in P_{lc}. Suppose that $A \rightarrow p[A,p]$ is a production of $H = \tau(G_{lc})$ introduced in step 1 of the construction of H. Then $[A,p] \rightarrow [A,B]$ is a production of H introduced in step 2 of the construction of H. By

definition of ϕ_τ, $\phi_\tau(\lambda'(A \to p\,[A,p\,])) = \epsilon$ and $\phi_\tau(\lambda'([A,p\,] \to [A,B\,])) = p$. Let π_j in π denote the production $p \to \epsilon$ in P'_H. Then π'_{j+1} equals $\lambda'([A,p\,] \to [A,B\,])$. Thus $\phi_\tau(1{:}\pi'_{j+1}) = \pi_j$.
End of the proof of the Claim. □

Now we should obtain $\tau(G)$ from H'. Define $G' = (N_H, \Sigma, P'_G, S)$ where $P'_G = \{ A \to h_\Delta(\alpha) \mid A \to \alpha \text{ in } P_H \}$. Let Δ'_G be the production label set for G' and let λ'_G denote a label function which satisfies the following: for all productions in P'_G, $\lambda'_G(A \to h_\Delta(\alpha)) = \lambda'(A \to \alpha)$.

It is not difficult to see that $G' = \tau(G)$ (Equality is meant here up to renaming of some nonterminal symbols). Thus we have: $L(G') = L(\tau(G)) = L(G) = L(H')$.

Define the homomorphism ψ from Δ'^*_G to Δ'^*_H as follows:

$\psi(\lambda'_G(A \to h_\Delta(\alpha))) = \lambda'(A \to \alpha)$, if $\alpha = h_\Delta(\alpha)$.

$\psi(\lambda'_G(A \to h_\Delta(\alpha))) = \lambda'(A \to \alpha)\lambda'(p_i \to \epsilon)$, if $1{:}\alpha = p_i \in \Delta$.

Lemma 5.4. *$G'[l/l]H'$ with respect to ψ.*

Proof: This follows immediately from the construction of grammar G' and the definition of ψ. □

Theorem 5.5. *$G'[l/lc]G$.*

Proof: Use Lemma 5.3 and Lemma 5.4 and the transitivity property of the cover relation. □

Example 5.6. See Example 4.2 in the previous section. Let Δ' be the set $\{1',2',\ldots,10'\}$ of unique labeling symbols of the productions in G' and let Δ be the set $\{1,2,3,4,5\}$ of unique labeling symbols of the productions in G. Define the homomorphism ϕ from Δ' into Δ^* as follows. $\phi(1') = \epsilon$, $\phi(2') = 4$, $\phi(3') = 2$, $\phi(4') = 3$, $\phi(5') = 1$, $\phi(6') = \epsilon$, $\phi(7') = \epsilon$, $\phi(8') = 4$, $\phi(9') = \epsilon$ and $\phi(10') = 3$.

The sentence $id \times id + id$ has left-parse $1'2'4'3'5'7'8'9'$ with respect to G'. This left-parse is mapped by ϕ on the left-corner-parse $4\,3\,2\,1\,4$ of $id \times id + id$ with respect to G. □

Example 5.7. See Example 4.3 in the previous section. With respect to grammar H the sentences *aacbaa* and *aacaab* have left-parses $1'3'1'3'2'4'5'6'7'7'7'$ and $1'3'1'3'2'4'7'7'5'6'7'$. The homomorphism ϕ_τ, given by the second column in Figure 5, maps both left-parses onto the left-corner parse $1\,1\,3\,4\,2$ with respect to G. □

Before we can present the following result we recall the definition of cycle-freeness of a context-free grammar.

A cfg $G = (N, \Sigma, P, S)$ is called *cycle-free* if for no $A \in N$ there is a derivation $A \Longrightarrow^+ A$ in G.

Theorem 5.8. *Any cycle-free context-free grammar is left-to-left-corner covered by a non-left-recursive grammar.*

Proof: The only thing left to show is that cycle-freeness of a cfg G implies non-left-recursiveness of $\tau(G)$.

Let $G = (N,\Sigma,P,S)$. First notice that a symbol in N cannot be left-recursive in $\tau(G)$. We show that there is in G a derivation

$$Y \Rightarrow^+ Z, \tag{5.8.1}$$

where $Y,Z \in N$, if there is in $\tau(G)$ a derivation of the form

$$[A,Z] \Rightarrow_r^+ [A,Y]x, \tag{5.8.2}$$

where $x \in \Sigma^*$. We use induction on the length of derivation (5.8.2). Let $[A,Z] \Rightarrow [A,Y]x$ be a derivation in $\tau(G)$. It follows from the construction of $\tau(G)$, that $Y \rightarrow Z$ is a production in P (and $x = \epsilon$).

Suppose that if there is a derivation in $\tau(G)$ of the form (5.8.2) with length less than or equal to n, then there is a derivation (5.8.1) in G. Consider a derivation of the form (5.8.2) with length $n+1$. This derivation has the form:

$$[A,Z] \Rightarrow_r^+ [A,X] \Rightarrow_r^+ [A,Y]z.$$

By the induction hypothesis we may conclude that $Y \Rightarrow^+ X$ and $X \Rightarrow^+ Z$ are derivations in G. Thus $Y \Rightarrow^+ Z$ in G. □

6. Two Classes of Grammars

In this section we give definitions of $LL(k)$ grammars and left-corner or $LC(k)$ grammars.

Definition 6.1. Let G be a cfg. Let $A \in N$, $\alpha,\beta,\gamma \in V^*$, $w \in \Sigma^*$ and let $A \rightarrow \alpha$ and $A \rightarrow \beta$ be two distinct productions of G. G is an *LL(k) grammar* if the conditions

(i) $S \Rightarrow_l^* wA\delta \Rightarrow_l w\alpha\delta \Rightarrow^* wz_1$

(ii) $S \Rightarrow_l^* wA\delta \Rightarrow_l w\beta\delta \Rightarrow^* wz_2$

(iii) $k{:}z_1 = k{:}z_2$

always imply that $\alpha = \beta$. □

In the following definition of the class of $LC(k)$ grammars, the notion of a *left-corner sentential form (lcsf)* is used. Informally, a left-corner sentential form is a left sentential form $uY\alpha$ such that the nonterminal or terminal symbol Y is not the left-corner of the production that introduced Y in the leftmost derivation $S \Rightarrow_l^* uY\alpha$. If $uY\alpha$ is a left-corner sentential form, we write: $S \Rightarrow_{lc}^* uY\alpha$. Formally, $S \Rightarrow_{lc}^* uY\alpha$ if and only if either this derivation has the form:

$$S \Rightarrow_l^* u'B\alpha' \Rightarrow_l u'\gamma_1 Y\gamma_2\alpha' \Rightarrow_l^* u'u''Y\gamma_2\alpha' = uY\alpha,$$

where $B \rightarrow \gamma_1 Y\gamma_2$ is a production rule in which $\gamma_1 \neq \epsilon$, or $uY\alpha = S$ (The first left sentential form in any derivation). We write $\underline{A} \Rightarrow_l^* \underline{B}\gamma$ if for some integer $n \geqslant 0$ there are B_i in N, γ_i in V^*, with $B_0 = A$, $B_n = B$ and $\gamma_n \ldots \gamma_1 = \gamma$, such that

$$A \Rightarrow_l B_1\gamma_1 \Rightarrow_l B_2\gamma_2\gamma_1 \Rightarrow_l \cdots \Rightarrow_l B_n\gamma_n \ldots \gamma_1.$$

Definition 6.2. Let $G = (N,\Sigma,P,S)$ be a cfg and let k be an integer $(k > 0)$. G is an $LC(k)$ grammar if and only if

1. the conditions

   (i) $S \Rightarrow_{lc}^* u_1\underline{A}\delta_1 \Rightarrow_l^* u_1\underline{B}_1\gamma_1\delta_1 \Rightarrow_l u_1X\beta_1\gamma_1\delta_1$

   $\Rightarrow_l^* u_1x_1\beta_1\gamma_1\delta_1 \Rightarrow_l^* u_1x_1z_1$

   (ii) $S \Rightarrow_{lc}^* u_2\underline{A}\delta_2 \Rightarrow_l^* u_2\underline{B}_2\gamma_2\delta_2 \Rightarrow_l u_2X\beta_2\gamma_2\delta_2$

   $\Rightarrow_l^* u_2x_2\beta_2\gamma_2\delta_2 \Rightarrow_l^* u_2x_2z_2$

   (iii) $u_1x_1 = u_2x_2$ and $k{:}z_1 = k{:}z_2$

   imply $B_1 = B_2$ and $\beta_1 = \beta_2$.

   Notice that if X is a terminal symbol then $X = x_1 = x_2$ and condition (iii) implies that $u_1 = u_2$. If $X\beta_1 = \epsilon$ then $X\beta_2 = \epsilon$, $u_1 = u_2$ and $x_1 = x_2 = \epsilon$.

2. (condition for ϵ-rules) If

   $$S \Rightarrow_{lc}^* u\underline{A}\delta_1 \Rightarrow_l^* u\underline{B}_1\gamma_1\delta_1 \Rightarrow_l u\gamma_1\delta_1 \Rightarrow_l^* uz_1$$

   is a derivation in G, then there is no derivation of the form:

   $$S \Rightarrow_{lc}^* u\underline{A}\delta_2 \Rightarrow_l^* u\underline{B}_2\gamma_2\delta_2 \Rightarrow_l ua\beta\gamma_2\delta_2 \Rightarrow_l^* uaz_2$$

   in G, such that $k{:}z_1 = k{:}az_2$.

3. (condition for left recursive nonterminal symbols) If

   $$S \Rightarrow_{lc}^* u_1\underline{A}\delta_1 \Rightarrow_l^* u_1\underline{B}_1\gamma_1\delta_1 \Rightarrow_l u_1A\beta_1\gamma_1\delta_1$$

   $$\Rightarrow_l^* u_1x_1\beta_1\gamma_1\delta_1 \Rightarrow_l^* u_1x_1z_1$$

   is a derivation in G, then there is no derivation of the form

   $$S \Rightarrow_{lc}^* uA\delta \Rightarrow_l^* ux\delta \Rightarrow_l^* uxz$$

   such that $ux = u_1x_1$ and $k{:}z_1 = k{:}z$. □

This definition is equivalent with the definition of $LC(k)$ grammars in terms of right-most derivations given by Soisalon-Soininen [11]. For a proof of this equivalence see [2]. Any $LL(k)$ grammar is $LC(k)$ and any $LC(k)$ grammar is $LR(k)$ [11].

Example 6.3. Grammar G in Example 4.2 is an $LC(1)$ grammar. □

7. Special Properties of Transformation τ

In this section we show that the transformation τ yields an $LL(k)$ grammar if and only if it is applied to an $LC(k)$ grammar.

Lemma 7.1. *Let $G = (N,\Sigma,P,S)$ be a context-free grammar. For all $A_0 \in \overline{N}$ (For the meaning of $\overline{N}$ see Section 4.), there exists in G a derivation*

$$\begin{aligned} A_0 &\Rightarrow_l^* B_1\delta_1 \Rightarrow_l \alpha_1 A_1\gamma_1\delta_1 \Rightarrow_l^* w_1 A_1\gamma_1\delta_1 \\ &\Rightarrow_l^* w_1 B_2\delta_2\gamma_1\delta_1 \Rightarrow_l w_1\alpha_2 A_2\gamma_2\delta_2\gamma_1\delta_1 \\ &\Rightarrow_l^* w_1 w_2 A_2\gamma_2\delta_2\gamma_1\delta_1 \Rightarrow_l^* \cdots \\ &\Rightarrow_l^* w_1 w_2 \cdots w_n A_n\gamma_n\delta_n \cdots \gamma_1\delta_1, \end{aligned}$$

if and only if in grammar $\tau(G)$ the derivation

$$\begin{aligned} A_0 &\Rightarrow_l^* w_1 A_1\gamma_1[A_0,B_1] \Rightarrow_l^* w_1 w_2 A_2\gamma_2[A_1,B_2]\gamma_1[A_0,B_1] \\ &\Rightarrow_l^* \cdots \Rightarrow_l^* \\ &\Rightarrow_l^* w_1 w_2 \cdots w_n A_n\gamma_n[A_{n-1},B_n]\gamma_{n-1}[\ldots]\ldots\gamma_1[A_0,B_1], \end{aligned}$$

exists, such that for all i, if $1 \leqslant i \leqslant n$ then

$$[A_{i-1},B_i] \Rightarrow_r^* \delta_i[A_{i-1},A_{i-1}] \Rightarrow_r \delta_i.$$

Proof: By induction on the length of the derivations. □

Lemma 7.2. *For any $k > 0$, if G is an $LC(k)$ grammar, then $\tau(G)$ is an $LL(k)$ grammar.*

Proof: Let $G = (N,\Sigma,P,S)$ be an $LC(k)$ grammar for some $k > 0$. G' denotes the grammar $\tau(G)$. Suppose that G' is not an $LL(k)$ grammar. Then for some $Z \in N'$,

$$S \Rightarrow_l^* wZ\delta \Rightarrow_l w\omega_1\delta \Rightarrow^* wz_1$$

and

$$S \Rightarrow_l^* wZ\delta \Rightarrow_l w\omega_2\delta \Rightarrow^* wz_2$$

are derivations in G', where $\omega_1 \neq \omega_2$, although $k{:}z_1 = k{:}z_2$.

We distinguish three cases: I) Z is a symbol in N, II) Z is of the form $[A,Y]$, where $A \in N$, $Y \in V-\{A\}$ and III) Z is of the form $[A,A]$.

Case I. It follows from the construction of the grammar G', that the productions $Z \to \omega_1$ and $Z \to \omega_2$ both have the form $A \to a[A,a]$, where $a \in \Sigma_\epsilon$. Since $k > 0$, $k{:}z_1 = k{:}z_2$ and $\gamma_1 \neq \gamma_2$, the following derivations exist in G'.

$$S \Rightarrow_l^* wA\delta \Rightarrow_l wa[A,a]\delta \Rightarrow_l^* wau_1\delta \Rightarrow_l^* wau_1v_1 = wz_1$$

$$S \Rightarrow_l^* wA\delta \Rightarrow_l w[A,\epsilon]\delta \Rightarrow_l^* wu_2\delta \Rightarrow_l^* wu_2v_2 = wz_2$$

Because of the first part of these derivation in G' we may conclude, using Lemma 7.1, that $S \Rightarrow_{lc}^* wA\delta'$ in G such that $\delta \Rightarrow^* \delta'$ in G'.

Notice that in general we may not conclude from $\delta \Rightarrow^* \delta'$ and $\delta \Rightarrow^* v$ that $\delta' \Rightarrow^* v$. However, by Lemma 7.1 the derivation $\delta \Rightarrow^* \delta'$ has a special form.

If in Lemma 7.1 $[A_{i-1},B_i] \Rightarrow^* y_1$ then $[A_{i-1},B_i] \Rightarrow_r^* y_1$ and thus $[A_{i-1},B_i] \Rightarrow_r^* \delta_i[A_{i-1},A_{i-1}] \Rightarrow_r^* \delta_i \Rightarrow_r^* y_1$. Therefore we may conclude here that $\delta' \Rightarrow^* v_1$ and $\delta' \Rightarrow^* v_2$ in G. Since $A \to a[A,a]$ is a production in G', it follows from the construction of G' that $\underline{A} \Rightarrow_l^* \underline{B}_1\gamma_1 \Rightarrow a\beta_1\gamma_1$ is a derivation in G. In the same way, since $A \to [A,\epsilon]$ is a production of G', there is a derivation $\underline{A} \Rightarrow_l^* \underline{B}_2\gamma_2 \Rightarrow \gamma_2$ in G. Furthermore we know (See the proof of Lemma 5.1.) that $a\beta_1\gamma_1 \Rightarrow^* au_1$ and $\gamma_2 \Rightarrow^* u_2$ in G. Thus derivations

$$S \Rightarrow_{lc}^* w\underline{A}\delta' \Rightarrow_l^* w\underline{B}_2\gamma_2\delta' \Rightarrow_l w\gamma_2\delta' \Rightarrow_l^* wu_2v_2 = wz_2$$

and

$$S \Rightarrow_{lc}^* w\underline{A}\delta' \Rightarrow_l^* w\underline{B}_1\gamma_1\delta' \Rightarrow_l wa\beta_1\gamma_1\delta' \Rightarrow_l^* wau_1v_1 = wz_1$$

exist in G. Since $k{:}z_1 = k{:}z_2$, we conclude that G does not satisfy the condition for ϵ-productions in Definition 6.2 and so we have shown that the assumption that G is $LC(k)$ leads to a contradiction.

Case II. In this case the productions $Z \to \omega_1$ and $Z \to \omega_2$ in the introduction of this proof have the form $[A,Y] \to \gamma_1[A,Y_1]$ and $[A,Y] \to \gamma_2[A,Y_2]$. Suppose that

$$S \Rightarrow_l^* w[A,Y]\delta \Rightarrow_l w\gamma_1[A,Y_1]\delta \Rightarrow_l^* wu_1[A,Y_1]\delta$$
$$\Rightarrow_l^* wu_1y_1\delta \Rightarrow wu_1y_1v_1 = wz_1$$

and

$$S \Rightarrow_l^* w[A,Y]\delta \Rightarrow_l w\gamma_1[A,Y_1]\delta \Rightarrow_l^* wu_1[A,Y_1]\delta$$
$$\Rightarrow_l^* wu_1y_1\delta \Rightarrow wu_1y_1v_1 = wz_1$$

are derivations in G', where $\gamma_1[A,Y_1] \neq \gamma_2[A,Y_2]$, although $k{:}z_1 = k{:}z_2$. From the construction of G' it follows that the first part of these derivations has the form: $S \Rightarrow_l^* w'A\,\delta \Rightarrow_l^* w'w''[A,Y]\delta$, where $w = w'w''$ and $Y \Rightarrow^* w''$ is a derivation in G. By Lemma 7.1 we know that $S \Rightarrow_{lc}^* w'A\,\delta'$ is a derivation in G, such that $\delta \Rightarrow^* \delta'$ in G'. It will be clear that in G the derivations

$$\underline{A} \Rightarrow_l^* \underline{Y}_1\beta_1 \Rightarrow Y\gamma_1\beta_1 \Rightarrow_l^* w''\gamma_1\beta_1$$

and

$$\underline{A} \Rightarrow_l^* \underline{Y}_2\beta_2 \Rightarrow Y\gamma_2\beta_2 \Rightarrow_l^* w''\gamma_2\beta_2$$

exist, such that $[A,Y_1] \Rightarrow_r^* \beta_1$ and $[A,Y_2] \Rightarrow_r^* \beta_2$ in G'. Notice that we may conclude that $\beta_1 \Rightarrow^* y_1$ and also that $\beta_2 \Rightarrow^* y_2$ (see Case I for the justification of this). Thus we know that

$$S \Rightarrow_{lc}^* w'\underline{A}\delta' \Rightarrow_l^* w'\underline{Y}_1\beta_1\delta' \Rightarrow w'Y\gamma_1\beta_1\delta'$$

$$\Rightarrow w'w''\gamma_1\beta_1\delta' \Rightarrow^* w'w''u_1y_1v_1 = w'w''z_1 \qquad (7.2.1)$$

and

$$S \Rightarrow_{lc}^* w'\underline{A}\,\delta' \Rightarrow_l^* w'\underline{Y}_2\beta_2\delta' \Rightarrow w'Y\gamma_2\beta_2\delta'$$
$$\Rightarrow w'w''\gamma_2\beta_2\delta' \Rightarrow^* w'w''u_2y_2v_2 = w'w''z_2 \qquad (7.2.2)$$

are derivations in G. Since $k{:}z_1 = k{:}z_2$ we have $k{:}w''z_1 = k{:}w''z_2$. From this last equality and derivations (7.2.1) and (7.2.2) we conclude that clause 1 of Definition 6.2 is not satisfied. This, however, contradicts the assumption that G is $LC(k)$.

Case III. We consider the case in which the productions $Z \to \omega_1$ and $Z \to \omega_2$ in the introduction of this proof have the form $[A,A] \to \epsilon$ and $[A,A] \to \beta[A,B]$. Suppose that in G' derivations

$$S \Rightarrow_l^* w[A,A]\delta \Rightarrow_l w\delta \Rightarrow_l^* wz_2 \qquad (7.2.3)$$

and

$$S \Rightarrow_l^* w[A,A]\delta \Rightarrow_l w\beta[A,B]\delta \Rightarrow_l^* wy_1[A,B]\delta$$
$$\Rightarrow_l^* wy_1v_1\delta \Rightarrow wy_1v_1z_1 \qquad (7.2.4)$$

exists, such that $k{:}z_2 = k{:}y_1v_1z_1$.

From the construction of G' it follows that the first part of derivations (7.2.3) and (7.2.4) has the form

$$S \Rightarrow_l^* w'A\,\delta \Rightarrow_l^* w'w''[A,A]\delta$$

and $A \Rightarrow^* w''$ in G. By Lemma 7.1 we know that in G derivation $S \Rightarrow_{lc}^* w'A\,\delta'$ exists, such that $\delta \Rightarrow^* \delta'$ in G'. From derivation (7.2.4) we conclude that in G the derivation

$$\underline{A} \Rightarrow_l^* \underline{B}\gamma \Rightarrow_l A\beta\gamma$$

exist, where $[A,B] \Rightarrow_r^* \gamma$ in G'. We may conclude that $\gamma \Rightarrow^* v_1$.

Thus in G derivations

$$S \Rightarrow_{lc}^* w'A\,\delta' \Rightarrow_l^* w'w''\,\delta' \Rightarrow^* w'w''z_2$$

and

$$S \Rightarrow_{lc}^* w'\underline{A}\,\delta' \Rightarrow_l^* w'\underline{B}\delta' \Rightarrow_l w'A\,\beta\gamma\delta' \Rightarrow_l^* w'w''\,\beta\gamma\delta'$$
$$\Rightarrow_l^* w'w''y_1\gamma\delta' \Rightarrow_l^* w'w''y_1v_1z_1$$

exist. Since $k{:}z_2 = k{:}y_1v_1z_1$ we conclude from these derivations that G doesn't satisfy clause 3 in Definition 6.2. This contradicts the assumption that G is $LC(k)$.

We finally conclude that G' must be $LL(k)$. □

We now show the converse of Lemma 7.2.

Lemma 7.3. *Let G be a context-free grammar. For any $k > 0$, if G is not an $LC(k)$ grammar, then $\tau(G)$ is not an $LL(k)$ grammar.*

Proof: Let $G = (N,\Sigma,P,S)$ be a context-free grammar which is not $LC(k)$. G' will denote $\tau(G)$.

Case I. Suppose that G does not satisfy clause 1 in the definition of $LC(k)$ grammars. Then there exist derivations

$$S \Rightarrow_{lc}^* u_1\underline{A}\delta_1 \Rightarrow_l^* u_1\underline{B}_1\gamma_1\delta_1 \Rightarrow_l u_1X\beta_1\gamma_1\delta_1$$
$$\Rightarrow_l^* u_1x_1\beta_1\gamma_1\delta_1 \Rightarrow_l^* u_1x_1y_1v_1z_1 \qquad (7.3.1)$$

and

$$S \Rightarrow_{lc}^* u_2\underline{A}\delta_2 \Rightarrow_l^* u_2\underline{B}_2\gamma_2\delta_2 \Rightarrow_l u_2X\beta_2\gamma_2\delta_2$$
$$\Rightarrow_l^* u_2x_2\beta_2\gamma_2\delta_2 \Rightarrow_l^* u_2x_2y_2v_2z_2 \qquad (7.3.2)$$

in G, where $B_1 \neq B_2$ or $\beta_1 \neq \beta_2$, although $u_1x_1 = u_2x_2$ and $k{:}y_1v_1z_1 = k{:}y_2v_2z_2$.

Consider derivation (7.3.1). By Lemma 7.1 we conclude from the first part of this derivation that in G' the derivation $S \Rightarrow_l^* u_1A\delta'_1$ exists, such that $\delta'_1 \Rightarrow^* \delta_1$. From the second and third part of the derivation we may conclude that $[A,X] \rightarrow \beta_1[A,B_1]$ is a production of G' and $[A,B_1] \Rightarrow^* \gamma_1$ in G'. Moreover, we may conclude that $A \Rightarrow_l^* x_1[A,X]$ in G'. Similar conclusions can be derived from derivation (7.3.2) in G. Thus in G' the derivations

$$S \Rightarrow_l^* u_1A\delta'_1 \Rightarrow_l^* u_1x_1[A,X]\delta'_1 \Rightarrow u_1x_1\beta_1[A,B_1]\delta'_1$$
$$\Rightarrow_l^* u_1x_1y_1[A,B_1]\delta'_1 \Rightarrow_l^* u_1x_1y_1v_1z_1$$

and

$$S \Rightarrow_l^* u_2A\delta'_2 \Rightarrow_l^* u_2x_2[A,X]\delta'_2 \Rightarrow u_2x_2\beta_2[A,B_2]\delta'_2$$
$$\Rightarrow_l^* u_2x_2y_2[A,B_2]\delta'_2 \Rightarrow_l^* u_2x_2y_2v_2z_2$$

exist. Since $B_1 \neq B_2$ or $\beta_1 \neq \beta_2$, the productions $[A,X] \rightarrow \beta_1[A,B_1]$ and $[A,X] \rightarrow \beta_2[A,B_2]$, used in these derivations, are not the same. Because of the equalities $u_1x_1 = u_2x_2$ and $k{:}y_1v_1z_1 = k{:}y_2v_2z_2$, we conclude that G' is not $LL(k)$.

Case II. Suppose that G does not satisfy clause 2 in the definition of $LC(k)$ grammars. Then there exist derivations

$$S \Rightarrow_{lc}^* u\underline{A}\delta_1 \Rightarrow_l^* u\underline{B}_1\gamma_1\delta_1 \Rightarrow_l u\gamma_1\delta_1$$
$$\Rightarrow_l^* uy_1\delta_1 \Rightarrow_l^* uy_1z_1 \qquad (7.3.3)$$

and

$$S \Rightarrow_{lc}^* u\underline{A}\delta_2 \Rightarrow_l^* u\underline{B}_2\gamma_2\delta_2 \Rightarrow_l ua\beta\gamma_2\delta_2$$
$$\Rightarrow_l^* ua\beta_2\gamma_2\delta_2 \Rightarrow_l^* uavy_2z_2 \qquad (7.3.4)$$

in G, such that $k{:}y_1z_1 = k{:}avy_2z_2$.

From derivation (7.3.3) we conclude that in G' derivation $S \Rightarrow_{lc}^* uA\delta'_1$ exists, such that $\delta'_1 \Rightarrow^* \delta_1$. In addition, $A \rightarrow [A,\epsilon]$ and $[A,\epsilon] \rightarrow [A,B_1]$ are productions of G' and $[A,B_1] \Rightarrow^* \gamma_1$ in G'. From derivation (7.3.4) we conclude that in G' derivation $S \Rightarrow_{lc}^* uA\delta'_2$ exists, such that $\delta'_2 \Rightarrow^* \delta_2$. Moreover, $A \rightarrow a[A,a]$ and $[A,a] \rightarrow \beta[A,B_2]$ are productions of G' and $[A,B_2] \Rightarrow^* \gamma_2$ in G'. Thus in G' derivations

$$S \Rightarrow_l^* uA\delta'_1 \Rightarrow_l [A,\epsilon]\delta'_1 \Rightarrow_l^* u[A,B_1]\delta'_1 \Rightarrow^* u\gamma_1\delta'_1 \Rightarrow^* uy_1z_1$$

and

$$\begin{aligned} S &\Rightarrow_l^* uA\delta'_2 \Rightarrow_l ua[A,a]\delta'_2 \Rightarrow_l^* ua\beta[A,B_2]\delta'_2 \\ &\Rightarrow_l^* uav[A,B_2]\delta'_2 \Rightarrow_l^* uavy_2\delta'_2 \Rightarrow_l^* uavy_2z_2 \end{aligned}$$

exist. Since $k{:}avy_2z_2 = k{:}y_1z_1$ we conclude that G' is not $LL(k)$.

Case III. Suppose that in G the derivations

$$\begin{aligned} S &\Rightarrow_{lc}^* u_1\underline{A}\delta_1 \Rightarrow_l^* u_1\underline{B}_1\gamma_1\delta_1 \Rightarrow_l u_1A\beta_1\gamma_1\delta_1 \\ &\Rightarrow_l^* u_1x_1\beta_1\gamma_1\delta_1 \Rightarrow_l^* u_1x_1y_1v_1z_1 \end{aligned} \tag{7.3.5}$$

and

$$S \Rightarrow_{lc}^* uA\delta \Rightarrow_l^* ux\delta \Rightarrow_l^* uxz$$

exist, such that $ux = u_1x_1$ and $k{:}u_1y_1z_1 = k{:}z$. In a way similar as in the other two cases we may conclude from these derivations that the derivations

$$\begin{aligned} S &\Rightarrow_l^* u_1A\delta'_1 \Rightarrow_l^* u_1x_1[A,A]\delta'_1 \Rightarrow u_1x_1\beta_1[A,B_1]\delta'_1 \\ &\Rightarrow_l^* u_1x_1y_1[A,B_1]\delta'_1 \Rightarrow_l^* u_1x_1y_1v_1z_1 \end{aligned}$$

and

$$S \Rightarrow_l^* uA\delta' \Rightarrow_l^* ux[A,A]\delta' \Rightarrow_l ux\delta' \Rightarrow^* uxz$$

exist in G'. Since $ux = u_1x_1$ and $k{:}z = k{:}y_1v_1z_1$ we conclude that G' is not $LL(k)$. □

From Lemma 7.2 and Lemma 7.3 we may conclude the following result.

Theorem 7.4. *The transformation τ yields an $LL(k)$ grammar $(k>0)$ if and only if it is applied to an $LC(k)$ grammar.* □

8. Semantical Covering of Attribute Grammars

In the introduction we already noticed that it is sometimes possible to evaluate attributes of the nodes of the parse tree during parsing. Therefore it makes some sense to consider transformations on attribute grammars which yield a semantically equivalent attribute grammar based on a cfg which covers the original cfg. It lies at hand to call the

result of such a transformation a *semantical covering grammar* of the original attribute grammar. Here we consider the question whether all translations definable by a class X of attribute grammars can also be defined by attribute grammars in a class Y which semantically covers class X. A similar question is posed by Aho and Ullman for syntax-directed translation schemes [1]: Suppose that G_2 left or right covers G_1. Is every SDTS with G_1 as underlying grammar equivalent to an SDTS with G_2 as underlying grammar? A partial answer to this question is given by Shyamasundar in [14]. For special classes of SDTS this question has been studied by Rosenkrantz and Lewis II [9] and by Soisalon-Soininen [11]. These studies consider the semantical cover relation between classes of simple syntax directed translation schemes (simple SDTS) [1]. SDTS can be viewed as a special class of attribute grammars (See Filé [13] for a precise description of SDTS in the formalism of attribute grammars).

The semantic equivalence of covering attribute grammars is also studied by Bochmann [4]. However, the notion of semantical covering introduced by Bochmann is quite different from the semantical cover relation we have in mind. Let us first introduce some notions and notation.

Let G be an attribute grammar (AG). The reader is referred to Filé [13] for a definition. Notice that the specification of the semantic domain, that is a set of sets of values of the attributes together with the set of functions denoted by the evaluation rules of the attributes, is a part of the definition of an AG. Let δ denote the distinguished synthesized attribute of the start symbol of the AG G. In an evaluated complete grammatical tree (parse tree), δ is a special attribute of the root of the tree of which the value represents the meaning or the translation of the yield of the tree. Let D denote the value set of δ. Let Tr denote the set of complete grammatical trees of the underlying cfg G_0 of G. Let $\delta(t) \in D$ denote the value of δ of tree $t \in Tr$ and let the *translation of* $x \in L(G)$ be $\psi(x) = \{\delta(t) \mid t \in Tr \text{ and } \mathit{yield}(t) = x\}$. ψ is called the *translation function* of G. Even if G_0 is (syntactically) ambiguous $\psi(x)$ may contain only one element (We assume that the AG is non-circular so $\delta(t)$ is always defined). The translation $TRANS(G)$ defined by the AG G is:

$$TRANS(G) = \{(x, \psi(x)) \mid x \in L(G)\}.$$

Let 2^D denote the power set of D. The following definition is from Bochmann [4].

Definition 8.1. Let G_1 and G_2 be AG's over the same terminal alphabet, δ_1 and δ_2 the distinguished attributes of G_1 and G_2, respectively, D_1 and D_2 their value sets and ψ_1 and ψ_2 the translation functions. G_2 is *semantically finer then* G_1 if there exists a mapping $\phi : 2^{D_2} \to 2^{D_1}$, such that: for all $x \in L(G_1) \cap L(G_2)$, $\psi_1(x) \subseteq \phi(\psi_2(x))$. □

Thus G_2 is semantically finer then G_1 implies that for all x in both languages the translation according to G_1 can be obtained by applying the mapping ϕ to the translation according to G_2. Because of the similarity between this definition and the definition of cover Bochmann uses the phrase "semantical covering". In order to explain our idea of semantical covering a little more, we consider for a start the following definition.

Definition 8.2. An AG G_2 *semantically covers* an AG G_1 if

i. the underlying cfg of G_2 covers the underlying cfg of G_1,

ii. $TRANS(G_1) = TRANS(G_2)$. □

If an attribute grammar has a deterministically parsable underlying cfg and all attributes of all parse trees are evaluable during parsing following a parsing method suitable for the cfg, then we call the AG a *one-pass AG*. We are specially interested in one-pass AG's. All one-pass AG's are L-attributed, at least if we adopt the strict one-pass evaluation strategy as defined in [3]. L-attributed $LL(k)$ grammars are one-pass AG's since all attributes are evaluable during top-down parsing. For other classes of one-pass AG we refer to [3] where also the class of LC-attributed grammars is defined. Let X-AG and Y-AG be classes of one-pass AG's over a specific semantic domain. If we want to compare different classes of AG with respect to their ability to define translations (or their "formal power", cf. [12]) we must explicitly mention the semantic domain because this ability not only depends on the number of attributes and the kind of attribute dependencies in the AG but also on the types of attributes and the function types in the semantic domain. Knuth already showed in [5] that any translation defined by an AG can be defined by an AG which has only synthesized attributes. The question whether a class X-AG semantically covers a class Y-AG asks for a transformation which yields a cfg in class X when applied to a cfg in Y and a redefinition of attributes and attribute-rules such that the translation is preserved and the AG obtained is in X-AG. By Definition 8.2 semantically covering of attribute grammars does not imply any correspondence between attribute values of internal nodes of corresponding parse trees. However, if we want to show that an AG G_1 semantically covers an AG G_2, we need some inductive argument on the construction of corresponding derivation trees of the involved underlying cfgs. Therefore we can use a *stronger form of semantical covering* which implies the equality of attribute values of the roots of corresponding subtrees of corresponding parse trees. What "corresponding subtrees" are should follow from a particular transformation by means of which the covering grammar is obtained. For example the left factoring transformation applied to a grammar which is not left factored yields a left factored grammar that right covers the original grammar. If a grammar is not left factored then there exist productions $A \to \alpha\beta$ and $A \to \alpha\gamma$, with $\alpha \neq \epsilon$ and

$\beta \neq \gamma$. A step in the process of left factoring consists of replacing the productions $A \rightarrow \alpha\beta$ and $A \rightarrow \alpha\gamma$ by the productions $A \rightarrow \alpha H$, $H \rightarrow \beta$ and $H \rightarrow \gamma$, where H is a newly introduced nonterminal symbol. In [3] an informal algorithm is given for transforming AG's in the class *LP-AG* based on left-part grammars into the class *LL-AG*, the class of L-attributed $LL(k)$ grammars. This transformation is an attributed variant of the left factoring transformation. Here it is immediately clear what the corresponding subtrees of corresponding parse trees are.

If we want to consider semantical cover relations between AG's with semantical conditions or disambiguating predicates — which play a role in making parsing decisions based on attribute values [3] — Definition 8.2 is not suitable. In this case the language generated by the AG is a subset of the language generated by the underlying cfg so we cannot say anything about the existence of a cover relation between the underlying cfgs without considering attribute values of internal corresponding nodes.

References

1. A.V. Aho & J.D. Ullman: *The Theory of Parsing, Translation and Compiling*, Volume 1, Prentice-Hall, Englewood Cliffs, N.J., 1972.
2. R. op den Akker: A left-corner property for context-free grammars, Memorandum INF-86-8 (1986), Twente University of Technology, Department of Informatics, Enschede, The Netherlands.
3. R. op den Akker: Deterministic parsing of attribute grammars, Part I: Top-down oriented strategies, Memorandum INF-86-19 (1986), Part II: Left-corner strategies, (to appear), Twente University of Technology, Department of Informatics, Enschede, The Netherlands.
4. G.V. Bochmann: Semantic equivalence of covering attribute grammars, *Internat. J. Comp. Inform. Sci.*, **8** (1979) 523-539.
5. D.E. Knuth: Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145. Correction in: *Math. Systems Theory* **5** (1971) 95-96.
6. J.N. Gray & M.A. Harrison: On the covering and reduction problems for context-free grammars, *J. Assoc. Comput. Mach.* **19** (1972) 675-698.
7. H.B. Hunt & D.J. Rosenkrantz: Efficient algorithms for structural similarity of grammars, Conf. Record of the 7th ACM Symp. on Principles of Progr. Languages (1980) 213-219.
8. A. Nijholt: *Context-Free Grammars: Covers, Normal Forms and Parsing*, Lect. Notes Comp. Sci. **93** (1980), Springer-Verlag, Berlin, Heidelberg, New York.

9. D.J. Rosenkrantz & P.M. Lewis II: Deterministic left corner parsing, Conf. Rec. of the 11th Annual IEEE Symp. on Switching and Automata Theory (1970) 139-152.

10. E. Soisalon-Soininen: On the covering problem for left-recursive grammars, *Theoret. Comput. Sci.* **8** (1979) 1-11.

11. E. Soisalon-Soininen: Characterization of LL(k) languages by restricted LR(k) grammars, Report A-1977-3 (1977), Department of Computer Science, University of Helsinki, Helsinki.

12 J. Engelfriet & G. Filé: The formal power of one-visit attribute grammars, *Acta Inform.* **16** (1981) 275-302.

13. G. Filé: Machines for attribute grammars, *Inform. and Control* **69** (1986) 41-124.

14 R.K. Shyamasundar: On the covering of syntax-directed translations for context-free grammars, *Proc. Indian Acad. Sci.* **88 A** Part 3 (1979) 1-19.

Programming Language Concepts — The Lambda Calculus Approach

Maarten M. Fokkinga

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

The Lambda Calculus is a formal system, originally intended as a tool in the foundation of mathematics, but mainly used to study the concepts of algorithm and effective computability. Recently, the Lambda Calculus and related systems acquire attention from Computer Science for another reason too: several important programming language concepts can be explained elegantly and can be studied successfully in the framework of the Lambda Calculi. We show this mainly by means of examples. We address ourselves to interested computer scientists who have no prior knowledge of the Lambda Calculus. The concepts discussed include: parameterization, definitions, recursion, elementary and composite data types, typing, abstract types, control of visibility and life-time, and modules.

1. Introduction

The Lambda Calculus is a completely formally defined system, consisting of *expressions* (for functions or rather algorithms) and *rules* that prescribe how to evaluate the expressions. It has been devised in the thirties by Alonzo Church to study the concept of function (as a *recipe*) and to use it in the foundation of mathematics. This latter goal has not been achieved (although recent versions of the Lambda Calculus come quite close to it, see Martin-Löf [16], Coquand & Huet [6]); the former goal, the study of the concept of function, has led to significant contributions to the theory of effective computability.

Recently, the Lambda Calculus and related systems, together called Lambda Calculi, have aroused much interest from computer science because several important programming language concepts are present — or can be expressed faithfully — in them in the most pure form and without restrictions that are sometimes imposed in commercial programming languages. Expressing a programming language concept in the Lambda Calculus has the following benefits:

- It may shed some light upon the concept and thus give some insight.
- It may answer fundamental questions about the concept via theorems already available in the Lambda Calculus. And there are quite a lot of them.

- It proves that the concept does not add something essentially new to the language. In particular it guarantees that there can not be nasty interferences with other such concepts. (It has always been a problem in programming language design to combine concepts that are useful in isolation but have unexpected and undesired interferences when taken together.)

Some programming language concepts cannot be expressed in the Lambda Calculus. In view of the expressive power of the Lambda Calculus one should first become suspicious of such a concept. Secondly, one may try to extend or adapt the Lambda Calculus in such a way that the concept is expressible. The techniques and tools developed for the Lambda Calculus may then prove useful to study the extension. Examples of such concepts are *assignment* and *exception handling*.

In this paper our aim is to show the significance of the Lambda Calculus approach to "Programming Language Concepts", and to raise interest in the Lambda Calculi. We address ourselves to (experienced) programmers; no knowledge of the Lambda Calculus is assumed. To this end we keep the formalism to a bare minimum and use computer science terms and notations as much as possible. We discuss a variety of programming language concepts, such as parameterization, definition, recursion, elementary and composite data types, typing, abstract types, control of visibility and life-time, and modules. All this is preceded by a brief exposition of the Lambda Calculus and its role as an area of research in itself.

The importance of the Lambda Calculus for the design of programming languages has already been recognized in the sixties by Landin [12, 13, 14]. Algol 68's orthogonality is very similar to the simplicity of the Lambda Calculus. Reynolds [26] explains the essence of Algol as follows:

> "Algol is obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus."

2. The Lambda Calculus

We describe the Lambda Calculus as a mini programming language in a notation and a terminology that is conventional in computer science. (Thus the title of this section might have read: "Lambda Calculus Concepts — the programming language approach"). Some topics of past and current research are mentioned.

Expressions. We assume that some set of *identifiers* is available, and we let x denote an arbitrary identifier. For *expressions* there are three syntactic formation rules:

$e ::= x$ identifier

e ::= (**fn** $x \bullet e$) function expression
e ::= $e(e)$ function call

We let e, ef, ea, eb,... denote arbitrary expressions; (f is mnemonic for function, a for argument and b for body). Fully capitalized *WORDS* will abbreviate specific expressions. The sign $\equiv$ stands for syntactic equality.

Notes

2.1. In an expression (**fn** $x \bullet eb$), x is called *its parameter* and eb *its body*; the parameter is a local name, so that renaming of the parameter and all its occurrences in the body is allowed and is considered not to change the expression. We thus identify (**fn** $x \bullet x$) and (**fn** $y \bullet y$), both denoting the identity function as we shall see below.

2.2. We leave out parentheses if no confusion can result; this is often the case with the outermost parentheses of a function expression under the convention that the expression following the dot $\bullet$ should be taken as large as possible.

2.3. An expression **fn** $x \bullet eb$ is an anonymous function. In contrast to conventional languages, the concepts of function and of naming are separated here syntactically. Naming is discussed in §3.1.

2.4. Church originally wrote $\hat{x}e$ for **fn** $x \bullet e$, but for typographical reasons changed this to $\wedge xe$ and later to λxe. This is still the standard notation and clearly explains the part 'Lambda' in the name.

2.5. As an example, consider the expression **fn** $x \bullet f(f(x))$ which we shall call $TWICE_f$: called with argument x_0 this function will return $f(f(x_0))$, i.e., the result of calling f twice. The function **fn** $f \bullet TWICE_f$ will return for each argument function f the function that calls f twice. So both the argument and the result of a function may be functions themselves. Actually, each expression denotes a function.

We shall now formally define the semantics of expressions analogously to the way in primary school children are taught to evaluate fractions like $(5\times8+8)/(10\times8)$: they are given some simplification rules that may be applied in any order.

Evaluation. An expression e *evaluates to* an expression e', notation $e \Longrightarrow e'$, if e' is obtained from e by repeatedly (zero or more times) applying the following *evaluation* rule:

replace a part (**fn** $x \bullet eb$)(ea) by $[ea/x]eb$.

Here, and in the sequel, $[ea/x]eb$ denotes the result of substituting ea for each occurrence of x in eb, (taking care to avoid clash of names by renaming local identifiers where appropriate).

Notes

2.6. Substitution is a syntactic manipulation that is tedious to define formally. Let it suffice here to say that $[ea/f]\,f\,(f\,(\mathbf{fn}\,x \bullet x))$ equals $ea\,(ea\,(\mathbf{fn}\,x \bullet x))$, and that

$$[\ldots x \ldots/f]\,f\,(\mathbf{fn}\,x \bullet f\,(x)) \;\equiv\; \ldots x \ldots(\mathbf{fn}\,x' \bullet \ldots x \ldots(x')),$$

where x' is a new identifier distinct from x.

2.7. As an example we have

$$\begin{aligned}&(\mathbf{fn}\,f \bullet TWICE_f)(sin)(zero)\\ &\Rightarrow TWICE_{sin}(zero) \quad \text{i.e. } (\mathbf{fn}\,x \bullet sin\,(sin\,(x)))(zero)\\ &\Rightarrow sin\,(sin\,(zero))\end{aligned}$$

and this can not be evaluated further at this point.

2.8. In Algol 60 jargon [23] the evaluation rule is the *body replacement rule*: the effect of a function call is explained by replacing it by the function body with substitution of the argument for the parameter. In the Lambda Calculus the rule is called the β-rule, and evaluation is called *reduction*.

This seemingly simple mini programming language gives rise to a large number of thorough questions that in turn have led to substantial research efforts and a lot of results. We mention but a few.

1. Do there exist expressions whose evaluation may not terminate? Answer: yes there are, for we shall see that arbitrary recursive definitions are expressible.

2. Is the evaluation strategy (the choice what part to evaluate next) of any importance? Answer: different strategies cannot yield different final outcomes, but one may terminate in cases where the other does not. Also the number of evaluation steps to reach the final outcome, if any, depends on the strategy.

3. Is it possible to express numbers and to do arithmetic in the Lambda Calculus? Answer: yes, see §3.4.

4. Clearly function expressions denote functions in the sense of *recipes* of how to obtain the result when given an argument. Is it possible to interpret expressions as functions in the sense of *a set of (argument, result)-pairs*, such a set itself being a possible argument or result? Answer: this has been a long standing problem. D. Scott formulated the first such models in 1969; a lot of others have been found since.

5. When may or must expressions be called semantically *equivalent*? (Of course we want semantic equivalence to satisfy the usual laws of equality and to be preserved under evaluation.) If two expressions may both evaluate to a common intermediate or final outcome, they must be called equivalent. However, it is possible that they do so only "in the limit", after an infinite number of evaluation steps; in this case they may be called equivalent. And what about calling expressions equivalent if they have no outcome, not even in the limit?

6. What are the consequences of the restriction that in **fn** $x \bullet eb$ parameter x must occur at least once in the body eb? And of the extra evaluation rule

replace **fn** $x \bullet ef(x)$ by ef

(because both denote the same function intuitively)?

More information about the Lambda Calculus may be obtained from [1, 11, 29].

3. Basic Programming Language Concepts

In this section we express various basic programming language concepts in the Lambda Calculus. Justified by this, we also give specific syntactic forms for each concept together with *derived* evaluation rules.

3.1. Definitions

We extend the syntactic formation rules for expressions by:

e ::= (**df** $x = e \bullet e$) definition expression

Within (**df** $x = ea \bullet eb$) the part $x = ea$ is a *local definition* that extends over the body eb. We consider this new expression as an abbreviation for (**fn** $x \bullet eb$)(ea), so that the Lambda Calculus is not extended in an essential way, and the evaluation rule has to read:

replace a part (**df** $x = ea \bullet eb$) by $[ea/x]eb$.

Notes

3.1.1. The definition $x = ea$ in **df** $x = ea \bullet eb$ is nonrecursive. This is a consequence of our choice to let it abbreviate (**fn** $x \bullet eb$)(ea).

3.1.2. By construction there is a close correspondence, or rather identity, between definitions **df** $x = ea \bullet eb$ and parameterizations as in (**fn** $x \bullet eb$)(ea). This can be taken as a guiding principle in the design of programming languages:

> for each kind of parameter (think of **value**, **in**, **out**, **ref** and **name**) there exists a semantically identical definition, and conversely.

The consequences of adhering to this *Principle of Correspondence* have been worked out by Tennent [30]. Pascal strongly violates it.

3.1.3. There is another principle involved here, the *Principle of Naming*. In **df** $x = ea \bullet eb$ identifier x names ea locally in eb, and both ea, eb and x are arbitrary. This principle is violated in Pascal, because e.g. statements cannot be named and naming can be done only locally to procedure and function bodies.

3.1.4. We have explained the local name introduction of **df** $x = ea \bullet eb$ in terms of the **fn**-construct. Reynolds [26] makes this to a guiding principle for the design of programming languages:

every local name introduction can be explained by the **fn**-construct.

We shall apply this *Principle of Locality* to the expression for recursion, below.

3.1.5. As an example, we now name the function *TWICE*:

$$\textbf{df } twice = (\textbf{fn } f \bullet (\textbf{fn } x \bullet f\,(f\,(x)))) \bullet \ \ldots\ldots twice\,(sin\,)(zero\,)\ldots\ldots$$

Here *sin* and *zero* are just identifiers for which a definition may be provided in the context; (Principle of Naming).

3.2. Multiple Parameters and Definitions

Consider once again the expression $TWICE(f_0)(x_0)$, where $TWICE \equiv \textbf{fn } f \bullet (\textbf{fn } x \bullet f\,(f\,(x)))$. One may easily verify that $TWICE(f_0)(x_0) \Longrightarrow (\textbf{fn } x \bullet f_0(f_0(x)))(x_0) \Longrightarrow f_0(f_0(x_0))$. We might say that *both* f *and* x are parameters, and *both* f_0 *and* x_0 are arguments. Thus multiple parameters are possible, for which we design a special syntax:

$$e ::= (\textbf{fn } x_1,\ldots,x_n \bullet e) \qquad \text{for distinct } x_1,\ldots,x_n$$
$$e ::= ef(e,\ldots,e)$$

These expressions are to abbreviate $(\textbf{fn } x_1 \bullet (\cdots (\textbf{fn } x_n \bullet e)\cdots))$ respectively $ef(e_1)\cdots(e_n)$, so that the evaluation rule has to read:

$$\text{replace } (\textbf{fn } x_1,\ldots,x_n \bullet e)(e_1,\ldots,e_n) \text{ by } [e_1,\ldots,e_n/x_1,\ldots,x_n]e.$$

Guided by the Principle of Correspondence we also design the corresponding definition form:

$$e ::= (\textbf{df } x_1 = e_1,\ldots,x_n = e_n \bullet e) \qquad \text{for distinct } x_1,\ldots,x_n$$

with evaluation rule:

$$\text{replace } \textbf{df } x_1 = e_1,\ldots,x_n = e_n \bullet e \text{ by } [ea_1,\ldots,ea_n/x_1,\ldots,x_n]e.$$

Notes

3.2.1. For example, we may now write $TWICE'(f_0,x_0)$ where $TWICE' \equiv \textbf{fn } f,x \bullet f\,(f\,(x))$. We can also write $\textbf{df } f = f_0,\ x = x_0 \bullet f\,(f\,(x))$.

3.2.2. The industrious reader may verify that the multiple definitions and substitutions are *simultaneous* rather than sequential: it turns out that the definition $x_i = e_i$ extends only over e and not over e_1 through e_n. The distinctness of $x_1,\ldots,x_n$ is necessary to formulate the evaluation rule so simple (and to guarantee that the substitution is well defined).

3.2.3. Exercise. Let $\textbf{df } x_1 = e_1;\ \cdots;\ x_n = e_n \bullet e$ abbreviate the *sequential* definition $(\textbf{df } x_1 = e_1 \bullet (\cdots (\textbf{df } x_n = e_n \bullet e)\cdots))$. Now think about the corresponding "sequential parameterization".

3.3. Recursion

A recursive definition is a definition in which the defined name occurs in the defining expression. A stupid evaluation strategy that first of all tries to eliminate the recursively defined name will therefore certainly get into an infinite loop of evaluation steps. However, sometimes (unfortunately not always) the recursively defined name occurs in a subexpression (**then**- or **else**-branch in particular) whose evaluation is not needed to reach the final outcome. A moderately clever evaluation strategy will not attempt to evaluate such needless occurrences. Thus the concept of recursion is fully captured by an expression in which designated occurrences evaluate — if time has come — to the expression itself. We "extend" (not really, see Note 3.3.4 below) the Lambda Calculus by the following grammar and evaluation rules:

e ::= (**rec** $x \bullet e$) recursion expression
replace (**rec** $x \bullet e$) by $[(\textbf{rec}\ x \bullet e)/x]e$.

Notes

3.3.1. Within (**rec** $x \bullet e$) the occurrences of x in e are the points of recursion: such an occurrence evaluates to the original recursive expression. (But if such an occurrence is contained in a **then**- or **else**-branch, it may happen that after one expansion it is not any more subject to the above evaluation rule.)

3.3.2. The concepts of recursion and of definition have been separated syntactically. We may combine them by abbreviating (**df** x = (**rec** $x \bullet ea$)$\bullet eb$) by (**df rec** $x = ea \bullet eb$): the occurrences of x in eb as well as in ea will evaluate — if time has come — to the recursive expression (**rec** $x \bullet ea$).

3.3.3. Assuming that **if then else** and arithmetic are possible, we may write the definition of the factorial function as follows:

df rec fac = (**fn** $n \bullet$ **if** $n = 0$ **then** 1 **else** $n \times fac\,(n-1)$)
i.e. **df** fac = (**rec** $fac \bullet$ (**fn** $n \bullet$ **if** $n = 0$ **then** 1 **else** $n \times fac\,(n-1)$))
≡ **df** fac = (**rec** $f \bullet$ (**fn** $n \bullet$ **if** $n = 0$ **then** 1 **else** $n \times f\,(n-1)$))

The part (**rec** $f \bullet$ ..... $f\,(n-1)$) denotes the factorial function without giving it a name that can be used elsewhere.

3.3.4. In (**rec** $x \bullet e$) the identifier x is a local name whose scope extends over e. Following the Principle of Locality we explain that local naming in terms of the **fn** -construct: provided that *REC* satisfies the property

REC (**fn** $x \bullet e$) evaluates to $[REC\ (\textbf{fn}\ x \bullet e)/x]e$

we may consider (**rec** $x \bullet e$) to abbreviate *REC* (**fn** $x \bullet e$). We could now add a constant '*REC*' to the Lambda Calculus with the above evaluation rule, but it turns out (space limitations prohibit to give the motivation) that we may take:

$$REC \equiv \mathbf{fn}\, f \bullet W_f(W_f) \quad \text{where } W_f \equiv \mathbf{fn}\, y \bullet f(y(y))$$

as is easily verified.

3.3.5. Notice that recursion (as in the **rec** -expression), circularity (as in the **df rec** -definition), self-activation (as in the evaluation rule for **rec**) and self-application (as in W_f: y is applied to itself) are intimately related.

3.3.6. Mutual recursion can also be expressed, but we shall not do so here.

3.4. Truth Values and Enumerated Types

We shall choose two expressions *TRUE* and *FALSE* and some function expressions *AND*, *OR*, *NOT* and *IF* such that the laws that we expect to hold, are indeed true of these expressions. The observable behaviour of *TRUE* and *FALSE* is in their being used as the condition part of an *IF* call: we wish to have

$$IF(TRUE, e_1, e_2) \Longrightarrow e_1,$$
$$IF(FALSE, e_1, e_2) \Longrightarrow e_2.$$

Hence we let *TRUE* and *FALSE* be selector functions:

$$TRUE \equiv \mathbf{fn}\, x, y \bullet x$$
$$FALSE \equiv \mathbf{fn}\, x, y \bullet y$$

so that we may choose

$$IF \equiv \mathbf{fn}\, b, x, y \bullet b(x, y).$$

The evaluation property for *IF* is true indeed. Now functions *AND*, *OR* and *NOT* are easy to define:

$$AND \equiv \mathbf{fn}\, b1, b2 \bullet IF(b1, b2, FALSE),$$
$$OR \equiv \mathbf{fn}\, b1, b2 \bullet IF(b1, TRUE, b2),$$
$$NOT \equiv \mathbf{fn}\, b \bullet IF(b, FALSE, TRUE).$$

Notes

3.4.1. Plugging in the expression *IF* into the expression *NOT* and performing some evaluation steps, we see that we also may set $NOT \equiv \mathbf{fn}\, b \bullet b(FALSE, TRUE)$. Similarly for *AND* and *OR*.

3.4.2. Suppose *PROG* is a program (an expression) in which identifiers *True*, *False*, *If*, *And*, *Or* and *Not* occur and have been assumed to satisfy the usual Boolean laws. We may then form

$$\mathbf{df}\ True = TRUE,\ False = FALSE, \ldots,\ Not = NOT \bullet PROG.$$

In other words, the definitions $True = TRUE, \ldots, Not = NOT$ can be considered to belong to the standard environment and the application programmer need not know the particular representation choices made for truth values. We shall see in Section 4 how to hide the representation

choices so that the application programmer is not allowed to write *True* (e_1, e_2) (but has to use the *If*-function explicitly).

3.4.3. Rather than providing a standard environment we can also design specific syntactic expressions for truth values, thus:

e ::= **true** | **false** | (e **and** e) | (e **or** e) | (**not** e)
e ::= **if** e **then** e **else** e

together with the *derived* evaluation rules:

replace (**true and** e) by e
replace (**false and** e) by e
replace **if true then** e_1 **else** e_2 by e_2
⋮

3.4.4. Elements of a finite enumeration type can be represented analogously: selector function **fn** $x_1,\ldots,x_n \bullet x_i$ represents the ith element, and *elt* $(e_1,\ldots,e_n)$ is the implementation of

case *elt* **in** 1: e_1, $\cdots$, n: e_n **endcase**.

3.4.5. The above representation has been chosen in the assumption that the evaluation strategy does not evaluate the argument expressions before the body replacement rule is applied. Otherwise both the **then**- and the **else**-branch are always evaluated, and that is undesirable. We can, however, adapt the representation to that strategy, but we shall not discuss it here.

3.5. Arithmetic: (Natural) Numbers

Throughout the paper we say 'number' instead of 'natural number' (0,1,2,...). As motivated below in Note 3.5.3 we choose to represent number n by an n-fold repeated call of a function f on an initial argument a, where both f and a are parameters:

fn $f,a \bullet f(\cdots(f(f(a))\cdots)$ (n times an f)

In particular we set

ZERO ≡ **fn** $f,a \bullet a$
ONE ≡ **fn** $f,a \bullet f(a)$
TWO ≡ **fn** $f,a \bullet f(f(a))$.

The successor function *SUCC* may be implemented by

SUCC ≡ **fn** $n \bullet$ "$n+1$-fold iteration"
≡ **fn** $n \bullet$ (**fn** $f,a \bullet f$ ("n-fold iteration of f on a"))
≡ **fn** $n \bullet$ (**fn** $f,a \bullet f(n(f,a))$)

For example, one easily verifies that

$$SUCC(TWO) \Longrightarrow \mathbf{fn}\, f,a \bullet f(TWO(f,a)) \Longrightarrow \mathbf{fn}\, f,a \bullet f(f(f(a)))$$

which represents 3. The test for equality is also easy:

$$EQ0 \equiv \mathbf{fn}\, n \bullet n(F,TRUE) \text{ where } F \equiv \mathbf{fn}\, x \bullet FALSE$$

so that

$$EQ0(ZERO) \Longrightarrow ZERO(F,TRUE) \Longrightarrow TRUE,$$
$$EQ0(ONE) \Longrightarrow ONE(F,TRUE) \Longrightarrow F(TRUE) \Longrightarrow FALSE.$$

The construction of a predecessor function is more complicated. The idea is to reconstruct the number itself, n say, and simultaneously "remember" at each step in the reconstruction the outcome of the previous step. So each intermediate result consists of a pair, in which one component is a number (initially 0 and at most n) and the other component its predecessor. We use here pair-expressions of the form $<e_1,e_2>$ and suffixes .1 and .2 for selection of the first and second component of a pair; in §3.6 we show how to express these in the Lambda Calculus. Now we set

$$PRED \equiv \mathbf{fn}\, n \bullet FINISH(n(F,A))$$
where
$$A \equiv <ZERO,DONTCARE>$$
$$F \equiv \mathbf{fn}\, pair \bullet <SUCC(pair.1),pair.2>$$
$$FINISH \equiv \mathbf{fn}\, pair \bullet pair.2$$

Notes

3.5.1. The choice for *DONTCARE* in *PRED* determines the outcome of $PRED(ZERO)$. If no outcome is wanted, because within the set of numbers zero has no predecessor, we may take a nonterminating expression like $(\mathbf{rec}\, x \bullet x)$.

3.5.2. One way to define addition is:

$$\mathbf{df\ rec}\ add = \mathbf{fn}\, m,n \bullet IF(EQ0(m),n,add(PRED(m),SUCC(n)))$$

However, use of recursion expressions is not necessary:

$$\mathbf{df}\ add = \mathbf{fn}\, m,n \bullet m(SUCC,n).$$

One can prove that all effectively computable total functions on numbers can be defined solely in terms of the number zero, the successor function and so-called primitive recursions (of higher order). For our representation it turns out that we can express primitive recursion of f and a as: $\mathbf{fn}\, n \bullet n(f,a)$. So let $NREC \equiv \mathbf{fn}\, f,a \bullet (\mathbf{fn}\, n \bullet n(f,a))$. Then knowledge of the representation of numbers is not needed any more, and in particular we can replace all expressions '$n(ef,ea)$' above by '$NREC(ef,ea)(n)$'.

3.5.3. The representation choice might be motivated thus: we have represented the data structure "number" by its most characteristic

control structure, namely the use of it to control a repetition (or: repeated call; compare **var** $x := a$; **for** $i := 1$ **to** n **do** $x := f(x)$ with $f(\cdots(f(f(a)))\cdots)$). A better motivation reads as follows. Numbers form an inductively definable data type: Zero is a "number" and if n is a "number" then so is Succ(n). If we are able to replace Zero and Succ in an arbitrary "number" Succ($\cdots$ (Succ(Succ(Zero)))$\cdots$) by any a and f, then we are effectively able to construct all functions on "numbers" that are definable by (structural) induction. Thus Succ($\cdots$ (Succ(Succ(Zero)))$\cdots$) is represented by

$$\mathbf{fn}\ Succ, Zero \bullet Succ(\cdots(Succ(Succ(Zero)))\cdots) \equiv$$
$$\mathbf{fn}\ f, a \bullet f(\cdots(f(f(a)))\cdots),$$

and primitive recursion has been built in.

3.5.4. One might object to the above representation of numbers: it can hardly be called a faithful modeling of commercial programming languages, because the representation length, and therefore storage space too, for a number n is linear in n and also the number of evaluation steps to compute the predecessor of m, or the sum of m and n, is linear in m. We can however improve upon this drastically. Observe that the above representation is close to the unary notation of numbers: number 1...11 (in unary notation) has been represented by $\mathbf{fn}\ f, a \bullet f(\cdots(f(f(a)))\cdots)$. Now we represent e.g. number 1001101 (in binary notation) by $\mathbf{fn}\ f, g, a \bullet f(g(g(f(f(g(f(a)))))))$; an f for 1 and a g for 0. The representation length grows only logarithmically. The successor function may be defined thus:

$$SUCC' \equiv \mathbf{fn}\ n \bullet \mathbf{fn}\ f, g, a \bullet FINISH(n(F, G, <CARRY, a>))$$
where
$$CARRY \equiv \mathbf{fn}\ x, y \bullet x \quad (\equiv TRUE)$$
$$NOCARRY \equiv \mathbf{fn}\ x, y \bullet y \quad (\equiv FALSE)$$
$$F \equiv \mathbf{fn}\ <carry, result> \bullet carry(g, f)(result)$$
$$G \equiv \mathbf{fn}\ <carry, result> \bullet carry(f, g)(result)$$
$$FINISH \equiv \mathbf{fn}\ <carry, result> \bullet carry(f(result), result)$$

In a similar way addition can be defined. It turns out that evaluation of both $SUCC(n)$ and $PRED(n)$ takes $O(\log n)$ steps, and $ADD(m,n)$ takes $O(\log m + \log n)$ steps. No programming language can improve upon this whenever it allows unbounded numbers.

3.5.5. Similar remarks as in Notes 3.4.2-3 apply here as well. In view of the complicated representation of numbers, and implementation of the operations, this is very welcome.

3.6. Composite Data Types: Records and Lists

We can be very brief with respect to lists. Note 3.5.3 provides the clue to choose the representation: the list

$$\mathrm{Cons}(x_1, \mathrm{Cons}(x_2, \ldots, \mathrm{Cons}(x_n, \mathrm{Nil})\ldots))$$

is represented by

$$\mathbf{fn}\ Cons, Nil \bullet Cons(x_1, Cons(x_2, \ldots, Cons(x_n, Nil)\cdots)) \equiv \mathbf{fn}\ f, a \bullet f(x_1, f(x_2, \ldots, f(x_n, a)\cdots)).$$

We leave it to the reader to define functions *NIL*, *CONS* and *LREC* (cf. *ZERO*, *SUCC* and *NREC* of §3.5), and to build *HEAD*, *TAIL*, *EQNIL* and so on in terms of them. (Typed versions will be given in §5.3).

In the next section we discuss typing and shall require that lists be homogeneous: all elements of a list must belong to the same data type. So we need a kind of **record**-construct for inhomogeneous aggregates. For simplicity we discuss pairs (2-tuples) only; the generalization to n-tuples is straightforward. The tuple Pair(x, y) is represented by $\mathbf{fn}\ Pair \bullet Pair(x, y)$, i.e. $\mathbf{fn}\ f \bullet f(x, y)$. The constituting elements can be retrieved by applying the pair to the appropriate selector functions. Thus we let

$$\begin{aligned} &\langle e_1, e_2 \rangle \equiv \mathbf{fn}\ f \bullet f(e_1, e_2) \\ &e.1 \equiv e(\mathbf{fn}\ x, y \bullet x) \\ &e.2 \equiv e(\mathbf{fn}\ x, y \bullet y) \end{aligned}$$

or we introduce the left-hand sides as new syntactic forms, together with the appropriate, derived, evaluation rules:

$$\begin{aligned} &e ::= \langle e, e \rangle \mid e.1 \mid e.2 \\ &\text{replace } \langle e_1, e_2 \rangle.1 \text{ by } e_1 \\ &\text{replace } \langle e_1, e_2 \rangle.2 \text{ by } e_2. \end{aligned}$$

3.7. Concluding Remarks

3.7.1. We have shown how data structures may be represented by functions. Reynolds [25] and Meertens [17] show the usefulness of such representations in practice. (However, they term the technique *procedural data abstraction* rather than procedural ($\approx$ functional) data representation.)

3.7.2. In a similar way arbitrary Turing machines and similar devices can be represented by functions, see Fokkinga [7] and Langmaack [15]. It turns out that the functions do accept functions as parameters, but do not yield functions as result: the representation can therefore be carried out in conventional languages (if recursive types are available, as in Algol 68). From this, one immediately concludes several fundamental limitations of compile-time checks.

4. Typing

We consider *typing* a well-formedness check where attributes, called *types*, are assigned to subexpressions and the type of a subexpression has to satisfy specific requirements in relation to the types of its direct

constituent parts. An expression that passes the check is said to be *typed* or *typable*.

The assignment of types to expressions may be facilitated by an explicitly written type at each introduction of a local name; but this is not necessary. In the former case we speak of *explicit* typing, in the latter case of *implicit* typing or type deduction. An expression that can be assigned only one type is called *monomorphic*. An expression is called *polymorphic* if it is assigned many related types, a type scheme so to speak. In particular, a function expression is polymorphic if it may be applied to arguments of various but schematically the same types. We call a function *generic* if it may be applied to a type (which may determine the types of the following arguments and final result). (Another term for genericity is *parametric polymorphism*.) Examples will be given in the sequel. The type of a function whose arguments must have type **nat** and whose result has type **bool**, is written (**nat**→ **bool**).

In this section we discuss the Monomorphic Typing M, the Polymorphic Typing P and the Generic Typing G. These are extensively studied by Hindley & Seldin [11]. Other overviews on typing are given by Reynolds [28] and Cardelli & Wegner [4]; they cover more features than we do.

4.1. The Usefulness of Typing

Typing proves its usefulness if the typable expressions satisfy a useful semantic property, (chosen by the designer of the typing). We list here some properties that may or may not be aimed at in the design of a typing.

1. Set theoretic interpretation. For the class of typable expressions a simple set-theoretic interpretation is possible, in which expressions of type (**nat**→ **bool**) are interpreted as *mappings* from the set of numbers to the set of truth values, rather than *recipes* that prescribe how to obtain the outcome when given an argument. (This property precludes self-application and therefore also the unrestricted use of the **rec** - expression.)

2. Termination. The evaluation of typable expressions terminates. One might argue that non-terminating evaluations are useless, but apart from that, the existence of nonterminating expressions invalidates conventional mathematical laws such as

$$0 \times e = 0 \quad \text{for any expression } e \text{ of type } \mathbf{nat}.$$

(This property too precludes general recursion.)

3. Implementation ease. For typable expressions the size of the storage space for the values that appear during the evaluation, is compile-time computable. This property is aimed at by the Pascal typing; consequently the programmer is forced to specify the size of arrays by

constants. The property eases the task of the implementor, not of the programmer.

4. Representation independence. The outcome of typable expressions does not depend on the representation chosen for internally used data like truth values, numbers and other data types. This property allows the implementor to switch freely from the unary representation to the binary representation; cf. §3.5 and Note 3.5.4. Moreover, the implementor may even implement arithmetic in hardware: the outcome of typable expressions will not change. This property, as well as property 1 precludes the use of **nat**-expressions as functions even if we know they are; cf. §3.5.

5. Error prevention. For typable expressions many errors of the kind "Ah, of course, I see, this is a misprint" and "Ah, of course, this is an oversight" are impossible. This is a rather fuzzy property and much of it is implied by properties 1 and 4.

4.2. The Monomorphic Typing M

We describe here a simple typing M that gives the essence of Pascal-like typing. We concentrate on the Lambda Calculus and shall *derive* the M-typing requirements for the derived expressions.

M-types. The attributes assigned to expressions, and called M-*types*, are syntactic forms defined by the following grammar:

$$t ::= (t \to t) \mid \mathbf{nat} \mid \mathbf{bool} \mid \mathbf{char} \mid \cdots .$$

We let t, ta, tb denote arbitrary types.

M-typable expressions. We write the type assigned to a (sub)expression as a superscript. It is required that within $(\mathbf{fn}\, x \bullet e)$ all occurrences of x in e have the same type; this type is written at the parameter position: $(\mathbf{fn}\, x^t \bullet e)$. Now consider the following infinite set of grammar rules, one for each choice of t, ta, and tb:

$$\begin{aligned} e^t &::= x^t \\ e^{ta \to tb} &::= (\mathbf{fn}\, x^{ta} \bullet e^{tb}) \\ e^{tb} &::= e^{ta \to tb}(e^{ta}) \end{aligned}$$

The grammar generates, by definition, the M-typed expressions. Alternatively we may consider it as a formalization of the requirements for a M-type assignment:

- if x has been assigned type ta and e type tb, then $(\mathbf{fn}\, x \bullet e)$ may be assigned type $(ta \to tb)$;
- if ef has been assigned type $ta \to tb$ and ea type ta, then $ef(ea)$ may be assigned type tb;
- if identifier x has been assigned type t, then considered as a subexpression it may be assigned type t.

For example, for any t the expression

$$((\mathbf{fn}\ x^{tt} \bullet x^{tt})^{tt \to tt}\ (\mathbf{fn}\ x^{t} \bullet x^{t})^{t \to t})^{tt}$$

where $tt \equiv t \to t$, is M-typed. But

$$(\mathbf{fn}\ id \bullet id\ (id))(\mathbf{fn}\ x \bullet x)$$

is not M-typable although it evaluates in one step to the preceding expression.

Notes

4.2.1. One may succeed easily in deriving the M-typing requirements for derived expressions like **fn** $x,y \bullet e$ and **df** $x = ea \bullet eb$. Extend the grammar for types by

$$t ::= (t_1,\ldots,t_n \to t)$$

where $t_1,\ldots,t_n \to t$ is thought of as an abbreviation of $t_1 \to (t_2 \to \cdots (t_n \to t)\cdots)$. Then the grammar for typed expressions may be extended by

$$e^{t1,\ldots,tn \to t} ::= (\mathbf{fn}\ x_1^{t1},\ldots,x_n^{tn} \bullet e^{t})$$
$$e^{t} ::= e^{t1,\ldots,tn \to t}(e^{t1},e^{t2},\ldots,e^{tn})$$
$$e^{t} ::= (\mathbf{df}\ x^{ta} = e^{ta} \bullet e^{t}).$$

4.2.2. We may decide to assign *ZERO, ONE, TWO*... type **nat** and *SUCC* type **nat**→**nat**; and so on:

$$e^{\mathbf{nat}} ::= ZERO \mid ONE \mid TWO \mid \cdots$$
$$e^{\mathbf{nat} \to \mathbf{nat}} ::= SUCC$$
$$e^{\mathbf{bool}} ::= TRUE \mid FALSE$$
$$e^{\mathbf{bool},\mathbf{bool} \to \mathbf{bool}} ::= AND \mid OR$$
$$e^{\mathbf{bool},t,t \to t} ::= IF.$$

A justification for this decision is given in §5.2. Notice that different occurrences of *IF* may be assigned different types; *IF* is a polymorphic expression. However, in

$$\mathbf{df}\ If = IF \bullet \ldots If \ldots If \ldots If \ldots$$

there is only one occurrence of *IF*, so that all occurrences of *If* are assigned the same type $\mathbf{bool},T,T \to T$ for one specific type T. In this way the programmer is forced to spell out *IF* each time again. One solution to this problem is given in §4.3: polymorphic typing. Another solution is to extend the grammar by:

$$e^{t} ::= \mathbf{if}\ e^{\mathbf{bool}}\ \mathbf{then}\ e^{t}\ \mathbf{else}\ e^{t}.$$

This is justified by considering **if** e **then** e_1 **else** e_2 as an abbreviation of $IF(e,e_1,e_2)$. Yet another solution is given in §4.4: generic typing.

4.2.3. We may treat the other polymorphic expressions similarly to *IF*: build the polymorphism into the derived syntax and typing. For

example for pairs:

$t ::= \langle t,t \rangle$ type for pairs
$e^{\langle t1,t2 \rangle} ::= \langle e^{t1}, e^{t2} \rangle$
$e^{t} ::= e^{\langle t,t' \rangle}.1$
$e^{t} ::= e^{\langle t',t \rangle}.2$

and analogously for n-tuples, lists, arrays and so on.

4.2.4. The M-typing as described so far validates properties 1, 2, 4 and 5 of §4.1.

4.2.5. Expression *REC* is not M-typable. So the following typing rule properly extends the set of M-typable expressions:

$$e^{t} ::= (\mathbf{rec}\ x^{t} \bullet e^{t})$$

or equivalently; cf. Note 4.2.1.

$$e^{(t \to t) \to t} ::= REC.$$

Now property 2 (Termination) is invalidated, the other three are preserved.

4.2.6. One may extend the monomorphic typing by allowing recursive types, as in Algol 68. It turns out that every expression of the pure Lambda Calculus is typable, with the recursive type $fun = fun \to fun$. Nevertheless not all expressions are typable, and properties 4 and 5 remain valid, if we require that *ZERO* is assigned type **nat** only, and *SUCC* type **nat**→**nat** and so on. We shall not discuss recursive types any further.

4.3. Polymorphic Typing P

The monomorphic typing M has a flagrant deficiency: there are only monomorphic types and consequently one is forced to duplicate expressions solely for the purpose of letting different occurrences be assigned different types. For example consider

df *id* = (**fn** $x \bullet x$)
• ... *id* (*zero*$^{\mathbf{nat}}$) ... *id* (*true*$^{\mathbf{bool}}$) ... *id* (*id*) ...
df *compose* = (**fn** $f,g \bullet$ (**fn** $x \bullet f\,(g\,(x))$))
• ... *compose* (*not*$^{\mathbf{bool} \to \mathbf{bool}}$, *not*$^{\mathbf{bool} \to \mathbf{bool}}$) ...
... *compose* (*sqr*$^{\mathbf{nat} \to \mathbf{nat}}$, *ord*$^{\mathbf{char} \to \mathbf{nat}}$) ...
df *sort* = sorting function
• ... *sort* (number list) ... *sort* (character list) ...

These expressions are not M-typable, but after substituting the defining expressions for the defined identifiers (or multiplicating the definitions, one for each use) they are M-typable. The solution to this deficiency is simple: polymorphism. It has been introduced into computer science by Milner [20], but was already known in the Lambda Calculus as

Principal Typing.

P-types. We let z range over identifiers. (One may stipulate that the identifiers denoted by z are distinct from those denoted by x, but this is not necessary). The P-types are now defined thus:

$$t ::= z \mid (t \to t) \mid \mathbf{nat} \mid \mathbf{bool} \mid \mathbf{char} \mid \cdots.$$

The identifiers occurring in P-types shall play the role of place holders for which arbitrary types may be substituted consistently. A P-type may therefore be considered as a "M-type *scheme*".

P-typable expressions. There is only one difference in the type assignment rules in comparison with those of the M-typing:

> within **df** $x = ea^t \bullet eb$ the occurrences of x in eb may be assigned *instantiations* of t, more precisely: P-types must be substituted for the identifiers in t that have been used in the typing of ea^t only (and not in its context); different occurrences of x may be assigned different instantiations of t.

All the other rules for the M-typing are valid for the P-typing as well.

Notes

4.3.1. The examples above are all P-typable. For instance:

df $id = (\mathbf{fn}\ x^a \bullet x^a)^{a \to a}$
• ... $id^{\mathrm{nat} \to \mathrm{nat}}(zero^{\mathrm{nat}})$ ...
... $id^{\mathrm{bool} \to \mathrm{bool}}(true^{\mathrm{bool}})$ ...
... $id^{(b \to b) \to (b \to b)}(id^{(b \to b)})$ ...

But unfortunately, $(\mathbf{fn}\ id \bullet ...id\ (zero^{\mathrm{nat}})...id\ (true^{\mathrm{bool}})...)(\mathbf{fn}\ x \bullet x)$ is not P-typable. This also shows that the P-typing violates the Principle of Correspondence.

4.3.2. The polymorphic typing is used in modern functional languages, like Miranda [31, 32], as well as in the modern imperative language ABC [18].

4.3.3. One obvious advantage of the P-typing over the more "powerful" G-typing of the next subsection, is that types need not be written explicitly in the program text (although it is permitted): the type-checker will deduce them anyway, (and show or insert them on request).

4.3.4. The language can be enriched by further constructs for the definition of user defined types. One particularly simple and elegant way has been built in in Miranda [31, 32]. We discuss type definitions more fundamentally in the next subsection.

4.4. Generic Typing G

The P-typing, although quite successful for programming in the small, is not very satisfactory for at least two reasons. First, as we have seen in Note 4.3.1 parameters can not be used polymorphically. Second, the

facility of assigning **nat** to *ZERO*, *ONE*, ... and **nat**→**nat** to *SUCC* is not generally available to the programmer. (Recall that *ZERO*, *ONE*, ... *SUCC* are merely ordinary expressions.) The programmer does need such a facility in order to get Representation Independence for his own devised data types. The solution is to control the type assignment explicitly, by indicating for each parameter the desired type and in addition allowing types to be parameters ("genericity"). The resulting language is often called Second Order Lambda Calculus and was invented by J.-Y Girard and, independently, Reynolds [24].

G-types. As before z ranges over identifiers. G-types are defined by the following grammar.

$$t ::= z \mid (t \to t) \mid (z : \mathbf{tp} \to t).$$

The third form of type is called a *generic type*. Within $(z : \mathbf{tp} \to t)$ identifier z is a local name whose scope extends over t; of course systematic renaming is allowed. We let t, ta, tb,... denote arbitrary types. (Type constants like **nat** and **bool** are no longer needed. We shall see that the programmer can "define" them.)

G-Typable expressions. As for the M-typing we define:

$$\begin{array}{lll} e^{t} & ::= & x^{t} \\ e^{ta \to tb} & ::= & (\mathbf{fn}\ x^{ta} : ta \bullet e^{tb}) \qquad \text{Note the explicit type for } x \\ e^{tb} & ::= & e^{ta \to tb}(e^{ta}) \end{array}$$

Henceforth we shall omit a type superscript at a parameter, if it also occurs explicitly. We add two new expressions:

$$\begin{array}{lll} e^{z:\mathbf{tp}\to t} & ::= & (\mathbf{fn}\ z : \mathbf{tp} \bullet e^{t}) \qquad \text{generic function expression} \\ e^{[ta/z]t} & ::= & e^{z:\mathbf{tp}\to t}(ta) \qquad \text{generic instantiation/call} \end{array}$$

Deliberately generic instantiation looks like a normal function call, but it is not: it is a new kind of expression with a *type* as one of its direct constituents. Similarly for generic function expression. Within $(\mathbf{fn}\ z : \mathbf{tp} \bullet e)$ identifier z is a local name whose scope extends over e; z may e.g. occur in the explicit types in e. For simple examples see the first note below; Section 5 contains further examples.

Evaluation. For the new expressions we have to define an evaluation rule. The rule is evident:

replace $(\mathbf{fn}\ z : \mathbf{tp} \bullet e)(ta)$ by $[ta/z]e$.

Notes

4.4.1. For example, the generic identity function reads $GID \equiv \mathbf{fn}\ z : \mathbf{tp} \bullet (\mathbf{fn}\ x : z \bullet x^{z})^{z \to z}$ and has type $z : \mathbf{tp} \to (z \to z)$. The generic instantiation $GID(nat)$ has type $[nat/z](z \to z) \equiv nat \to nat$ and evaluates to $(\mathbf{fn}\ x : nat \bullet x^{nat})^{nat \to nat}$ as expected and desired. Similarly, $GID(bool)$ has type $[bool/z](z \to z) \equiv bool \to bool$, and evaluates to the identity function for *bool*-expressions.

4.4.2. In Section 5 we shall introduce some syntactic sugar like we did before. Let it suffice here that we may write the left-hand sides for the right-hand sides:

$(z:\mathbf{tp}, x:z\to z)$	$(z:\mathbf{tp}\to(x:z\to z))$
$(\mathbf{fn}\ z:\mathbf{tp}, x:z\bullet z)\ (\equiv GID')$	$\mathbf{fn}\ z:\mathbf{tp}\bullet(\mathbf{fn}\ x:z\bullet x)\ (\equiv GID)$
$GID'(nat, zero^{nat})$	$GID(nat)(zero^{nat})$
and	
$(\mathbf{df}\ z:\mathbf{tp} = nat,\ x:z = zero^{nat}\bullet x)$	$GID'(nat, zero^{nat})$.

4.4.3. The richness of the G-typable expressions is already perceptible from the possibility of generically calling *GID* with its own type: $GID(z:\mathbf{tp}\to(z\to z))$ has type $(z:\mathbf{tp}\to(z\to z))\to(z:\mathbf{tp}\to(z\to z))$ and evaluates to the identity function $\mathbf{fn}\ x:(z:\mathbf{tp}\to(z\to z))\bullet x$. This in turn may be applied to *GID* and then evaluates to *GID* again.

4.4.4. The following theorems have been proved for the class of G-typed expressions. For (a)-(e) see [9] and for (f) [27].

a. Different evaluation strategies cannot produce different outcomes.

b. Any evaluation of any expression terminates.

c. G-typability is preserved under evaluation.

d. The G-type of an expression is uniquely determined and compile-time computable.

e. The generic function expressions and instantiations are semantically insignificant. That is, they can be eliminated compile-time from any expression (by compile-time evaluation), provided, of course, that neither the expression nor the global identifiers in it have a generic part $(z:\mathbf{tp}\to\cdots)$ in their type. For example $GID(nat)$ has type $nat\to nat$; it evaluates compile-time to $\mathbf{fn}\ x:nat\bullet x$ that contains no generic constructs any more. *GID* itself does contain a generic construct that can not be eliminated, for its type explicitly demands so.

f. A classical set-theoretic interpretation of expressions and types is not possible.

4.4.5. It is still a topic for research how much of the explicit types and generic functions and instantiations can be left out of expressions, while still keeping G-typability decidable.

4.4.6. It is easy to extend the language with a recursive construct; this however invalidates the Termination property (b) above.

4.4.7. A technical detail. Consider $(\mathbf{fn}\ z:\mathbf{tp}\bullet e^{t})^{z:\mathbf{tp}\to t}$ and assume that some global identifier x with type $\ldots z\ldots$ occurs in it. There are now a global z and a local z involved in the type assignment to e. To avoid problems one should either forbid such occurrences of x or else require that $[z'/z]e$ has type $[z'/z]t$ for some brand-new identifier z', rather than that e has type t).

5. Type Definitions, Abstract Types and Modules

Clearly a typing is not satisfactory if there is no facility for something like "user defined types", "abstract types" and "modules". Pascal, Algol 68, Ada, Modula 2 and others all have their own way to do so, and the result is an astonishing diversity of different constructs; (think only of type definitions and the problem of choosing between occurrence equivalence, name equivalence and structural equivalence). We have refrained from designing such facilities in an ad-hoc way, because we get them for free, in a fundamental way, from the G-typing: according to the Principle of Correspondence we may write a generic instantiation of a generic function expression as a definition: a type definition. This is done in §5.1; § 5.2 and 5.3 give some examples and §5.4 discusses modules. Throughout §5.1-3 we use the G-typed Lambda Calculus.

5.1. User Defined Types

Like we did for the un-, M- and P-typed Lambda Calculus, we introduce some special syntactic forms for special (frequently used) composite expressions. The abbreviations for (normal) multiple parameters, arguments and definitions are straightforward; both with respect to the form of the expressions, as well as with respect to the typing and evaluation rules. But generic types, functions and instantiations call for an abbreviation too; in particular we write the left-hand sides, below, for the right-hand sides:

$z:\mathbf{tp}, t_1, \ldots, t_n \to t$	$z:\mathbf{tp} \to (t_1 \to (\cdots (t_n \to t)\cdots))$
$\mathbf{fn}\ z:\mathbf{tp}, x_1:t_1, \ldots, x_n:t_n \bullet e$	$\mathbf{fn}\ z:\mathbf{tp} \bullet (\mathbf{fn}\ x_1:t_1 \bullet (\cdots (\mathbf{fn}\ x_n:t_n \bullet e)\cdots))$
$ef(t, e_1, \ldots, e_n)$	$ef(t)(e_1)\cdots(e_n)$
$\mathbf{df}\ z:\mathbf{tp}, x_1:t_1 = e_1, \ldots, x_n:t_n = e_n \bullet e$	$(\mathbf{fn}\ z:\mathbf{tp}, x_1:t_1, \ldots, x_n:t_n \bullet e)(t, e_1, \ldots, e_n)$

Notice that in all these expressions the scope of z extends over $t_1, \ldots, t_n$ and e but not over $e_1, \ldots, e_n$. This is particularly true of the **df**-expression. Hence the *derived* typing rules have to read

$$e^{z:\mathbf{tp}, t1, \ldots, tn \to t} ::= (\mathbf{fn}\ z:\mathbf{tp}, x_1:t1, \ldots, x_n:tn \bullet e^t)$$
$$e^{[t/z]tb} ::= e^{z:\mathbf{tp}, t1, \ldots, tn \to tb}(t, e_1^{[t/z]t1}, \ldots, e_n^{[t/z]tn})$$
$$e^{[t/z]tb} ::= (\mathbf{df}\ z:\mathbf{tp} = t, x_1:t1 = e_1^{[t/z]t1}, \ldots, x_n:tn = e_n^{[t/z]tn} \bullet e^{tb})$$

In words: if z is defined to be t and an expression e_i outside the scope of $z:\mathbf{tp}$ is required to have — formulated inside the scope of z — type ti, then e_i must actually have $[t/z]ti$, i.e. the required type ti in which t is read for z. Within the scope of a type definition $z = t$ identifier z is a type that is unrelated to t as far as type-checking is concerned. (Thus far all our type-checking rules require exact matching, i.e. equality; there has not been introduced any notion of type

equivalence.)

Notes

5.1.1. Referring to the expressions discussed above, the collection

$$z : \mathbf{tp},\ x_1 : t_1, \ldots,\ x_n : t_n$$

constitutes the signature of an abstract data type, z being the name for the carrier. The collection

$$t,\ e_1, \ldots,\ e_n$$

constitutes the/an implementation; t being the representation type, i.e. the type to represent the "abstract z-values", and $e_1, \ldots, e_n$ being the implementation of $x_1 : t_1, \ldots, x_n : t_n$. The G-typed Lambda Calculus provides no way to express laws between the $x_1, \ldots, x_n$ that one might wish to hold. See also Section 6.

5.1.2. One might introduce a new expression

$$\mathbf{df}\ z \equiv t \bullet e \quad \text{to stand for} \quad [t/z]e.$$

In this expression, z and t may be used interchangeably within e: as is to be seen in the right-hand side all z's are replaced by t. So here the definition $z \equiv t$ is completely transparent for e. We shall not use this construct in the sequel.

5.2. Simple Abstract Types: *nat*

We have already seen how numbers may be represented by function expressions and that the definitions for zero, successor and primitive recursion in principle suffice to define the other total functions on numbers. We shall now adapt the expressions to the G-typing and provide suggestive names (identifiers) for them.

Remember, number n was represented by

$$\mathbf{fn}\ f, a \bullet f(\cdots(f(f(a)))\cdots).$$

This may be typed

$$(\mathbf{fn}\ f : (t \to t),\ a : t \bullet f(\cdots(f(f(a)))\cdots))^{(t \to t),\, t \to t}$$

for any type t. Therefore we make t to an explicit parameter (called z), getting

$$\mathbf{fn}\ z : \mathbf{tp},\ f : (z \to z),\ a : z \bullet f(\cdots(f(f(a)))\cdots).$$

The type of these expressions is abbreviated *NAT*, so

$$NAT \equiv (z : \mathbf{tp},\ (z \to z),\ z \to z).$$

Now we form, given a user program *PROG*:

$$\begin{aligned} &\mathbf{df}\ nat : \mathbf{tp} = NAT, \\ &\quad zero : nat = (\mathbf{fn}\ z : \mathbf{tp},\ f : (z \to z),\ a : z \bullet a), \\ &\quad succ : nat \to nat \end{aligned}$$

$$= \mathbf{fn}\ n : NAT \bullet \mathbf{fn}\ z : \mathbf{tp}, f : z \to z, a : z \bullet f\,(n\,(z,f,a)),$$
$$nrec : (z : \mathbf{tp}, (z \to z), z \to (nat \to z))$$
$$= \mathbf{fn}\ z : \mathbf{tp}, f : z \to z, a : z \bullet (\mathbf{fn}\ n : NAT \bullet n\,(z,f,a))$$
$$\bullet\ PROG$$

Notes

5.2.1. Notice that uses of number representations have to get an explicit type argument, which determines (see *NAT*) the types of the subsequent arguments and the final result.

5.2.2. Within *PROG* the identifiers can only be used as prescribed by their type at the left-hand sides of the definitions; other use within *PROG* is not G-typable. In particular, although *zero* evaluates to a function, the "expression" $zero\,(bool, (\mathbf{fn}\ x : bool \bullet x), true^{bool})$ is type-incorrect.

5.2.3. The right-hand sides may be replaced by G-typed versions of the "binary" representation suggested in Note 3.5.4: the entire **df**-expression remains G-typable. Also, as long as *PROG* has no *nat* in its type, the outcome does not change by this replacement. Cf. 4.1.4.

5.2.4. Nothing prevents us from replacing a by $f\,(f\,(f\,(a)))$ in the right-hand side of the definition of *zero*: what results is G-typable again, but does not have the intended semantics.

5.2.5. Suppose *PROG* has type *nat*. Then the entire expression has type *NAT* (not *nat*). Hence the context of the entire expression "knows" that the outcome is a generic iterator function, rather than a number, and it may use the outcome accordingly. This is not at all surprising if one realizes that it is the very context writer who also provides the right-hand sides (and possibly delegates the construction of *PROG* to another programmer, the left-hand sides and the type of *PROG* being the interface between the two).

5.2.6. Within *PROG* one may define other arithmetic functions, e.g.

$$\mathbf{df}\ eq0 : nat \to bool = nrec\ (bool, (\mathbf{fn}\ x : bool \bullet false), true) \bullet \ldots$$

assuming that the global identifiers *bool*, *false* and *true* have been defined properly in that context. Similarly one can adapt the expression *PRED* to the G-typing, and use it to define $pred : nat \to nat$ within *PROG*.

5.2.7. An alternative type and definition for *nrec* is:

$$nrec' : (nat \to NAT)$$
$$= \mathbf{fn}\ n : NAT \bullet (\mathbf{fn}\ z : tp, f : (z \to z), a : z \bullet n\,(z,f,a))$$

and the right-hand side may even be replaced by $\mathbf{fn}\ n : NAT \bullet n$ and $GID\,(NAT)$. But notice that with the definition

$$nrec'' : (nat \to nat) = \ldots\ldots\text{as for } nrec' \ldots\ldots$$

we cannot use *nrec''* differently from the identity function on *nat*-

expressions.

5.2.8. Let us abbreviate the sequence of the left-hand sides by SIG_{nat} and the sequence of right-hand sides by $IMPL_{NAT}$. ('*SIG*' is mnemonic for signature and '*IMPL*' for implementation.) Then we may also write

$$(\mathbf{fn}\ SIG_{nat} \bullet PROG)(IMPL_{NAT})$$

and this is G-typed and equivalent to the previous program (w.r.t. both typing and evaluation). The expression shows more clearly that the implementation may be changed independently of the signature.

5.3. Parameterized Abstract Types: List of Elements

We shall construct something like "*list*(*elt*)" where *elt* is a parameter, in such a way that the construct "*list*(*elt*)" can be used with different choices for *elt*. The problem here is that "*list*(*elt*)" can not be a G-type, because we would then have $(\ldots \to \mathbf{tp})$ as type for *list* and such G-types do not exist. Nevertheless a satisfactory solution is possible and is easily generalized to, say, "*array*(*elt*,*n*)" for arrays of run-time determined fixed length *n*.

As for numbers it suffices to have nil (the empty list), cons (for constructing an element and a list into a new list) and lrec (for generic primitive recursion over lists) as the primitive operations and constant of lists. Head, tail, eqnil, append, map and so on can be defined in terms of them. (There is however no objection at all to enlarge the set of primitives.) We set

$$\begin{aligned} SIG \equiv\ & list : \mathbf{tp}, \\ & nil : list, \\ & cons : (elt, list \to list), \\ & lrec : (z : \mathbf{tp}, (elt, z \to z), z \to (list \to z)). \end{aligned}$$

So SIG gives the signature of the abstract data type of lists. It is not an expression, but a series of left-hand sides of definitions or formal parameters. Notice also that *elt* occurs globally in *SIG*; it will be used as the type of the list elements. For the time being we assume that we have an implementation for the signature, i.e. a series of expressions that we call *IMPL*:

$$IMPL \equiv LIST, NIL, CONS, LREC$$

where, again, identifier *elt* occurs globally in *LIST*,..., *LREC*. We postpone the construction of *IMPL* and first focus on instantiating *IMPL* by different choices for *elt*.

Let *PROG* be a user program that computes with lists of numbers: within *PROG* *list* is assumed to be a type, *cons* to be of type $(nat, list \to list)$ (rather than $(elt, list \to list)$), and similarly for *nil* and *lrec*. In short, *PROG* is G-typed under the typing assumptions

[*nat /elt*]*SIG*. We may then form

(**fn** [*nat /elt*]*SIG* • *PROG*)([*nat /elt*]*IMPL*)

to obtain a G-typed expression with the desired behaviour. Now suppose that *PROG* uses both lists of *nat*s and lists of *bool*s. Let [*nat /elt*]*SIG'* and [*bool /elt*]*SIG''* be the assumptions under which *PROG* is G-typed. (By a single/double prime on *SIG* we mean that each of *list*,...,*lrec* gets a single/double prime.) As before we may now form

(1) (**fn** [*nat /elt*]*SIG'*, [*bool /elt*]*SIG''* • *PROG*)
([*nat /elt*]*IMPL*, [*bool /elt*]*IMPL*)

to obtain a G-typed expression with the desired behaviour. However, we have duplicated *IMPL* and performed the substitutions *nat /elt* and *bool /elt* in *IMPL* manually. This is quite unsatisfactory, and can not claimed to be (a good model of) a practical programming language concept. Fortunately, there is better way by using parameterization. In the following expression 'AT' is mnemonic for 'abstract type', 'gen' for 'generic' or 'generate', and *T* is the type of *PROG*.

(2)
df *genListAT* : (*elt* : **tp**, (*SIG* → *T*) → *T*)
= (**fn** *elt* : **tp**, *p* : (*SIG* → *T*)• *p* (*IMPL*))
• *genListAT* (*nat*, (**fn** [*nat /elt*]*SIG* • *PROG*))
respectively
genListAT (*nat*, (**fn** [*nat /elt*]*SIG'*
• *genListAT* (*bool*, (**fn** [*bool*,*elt*]*SIG''*
• *PROG*))))

So inside *genListAT* the user program receives the implementation, and thanks to the argument for parameter *elt* the implementation is suitably instantiated. Notice also that the nested call of *genListAT* is not at all recursive. There is yet one adaptation necessary; it concerns *PROG*'s result type *T*. As it stands, *T* is fixed within *genList AT* but naturally we want *T* to vary with the argument for *p*. Hence *T* should be made a parameter and we get:

(2')
df *genListAT* : (*elt* : **tp**, *t* : **tp**, (*SIG* → *t*) → *t*)
= (**fn** *elt* : **tp**, *t* : **tp**, *p* : (*SIG* → *t*)• *p* (*IMPL*))
• *genListAT* (*nat*, *T*, (**fn** [*nat /elt*]*SIG* • *PROG*))
and so on...

Actually the type-checker may deduce *T* from *PROG* and the programmer need not write it explicitly.

Notes

5.3.1. One might now go on and design new expression forms for the definition and use of abstract types. This has been done indeed, and

gives rise to the introduction of a $\forall$- and a $\exists$-type and a special syntax for program scheme (2'). Essentially $\forall elt \bullet t$ is a generic type and abbreviates $(elt : \mathbf{tp} \rightarrow t)$, whereas $\exists\, elt \bullet t$ is a generic signature and abbreviates $elt : \mathbf{tp}, t$ (where t may be a cartesian product $t_1, t_2, \ldots, t_n$). See Cardelli & Wegner [4] and Mitchell & Plotkin [21]. A formal Representation Independence has been proved for the typing system of [21], see [22].

5.3.2. Consider once more expressions (1) and (2') and in particular type T of *PROG*. In (1) T may be expressed in terms of *list'* and *list''* and there is no problem whatsoever with the G-typability of the entire expression (1). E.g. if $T \equiv list'$ then (1) has type $[nat/elt]LIST$; cf. also Note 5.2.5. Within (2') however it seems impossible to arrange that $T \equiv list'$. The reason is that T falls outside the scope of $[nat/elt]SIG$, i.e. T is not in the scope of the locally defined *list*. This forms *our* explanation of the requirement AB.3 in [21], viz. that a program may not deliver a value of a locally defined abstract type.

5.3.3. Constructions like "list of list of elements" are possible too. For example, replace in (2) *bool* by *list'* (i.e. list of *nat* s).

5.3.4. Within *PROG* other list manipulating functions can be defined, thereby using the entries of $[nat/elt]SIG$. For example

$$\mathbf{df}\ hd : (list \rightarrow nat)$$
$$= lrec\ (nat, (\mathbf{fn}\ x : nat, y : nat \bullet x), DONTCARE)$$

defines the head-function for lists of numbers, (with *DONTCARE* determining the outcome for "the head of the empty list *nil*").

It remains to construct some *IMPL*, i.e. some *LIST*, *NIL*, *CONS*, *LREC*. The construction below is quite analogous to the definitions of *nat*, *zero*, *succ* and *nrec* given in §5.2, and follows the suggestion of §3.6. Here are the type and expressions:

$$LIST \equiv (z : \mathbf{tp}, (elt, z \rightarrow z), z \rightarrow z)$$
$$NIL \equiv \mathbf{fn}\ z : \mathbf{tp}, f : (elt, z \rightarrow z), a : z \bullet a$$
$$CONS \equiv \mathbf{fn}\ x : elt, l : LIST \bullet (\mathbf{fn}\ z' : \mathbf{tp}, f : (elt, z' \rightarrow z'), a : z' \bullet f(x, l(z', f, a)))$$
$$LREC \equiv \mathbf{fn}\ z : \mathbf{tp}, f : (elt, z \rightarrow z), a : z \bullet (\mathbf{fn}\ l : LIST \bullet l(z, f, a)).$$

5.4. Modules

For large scale programs modularity is of utmost importance. The wide variety in modern programming languages is substantially due to the constructs for modularity: **packages** in Ada, **modules** in Modula 2, **clusters** in CLU, **programs** in Modular Pascal and so on. We shall express a very general module concept in the Lambda Calculus and design a new syntactic form for this particular expression scheme. For simplicity we do not consider typing.

In order to demonstrate the generality (not to *express* the concept) we assume in this subsection that the Lambda Calculus has been extended with assignment, assignable variables, sequencing and, if you wish, exception handling. (These extensions surely invalidate so much of the properties of the Lambda Calculus that no one would ever call it "Lambda Calculus" any more.)

The example problem that we tackle is a classical one: it is requested to write a "module" for a random number generator that allows the user to specify the "seed" (which determines the pseudo-random sequence completely) and that "exports" a parameterless function for "drawing" a next random number from the sequence. It should also be possible that several instantiations of the module be active simultaneously for several independent pseudo-random sequences.

It is known that $a_0, a_1, a_2, \cdots$ is a pseudo-random sequence if, for some suitable constants m and d, $a_i = a_{i-1} \times m \bmod d \quad (i > 0)$; a_0 is the seed. Therefore we wish to construct the solution from the following three ingredients:

var a : *nat* • ..... local store for the a_i
a := *seed* initialization
DRAW ≡ (**fn** ()• $a := a \times m \bmod d$; {result is} a)
function that yields the next random number

There are two main problems: *to control the visibility* so that the scope of **var** a does not extend over the user's program, and *to control the life-time* of **var** a so that storage is allocated for a precisely during the evaluation of the user's program *PROG*.

Quite surprisingly our successful attempts to express parameterized abstract types in the G-typed Lambda Calculus provide already the solution. In expressions (2) and (2') above, *IMPL* is invisible in *PROG even if all typing were omitted*! Moreover, had there been Pascal-like variables in the body of *genListAT*, these would exist as long as the evaluation of p (and therefore of *PROG*) would last. Thus we find:

df *rng* = (**fn** *seed*, p • **var** a : *nat* • a := *seed*; p (*DRAW*))
• .....
rng (041130, (**fn** *draw* • *PROG*))
respectively
rng (041130, (**fn** *draw* •
rng (161087, (**fn** *draw'* • *PROG'*))))
.....

It seems worthwhile to design a special syntactic form for the above scheme: module expressions and module invocations. First we show their use and then we define them formally. Here is the above program written with the module constructs.

```
df rng = (fn seed • module
                      var a : nat • a := seed; export(DRAW)
                    endmod)
• .....
  (invoke draw = rng (041130)• PROG)
  respectively
  (invoke draw = rng (041130)•
     (invoke draw' = rng (161087)• PROG'))
  .....
```

The module consists of an expression in which one subexpression is tagged with **export**. An **invoke**-expression is syntactically similar to a **df** -expression. The evaluation of **invoke** $x = em \bullet eb$ consists of evaluating the module expression *em* after replacing the part **export**(*ea*) in it by **df** $x = ea \bullet eb$. Thus it may be better to say that *eb* is imported into *em* rather than that *ea* is exported to *eb*. Formally, we consider the left-hand sides, below, as abbreviations for the right-hand sides:

module .....**export**(*ea*)....**endmod**	**fn** $p \bullet$ ..... p (*ea*)....
invoke $x = em \bullet eb$	*em* (**fn** $x \bullet eb$)

Hence, the derived evaluation rule reads:

replace	**invoke** $x =$ (**module** .....**export**(*ea*)....**endmod**)• *eb*
by	.....(**df** $x = ea \bullet eb$)....

Notes

5.4.1. One may, of course, replace $a := seed; p(DRAW)$ by **df** $drw = DRAW \bullet a := seed; p(drw)$. It thus turns out that the (first) solution can be transliterated to Pascal, so that in principle no extra module construct is needed in Pascal.

5.4.2. A formal proof that "storage for *a* is allocated precisely during the evaluation of *PROG*" can not be given before assignment and variables have been added formally to the Lambda Calculus.

5.4.3. Neither Pascal-like dynamically allocated variables, nor Algol 68 **heap** variables, facilitate a solution to the problem of controlling the life-time of *a* satisfactorily. Algol 60 has the concept of **own** variable for this purpose. But, whereas the interference of recursion and **own** variables gives problems in Algol 60, there are no such problems here (because both recursion and the above solution are expressed entirely within the Lambda Calculus).

5.4.4. Generalization to multiple export and invocation is straightforward.

5.4.5. Not only initialization "before the export" is possible, but also finalization "after the export", and even exception handling "around the export". For the latter, imagine an "exception" defined locally within the module, possibly "raised" from within *DRAW* when used

in *PROG*, and "handled" at/around the export expression within the module. Other arrangements are possible too, e.g. exporting a locally defined "exception" jointly with *DRAW* so that it may be handled from within *PROG* as well. Also, by a slight adaptation of the *rng* definition, we get a module that yields *initialized variables*: export *a* itself rather than *DRAW*. For details see Fokkinga [7].

6. Beyond Generic Typing

As shown informally in Section 5, generic typing is quite expressive; a precise characterization of the G-typable arithmetic functions is given by Fortune et al. [9]. Nevertheless there are reasons for further generalization:

- There still exist expressions that are semantically meaningful but not G-typable; e.g. $TW(TW)(K)$ where $TW \equiv \mathbf{fn}\, f \bullet (\mathbf{fn}\, x \bullet f\,(f\,(x)))$ and $K \equiv \mathbf{fn}\, x \bullet (\mathbf{fn}\, y \bullet x)$, [10].
- Functions like $\mathbf{fn}\, n, x_1, \ldots, x_n \bullet x_1 + \cdots + x_n$ for which the first argument determines the number of following arguments, are not G-typable.
- Referring to *SIG* and *IMPL* of Section 5, it seems natural to make the implementation *IMPL* into one tuple expression $<IMPL>$, from which the individual components can be retrieved by selections .1, .2 ...; the type of $<IMPL>$ would then be a tuple type $<SIG>$. (Notice the *dependency* between the first and following components within $<SIG>$.)
- One might wish to extend the type formation rules in such a way that arbitrary properties can be expressed in types, and typability means total correctness with respect to the properties expressed in the types.

Much work is, and has been, done towards the fulfillment of the last point above: the AUTOMATH project [2], Martin-Löf's Intuitionistic Theory of Types [16], and recently the Theory of Constructions [5]. Space limitations do not permit us to discuss these very promising approaches. Instead, we briefly present our own devised typing, called SVP-typing [8]. Due to its far going generalization, it is quite simple to define but, as a price to be paid, has some weak points that have been avoided consciously in AUTOMATH, the Intuitionistic Theory of Types and the Theory of Constructions:

- there is no distinction any more between types and normal value expressions;
- the evaluation of typed expressions may not terminate.

Consequently, compile-time type-checking may sometimes not terminate. This is really a pity, but hopefully not disastrous:

- we expect that type errors will be detected far more often than that the type-checker does not terminate;
- nontermination of the type-checker can be treated in the same way as nontermination of programs nowadays: an unexpectedly long type-checking time should make someone suspicious and suggests to prove termination or change the program (or typing) otherwise.

It is left open for future research whether this is a sensible approach.

6.1. SVP-Typing

The SVP-typing is a generalization of the G-typing that was already anticipated when we designed the syntax for generic constructs. Basically, types are now merely expressions that have type **tp**. In particular they may be the result of functions and components of tuples, and **tp** itself is a type (so that **tp** has type **tp**).

SVP'-typed expressions. The following grammar generates the SVP'-typable expressions. In each rule we distinguish constituent parts by suffixes f, a, b, x and r, and we stipulate that equally named constituents are equal. Moreover, for readability we write 't' for '$e^{\mathbf{tp}}$', 'tx' for '$ex^{\mathbf{tp}}$', 'tr' for '$er^{\mathbf{tp}}$', and so on.

$t ::= \mathbf{tp}$ — the type of types

$e^{tx} ::= x^{tx}$

$t ::= (x^{tx}:tx \rightarrow tr)$ — the type of functions

$e^{x:tx\rightarrow tr} ::= (\mathbf{fn}\ x^{tx}:tx \bullet eb^{tr})$

$e^{[ea/x]tr} ::= ef^{x:tx\rightarrow tr}(ea^{tx})$

$t ::= \langle x^{t1}:t1, t2\rangle$ — the type of tuples

$e^{\langle x:t1,t2\rangle} ::= \langle e\,1^{t1}, e_2^{[e_1/x]t2}\rangle$

$e^{t1} ::= ep^{\langle x:t1,t2\rangle}.1$

$e^{[ep.1/x]t2} ::= ep^{\langle x:t1,t2\rangle}.2$

An expression of type **tp** is called a *type*; we let t, ta,.... denote arbitrary types. Within $(x:t'\rightarrow t'')$, $(\mathbf{fn}\ x:t' \bullet eb^{t''})$ and $\langle x:t', t''\rangle$ identifier x is a local name whose scope extends over t'' and eb, but not over t'. If x does not occur in t'', we simply write $(t'\rightarrow t'')$, respectively $\langle t', t''\rangle$.

Notes

6.1.1. The evaluation rules and the generalization to multiple parameters, definitions and tuples are obvious and tacitly used in the sequel.

6.1.2. Due to the last rule the type of an expression is not uniquely determined.

6.1.3. Clearly, the SVP'-typing subsumes the G-typing. Thanks to the concrete notation that we have designed both a G-type and a G-typed expression are SVP'-typed expressions.

6.1.4. Functions that yield types, and tuples that contain types, are now possible. For instance, with *SIG* and *IMPL* of the previous section, we may now write:

$$\textbf{df}\ genListAT'' : (elt : \textbf{tp} \to <SIG>) = (\textbf{fn}\ elt : \textbf{tp} \bullet\ <IMPL>) \bullet \ldots\ldots$$

On the dots $genListAT''(nat)$ has type $[nat/elt]<SIG>$ so that, according to the typing rules for tuple selection:

$genListAT''(nat).1$	$(\equiv NLIST)$ has type **tp** and is the representation type for lists,
$genListAT''(nat).2$	has type $NLIST$ and is the nil for lists of numbers,
$genListAT''(nat).3$	has type $(nat, NLIST \to NLIST)$ and is the cons for lists of numbers, and
$genListAT''(nat).4$	has type $(z : \textbf{tp}, (z \to z), z \to (NLIST \to z))$ and is the lrec for lists of numbers.

6.1.5. The dependency has been generalized too. Not only type parameters and components may be referred to in later parameters, result and components, but also normal value parameters and components; e.g. $sort : (n : nat, elt : \textbf{tp}, a : array(elt, n) \to array(elt, n))$. This kind of dependency has been strived for in the design of PEBBLE, a typing system for large scale modularity [3].

6.1.6. Given *array* of type $(elt : \textbf{tp}, n : nat \to \textbf{tp})$, as above, we find that

$$ef^{(a : array(bool,7) \to \cdots)}(ea^{array(bool,3+4)})$$

is not SVP'-typed: the rule for function call requires that the parameter type and the argument type be syntactically equal. This leads to the extension below.

SVP'' -typed expressions. The grammar consists of all rules for the SVP'-typed expressions, and in addition:

$e^{t'} ::= e^{t''}$ whenever t' and t'' are semantically equivalent.

(There are several ways to define semantic equivalence; one way is to say that expressions are semantically equivalent if they can be evaluated to a common intermediate result.)

Notes

6.1.7. It is the very combination of this rule with $\textbf{tp}^{\textbf{tp}}$ that seems to allow for SVP''-typed expressions with nonterminating evaluations; cf. Meyer & Reinhold [19].

6.1.8. We conjecture that it is impossible to express the tuple constructs in the others, i.e. to replace the tuple constructs by SVP''-typed equivalent functions.

6.1.9. Now that evaluation on type positions (superscripts) is allowed, we can reformulate the grammar rules for function constructs:

$$t ::= (\mathbf{fn}\ x:\mathbf{tp} \bullet tr) \tag{1}$$

$$e^{\mathbf{fn}\ x:\mathbf{tp}\bullet tr} ::= (\mathbf{fn}\ x^{tx}:tx \bullet eb^{tr}) \tag{2}$$

$$e^{tf(ea)} ::= ef^{tf^{(\mathbf{fn}\ x:\mathbf{tp}\bullet tp)}}(ea^{tx}) \tag{3}$$

That is, $\mathbf{fn}\ x:tx \bullet tr$ plays the role of the type $(x:tx \to tr)$; it contains the same information and, indeed, $(\mathbf{fn}\ x:tx \bullet tr)(ea)$ is semantically equivalent to $[ea/x]tr$.

6.1.10. According to rule (1) above $\mathbf{fn}\ x:t' \bullet t''$ has type $\mathbf{tp}$, and according to rule (2) [taking tx, eb, tr to be $t, t', \mathbf{tp}$] it has type $\mathbf{fn}\ x:tx \bullet \mathbf{tp}$ too. This suggests to replace rule (1) by

$e^{t'} ::= e^{t''}$ whenever $t'' \leqslant t'$
$\leqslant$ is the reflexive and transitive closure generated by
$(\mathbf{fn}\ x:tx \bullet eb^{tr}) \leqslant (\mathbf{fn}\ x:tx \bullet tr)$

This approach has been studied in the context of AUTOMATH and is incorporated in some version of the Theory of Constructions.

7. Concluding Remarks

Much of the programming language concepts that we have discussed, deal with the — intuitive — notion of "abstraction", which is to neglect, consciously, some aspects of the subject under consideration. It turns out that the **fn** -construct facilitates this abstraction. Since the syntactic manipulation of forming $\mathbf{fn}\ x \bullet e$ out of e and x, is called Lambda-abstraction, we conclude from our exposition that

Lambda-Abstraction is the key to Intuitive Abstraction.

Many programming language concepts have not been discussed here, notably assignment and assignable variables, and exception handling. These two concepts in particular require a drastical extension/change of the Lambda Calculus. In [7] we have done so, and it turns out that the formalism does not change much; the properties do. There we also show how the conventional stack-based implementations may be derived in a systematic way from the the "replacement" semantics of the Lambda Calculus. Thus the applicability of the Lambda Calculus approach to programming language concepts is wider than sketched in this paper.

Finally we remark that one should not confuse "programming language concepts" with "programming concepts".

Acknowledgement. I have had much profit from our recent study group on Lambda Calculus and, earlier, from discussions with Mirjam Gerritsen and Gerrit van der Hoeven. It has been a great stimulus for

me to know that this paper would be dedicated to Leo Verbeek.

References

1. H.P. Barendregt: *The Lambda Calculus — Its Syntax and Semantics*, Studies in Logic 103, North-Holland, Amsterdam, 1981. (2nd edition 1984)
2. N.G. de Bruijn: A survey of the Automath project, *in:* J.P. Seldin & J.R. Hindley (Eds.): *To H.B. Curry — Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London, 1980.
3. R. Burstall & B. Lampson: A kernel language for abstract data types and modules, *in:* G. Kahn, D.B. MacQueen & G. Plotkin (Eds.): *Semantics of Data Types*, Lect. Notes Comp. Sci. **173** (1984) 1-50, Springer-Verlag, Berlin - Heidelberg - New York.
4. L. Cardelli & P. Wegner: On understanding types, data abstraction, and polymorphism, *Comput. Surveys* **17** (1985) 471-522.
5. Th. Coquand & G. Huet: Constructions — a higher order proof system for mechanizing mathematics, *EUROCAL 85*, Lect. Notes Comp. Sci. **203** (1985) 151-184, Springer-Verlag, Berlin - Heidelberg - New York.
6. Th. Coquand & G. Huet: A selected bibliography on constructive mathematics, intuitionistic type theory and higher order deduction, *J. Symbolic Comput.* **1** (1985) 323-328.
7. M.M. Fokkinga: *Structuur Van Programmeertalen*, University of Twente, Enschede, Netherlands, 1983. (Lecture Notes in Dutch, "Structure of Programming Languages".)
8. M.M. Fokkinga: *Over het nut en de mogelijkheden van typering*, University of Twente, Enschede, Netherlands, 1983. (Lecture Notes, in Dutch, "On the use and the possibilities of typing".)
9. S. Fortune, D. Leivant & M. O'Donnell: The expressiveness of simple and second order type structures, *J. Assoc. Comput. Mach.* **30** (1983) 151-185.
10. M. Gerritsen & G.F. van der Hoeven: Private communication, 1987.
11. J.R. Hindley & J.P. Seldin: *Introduction to Combinators and Lambda Calculus*, London Mathematical Society Student Texts **1**, Cambridge University Press, Cambridge (U.K.), 1986.
12. P.J. Landin: The mechanical evaluation of expressions, *Computer J.* **6** (1964) 308-320.
13. P.J. Landin: A correspondence between Algol 60 and Church's Lambda notation, *Comm. Assoc. Comput. Mach.* **8** (1965) 89-101, 158-165.

14. P.J. Landin: The next 700 programming languages, *Comm. Assoc. Comput. Mach.* **9** (1966) 157-166.

15. H. Langmaack: On procedures as open subroutines I, *Acta Inform.* **2** (1973) 311-333.

16. P. Martin-Löf: An intuitionistic theory of types: predicative part, *in: Logic Colloquium 1973*, pp. 73-118, North-Holland, Amsterdam, 1975.

17. L.G.L.T. Meertens: Procedurele datastructuren, *in: Colloquium Datastructuren*, Mathematisch Centrum (Currently CWI), Amsterdam, 1978.

18. L.G.L.T. Meertens & S. Pemberton: Description of B, *ACM SIGPLAN Notices* **20** (1985) 58-76.

19. A.R. Meyer & M.B. Reinhold: 'Type' is not a type — Preliminary Report, *in: ACM Conf. Record of the 13th Annual Symposium on Principles of Programming Languages* **13** (1986) 187-295.

20. R. Milner: A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978) 348-375.

21. J.C. Mitchell & G.D. Plotkin: Abstract types have existential type, *in: ACM Conf. Record of the 12th Annual Symposium on Principles of Programming Languages* **12** (1985) 37-51.

22. J.C. Mitchell: Representation independence and data abstraction (preliminary version), *in: ACM Conf. Record of the 13th Annual Symposium on Principles of Programming Languages* **13** (1986) 263-276.

23. P. Naur (Ed.): Revised report on the algorithmic language ALGOL 60, *Comm. Assoc. Comput. Mach.* **6** (1963) 1-17.

24. J.C. Reynolds: Towards a theory of type structure, *in:* B. Robinet (Ed.): *Programming Symposium*, Lect. Notes Comp. Sci. **19** (1974) 408-425, Springer-Verlag, Berlin - Heidelberg - New York.

25. J.C. Reynolds: User-defined data types and procedural data structures as complementary approaches to data abstraction, *in:* S.A. Schuman (Ed.): *New Directions in Algorithmic Languages 1975*, pp. 154-165, IRIA, France, 1976.

26. J.C. Reynolds: The essence of ALGOL, *in:* J.W. de Bakker & J.C. van Vliet (Eds.): *Algorithmic Languages*, pp. 354-372, North-Holland, Amsterdam, 1981.

27. J.C. Reynolds: Polymorphism is not set-theoretic, *in:* G. Kahn, D.B. MacQueen, G. Plotkin (Eds.): *Semantics of Data Types*, Lect. Notes Comp. Sci. **173** (1984) 145-156, Springer-Verlag, Berlin - Heidelberg - New York.

28. J.C. Reynolds: Three approaches to type structure, *in:* H. Ehrig et al. (Eds.): *Mathematical Foundations of Software Development*, Lect. Notes Comp. Sci. **185** (1985) 97-138, Springer-Verlag,

Berlin - Heidelberg - New York.

29. A. Rezus: *A Bibliography of Lambda Calculi, Combinatory Logics and related topics*, Mathematisch Centrum, Amsterdam, 1982.
30. R.D. Tennent: *Principles of Programming Languages*, Prentice Hall, 1981.
31. D. Turner: Miranda — a non-strict functional language with polymorphic types, *in: Proc. Int. Conf. on Functional Programming Languages and Computer Architecture*, Lect. Notes Comp. Sci. **201**, (1985) 1-16, Springer-Verlag, Berlin - Heidelberg - New York.
32. D. Turner: An overview of Miranda, *ACM SIGPLAN Notices* **21** (1986) 158-166.

14. P.J. Landin: The next 700 programming languages, *Comm. Assoc. Comput. Mach.* **9** (1966) 157-166.

15. H. Langmaack: On procedures as open subroutines I, *Acta Inform.* **2** (1973) 311-333.

16. P. Martin-Löf: An intuitionistic theory of types: predicative part, *in: Logic Colloquium 1973*, pp. 73-118, North-Holland, Amsterdam, 1975.

17. L.G.L.T. Meertens: Procedurele datastructuren, *in: Colloquium Datastructuren*, Mathematisch Centrum (Currently CWI), Amsterdam, 1978.

18. L.G.L.T. Meertens & S. Pemberton: Description of B, *ACM SIGPLAN Notices* **20** (1985) 58-76.

19. A.R. Meyer & M.B. Reinhold: 'Type' is not a type — Preliminary Report, *in: ACM Conf. Record of the 13th Annual Symposium on Principles of Programming Languages* **13** (1986) 187-295.

20. R. Milner: A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978) 348-375.

21. J.C. Mitchell & G.D. Plotkin: Abstract types have existential type, *in: ACM Conf. Record of the 12th Annual Symposium on Principles of Programming Languages* **12** (1985) 37-51.

22. J.C. Mitchell: Representation independence and data abstraction (preliminary version), *in: ACM Conf. Record of the 13th Annual Symposium on Principles of Programming Languages* **13** (1986) 263-276.

23. P. Naur (Ed.): Revised report on the algorithmic language ALGOL 60, *Comm. Assoc. Comput. Mach.* **6** (1963) 1-17.

24. J.C. Reynolds: Towards a theory of type structure, *in:* B. Robinet (Ed.): *Programming Symposium*, Lect. Notes Comp. Sci. **19** (1974) 408-425, Springer-Verlag, Berlin - Heidelberg - New York.

25. J.C. Reynolds: User-defined data types and procedural data structures as complementary approaches to data abstraction, *in:* S.A. Schuman (Ed.): *New Directions in Algorithmic Languages 1975*, pp. 154-165, IRIA, France, 1976.

26. J.C. Reynolds: The essence of ALGOL, *in:* J.W. de Bakker & J.C. van Vliet (Eds.): *Algorithmic Languages*, pp. 354-372, North-Holland, Amsterdam, 1981.

27. J.C. Reynolds: Polymorphism is not set-theoretic, *in:* G. Kahn, D.B. MacQueen, G. Plotkin (Eds.): *Semantics of Data Types*, Lect. Notes Comp. Sci. **173** (1984) 145-156, Springer-Verlag, Berlin - Heidelberg - New York.

28. J.C. Reynolds: Three approaches to type structure, *in:* H. Ehrig et al. (Eds.): *Mathematical Foundations of Software Development*, Lect. Notes Comp. Sci. **185** (1985) 97-138, Springer-Verlag,

Berlin - Heidelberg - New York.

29. A. Rezus: *A Bibliography of Lambda Calculi, Combinatory Logics and related topics*, Mathematisch Centrum, Amsterdam, 1982.
30. R.D. Tennent: *Principles of Programming Languages*, Prentice Hall, 1981.
31. D. Turner: Miranda — a non-strict functional language with polymorphic types, *in: Proc. Int. Conf. on Functional Programming Languages and Computer Architecture*, Lect. Notes Comp. Sci. **201**, (1985) 1-16, Springer-Verlag, Berlin - Heidelberg - New York.
32. D. Turner: An overview of Miranda, *ACM SIGPLAN Notices* **21** (1986) 158-166.

A Representation Principle for Sets and Functions

Jan Kuper

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

We present a representation principle for sets and functions, essentially meaning that sets and functions do exist in two different ways: as intuitive objects and as mathematical objects. In this paper some aspects of the relationship between these two ways are investigated. The principle has consequences for the concept of λ-calculus model and for the relationship between such models and set theory.

1. Introduction

In the literature on λ-calculus it is considered quite normal that a function can be applied to itself, whereas a set cannot be a member of itself. This difference appears to be a problem in the construction of models for the (untyped) λ-calculus, because functions (of one argument) cannot be thought of as sets of ordered pairs; for an exception, see [11]. This is a remarkable distinction, because intuitively both seem to be almost equally strange, at least if function and set are taken in their usual sense, i.e. — among other things — that the objects in the domain and range of a function are prior to that function itself, just as the members of a set are prior to that set.

In this paper we discuss a representation principle for sets and functions. For sets it says that, in a mathematical theory about sets, sets in their intuitive sense are represented by sets in their mathematical sense, i.e., as mathematical objects. For functions it works analogously. The principle is rather important to mathematical abstraction, and the possibility of both selfmembership for sets and selfapplication for functions can be understood as one of its consequences. Due to this representation principle, set-theoretical models for untyped λ-calculus, where functions are just sets of ordered pairs, become very elegant, attractive and rather trivial.

The intention of the paper is not to give a philosophical discussion of the notions of (mathematical) object, set and function. We suppose that these notions are more or less precisely understood in an intuitive way. The paper contains a sketch of the idea of representation, and intends to show its consequences for the interpretation of several axioms, rules and principles, which are used in set theory or λ-calculus.

As far as λ-calculus is concerned, a semantical point of view is taken; so the paper concentrates on functions rather than on terms. The same holds of course for set theory, but there a semantical point of view is usual.

2. The Representation Principle

The main statement of this section can roughly be formulated as follows: axiomatic set theory is not about sets, but about representatives of sets. These representatives are abstractions of sets and this form of abstraction is rather important to the mathematical way of looking at sets. In the second part of this section the same abstraction is discussed for functions.

2.1. Sets

Let us start with an example. Suppose we have a set s which consists of five objects a,b,c,d,e. Suppose further, that a,b,c together form the set p, and that t is the set which consists of d,e and p. According to ordinary mathematical usage we can now form the set $s \cup t$ which contains six objects: a,b,c,d,e and p. The question we want to discuss is: is this really possible if we think of sets in the normal intuitive way? We will argue that the answer to this question is negative.

In the normal intuitive sense the least we can say about sets is, that a set *consists of* its elements, a set is the *totality of* its elements, and the elements of a set are in some way or another *in* the set. In the example above this means that a,b,c are already in p, and that p is an object on a "higher" level than a,b and c. However, if we think of $s \cup t$ as a set in the usual way, then all elements of $s \cup t$ exist besides each other, and on the same level as each other. So if we consider p to be an element of $s \cup t$, then a,b,c cannot be elements of $s \cup t$ any more, and vice versa.

In a more daily situation we might say: if you sell your collection of stamps, you will get rid of your stamps too, and vice versa. If in the example above we think of the objects a,b,c,d,e as physical objects, teaspoons say, then it is possible to form the set s or we can form the sets p and t, but it is impossible to form the set $s \cup t$. We have to *choose* between considering a,b,c as *different* objects, or as *one* totality, i.e., as the set of a,b,c.

But also if we think of a,b,c,d,e as non-physical objects, then in order to form $s \cup t$ we have to think of p as existing on its own, independently of and on the same level as a,b,c. Thus within $s \cup t$, p is *not* the set of a,b,c in the normal, intuitive sense, but p is a (mathematical) object playing the rôle of this intuitive set. So axiomatic set theory deals with mathematical objects, called sets, and not with sets in their normal intuitive sense.

That does not mean, however, that people, at the moment they are engaged in (axiomatic) set theory, have lost their ability to form intuitive collections from these mathematical objects. So when we work on set theory, sets are present to us in two different ways: as intuitive collections of objects, and as mathematical objects representing these intuitive collections.[1] From now on the word *collection* is used for sets in their normal, intuitive sense, and the word *set* is reserved for objects representing a collection. A set is called the *representative* of a collection.[2]

There is still one point to make: the fact that a,b and c are *in* the *collection* of which p is the representative, is expressed by letting a,b,c have the $\in$*-relation* with the *set* p. So $x \in p$ holds if and only if x is in the collection of which p is the representative. As usual $\in$ is called the *membership relation* and x is called a *member* or *element* of p; see Figure 1.

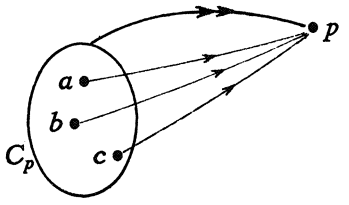

Figure 1. The collection C_p is represented by the set p. The single arrows denote the $\in$-relation, the double arrow visualizes representation.

It is important to realize that the above described abstraction is presupposed already in the iterative conception of sets, e.g., in normal *ZF*-set theory, with or without the axiom of choice, and with the axiom of foundation.

2.2. Functions

Usually a function (only functions of one variable are considered here) is thought of as a correspondence by which each object in the domain of the function is associated to precisely one object in its range. In this paper we are not concerned with functions as "rules" or as "operation processes", but we will restrict ourselves to functions in the

[1] Of course these intuitive collections are objects too, but they exist on a higher level. When in this paper the word "object" is used, first order object is meant. Furthermore, the word "collection" will mean: collection of first order objects.

[2] There is a difference between *collections* and *classes*. Classes are, at least in a Gödel-like set theory, mathematical objects which also represent collections. Sets are specific classes; proper classes represent collections which are not represented by sets (e.g., because they are too "big").

extensional sense, and consider a function as completely determined by all its individual correspondences. A function can then be seen, and in fact often is seen, as the totality of all these individual correspondences. In case of an ordinary mathematical theory this will do, but when we want to develop a mathematical theory *about* functions (i.e., when at least collections of functions are formed, and functions from functions to functions are constructed), we make the same abstraction to functions as described in Section 2.1 for collections. In other words: such a mathematical theory is about (mathematical) *objects* which play the rôle of functions as described above.

In the following the word *map* is used for functions in their normal, intuitive sense as described above, and the word *function* is used for the *representative* of a map. So a function exists on the same level as, and independently of the objects in its domain and range. On the contrary a map exists "above" the objects in its domain and range and certainly not independent of them; see Figure 2.

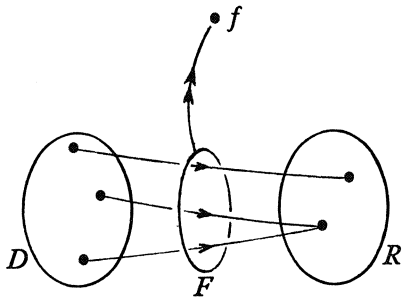

Figure 2. The map F is represented by the function f, D is the domain, R is the range of F. If x is in D, then: $f \bullet x = y$ iff $F(x) = y$.

When maps are represented by functions, then the (intuitive) act of applying a map F to an argument x — as usual, the result of this act is denoted as $F(x)$ — must be represented too. This can be done in different ways. One possibility is to have a binary *application operation* $(\bullet)$. Let f be a function representing F, then $f \bullet x = F(x)$ for all x in the domain of F, and $f \bullet x$ is undefined when F is undefined for x, i.e., when x is not in the domain of F.

A more general possibility is, however, to have a ternary *application relation* (*appl*), such that $appl(f,x,y)$ does hold when $F(x) = y$, and does not hold when $F(x) \neq y$ or when F is undefined for x, where again the function f represents the map F. The advantage of this ternary relation is, that $appl(f,x,y)$ simply does not hold for any y whenever F is undefined for x, i.e., when x is not in the domain of F, whereas in such a case $f \bullet x = y$ is meaningless.

2.3. Representation as a Mathematical Relation

Both sets and functions are by now considered as (first order) mathematical objects which, within the framework of a mathematical theory, play the rôle of (higher order) intuitive collections and maps respectively. Furthermore, between sets the binary membership relation can exist, and between functions the ternary application relation can exist.

The representation *relation* that exists between a collection and its representative, or between a map and its representative, can of course be mathematized in the sense as described in Sections 2.1 and 2.2, but we must not expect to learn much of that. Suppose we tried to do so for collections and sets, then the first thing to do would be to represent a collection A by an object a'. The next thing to do would be to examine the *mathematical* "representation relation" ρ. In this case ρ exists between a' and an already existing representative a of A, so $\rho(a',a)$ holds. That is: the best we would get is a mathematical representation of our (intuitive) representation relation, and the representation principle itself would be presupposed in the mathematical representation of it.

In the literature on models of λ-calculus it often is considered to be unsatisfying that a model of (untyped) λ-calculus looks like an abstract algebra $<D,\bullet>$, whereas interpretations of λ-terms are expected to be functions (maps); see [7,10]. A more "function-oriented" view on such a model is then given by introducing a map Φ from D to a subcollection of $D \to D$, being the collection of all one-place functions (maps) from D to D, such that $\Phi(a)(x) = a \bullet x$ for all x. An object a in D is then called a "representative" of the function (map) $\Phi(a)$ in $D \to D$.
However, the view held in this paper is, that, when introducing such a collection $D \to D$ and map Φ from D to $D \to D$, the elements of $D \to D$ are taken as mathematical objects, and so in the sense as described above already *are* representatives (called functions) of (intuitive) maps from D to D. From this point of view things do not get better when Φ is introduced. We can simply suppose D to be a subcollection of $D \to D$ (see also Sections 3.2 and 3.4), and Φ to be the identity map on D. Due to the representation principle a model of (untyped) λ-calculus is *expected* to be an algebra $<D,\bullet>$.

3. Some Questions

In this section some questions about the representation principle as described in Section 2, are discussed. These questions are:

- How many collections [maps] can be represented by the same set [function]?

- How many sets [functions] can represent the same collection [map]?
- Does every collection [map] have a representative?
- Which set [function] can (or must) represent a certain collection [map]?

These four questions are discussed in Sections 3.1 to 3.4 respectively.

3.1. Fundamental for the representation principle as described in Section 2, is that one set [function] can represent at most one collection [map] at a time.
Suppose there are two collections A_1 and A_2 represented by the same set a. The collection *generated* by this set a is the collection of all objects x for which $x \in a$ holds. Thus the only collection which can be *recognized* as the collection represented by a, is the union of A_1 and A_2. Generating is something like the *reverse* of representing, so every set must represent at most one collection.
For maps the same things can be said. Furthermore, if two maps F_1 and F_2 would be represented by the same function f, then the possibility of two different values $F_1(x)$ and $F_2(x)$ for the same argument x would be an additional problem, because what to choose for $f \bullet x$? So the same conclusion as for collections and sets holds even stronger for maps and functions.

In principle it is possible that an object does *not* represent a collection, but stands for itself (it represents teaspoons, drawing-pins or the like, so to say). The only way to express this, is by not calling such objects sets but "urelements", and to agree that only sets do generate collections. Still the collection of all objects x, for which $x \in e$ holds (where e is a non-set) is empty, and it is rather arbitrary to say that e does not represent this empty collection.
The same holds for functions: if f is a function, then the map F, generated by f, is defined for all x for which $f \bullet x$ is defined, and for those x: $F(x) = f \bullet x$. If e is a non-function (an "urargument", or an "urvalue"), then e does by convention (as in case of sets) not generate a map, but we can still think of the map E which is defined for all x for which $e \bullet x$ is defined. For non-functions there are no such x, so E is the *totally undefined* map (the *empty* map).[3] It is arbitrary to say that non-functions do not represent the empty map but something else, or maybe nothing at all.
For reasons of elegance and simplicity we will restrict ourselves to *pure sets*, i.e., sets of which all members are sets, and to *pure functions*, i.e., functions whose domains and ranges contain only functions. So all

[3] Because this map is undefined for all possible arguments, it is *as a map* precisely defined. It has just an empty domain, and there is exactly one map with domain the empty collection (though it can have more than one representative, see Section 3.2).

objects are either sets or functions, representing collections of sets, or maps from functions to functions respectively.

3.2. In general there is no a priori restriction on the *number* of sets [functions] that can represent a certain collection [map].
For collections this means that a given collection of sets can be represented by zero, one, or more than one set. If we start with a universe of objects (sets), then (in principle) every (sub)collection of these objects can exist in the intuitive way. A collection consists of its elements, so every collection is unique and completely determined by its objects. For sets however, there are some choices concerning these points to be made.

In the first place it is not necessary that there is a representative for *every* collection. It is possible that in the intuitive way there exists a collection of objects, whereas in the universe of sets there is no set with these objects as elements, i.e., which represents the given collection. If we want to be sure that certain collections are represented, then we have to require this explicitly. In set theory this is usually done by formulating some axioms and axiom schemes. Examples are:

— The *axiom of pairing*:

$$\forall x,y \exists a \forall u (u \in a \leftrightarrow u = x \vee u = y)$$

Of course the *collection* consisting of any two objects x and y simply exists. According to the axiom of pairing there is also a *set* with x and y as elements.

— The *axiom scheme of separation*:

$$\forall a \exists b \forall x (x \in b \leftrightarrow x \in a \wedge \phi(x))$$

(Where $\phi(x)$ is a formula not containing b as a free variable). Here too, any subcollection of the collection generated by a set a exists, at least in principle, but that does not mean that all these subcollections are represented. According to the axiom scheme of separation at least those sub*sets* of a exist, for which we can formulate (in the language of the theory) a property that is distinctive for the elements of this subset.

— The *power set axiom*:

$$\forall a \exists b \forall x (x \in b \leftrightarrow x \subseteq a).$$

This axiom states that there exists a representative of the collection of all subsets of a given set, i.e., of all sets which represent subcollections of the collection generated by this given set. By the way, this does not imply that for all of these subcollections there exists a representative, i.e., we can have "incomplete" power sets.
These axioms of set theory are formulated in some general way, i.e., they say that there exist representatives for all collections of a certain

sort, e.g., for all collections consisting of two elements.

In the second place it is not necessary that there is *at most* one set representing a given collection. If we want any collection to be represented by at most one set, we also must state this requirement explicitly. This is what the *axiom of extensionality* does:

$$\forall a,b,x\,((x \in a \leftrightarrow x \in b) \rightarrow a = b).$$

Usually in literature on set theory the distinction between sets and collections, as described here, is not made, at least not explicitly. Mostly "set" and "collection" are considered to be synonyms, as are other notions like family, class, etc. A problem different from the one discussed in Section 2.1, that arises from this point of view, is that it is hard to understand *why* stating several axioms of set theory is necessary. Especially the axiom of extensionality seems to be a necessary truth (cf. [3], where the notion "analytic" is used), for how can there be two different collections consisting of the same elements, if a collection is precisely that thing which consists of its elements? Extensionality is an intrinsic property of the notion of set or collection. One gets the impression that formulating the axiom of extensionality is superfluous, or at least it feels like an axiom of underlying logic. According to the representation principle this axiom is not a necessary truth, and indeed must be stated explicitly (as must the other axioms). In the light of the representation principle it can also easily be understood that axioms, e.g., the axiom of extensionality, can be rejected from most theories on sets (*ZF* for instance), and that we can replace them by other axioms. The axiom of extensionality can for instance be replaced by an axiom saying that, if there is a representative for a collection, then there are two of them. Or even that non-empty collections have as many representatives as elements. Of course we can also leave it open, and then we will not know how many representatives a given collection has.

For maps and functions the situation is basically the same. Starting with a universe of objects (functions) any *map* from any (sub)-collection of this universe to any other (sub)collection can exist in the intuitive way (at least in principle). Because we are only concerned with maps in the extensional sense, every map can be seen as the totality of all its correspondences, so every map is unique and completely determined by its correspondences.
Now the same possibilities which occurred for sets, arise for functions, understood as representatives of maps. Every map can in principle be represented by zero, one, or more functions, i.e., by zero, one, or more objects in the given universe. Here too requirements for functions must be stated explicitly. Extensionality for instance is formulated (in the more general way) as:

$$\forall f,g,x,y\,((appl(f,x,y) \leftrightarrow appl(g,x,y)) \rightarrow f = g).$$

When the representation relation between maps and functions is itself considered as a mathematical relation (cf. Section 2.3), then extensionality of functions is sometimes defined by requiring that the map Φ from D to $D \to D$ is one-to-one; see [10] and also Section 2.3 for the meaning of D and Φ. In such a definition it is tacitly assumed that $D \to D$ *itself* is extensional. However, we may not rely on this, because $D \to D$ does not contain *maps* from D to D, but *functions* representing these maps.

Concerning the *existence* of functions no (examples of) axioms will be given, but a universe of functions will be defined as a subcollection of some given universe of sets. Advantages of this approach are that it is rather straightforward and in accordance with our usual conception of function.

In order to keep the definitions below readable, we assume that the theory of sets describing the structure of the given universe of sets, is strong enough to guarantee the existence of certain sets needed in the definitions. We also assume that this theory contains the axiom of extensionality. Later on the definitions may be reformulated in such a way that they are independent of the chosen theory of sets, but that will make them more complicated and less readable.

The first definition is a well-known definition of *ordered pair* $<x,y>$.

Definition 1. $<x,y> = \{\{x\},\{x,y\}\}$. □

In this definition $\{x\}$, $\{x,y\}$ and $\{\{x\},\{x,y\}\}$ are *sets*, so $<x,y>$ is also a set, i.e., an object in the given universe of sets.

Definition 2. The *field* Δa of a set a is

$$\Delta a = \{x \mid \exists y(<x,y> \in a \vee <y,x> \in a)\}.$$ □

Definition 3. A set a is *applicatively closed* iff

$$\forall x(x \in a \to \Delta x \subseteq a).$$ □

Definition 4. A set b is the *applicative closure* of a set a iff

- $\Delta a \subseteq b$,
- b is applicatively closed,
- if c is an applicatively closed set and $\Delta a \subseteq c$, then $b \subseteq c$.

The applicative closure of a is denoted as $APC(a)$.

Furthermore, $APC^*(a) = \{a\} \cup APC(a)$. □

Definition 5. A set f is a *function* (*in the sense of set theory*) iff

- $\forall u \in f \; \exists x,y \; (u = <x,y>)$
- $\forall x,y_1,y_2 \; (<x,y_1> \in f \wedge <x,y_2> \in f \to y_1 = y_2)$.

Notation: $fnc_s(f)$. □

Note that if f is a function then $\Delta f = D(f) \cup R(f)$, where $D(f)$ and $R(f)$ are the domain and the range of f respectively.

Definition 6. A set f is a *pure function (in the sense of set theory)* iff

$$\forall x\,(x \in APC^*(f) \rightarrow fnc_s(x)).$$ □

It can easily be seen that if f is a pure function, then every $x \in APC^*(f)$ is a pure function.

Clearly, the ternary *application relation (in the sense of set theory)* for functions is defined as follows.

Definition 7. $appl_s(a,x,y) \leftrightarrow fnc_s(a) \wedge <x,y> \in a$. □

As a universe of functions we can take from the universe of sets the collection of all pure functions (in the sense of set theory), or even a certain subcollection of it. I.e., from the universe of sets we can leave out all sets which are not relevant in the definition of pure function (like sets with more than two elements, if such a set is not a pure function itself), or which are superfluous from a functional point of view (like $\{x\}$, $\{x,y\}$, $<x,y>$ in a case that $f \bullet x = y$ for some pure function f).

As mentioned before, these definitions may be reformulated independently of the axioms of the underlying theory of sets. For example, the resulting universe of functions can then be non-extensional.

Summarizing: it depends on the chosen theory if there exist zero, one or more representatives of a certain collection or map.

3.3. In the previous section one of the questions discussed was, whether it is necessary that every collection [map] is represented. In this section the question is, whether it is *possible* that every collection [map] is represented by an object. Because of Russell's paradox and related antinomies (e.g., Cantor's paradox), the answer to this question is definitely negative.

In case of set theory we can form the intuitive collection R of all sets x for which $x \in x$ does not hold. But we can *not* represent this collection by a set r. For suppose we can, and suppose r is in R. Because r is a representative of R it follows that $r \in r$. Then, by definition of R, r not in R. Thus, while r is a representative of R, $r \notin r$. But then, by definition of R again, r is in R.
So if the collection R can be represented by a set r, we get *two* forms of Russell's paradox:

$$r \text{ is in } R \quad \text{iff} \quad r \text{ is not in } R,$$

$$r \in r \quad \text{iff} \quad r \notin r.$$

For functions Russell's paradox is present in exactly the same way. Suppose we have a universe of functions, and any intuitive map from any subcollection of this universe to any (possibly other) subcollection can be represented by a function. Suppose in particular, that there are

functions representing maps from the entire universe to itself (e.g., a function i representing the identity map I which is supposed to be defined for the entire universe). Such a function can be applied to itself (e.g., $i \bullet i = i$ or, alternatively, $appl(i,i,i)$ holds). Now we can define the *map* F, with domain the entire universe of functions, as follows:

- if $appl(x,x,x)$ does *not* hold, then $F(x) = x$,
- if $appl(x,x,x)$ *does* hold, then $F(x) = y$, where y is arbitrary, but $y \neq x$.[4]

Russell's paradox for functions can be shown as follows: let f be the representative of F. Suppose $appl(f,f,f)$ holds. Then by definition of F, $F(f) \neq f$. Because f represents F, this implies that $appl(f,f,f)$ does not hold. But then, by definition of F again, $F(f) = f$. And so, because f represents F, $appl(f,f,f)$ holds.
So here too we have two forms of Russell's paradox:

$$F(f) = f \quad \text{iff} \quad F(f) \neq f,$$

$$appl(f,f,f) \quad \text{iff} \quad \text{not } appl(f,f,f).$$

The conclusion for both sets and functions is the same: not every collection [map] can be represented by a set [function]. It depends on the set or function theory at hand which collections or maps cannot be represented. Mostly however, there will be present some principle like the axiom of separation (in set theory), which is not a very strong principle with regard to the existence of sets. It plays an important rôle in preventing "dangerous" collections from being represented. For set theory the axiom of separation implies that collections that are too "big" (for example the collection of all sets) can not be represented.
For function theory such a principle might state something like: given a function f and a formula $\psi(x,y)$, then there exists a function g (of course, g may not be free in $\psi(x,y)$) with the same domain as f such that, for each x in this domain:

- if there is a y such that $\psi(x,y)$ holds, then $g \bullet x = y$ (of course y must be unique),
- otherwise: $g \bullet x = f \bullet x$.

Such a principle for function theory would imply that maps which are too "powerful" (e.g., having the entire universe as their domain, like the total identity map) cannot be represented.

Of course there can be (mathematical) objects representing collections [maps] which cannot be represented by a set [function], but these

[4] y is a constant value, say. Then y must be such that $appl(y,y,y)$ does not hold. Because of the assumption that every map is represented, such a y exists, e.g., let y be the representative of the constant map $Y(x) = i$ for all x, then $y \bullet y = i$ and $y \neq i$.

objects cannot exist *inside* the given universe of sets [functions]. *Outside* this universe, however, there can exist objects which do the job (called *proper classes* in the case of set theory, and for the moment without a name in the case of function theory). Then we have to suppose a bigger universe of objects and the problem reoccurs.

The presence of Russell's paradox does not mean — as is often concluded from it — that (possibly indirect) selfreference is not allowed, e.g., a set being a member of itself, or a function being applied to itself. In the next section some remarks on this point will be made.

3.4. In this section the question is discussed *which* set [function] may represent a specific collection [map].
Clearly, there are no a priori restrictions at this point. Any set can represent any not too big collection and any function can represent any not too powerful map, provided of course that every set or function is the representative of at most one collection of sets or of at most one map from functions to functions respectively. For sets there is no reason why, for instance, the representative a of a collection A of sets should not lie inside this very collection. If a is in A, then the representation principle implies, that $a \in a$ is the case. It will be clear that this is no *real* selfmembership, but just a mathematical form of it (as are all instances of the membership relation).[5]
There also is no reason why the representative t of a collection T should not lie inside a collection U, which in turn is represented by a set u in the collection T. In such a case both $t \in u$ and $u \in t$ hold. And so on.[6]

In case of sets, these possibilities are usually excluded by the *axiom of foundation*, saying (if the axiom of extensionality is accepted):

$$\forall a \neq \emptyset \; \exists x \in a (x \cap a = \emptyset).$$

which is motivated by the iterative conception of sets. In the iterative conception we start with a possibly empty collection of urelements. We then proceed in stages, and the first stage after the basic stage contains all subcollections of this basic stage. If the basic stage contained no urelements, then the first stage contains only one member: the empty collection. The next stage again contains all the subcollections of this first stage, and so on. It is thereby understood that a stage contains all the collections of all the previous stages as well.
In such a process it is obvious that for every collection there is a first stage at which this collection comes into existence, and that all members of this collection already existed at lower stages. That is

[5] Because of the mental state one gets in when imagining "real" selfmembership, nonfounded sets might be called "dizzy sets".

[6] If in the sequel "selfmembership" is used, cases like $t \in u$ and $u \in t$, etc., will be meant too.

precisely what the axiom of foundation intends to say, and it is clear that according to it no collection can be a member of itself.
The iterative conception of sets proceeds in forming collections of collections and calls them sets, so in this conception sets *are* collections of sets, which in turn *are* collections of sets, etcetera. So the iterative conception of sets admits sets like $\{a,b,c,p\}$, where $p = \{a,b,c\}$. But as was argued in Section 2, these sets can only exist when we think of p as in some way or another *representing* the collection of a, b and c.[7]
The possibility of non-founded sets is a simple consequence of what is considered here to be the normal mathematical way of looking at collections and objects.
Of course selfmembership strongly violates our intuition on collections, but the iterative conception already violates this intuition in the same way, it only is less apparent.[8]

For functions the situation is basically the same. If we think of a map in the normal intuitive and extensional way, then the level on which a map exists is higher then the level on which the objects in the domain and range of the map exist. We might describe an iterative conception of functions in the same way as we did for sets: we start with a possibly empty domain of "urarguments" and a possibly empty range of "urvalues". The next stage contains all maps from the given domain to the given range, though this could be only the empty map (the totally undefined map). The process is the same as described for sets, i.e., a stage contains all maps from any subcollection to any (possibly other) subcollection of the union of all foregoing stages, and map and function are just different names for the same thing.
In such an iterative conception of functions, functions cannot be applied to themselves. But due to mathematical abstraction a mathematical theory *about* maps does not deal with maps, but with representatives of maps, called functions. Here too, there is a priori no reason why a function f should not lie inside the domain of the map F represented by f. But then $F(f)$ is defined, and so $f \bullet f$ too.
In an analogous fashion it is possible that $f \bullet x$ and $x \bullet f$ are both well-defined. And so on.[9]

---

[7] Also the necessity of stating several axioms of set theory does not follow from it; cf. Section 3.2.

[8] There are other arguments defending the axiom of foundation, like the possibility of simple and powerful induction principles. In a universe with non-founded sets such principles are only valid in its well-founded part. For the non-founded part however, some weaker or more complicated induction principles can be found; see [1,2]. But such an argument is concerned with technical, and not with principal aspects. Furthermore there are nice and technically simple applications of non-founded sets, e.g., in models of λ-calculus [11] and in models for (projective) geometry [9].

[9] If in the sequel "selfapplication" is used, cases like $f \bullet x$ and $x \bullet f$, etc., will be meant too.

The conclusion of this is, that the possibility of both selfapplication of functions and selfmembership of sets is the same sort of consequence from the same mathematical point of view. It is therefore remarkable that in the literature on models of λ-calculus selfapplication for functions is considered to be quite normal, whereas selfmembership for sets is considered to be impossible, or at least undesirable.

In order to understand selfapplication maps (functions) are often considered intensionally (e.g., in [7]), whereas collections (sets) are mostly thought of extensionally. Intensionally a function or map is considered as a *rule* of correspondence, which can (often) be applied to any object whatever and not only to objects in a certain domain. Because a rule of correspondence can itself be considered as an object, selfapplication is possible (at least in principle).
When collections (sets) are understood intensionally, then we think of them as containing all objects whatever which have a specific property, or sometimes "set" and "property" are even considered as identical notions. But then the set of all sets (for the moment "set" has its usual meaning) has itself the property of being a set, and so is a member of itself. Thus intensionally selfmembership can be understood equally well as selfapplication. However, thinking intensionally of sets (in an unrestricted way) leads to Russell's paradox, and so does an (unrestricted) intensional approach to functions; see Section 3.3, where the map F was defined as a rule.

As stated before, set-theoretical models for λ-calculus, in which functions are considered as sets of ordered pairs, are mostly excluded. The possibility of applying a function f to itself would imply that there is an ordered pair

$$<f,y> \in f.$$

With the standard definition of ordered pair

$$<a,b> = \{\{a\},\{a,b\}\},$$

this would give

$$f \in \{f\} \in <f,y> \in f$$

which is generally rejected. But as argued above, selfmembership is just as strange, or maybe just as nice, as selfapplication, both being the same consequence of the same mathematical abstraction. So if we accept one of them, we have to accept the other, and if we reject one of them, we have to reject the other.
By now we do expect that set-theoretical models of (untyped) λ-calculus are possible, elegant and rather trivial. Indeed they are as is shown in [11].

Returning to one of the questions discussed in Section 3.2 we have to ask *which* non-founded sets do exist in the universe of sets.

The same question can be posed for a universe of functions.
As far as set theory is concerned, a rather general way to decide this, is by means of the *axiom of universality*, intuitively saying that there exist copies of every possible sort of non-founded sets; see [1,12], where non-founded sets are considered within a theory of sets including the axiom of extensionality. In order to describe the axiom of universality more formally, here too the axiom of extensionality will be accepted. The following definitions are needed.

Definition 8. A set a is *transitive* (notation: $tr(a)$) iff for all x

$$x \in a \rightarrow x \subseteq a. \qquad \square$$

Definition 9. A *structure* is an ordered pair $<a,r>$ where a is a set, and $r \subseteq a \times a$. $\square$

Note that a structure is a set.

Definition 10. A structure $s = <a,r>$ is *extensional* (notation: $es(s)$) iff for all $x,y,z \in a$

$$(<x,y> \in r \leftrightarrow <x,z> \in r) \rightarrow y = z. \qquad \square$$

Definition 11. The *internal* $\in$*-structure* $\in(t)$ of a set t is the structure $<t,e>$, where

$$e \subseteq t \times t$$

and

$$<x,y> \in e \leftrightarrow x \in y. \qquad \square$$

The *axiom of universality* now states that for every extensional structure s there exists a transitive set t whose internal $\in$-structure is isomorphic with s (where the meaning of isomorphism is obvious):

$$\forall s\,(es(s) \rightarrow \exists t\,(tr(t) \wedge \in(t) \sim s)).$$

With respect to functions we can again take the subcollection from the universe of sets consisting of all pure functions (in the sense of set theory) and, if preferable, we can leave out extensionality. The result is a rich universe of functions where selfapplication is possible, though not for all functions. It seems that it contains a rather universal model for typed and untyped λ-calculus, where untyped λ-calculus can be a part of typed λ-calculus, i.e., in which untyped terms are of a specific type.

4. Rules

In Section 3 some aspects of the representation principle were discussed from a point of view which is common practice in set theory, i.e., axiomatic, concerning the existence of objects, etc. In this section an approach from the rule-oriented point of view of λ-calculus is given, be it that the rules are interpreted in a semantical way, whereas in λ-calculus they play a syntactical rôle in the first place. In this section it

will become even more obvious that the representation principle is the same for collections and maps, i.e., that we make the same abstraction when we think of collections or maps in a mathematical way. In Section 4.1 some remarks are made on notational matters, and in Section 4.2 some rules are discussed.

4.1. Notation

In λ-calculus the λ-notation was invented to denote maps (functions) as such; see [4]. Let $\theta(x)$ be an expression which has for a given x a certain value in some range of values (the natural numbers, say). The relationship between all the values of x and the corresponding values of $\theta(x)$ is a map (function). As we have seen we have to distinguish between a map and its representing function(s). The *map* described a moment ago is denoted as $\lambda x.\theta(x)$, the *function* representing this map as $\lambda x.\theta(x)$.[10]

Because more than one function can represent the same map, there is a problem at this point: we have to *choose which* of these representatives is denoted by $\lambda x.\theta(x)$. The choice is arbitrary in principle.[11] Depending on the specific system of functions, it is also possible that $\lambda x.\theta(x)$ does not exist. Due to the paradox of Russell, for some θ it is even necessary that $\lambda x.\theta(x)$ does not exist; cf. Sections 3.2 and 3.3.
For collections and sets the same things can be said: the *collection* of all objects characterized by some formula $\phi(x)$ is denoted by $\{\!|x\,|\phi(x)|\!\}$, and an (in principle arbitrary) *representative* of this collection is denoted by $\{x\,|\phi(x)\}$. Here too it is possible, and for some ϕ necessary, that $\{x\,|\phi(x)\}$ does not exist.

A first similarity in notation is almost immediate: if we replace $\{x\,|\phi(x)\}$ by $\sigma x.\phi(x)$, likewise expressed as "the set of all x such that $\phi(x)$", then the notations for sets and functions reflect the same abstraction principle in the same way. We will stick here to the usual set-theoretic notation with braces.

Furthermore, if we do not think of true and false on an intuitive level, but as two values in the collection of truth values $\{\!|true, false|\!\}$, then we can express that x is a member of a set a as follows: $x \in a = true$. From this point it is only a small step to think of a as a *function* which is defined for all members of a, such that $a \bullet x = true$ for all x in the collection represented by a. Taken as a function we

[10] Denoting the domain of a map or function within the notation, as in $\lambda x \in \alpha.\theta(x)$, will not be considered here.

[11] This is in agreement with a usual definition of a model of λ-calculus; see [7,10]. In (models of) combinatory logic λ-abstraction (sometimes called λ^*, [.], <.>) is defined as an "applicative product" of some values, thus *fixing* the choice of the representative denoted by $\lambda^* x.\theta(x)$ to a certain value (for instance: if a does not depend on x then $\lambda^* x.a = k \bullet a$, where k is a specific function for which $k \bullet x \bullet y = x$ holds for all values of x and y, so $k \bullet a$ is a function with constant value a).

agree that a is undefined for all other x. If we take for *true* and *false* two specific functions, then this gives us a possibility for embedding sets within a universe of functions.
The point to be made here however, is a second similarity in notation: $x \in a$ can simply be elaborated to $a \bullet x = true$, so $x \in a$ for sets corresponds to $f \bullet x = y$ for functions. We will stick here to usual set-theoretic notation, and consider truth values in the normal intuitive way.

In the sequel $\theta(x)$ denotes an expression whose value can depend on x, i.e., x can be free in $\theta(x)$, and $\phi(x)$ denotes a formula whose truth can depend on x, i.e., x can be free in $\phi(x)$. It is thereby understood that $\theta(x)$ and $\phi(x)$ contain as non-logical constants only $\bullet$ and $\in$ respectively, and variables range over functions and sets respectively.

4.2. Some Rules Compared

The first rule is the *rule of α-conversion*. For functions this rule says

$$\lambda x.\, \theta(x) = \lambda y.\, \theta(y),$$

if $\theta(x)$ does not contain y.

For sets we get an obvious translation of this rule

$$\{x \mid \phi(x)\} = \{y \mid \phi(y)\},$$

if $\phi(x)$ does not contain y.

So the α-rule says that the object denoted by $\lambda x.\, \theta(x)$ or $\{x \mid \phi(x)\}$ does not depend on the choice of the bound variable, provided of course that this object exists (which is supposed to be the case whenever appropriate).

The second rule is the *rule of β-conversion*. For functions this rule says

$$(\lambda x.\, \theta(x)) \bullet a = \theta(a).$$

For sets it says

$$a \in \{x \mid \phi(x)\} \leftrightarrow \phi(a).$$

The rules μ and ν say for functions

$$\text{if } a = b \text{ then for all } c,\ c \bullet a = c \bullet b,$$
$$\text{if } a = b \text{ then for all } c,\ a \bullet c = b \bullet c.$$

(It is supposed here, that $c \bullet a$ is defined if and only if $c \bullet b$ is defined. The same holds for $a \bullet c$ and $b \bullet c$.)

For sets these rules say (in the same order)

$$\text{if } a = b \text{ then for all } c,\ a \in c \leftrightarrow b \in c,$$
$$\text{if } a = b \text{ then for all } c,\ c \in a \leftrightarrow c \in b.$$

The rules β, μ and ν speak for themselves. In a semantical surroundings, as chosen here, they just express ordinary logical facts.

Two less trivial rules are the rules which are known in λ-calculus as ξ and η. Together they constitute extensionality. The rule ξ says for functions: if for all x, $\theta_1(x)$ is defined iff $\theta_2(x)$ is defined, and for those x, $\theta_1(x) = \theta_2(x)$, then

$$\lambda x.\theta_1(x) = \lambda x.\theta_2(x).$$

The meaning of the rule ξ can be understood as follows: if we have two expressions $\theta_1(x)$ and $\theta_2(x)$ which for all x either both have the same value, or both are undefined, then of course the *map* denoted as $\lambda\!\!\!\lambda x.\theta_1(x)$ is the same map as denoted by $\lambda\!\!\!\lambda x.\theta_2(x)$.
As we saw, it is possible to have more than one function representing this map, and it is of course also possible to choose for $\lambda x.\theta_1(x)$ one of these representatives, and for $\lambda x.\theta_2(x)$ another. When the rule ξ is accepted, this cannot be the case, but on the contrary there is a *canonical* representative of the map in question, which is denoted by any λ-expression of the form $\lambda x.\theta(x)$ that denotes a function representing this map. The rule ξ does *not* say that there is at most one representative of a map, but only that one of the possibly more than one representatives is chosen as the canonical representative.

For sets the rule ξ says: if for all x, $\phi_1(x)$ holds iff $\phi_2(x)$ holds, then

$$\{x \mid \phi_1(x)\} = \{x \mid \phi_2(x)\}.$$

The same comment as for functions can be given here: of course the collection $\{\!|x \mid \phi_1(x)|\!\}$ is the same as the collection $\{\!|x \mid \phi_2(x)|\!\}$. But only when ξ is accepted there is a canonical representative of this collection, denoted as $\{x \mid \phi_1(x)\}$, $\{x \mid \phi_2(x)\}$ or any other "brace expression" of the form $\{x \mid \phi(x)\}$ denoting a set which represents the given collection. Here too it is still possible that there is more than one representative for the same collection.

In order to introduce the last rule (η) recall the notion of generation, which in some sense is the reverse of representation. Let f be an arbitrary function. This function generates the map $\lambda\!\!\!\lambda x.f{\bullet}x$. The η-rule now says that $\lambda x.f{\bullet}x = f$, meaning that the function denoted as $\lambda x.f{\bullet}x$ is the same function as f. One could say that if the η-rule is accepted, then $\lambda x.f{\bullet}x$, understood as representative of $\lambda\!\!\!\lambda x.f{\bullet}x$, "remembers" which function generated $\lambda\!\!\!\lambda x.f{\bullet}x$. There can for instance be two different functions f and g generating the same map, i.e., $f{\bullet}x = g{\bullet}x$ for all x, and according to the η-rule $\lambda x.f{\bullet}x \neq \lambda x.g{\bullet}x$.
Again, for sets the same remarks can be made. A set a can be seen as generating the collection $\{\!|x \mid x \in a|\!\}$. According to the η-rule $\{x \mid x \in a\} = a$, so the η-rule chooses for $\{x \mid x \in a\}$ the set that generated $\{\!|x \mid x \in a|\!\}$. Here too, when the η-rule is accepted, then $\{x \mid x \in a\} \neq \{x \mid x \in b\}$, even if a and b (with $a \neq b$) generate the same

collection, i.e., even if $x \in a \leftrightarrow x \in b$ holds for all x.

In the literature on λ-calculus the meaning of the η-rule sometimes is stated as: every object is a function; e.g., in [10]. For sets this naturally translates into: every object is a set. Although indeed the (unrestricted) η-rule is inconsistent with non-functions [non-sets], the existence of these is mainly a matter of convention; see Section 3.1. It is possible that the η-rule is rejected, and still every object is a function [set]. For instance, in such a case there can exist two functions a and b, with $\lambda x.a \bullet x = b$ and $\lambda x.b \bullet x = a$, and yet $a \neq b$ [or a and b can be two sets with $\{x \mid x \in a\} = b$ and $\{x \mid x \in b\} = a$, and yet $a \neq b$]. Thus the precise meaning of the η-rule is not that every object is a function [set], but it gives an answer to the question which object to choose for $\lambda x.a \bullet x$ [or for $\{x \mid x \in a\}$].

As can be seen from the above, and as is also well known in λ-calculus, ξ and η do not imply extensionality on their own. Both ξ and η are needed to obtain extensionality. In the light of the representation principle this is clear immediately: let, in the case of set theory, a and b be two sets generating the same collection. Then the ξ-rule says that $\{x \mid x \in a\}$ and $\{x \mid x \in b\}$ denote the same canonical representative of this collection. Thus according to the η-rule a and b must both denote this canonical representative, thus $a = b$. So when ξ and η are accepted, any collection has at most one representative which is precisely the meaning of extensionality for sets.
Of course, the same remarks hold for functions.

5. Concluding Remarks

The representation principle for sets and functions, as described in this paper, is relevant for a unification of different approaches to sets and functions. The axiomatic approach of set theory can be applied to functions, and the rule oriented approach of the λ-calculus can be applied to sets. However, the relationship between these approaches must be analyzed further.

The representation principle is rather universal, and can be applied to many branches of mathematics, though it is possible that the properties of the principle change slightly, when it is applied to other subjects. Numbers, for instance, can also be considered in two different ways: as intuitive objects, and as mathematical objects, where a number in its mathematical sense represents an intuitive number. It seems interesting to investigate what sorts of representation can be distinguished, and to which branches of mathematics they apply.

References

1. M. Boffa: Sur la théorie des ensembles sans axiome de fondement, *Bull. Soc. Math. Belg.* **21** (1969) 16-56.
2. M. Boffa: Induction et récursion en théorie des ensembles sans axiome de fondement, *Fund. Math.* **66** (1970) 241-253.
3. G. Boolos: The iterative conception of set, *J. of Philos.* **68** (1971) 215-232. Reprinted in: P. Benacerraf & H. Putnam (Eds.): *Philosophy of Mathematics* (second edition), Cambridge University Press, Cambridge, 1983.
4. A. Church: *The Calculi of Lambda-Conversion*, Princeton University Press, Princeton, 1941.
5. H.B. Curry & R. Feys: *Combinatory Logic* (Volume 1), North-Holland, Amsterdam, 1958.
6. K. Gödel: *The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis with the Axioms of Set Theory*, Princeton University Press, Princeton, 1940.
7. J.R. Hindley & J.P. Seldin: *Introduction to Combinators and λ-Calculus*, Cambridge University Press, Cambridge, 1986.
8. J.-L. Krivine: *Introduction to Axiomatic Set Theory*, Reidel, Dordrecht, 1971.
9. J. Kuper: Non-founded sets and their relevance to the foundations of geometry, Report 86-01 (preprint), Department of Computer Science, University of Leiden, Leiden, 1986.
10. A.R. Meyer: What is a model of the lambda calculus?, *Inform. and Control* **52** (1982) 87-122.
11. M. von Rimscha: Mengentheoretische Modelle des λK-Kalküls, *Arch. Math. Logik Grundlag.* **20** (1980) 65-73.
12. M. von Rimscha: Universality and strong extensionality, *Arch. Math. Logik Grundlag.* **21** (1981) 179-193.
13. D.S. Scott: Axiomatizing set theory, *in*: T.J. Jech (Ed.): *Axiomatic Set Theory* (Part II), Amer. Math. Soc., Providence, 1974, pp. 207-214.
14. D.S. Scott: Combinators and classes, *in*: C. Böhm (Ed.): *λ-calculus and Computer Science Theory*, Lect. Notes Comp. Sci. **37** (1975) 1-26, Springer-Verlag, Berlin, Heidelberg, New York.
15. J.R. Shoenfield: Axioms of set theory, *in*: J. Barwise (Ed.): *Handbook of Mathematical Logic*, North-Holland, Amsterdam, 1977, pp. 321-344.
16. H. Wang: The concept of set, *in*: H. Wang: *From Mathematics to Philosophy*, Routledge and Kegan Paul, London, 1974, pp. 181-223. Reprinted in: P. Benacerraf & H. Putnam (Eds.): *Philosophy of Mathematics* (second edition), Cambridge University Press, Cambridge, 1983.

Unification – An Overview

Rudolf Sommerhalder

Department of Mathematics and Computer Science
Delft University of Technology
P.O. Box 356, 2600 AJ Delft, The Netherlands

Unifying terms s and t in an equational theory E means finding a substitution σ such that $\sigma(s) =_E \sigma(t)$. Unification in the empty theory, i.e., the theory without axioms and so consisting of free terms only, is important for the implementation of programming languages such as Prolog; the general case is of importance for programming languages such as OBJ and for logic programming in general. In this overview, results about the decidability of the question whether given terms s and t are unifiable in an equational theory as well as unification algorithms for particular theories of different unification types are presented. Recent methods to construct a unification algorithm for a theory E_1+E_2 using given unification algorithms for the theories E_1 and E_2 are discussed. Finally, the complexity of unification is considered. Unification in the empty theory is complete for **P** and for *co*-**NL**; unification is NP-complete for instance in Boolean rings and worse in general.

1. Introduction

Unification is concerned with equation solving in a general setting. Given are two expressions e_1 and e_2 constructed using variables, constants and operations. Unifying these two expressions means finding an assignment ϕ to the variables occurring in e_1 or e_2 such that $\phi(e_1) = \phi(e_2)$; in other words, substituting the values assigned to the variables in the equation, results in an equality, i.e., the equation $e_1 = e_2$ is solved. The assignment ϕ is called a *unifying substitution*, or a *unifyer*.

For example, let $e_1 \triangleq f(x,g(a,b))$ and $e_2 \triangleq f(g(y,b),x)$. Then $\phi \triangleq \{x := g(a,b), y := a\}$ is a unifyer of e_1 and e_2, because $\phi(e_1) = f(g(a,b),g(a,b)) = \phi(e_2)$. In this example $\phi(e_1)$ is identical to $\phi(e_2)$. This will not always be required. We also consider the variant where a variable assignment ϕ is called a solution to the equation $e_1 = e_2$ if $\phi(e_1) =_E \phi(e_2)$ for some equivalence relation $=_E$. For example, consider once more the equation $f(x,g(a,b)) = f(g(y,b),x)$ and assume that the operation f is commutative, that is $f(x,y) = f(y,x)$ for all x and y. Now $\phi \triangleq \{y := a\}$ is also a solution, because

$$\phi(f(x,g(a,b))) = f(x,g(a,b)) =_E f(g(a,b),x) = \phi(f(g(y,b),x)),$$

where $=_E$ denotes the equivalence relation induced by the axiom of

commutativity.

Unification is an important problem; it arises in many areas of computer science such as:

- *Computational logic.* A central component of all current theorem provers is a procedure to unify first order terms. This problem was first studied by Herbrand [17], who also gave an algorithm to compute a unifier. Automatic theorem provers cannot adequately handle equational axioms such as commutativity and associativity, if these are treated as just any other axiom. A traditional approach to these problems is to develop unification algorithms which directly handle properties such as commutativity, by computing unifiers relative to an equivalence relation $=_E$.
- *Programming languages.* Traditionally, procedures are called by name. A deviation from this is *pattern directed* invocation as in PLANNER [19], QA4 [37], and Prolog, see [24] for example. Also, the fundamental mode of operation for the programming language SNOBOL is to detect the occurrence of a string within a larger string. If the substring contains SNOBOL *don't care variables*, then the occurrence problem is an instance of the string unification problem. Prolog is an attempt to use Horn clause logic, a subset of full first order predicate logic, as a programming language. The success of Prolog as a programming language critically depends on the availability of efficient implementations, which in turn depends on the availability of fast string unification algorithms.
- *Computer algebra* or *symbol manipulation.* Here matching algorithms also are of utmost importance. For example, the integrand in a symbolic integration problem may be matched against a set of patterns to determine the class of integration problems to which it belongs, so as to trigger the appropriate action.
- *Deductive databases.* In a deductive database not all information is explicitly stored as data. Instead, the data consists of facts and rules using which implied facts can be deduced and given input can be checked for integrity. Applying such rules heavily depends on unification.

In [41] Siekmann discusses these and other application areas in more detail.

2. Preliminaries on Formalism

The concepts used are from universal algebra, see for instance [15]. These may be set out as follows.

1. V is an alphabet of symbols denoting variables.
2. For each $n \geqslant 0$, F_n is an alphabet of symbols denoting n-ary functions, i.e., functions of the type $A^n \to A$, where A is the carrier of the algebra.

3. $F \triangleq \bigcup \{F_i \mid i \geqslant 0\}$.
4. T is the smallest set such that
   1. $V \cup F \subseteq T$,
   2. If $t_i \in T$, $1 \leqslant i \leqslant n$, and $f \in F_n$, then $f(t_1,\ldots,t_n) \in T$.

   The elements of T are called *terms*. They are formal objects replacing the intuitive "expressions" of the introduction.
5. $var(t)$ denotes the set of all variables occurring in the term t and $var(t_1,\ldots,t_n)$ denotes $\bigcup \{var(t_i) \mid 1 \leqslant i \leqslant n\}$. A term t is called *ground* if $var(t) = \emptyset$. The set of all ground terms is denoted by T_g.
6. The set T can be made into an algebra $(T,\hat{F})$ by specifying an operation $\hat{f} : T^n \to T$ for every $f \in F_n$ as follows: $f(t_1,\ldots,t_n) \triangleq f(t_1,\ldots,t_n)$.
7. A *substitution* σ is an endomorphism of $(T,\hat{F})$ such that there are finitely many $x \in V$ for which $\sigma(x) \neq x$. A substitution σ can therefore be represented by a finite set $\{x_1 := t_1,\ldots,x_n := t_n\}$, that will also be denoted by σ. For a substitution σ we define
   1. $D(\sigma) \triangleq \{x \in V \mid \sigma(x) \neq x\}$ and
   2. $X(\sigma) \triangleq \bigcup \{var(\sigma(x)) \mid x \in D(\sigma)\}$.

   The identity substitution is denoted by ϵ, in set representation $\epsilon = \emptyset$. The set of all substitutions is denoted by Σ.
8. Let σ be a substitution and $W \subseteq V$ a set of variables. Then σ *substitutes away from* W if and only if $X(\sigma) \cap W = \emptyset$.
9. An *equation* is a pair of terms $s,t \in T$ and is denoted by $s = t$. The equation is *valid* in an algebra A if and only if $\phi(s) = \phi(t)$ for every homomorphism $\phi : T \to A$.
10. Let E be a set of equations (axioms). The *equational theory* presented by E is the finest congruence relation $=_E$ of $(T,\hat{F})$ that contains $\{(\sigma(s),\sigma(t)) \mid \sigma \in \Sigma, (s = t) \in E\}$. The equational theory is recursively decidable [enumerable] if and only if $=_E$ as a set of pairs is recursively decidable [enumerable].
11. A substitution $\sigma \in \Sigma$ is an *E-unifier* of the terms s and t if $\sigma(s) =_E \sigma(t)$. The set of all E-unifiers of s and t is denoted by $U\Sigma_E(s,t)$.
12. A *unification problem* is a pair $\langle s,t \rangle_E$ for which the set $U\Sigma_E(s,t)$ has to be determined.
13. Let $W \subseteq V$. E-equality is extended to substitutions by defining $\sigma =_{(E,W)} \tau$ if and only if $\sigma(x) =_E \tau(x)$ for all $x \in W$, in which case σ and τ are said to be E-equal in W.
14. Substitution ρ is an *instance* of substitution σ, and σ is *more general* than ρ, in symbols $\rho \leqslant_{(E,W)} \sigma$, if and only if there exists a $\tau \in \Sigma$ such that $\rho =_{(E,W)} \tau \cdot \sigma$.

15. For a given unification problem $<s,t>_E$ we do not wish to compute the whole set $U\Sigma_E$, but a smaller set that can be used to represent $U\Sigma_E$. To that purpose we define:
1. A set $CU\Sigma_E(s,t) \subseteq U\Sigma_E(s,t)$ is a *complete set of unifiers* of s and t if and only if for every $\rho \in U\Sigma_E(s,t)$ there is a $\sigma \in CU\Sigma_E(s,t)$ such that $\rho \leqslant_{(E,W)} \sigma$, where $W = var(s,t)$.
2. The set $\mu U\Sigma_E(s,t)$ of *most general unifiers* of s and t is a complete set of unifiers $CU\Sigma_E(s,t)$ such that for all $\rho,\sigma \in \mu U\Sigma_E(s,t)$, $\rho =_{(E,W)} \sigma$ whenever $\rho \leqslant_{(E,W)} \sigma$, where $W = var(s,t)$.
3. If for all $\sigma \in \mu U\Sigma_E(s,t)$ and a set $Z \subseteq V$ of variables we have $X(\sigma) \cap Z = \emptyset$, then $\mu U\Sigma_E(s,t)$ is called the set of *most general unifiers away from* Z. (The concept *away from* for a complete set of unifiers $CU\Sigma_E(s,t)$ is defined in the same way).

16. An operation related to unifying is *matching*.
1. A term s *matches* a term t if and only if there is a substitution σ such that $\sigma(s) = t$.
2. The sets $M\Sigma_E(s,t)$, $CM\Sigma_E(s,t)$ and $\mu M\Sigma_E(s,t)$ are defined in the same way as for unifying substitutions in 15 above.
3. It follows from the results summarized in Section 6 on complexity, that it can be safely conjectured that matching is simpler than unifying.

17. In all of the above, we drop the subscript E when $E = \emptyset$, that is, if we are considering equality in the algebra $(T,\hat{F})$ instead of in the algebra $(T,\hat{F})/E$.

In the sequel attention is restricted to recursively decidable, finitely presentable, equational theories.

3. Decidability of Unifiability

First consider the following fundamental problem: is it recursively decidable whether or not terms s and t are unifiable? That is, is $\lambda st\,[(\exists\,\sigma \in U\Sigma_E)[\sigma(s) =_E \sigma(t)]]$ a recursively decidable predicate?

The answer to this question obviously depends on the equational theory involved. In the case of the empty theory, the answer to the question is "yes", this is a recursively decidable predicate. The algorithm below finds a unifier of a finite non-empty set S of terms if one exists and reports failure otherwise. The algorithm can be found in almost every book on Prolog and Logic Programming; see for example [27].

Definition. The *disagreement set* of a finite non-empty set S of terms is defined as follows. Find the leftmost position at which at least two terms in S have different symbols. Extract from each term in S the smallest subterm beginning at that position. The set of all these subterms is the disagreement set. □

For example, if $S = \{f(g(x),h(y),a),f(g(x),z,b)\}$, then the disagreement set is $\{h(y),z\}$.

The unification algorithm is as follows.

input: a finite non-empty set S of terms
output: a unifier σ
method:

$\sigma = \emptyset$;
while $\sigma(S)$ is not a singleton **do**
$D :=$ the disagreement set of $\sigma(S)$;
if there exist $x,t \in D$ such that
x is a variable and $x \notin var(t)$ **then** $\sigma := \sigma \cdot \{x := t\}$
else report that S is not unifiable and halt
od.

A proof of the total correctness of the above algorithm can be found in [27]. It can also be shown that for every s and t, $\mu U \Sigma(s,t)$ is either empty or a singleton and that the above algorithm produces a most general unifier.

The algorithm can be very inefficient, due to the *occur check*. This is the test $x \notin var(t)$ in the algorithm above. Consider the following example from [5]:

$$S = \{p(x_1,\ldots,x_n),p(f(x_0,x_0),f(x_1,x_1),\ldots,f(x_{n-1},x_{n-1}))\}.$$

In the different iterations of the **while**-loop the following substitutions are obtained:

$$\sigma = \{x_1 := f(x_0,x_0)\}$$
$$\sigma = \{x_1 := f(x_0,x_0), x_2 := f(f(x_0,x_0),f(x_0,x_0))\}$$
$$\sigma = \{x_1 := f(x_0,x_0), x_2 := f(f(x_0,x_0),f(x_0,x_0)), x_3 := f(f(f(x_0,x_0),f(x_0,x_0)),f(f(x_0,x_0),f(x_0,x_0)))\}$$

and so forth. In the final substitution we have $x_n := t$ where the term t has $2^n - 1$ occurrences of the symbol f; thus, performing the occur check takes exponential time.

In Section 6 on complexity results it will be shown that the most general unifier can be found in linear time.

Although a linear time algorithm is known, most Prolog implementations combat the complexity by simply dispensing with the occur check. This can result in non-termination and in strange results, as in the following example from [32].

```
:- op(100, xfx, less_than).
X less_than s(X).
3 less_than 2 :- s(X) less_than X.
```

The answer to the goal ?- 3 less_than 2. will be "yes". In [32] Plaisted describes a method that will insert occur checks where necessary and

claims that this does not appreciably slow down the execution of most Prolog programs.

Maluszynski and Komorowski go further than that. In [28] they give a sufficient condition to replace run-time unification in Prolog programs by term matching. The reason for this is that matching can be accomplished in $O(\lambda n\,[(\log n)^2])$ time on a parallel machine and there is no fast parallel algorithm for unification known while there are reasons to believe that it does not exist; see the discussion on complexity results in Section 6.

Returning to the question whether the predicate $\lambda st\,[(\exists\,\sigma \in U\,\Sigma_E)\,[\sigma(s) =_E \sigma(t)]]$ is recursively decidable. There exist non-empty equational theories for which this predicate is recursively decidable, and also theories for which unifiability is not recursively decidable.

Consider for example the equational theory of arithmetic as presented by Peano's axioms or in any other suitable way. The unification problem in this theory is precisely Hilbert's tenth problem and is therefore not recursively decidable.

In the sequel we will consider some special theories which are presented by combinations of the following axioms.

A	associativity	$f(f(x,y),z) = f(x,f(y,z))$
C	commutativity	$f(x,y) = f(y,x)$
$\mathbf{D_L}$	left-distributivity	$f(x,g(y,z)) = g(f(x,y),f(x,z))$
$\mathbf{D_R}$	right-distributivity	$f(g(x,y),z) = g(f(x,z),f(y,z))$
D	distributivity	$\mathbf{D_L} + \mathbf{D_R}$
I	idempotence	$f(x,x) = x$
U	unit	$f(1,x) = f(x,1) = x$

In the sequel, the empty theory, that this, the equational theory that does not have any axioms and terms are therefore equal if and only if they are identical, is denoted by $\varnothing$.

The status of the unifiability problem in different theories is as follows:

- it is recursively decidable in the theories mentioned in Figure 1.
- and recursively undecidable in those listed in Figure 2.
- the status of the unifiability problem is open in the theories given in Figure 3.

The **D** + **A** unification problem is also of interest with respect to Hilbert's tenth problem. An axiomatization of arithmetic sufficient to pose Hilbert's tenth problem involves the axioms **A** and **D** and some other axioms regarding the integers. The **D** + **A**-unsolvability result implies that the unsolvability of Hilbert's tenth problem does not

Theories	References
∅	[17,35,36]
A	[33,42]
C	[40,42]
I, C + I	[34]
D_L, D_R, U	[2]
A + C, A + C + I	[26,43,11]
A + I, D + A + I	[46]

Figure 1.

Theories	References
D + A, D + A + C	[46]
D_L + A + U	[2]

Figure 2.

Theories
D
D + C
D + U
D_L + A
D_L + U

Figure 3.

depend on any specific property of the integers.

4. Unification in Specific Theories

We have seen above that in the empty theory every unifyable pair of terms has precisely one unifier. This is not the case in all equational theories. There are equational theories where every unifiable pair of terms has a finite number of most general unifiers, equational theories where at least some unifiable terms have an infinite number of most general unifiers and also equational theories where some unifiable terms have no most general unifier at all. Thus the number of most general unifiers induces the following classification of equational theories.

Definition.

1. An equational theory is *unitary*, if for every pair of unifiable terms s and t the set $\mu U \Sigma_E(s,t)$ has precisely one element.
2. An equational theory is *finitary*, if for every pair of unifiable terms s and t the set $\mu U \Sigma_E(s,t)$ is non-empty and finite.

3. An equational theory is *infinitary*, if for every pair of unifiable terms s and t the set $\mu U \Sigma_E(s,t)$ is non-empty and there are terms s and t such that $\mu U \Sigma_E(s,t)$ is infinite.
4. An equational theory is *nullary*, or of unification type 0, if there are unifiable terms s and t such that $\mu U \Sigma_E(s,t) = \emptyset$. □

As we have seen above, the free theory $\emptyset$ is unitary.

The theory **C** is finitary. Consider for example $<f(x,y),f(a,b)>_C$, where f is commutative. There are *two* most general unifiers, namely, $\sigma_1 = \{x := a, y := b\}$ and $\sigma_2 = \{x := b, y := a\}$.

The standard $\emptyset$-unification algorithm can easily be adapted to commutative functions. To unify $f(s_1,s_2)$ and $f(t_1,t_2)$, where f is commutative, it is necessary to try to unify s_1 with t_1 and s_2 with t_2 and also try to unify s_1 with t_2 and s_2 with t_1. Continuing in this fashion, the **C**-unification problem is reduced to a number of $\emptyset$-unification problems exponential in the number of occurrences of the commutative function symbol.

The equational theory **A** is infinitary. Consider for example the terms $f(a,x)$ and $f(x,a)$. These two terms have the following most general unifiers.

$\sigma_1 = \{x := a\}$

$\sigma_2 = \{x := f(a,a)\}$

$\sigma_3 = \{x := f(a,f(a,a))\}$

$\sigma_4 = \{x := f(a,f(a,f(a,a)))\}$

and so on.

In [42,45,33] can be found descriptions of adaptations of the standard $\emptyset$-unification algorithm so that it can handle associative functions.

For associative functions it is convenient to drop the distinction between $f(x,f(y,z))$ and $f(f(x,y),z)$ and represent them both by an argument list $[_f x,y,z]_f$. Assume given the argument lists of two terms to be unified. The algorithm proceeds as follows.

1. Consider the elements of the list one-by-one in left-to-right order. Determine the first non-empty disagreement set. If one list is exhausted before the other, unification fails.
2. If the disagreement set does not contain a term t and a variable x, such that $x \notin var(t)$ then unification fails with this substitution.
3. Otherwise, two unification subprocesses are created, one exploring the substitution $x := t$, the other the substitution $x := f(t,u)$, where u is a new variable.
4. The algorithm returns all substitutions constructed by successfully terminating subprocesses.

Consider the following example, where f is an associative function symbol, a,b and c are constants and all other letters variables. The terms to be unified are $f(x,y)$ and $f(a,f(b,c))$, thus the input to the algorithm are the lists $[_f x,y]_f$ and $[_f a,b,c]_f$. In the sketch of the computation given below, argument lists are denoted just using square brackets, dropping the subscript f.

$[x,y] = [a,b,c]$
1. $x := a$, $[a,y] = [a,b,c]$
    1. $y := b$, $[a,b] = [a,b,c]$, FAIL
    2. $y := f(b,v)$, $[a,b,v] = [a,b,c]$
        1. $v := c$, $[a,b,c] = [a,b,c]$, O.K.
        2. $v := f(c,z_1)$, $[a,b,c,z_1] = [a,b,c]$, FAIL
2. $x := f(a,u)$, $[a,u,y] = [a,b,c]$
    1. $u := b$, $[a,b,y] = [a,b,c]$
        1. $y := c$, $[a,b,c] = [a,b,c]$, O.K.
        2. $y := f(c,z_2)$, $[a,b,c,z_2] = [a,b,c]$, FAIL
    2. $u := f(b,w)$, $[a,b,w,y] = [a,b,c]$
        1. $w := c$, $[a,b,c,y] = [a,b,c]$, FAIL
        2. $w := f(c,z_3)$, $[a,b,c,z_3,y] = [a,b,c]$, FAIL

Two most general unifiers are obtained, namely

$\sigma_1 = \{x := a, y := f(b,c)\}$ and

$\sigma_2 = \{x := f(a,b), y := c\}$.

Much work has been done on associative-commutative unification because of its practical significance in automatic theorem proving. The equational theory $\mathbf{A} + \mathbf{C}$ is finitary. Complete algorithms have been developed by Livesey and Siekmann [26] and by Stickel [43]. Let $+$ denote a binary, associative and commutative function symbol and let

$s = (x+(x+y)) + (f(a+(a+a)) + (b+c))$ and

$t = ((b+b) + (b+z)) + c$

be the terms to be unified, where x,y and z are variables and a,b and c constants. The terms are represented by argument lists, thus $s = [x,x,y,f(a+(a+a)),b,c]$ and $t = [b,b,b,z,c]$.

Stickel [43] has proved that arguments common two both lists can be canceled in pairs without changing $U_{\Sigma_{A+C}}(s,t)$. Thus the given problem is equivalent with unifying the lists $[x,x,y,f(a+(a+a))]$ and $[b,b,z]$. Consider first two argument lists, which contain only variables, $[x,x,y,u]$ and $[v,v,z]$ in our current example. If σ is a unifier and t a specific term, then twice the number of occurrences of t in $\sigma(x)$ plus the number of occurrences of t in $\sigma(y)$ plus the number of occurrences of t in $\sigma(u)$ must be equal to twice the number of occurrences of t in $\sigma(v)$ plus the number of occurrences of t in $\sigma(z)$. Thus unification of argument lists is related to solving linear homogeneous diophantine equations

$$\sum_{i=1}^{m} a_i x_i = \sum_{i=1}^{n} b_i y_i .$$

In our current example the equation is $2x+y+u = 2v+z$.

Any positive integer solution to such an equation can be obtained as a linear combination of elements from a finite basis set of solutions. This finite set can be enumerated by a backtracking procedure using a bound on the values of the variables. Huet [20] describes an algorithm to enumerate a basis set of solutions. The following seven solutions form a basis for our current example $2x+y+u = 2v+z$.

	x	y	u	v	z
s_1	0	0	1	0	1
s_2	0	1	0	0	1
s_3	0	0	2	1	0
s_4	0	1	1	1	0
s_5	0	2	0	1	0
s_6	1	0	0	0	2
s_7	1	0	0	1	0

Any linear combination of these is a solution to the equation. However, because we have no zero in the unification problem, we must consider all subsets of the basis with the constraints that the sum of coefficients in any column must be non-zero and must be equal to 1 if the corresponding term is not a variable. Fortenbacher [13] describes a method to reduce the number of subsets which need to be considered. The reduction can be very significant.

Consider for instance $\{s_1, s_2, s_6, s_7\}$. The corresponding solution is $x = s_6+s_7$, $y = s_2$, $u = s_1$, $v = s_7$ and $z = s_1+s_2+2s_6$. Substituting the constants and simplifying, we arrive at the unifier $\{x := s_6+b, z := f(a+(a+a))+y+s_6+s_6\}$.

Termination of **A + C**-unification has remained an open problem for a long time, a termination proof of Livesey and Stickel's algorithm has been given by Fages [11].

Little or nothing is known about why the combination of an infinitary theory **A** and a finitary theory **C** gives a finitary theory **A + C**, whereas the combination of another infinitary theory **D** and the finitary theory **C** results in an infinitary theory **D + C**.

The first theory of unification type 0 has been presented by Fages and Huet in [10]. The theory has two constants a and 1, a one-place function q, and a two place-function f. The axioms are

$f(1,x) = x$

$q(f(x,y)) = q(y)$

Consider the following unifiers of the terms $q(x)$ and $q(a)$.

$\sigma_0 = \{x := a\}$

$\sigma_1 = \{x := f(x_1, a)\}$

$\sigma_2 = \{x := f(x_2, f(x_1, a))\}$

... ...

$\sigma_i = \{x := f(x_i, f(..., a)...)\}$

Let $S \triangleq \{\sigma_i \mid i \geq 0\}$. Then S is a complete set of unifiers $CU\Sigma_E(s,t)$ and also σ_{i+1} is strictly more general than σ_i. Assuming this, it follows that there does not exist a $\mu U\Sigma_E(s,t)$. Let R be any $CU\Sigma_E(s,t)$ and $W = \{x\}$. For every $\sigma \in R$ there is an i such that $\sigma \leqslant_{(E,W)} \sigma_i <_{(E,W)} \sigma_{i+1}$, because S is complete. On the other hand, there is a $\rho \in R$ such that $\sigma_{i+1} \leqslant_{(E,W)} \rho$, because R is complete. But then $\sigma <_{(E,W)} \rho$, whence R is not a $\mu U\Sigma_E(s,t)$.

Recently there also have been found "natural" equational theories of unification type 0, namely the theory of idempotent semigroups; see [3,39].

We have seen equational theories of unification type 0, 1 and ∞. There exists no hierarchy of theories of bounded unification type. That is, if E is a suitable equational theory which is neither nullary nor unary, then there is no integer n such that for all unifiable pairs $\langle s,t\rangle_E$ the number of most general unifiers is at most n. A proof can be found in [6]. The proof is a generalization of the technique shown in the following example.

Let f be a commutative function symbol and h any free binary function symbol. The unification problem $\langle f(x,y), f(a,b)\rangle_C$, where a and b are constants, has $n = 2$ most general unifiers, namely $\{x := a, y := b\}$ and $\{x := b, y := a\}$. Construct a new unification problem namely $\langle h(f(x_1,y_1), f(x_2,y_2)), h(f(a,b), f(a,b))\rangle_C$. This problem has $n^2 = 2^2 = 4$ most general unifiers, namely

$\{x_1 := a, y_1 := b, x_2 := a, y_2 := b\}$,

$\{x_1 := a, y_1 := b, x_2 := b, y_2 := a\}$,

$\{x_1 := b, y_1 := a, x_2 := a, y_2 := b\}$ and

$\{x_1 := b, y_1 := a, x_2 := b, y_2 := a\}$.

The result of course does not depend on commutativity, but gives a general method to construct new terms which have n^2 most general unifiers from given terms having n most general unifiers.

5. Combining Equational Theories

For each equational theory of interest a unification algorithm must be designed and implemented. A general design methodology is not known. However, for equational theories presentable by convergent term rewriting systems, there is an algorithm that automatically

generates unification procedures; see Fay [12]. Some progress has been made by Yelick [50], Tiden [47], Herold [18] and Kirchner [23]. These authors describe algorithms to produce a unification algorithm for an equational theory E_1+E_2 given unification algorithms for theories E_1 and E_2. The combination algorithms cannot handle arbitrary theories, furthermore, there must be no interaction between the two theories. Yelick's method, which can be applied to *regular, collapse-free* theories, is described below.

Definition.

1. An equational theory is *regular* if and only if $var(s) = var(t)$ for every axiom $s = t$ of the theory.
2. An equational theory is *collapse-free* if and only if the theory does not contain an axiom $s = t$, where either s or t is a variable and the other term is not. □

First, it is assumed that the sets of function symbols handled by the different unification algorithms are mutually disjoint. Let E be a presentation of an equational theory. Then $\pi = \{E_1, E_2, \ldots, E_n\}$ is called a *partioned presentation of* E if and only if

1. $\emptyset \in \pi$,
2. $\bigcup \{E_i \mid 1 \leqslant i \leqslant n\} = E$,
3. $F(E_i) \cap F(E_j) = \emptyset$ for all i and j, $1 \leqslant i, j \leqslant n$; where $F(E_m)$ denotes the set of function symbols occurring in axioms of E_m, $F(E)$ denotes the set of all function symbols and $F(\emptyset)$ denotes the set of all free function symbols, that is, symbols which do not occur in any axiom.

Such a partioned presentation induces an equivalence relation $\sim$ on $F(E)$ as follows.

$f \sim g$ if and only if either

1. there is a block $E_i \in \pi$ such that $f, g \in F(E_i)$, or
2. $f \notin X$ and $g \notin X$, for all $X \in \pi$.

	$\pi = \{E_1, E_2, \emptyset\}$	$F(E) = \{+, *, a, b, f\}$
E_1:	$x+y = y+x$ $(x+y)+z = x+(y+z)$	$F(E_1) = \{+\}$
E_2:	$x*y = y*x$ $(x*y)*z = x*(y*z)$ $a*a = a$	$F(E_2) = \{*, a\}$
$\emptyset$:		$F(\emptyset) = \{b, f\}$

Figure 4.

Figure 4 contains an example of a partioned presentation of an equational theory.

The combined unification algorithm begins by transforming the input terms into simpler ones, containing only function symbols from a subset $F(E_i)$, for which by assumption an E_i-unification algorithm is known. The transformation consists of replacing subterms whose top-function symbol does not belong to the set $F(E_i)$ by new variables. The information lost in this way, is saved in the form of a substitution. For example, considering the set $F(E_2)$ of function symbols

term t	is replaced by term $\hat{t}$
$x*(a+y)$	$x*v_1$
$x*(a*b)$	$x*(a*v_2)$
$x+y$	v_3

where v_1, v_2 and v_3 are new variables. The newly constructed terms can then be unified using an E_2-unification procedure, which is assumed to be given. To restore a term $\hat{t}$ to its original t, a substitution σ such that $\sigma(\hat{t}) = t$, i.e., a matcher is constructed.

The recursive procedure to E-unify terms using E_i-unification algorithms to unify subterms, is as follows.

procedure E-unify

input: terms s and t
output: a set of E-unifiers of s and t
method:

1. If s and t are both variables, return $\{t := s\}$.
2. If s is a variable and t is not, return the result of var_unify(s,t). Similarly, if t is a variable and s is not, return the result of var_unify(t,s).
3. If the top-function symbols of s and t are not $\sim$-equivalent, return $\emptyset$.
4. Otherwise the top-function symbols of s and t are $\sim$-equivalent. Assume that these symbols belong to theory E_i.
    1. Compute the reduced terms $\hat{s}$ and $\hat{t}$ and the corresponding matchers σ_s and σ_t, such that $\sigma_s(\hat{s}) = s$ and $\sigma_t(\hat{t}) = t$.
    2. Let $\sigma = \sigma_s \cup \sigma_t$.
    3. Let $P = E_i$-unify$(\hat{s}, \hat{t})$.
    4. Return $\cup \{\text{map\_unify}(\rho,\sigma) \mid \rho \in P\}$.

In the above map_unify(ρ,σ) computes the set of E-unifiers of $\{<\rho(x), \sigma(x)> \mid x \in D(\rho) \cup D(\sigma)\}$ and the procedure var_unify is specified as follows.

procedure var_unify

input: a variable v and a term t

output: a set of unifiers of v and t
method:

Let $\hat{t}$ be the reduced term and σ_t the matching substitution, i.e., $\sigma_t(\hat{t}) = t$. Also assume that the top-function symbol of t belongs to the subtheory E_i.

1. If $v \notin var(t)$, return $\{v := t\}$.
2. If $v \in var(t)$ and $v \notin X(\sigma_t)$
   1. let $P = E_i\text{-unify}(v,\hat{t})$,
   2. return $\bigcup \{\text{map\_unify}(\pi,\sigma_t) \mid \pi \in P\}$.
3. If $v \in var(t)$ and $v \in X(\sigma)$, return $\emptyset$.

Consider the unification of $s = b+(x*y)$ and $t = a+z$ in the equational theory E with partioned presentation $\{E_1,E_2,\emptyset\}$ as given in Figure 4. Both are non-variable terms and the top-function symbol (+ in both cases), belongs to theory E_1, thus case 4 of E-unify applies. Thus $\hat{s} = v_1+v_2$ and $\hat{t} = v_3+z$ and the combined restoring substitution is $\sigma = \{v_1 := b, v_2 := x*y, v_3 := a\}$. E_1-unifying $\hat{s}$ and $\hat{t}$ produces the set $\{\rho_1,\rho_2\}$ where $\rho_1 = \{v_3 := v_1, z := v_2\}$ and $\rho_2 = \{v_3 := v_2, z := v_1\}$. Now map_unify$(\rho_1,\sigma)$ and map_unify(ρ_2,σ) are called. map_unify(ρ_1,σ) returns $\emptyset$, because $a \in F(E_2)$ must be unified with $b \in F(\emptyset)$. To compute map_unify(ρ_2,σ), the term a must be unified with term $x*y$. These terms are E_2-unifiable with a single most general unifier $\{x := a, y := a\}$. Using this, E-unify(s,t) returns $\{z := b, v_1 := b, v_2 := a*a, x := a, y := a\}$. We can check the result: $b+(a*a) =_E a+b$. In [50] Yelick sketches the proofs of correctness and termination. Full proofs are to be found in her Master's thesis, MIT Laboratory for Computer Science.

Tiden [47] gives an extension of this method and proves that it is correct for the whole class of finitary equational theories and that it terminates for the class of collapse-free equational theories. Herold [18] also gives an algorithm that is totally correct for the class of regular collapse-free equational theories. He does not replace subterms with new variables but with new constants in such a way that E-equal subterms are replaced with the same constant and claims that this is more efficient in many cases.

6. Complexity of Unification

In this section we will mainly be concerned with $\emptyset$-unification; most published results concern the unification of free terms. The following notation will be used.

L denotes the set of all languages that can be accepted by a $\lambda n[\log n]$ space-bounded *deterministic* Turing machine.

NL denotes the set of all languages that can be accepted by a $\lambda n[\log n]$ space-bounded *non-deterministic* Turing machine.

P denotes the set of all languages that can be accepted by a polynomial time-bounded *deterministic* Turing machine.

NP denotes the set of all languages that can be accepted by a polynomial time-bounded *non-deterministic* Turing machine.

The following notation is used to denote reductions between problems.

$A \leqslant_L B$ means that there is a function f that can be computed by a $\lambda n[\log n]$ space-bounded deterministic Turing transducer such that $(\forall x)[x \in A \text{ iff } f(x) \in B]$.

$A \leqslant_P B$ means that there is a function f that can be computed by a polynomial time bounded deterministic Turing transducer such that $(\forall x)[x \in A \text{ iff } f(x) \in B]$.

In Section 3 on decidability of unification an algorithm for Ø-unification has been described which has exponential running time in the worst case. Better algorithms are known, in particular, the most general unifier can be computed in linear time using the algorithm of Paterson and Wegman [31].

As we have seen the size of terms constructed during unification may be exponential in the size of the terms to be unified. Thus representing terms as trees is out of the question. Paterson and Wegman use *directed acyclic graphs* (*dags*) in which common subexpressions are represented by a single subgraph. The nodes are labeled by function symbols and variable symbols. A node labeled with a k-place function symbol has outdegree k and the outgoing arcs are labeled 1 to k, so that we can refer to the i^{th} son. Variable nodes have outdegree 0 and there is one node for each variable. The terms to be unified are represented by a single (not necessarily connected) dag with two distinguished nodes corresponding to the top-function symbols of the terms involved. Computing a most general unifier of two terms is equivalent to computing a certain equivalence relation on the nodes of the corresponding dag.

An equivalence relation on the nodes of a dag is *valid* if it has the following properties.

1. If two function nodes are equivalent, their corresponding sons are equivalent in pairs.
2. Each equivalence class is homogeneous, that is, it does not contain two nodes with distinct function symbols.
3. The equivalence classes may be partially ordered by the partial order on the given dag.

Paterson and Wegman [31] prove that the terms corresponding to nodes u and v are unifiable if and only if there is a valid equivalence relation, such that u and v are equivalent. In that case there also exists a unique minimal valid equivalence relation, that corresponds to the most general unifier. All the nodes in an equivalence class of a valid

equivalence relation represent the same term. Thus with a given valid equivalence relation corresponds a unifying substitution that assigns to every variable the term corresponding to the equivalence class which contains that variable.

The well-known *UNION-FIND* algorithm, see [1] can be used to handle the equivalence relation, which results in an $O(\lambda n\,[n\,\alpha(n)])$ time algorithm, where α is the inverse of Ackermann's function and thus grows extremely slowly. Setting sons of nodes equivalent when their fathers are, is called "propagating the equivalence". Paterson and Wegman achieve the linear running time of their algorithm by propagating the equivalence in a carefully ordered way, taking one completed equivalence class at a time.

Martelli and Montanari [30] describe an algorithm and its implementation in Pascal. The standard unification problem is an equation $<s=t>$. Martelli and Montanari's algorithm uses transformations of sets of equations into other sets of equations, which are equivalent to the given ones in that both have the same sets of unifiers. Examples of such transformation rules are

1. *Term reduction.* An equation $f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)$ may be replaced by the set of equations $\{s_i = t_i \mid 1\leqslant i \leqslant n\}$.
2. *Variable elimination* Let E be a set of equations containing the equation $x = t$, where x is a variable. The new set of equations is obtained by applying the substitution $\{x:=t\}$ to all terms occurring in the equations of $E-\{x = t\}$ and then adding the equation $x = t$.

The efficiency of the algorithm is obtained by handling equations in groups called multi-equations. The running time of the algorithm is $O(\lambda n\,[n \log n])$.

In typical applications, such as theorem provers, the unification algorithm is not used on very long terms but very often on rather small terms. In these circumstances the asymptotic difference between Paterson and Wegman's and Martelli and Montanari's algorithm cannot be exploited. Martelli and Montanari [30] claim that their algorithm performs better most of the time in these circumstances. An additional advantage of the algorithm is that it can be generalized to handle E-unification, see Kirchner [23].

Dwork, Kanellakis and Mitchell [8] also study the time complexity of the unification problem. They show that unification is complete for **P** with respect to log-space reducibility $\leqslant_L$, using the *Monotone Circuit Value* problem.

The monotone circuit value problem *MCV* is defined as follows.

$MCV \triangleq \{(\beta_0,\beta_1,\ldots,\beta_n) \mid$ for all $i\ (0\leqslant i<n)$, $\beta_i = IN(0)$ or $\beta_i = IN(1)$ or $\beta_i = AND(j,k)$ or $\beta_i = OR(j,k)$ such that
1. if β_i is an input ($IN(0)$ or $IN(1)$) then the index i appears at most once in the sequence $(\beta_0,\ldots,\beta_n)$; if β_i is a gate ($AND(j,k)$ or $OR(j,k)$) then the index i appears at

most twice in the sequence $(\beta_0,\ldots,\beta_n)$ and $i > j > k$;
2. β_n is an or-gate $OR(j,k)$ whose output according to the circuit and its given inputs is equal to 1}.

The problem *MCV* is complete for **P** with respect to $\leqslant_L$; see [14].

Dwork Kanellakis and Mitchell use labeled dags to represent terms. In order to get complexity results where the size of the instance corresponds with the length of a string representation of the terms, Dwork et al. introduce *simple dags* as dags where the only nodes with indegree greater than 1 are leaves. Thus there are two versions of the unification problem.

UNIFY $\triangleq$ {$(G,u,v)|G$ is a labeled dag, u and v are nodes of G and the terms corresponding to u and v are unifiable}.

UNIFY-SIMPLE $\triangleq$ {$(G,u,v)|G$ is a labeled simple dag, u and v are nodes of G and the terms corresponding to u and v are unifiable}.

Dwork et al. show that $MCV \leqslant_L UNIFY$ and also that $MCV \leqslant_L$ *UNIFY-SIMPLE*, whence both of these problems are complete for **P**.

The authors conclude that in consequence there most probably is no efficient parallel algorithm for unification. Here "having an efficient parallel algorithm" must be identified with membership in "Nick's Class" **NC**. **NC** is the class of all problems solvable on a parallel RAM using $\lambda n[(\log n)^k]$ parallel time for some k, and $\lambda n[n^m]$ processors for some m. It is clear that **NC** $\subseteq$ **P**; it is generally believed, but as yet unproved that the inclusion is strict.

Vitter and Simons [48] also study the possibilities of using parallel processors to perform unification. The parallelization of a number of sequential algorithms is discussed in detail, among which the algorithm of Paterson and Wegman and an algorithm using the algorithm for the *UNION-FIND* problem mentioned before. A limited speed-up of approximately the number of processors used is achieved.

Lewis and Statman [25] have studied the space complexity of the unification problem.

NON-UNIFY $\triangleq$ {$(G,u,v)|G$ is a labeled dag, u and v are nodes of G and the terms corresponding to u and v are not unifiable}.

Lewis and Statman give a space-efficient implementation of a naive, non-linear equivalence handling algorithm described by Paterson and Wegman [31], thus establishing that *NON-UNIFY* is in **NL**.

CDG $\triangleq$ {$G|G$ is a directed cyclic graph}.

The problem *CDG* is complete for **NL**; see [21]. Lewis and Statman then show that $CDG \leqslant_L NON\text{-}UNIFY$ and conclude that unifiability is complete for *co*-**NL**.

Unification has an important special case which does admit efficient parallel algorithms, in particular the matching problem.

Dwork, Kanellakis and Mitchell [8] show that matching is in NC and describe a parallel algorithm requiring $O(\lambda n[(\log n)^2]$ time and $O(\lambda n[M(n^2)])$ processors, where $M(n)$ is the time complexity of matrix multiplication. In [9] an improvement is given in the form of a randomized parallel algorithm requiring $O(M(n))$ processors with the same asymptotic running time on inputs of size n.

Many matching problems are NP-complete, for example the problem *ACM* of associative-commutative matching, see Benanav, Kapur and Narendran [4].

$ACM \triangleq \{(F,V,s,t) \mid$ F is a set of function symbols some of which may be associative and commutative, V is a set of variables, s and t are terms and there is a matching substitution σ such that $\sigma(s) = t\}$.

Similarly defined are the problems *AM* of associative matching, *CM* of commutative matching, *AIM* of associative idempotent and *ACIM* of associative, commutative, idempotent matching. Benanav, Kapur and Narendran [4] show that *ACM*, *AM* and *CM* are all NP-complete problems. Kapur and Narendran [22] show that *AIM* and *ACIM* are NP-hard.

Returning now to the unification problem. Kapur and Narendran [22] consider the complexity of unifying sets of terms in the form of the *Set Unification Problem SUP* defined as follows.

$SUP \triangleq \{(F,V,S,T) \mid F$ is a set of function symbols, V a set of variables, S and T sets of terms and there is a substitution σ such that $\sigma(S) = \sigma(T)\}$.

Kapur and Narendran [22] show that $SUP \in$ NP and that 3-*CNF-SAT* $\leqslant_p SUP$, so that *SUP* is NP-complete.

In closing this section and also this overview we want to mention a result of Mannila and Ukkonen [29]. These authors relate Prolog execution with sequences of *UNIFY-DEUNIFY* instructions and these with sequences of *UNION-FIND* instructions. Consider for example the following Prolog program

```
p(a).
p(b).
q(c).
q(b).
```

and the Prolog goal :- p(X), q(X). Solving this goal creates the following sequence of *UNIFY-DEUNIFY* instructions.

UNIFY(X,a), *UNIFY*(X,c), *UNIFY*(X,b), *DEUNIFY*,
UNIFY(X,b), *UNIFY*(X,c), *UNIFY*(X,b).

The instruction *UNIFY*(s,t) tries to unify the terms s and t and if successful returns the common instance of these terms; the instruction *DEUNIFY* cancels the last successful *UNIFY* instruction which has

not yet been canceled.

Mannila and Ukkonen show that this *UNIFY-DEUNIFY* problem is at least as difficult as the *UNION-FIND* problem and is therefore non-linear on a large class of algorithms.

7. Conclusions

We have given an impression of possibilities, complexities and problems with respect to unification in equational theories. We have restricted ourselves to homogeneous algebras. In most practical applications, variables are typed. The extension of the known results to many sorted algebras, depending on the relation between the sorts, is not trivial. Some results can be found in [7,38,49]. Using many sorted algebras also provides opportunities to reduce the complexity of unification algorithms.

References

1. A.V. Aho, J.E. Hopcroft & J.D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. 1974.
2. S. Arnberg & E. Tiden: Unification problems with one-sided distributivity, *in* J.-P. Jouannaud (Ed.): *Rewriting techniques and Applications*, Lect. Notes Comp. Sci. **202** (1985) 398-406, Springer-Verlag, Berlin, Heidelberg, New York.
3. F. Baader: The theory of idempotent semigroups is of unification type zero, *J. Autom. Reasoning* **2** (1986) 283-286.
4. D. Benanav, D. Kapur & P. Narendran: Complexity of matching problems, *in* J.-P. Jouannaud (Ed.): *Rewriting techniques and Applications*, Lect. Notes Comp. Sci. **202** (1985) 417-429, Springer-Verlag, Berlin, Heidelberg, New York.
5. W. Bibel: *Automated Theorem Proving*, Vieweg, Braunschweig, 1982.
6. R.V. Book & J.H. Siekmann: On unification: equational theories are not bounded, *J. Symbolic Comput.* **2** (1986) 317-324.
7. D. DeGroot & G. Lindstrom: *Logic Programming Functions, Relations and Equations*, Prentice-Hall, Englewood Cliffs, N.J. 1986.
8. C. Dwork, P.C. Kanellakis & J.C. Mitchell: On the sequential nature of unification, *J. Logic Programming* **1** (1984) 35-50.
9. C. Dwork, P.C. Kanellakis & L.J. Stockmeyer: Parallel algorithms for term matching, *in*: J.H. Siekmann (Ed.): *8th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **230** (1986) 416-430, Springer-Verlag, Berlin, Heidelberg, New York.
10. F. Fages & G. Huet: Complete sets of unifiers and matchers in equational theories, *in*: G. Ausiello & M. Protasi (Eds.): *CAAP'83:*

Trees in Algebra and Programming, 8th Colloquium, Lect. Notes Comp. Sci. **159** (1983) 205-220, Springer-Verlag, Berlin, Heidelberg, New York.

11. F. Fages: Associative-commutative unification, *in*: R.E. Shostak (Ed.): *7th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **170** (1984) 194-208, Springer-Verlag, Berlin, Heidelberg, New York.
12. M. Fay: First-order unification in an equational theory, *in*: Proceedings Fourth Workshop on Automated Deduction, Austin Texas, 1979, pp. 161-167.
13. A. Fortenbacher: An algebraic approach to unification under associativity and commutativity, *in*: J.-P. Jouannaud (Ed.): *Rewriting Techniques and Applications*, Lect. Notes Comp. Sci. **202** (1985) 381-397, Springer-Verlag, Berlin, Heidelberg, New York.
14. L.M. Goldschlager: The monotone and planar circuit value problems are log space complete for **P**, *ACM SIGACT News* **9** No. 2 (1977) 25-29.
15. G. Grätzer: *Universal Algebra*, Van Nostrand, Princeton, N.J. 1968.
16. J. Heijenoort: *From Frege to Gödel — A Source Book in Mathematical Logic, 1879-1931*, Harvard University Press, Cambridge, Mass. 1967, pp. 525-581.
17. J. Herbrand: Recherches sur la theorie de la demonstration, *Traveaux de la Soc. des Sciences et des Lettres de Varsovie* **III**, **33** (1930) 33-160. English translation: *Investigations in proof theory* in [16].
18. A. Herold: Combination of unification algorithms, *in*: J.H. Siekmann (Ed.): *8th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **230** (1986) 450-469, Springer-Verlag, Berlin, Heidelberg, New York.
19. C. Hewitt: *Description and theoretical analysis of PLANNER, a language for proving theorems and manipulating models in a robot*, Ph.D. Thesis, MIT, 1972.
20. G. Huet: An algorithm to generate a basis of solutions to homogeneous diophantine equations, *Inform. Process. Lett.* **7** (1978) 144-147.
21. N.D. Jones: Space-bounded reducibility among combinatorial problems, *J. Comput. Systems Sci.* **11** (1972) 68-85.
22. D. Kapur & P. Narendran: NP-completeness of the set unification and matching problems, *in*: J.H. Siekmann (Ed.): *8th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **230** (1986) 489-495, Springer-Verlag, Berlin, Heidelberg, New York.

23. C. Kirchner: A new equational unification method: a generalization of Martelli-Montanari's algorithm, *in*: R.E. Shostak (Ed.): *7th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **170** (1984) 224-247, Springer-Verlag, Berlin, Heidelberg, New York.

24. F. Kluzniak & S. Szpakowicz: *PROLOG for Programmers*, Academic Press, New York, 1985.

25. H.R. Lewis & R. Statman: Unifiability is complete for co-NlogSpace, *Inform. Process. Lett.* **15** (1982) 220-222.

26. M. Livesey & J.H. Siekmann: Unification of A+C-terms (bags) and A+C+I-terms (sets), Int. Bericht *5/76* Inst. für Informatik, University of Karlsruhe, 1976.

27. J.W. Lloyd: *Foundations of Logic Programming*, Springer-Verlag, Berlin, Heidelberg, New York, 1984.

28. J. Maluszynski & H.J. Komorowski: Unification-free execution of logic programs, Symposium on Logic Programming, IEEE Computer Society, 1985.

29 H. Mannila & E. Ukkonen: On the complexity of unification sequences, *in*: E. Shapiro (Ed.): *Third International Conference on Logic Programming*, Lect. Notes Comp. Sci. **225** (1986) 122-133, Springer-Verlag, Berlin, Heidelberg, New York.

30. A. Martelli & U. Montanari: An efficient unification algorithm, *ACM Trans. Program. Lang. Syst.* **4** (1982) 258-282.

31. M.S. Paterson & M.N. Wegman: Linear unification, *J. Comput. Systym Sci.* **16** (1978) 158-167.

32. D.A. Plaisted: The occur-check problem in Prolog, *New Generation Computing* **2** (1984) 309-322.

33. G. Plotkin: Building in equational theories, *in*: D. Michie: *Machine Intelligence* **7** (1972) 73-90, American Elsevier.

34. P. Raulefs & J.H. Siekmann: Unification of idempotent functions, Techn. Report, University of Karlsruhe, 1978.

35. J.A. Robinson: A machine oriented logic based on the resolution principle, *J. Assoc. Comput. Mach.* **12** (1965) 23-41.

36. J.A. Robinson: Computational logic: the unification computation, *in*: B. Mettzer & D. Michie: *Machine Intelligence* **6** (1971), American Elsevier.

37. J.F. Rulifson, J.A. Derksen & R.J. Waldinger: QA4: a procedural calculus for intuitive reasoning, Technical Note 73, Art. Intell. Center, Stanford 1972.

38. M. Schmidt-Schauss: Unification in many-sorted equational theories, *in*: J.H. Siekmann (Ed.): *8th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **230** (1986) 538-

552, Springer-Verlag, Berlin, Heidelberg, New York.

39. M. Schmidt-Schauss: Unification under associativity and idempotence is of type nullary, *J. Autom. Reasoning* **2** (1986) 277-281.
40. J.H. Siekmann: Unification of commutative terms, Int. Bericht Nr. 2/76 Inst. für Informatik, University of Karlsruhe, 1976
41. J.H. Siekmann: Universal unification, *in*: R.E. Shostak (Ed.): *7th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **170** (1984) 1-42, Springer-Verlag, Berlin, Heidelberg, New York.
42. J.R. Slagle: Automated theorem-proving for theories with simplifiers, commutativity and associativity, *J. Assoc. Comput. Mach*, **21** (1974) 622-642.
43. M.E. Stickel: A complete unification algorithm for associative-commutative functions, *J. Assoc. Comput. Mach.* **28** (1981) 423-434.
44. M.E. Stickel: An introduction to automated deduction, *in*: W. Bibel & Ph. Jorrand (Eds.): *Fundamentals of Artificial Intelligence*, Lect. Notes Comp. Sci. **232** (1986) 75-132, Springer-Verlag, Berlin, Heidelberg, New York.
45. M.E. Stickel: *Mechanical Theorem Proving and Artificial Intelligence Languages* Ph.D. Thesis, Carnegie-Mellon University, 1977.
46. P. Szabo: *Theory of first order unification*, Thesis (in German), University of Karlsruhe, 1982.
47. E. Tiden: Unification in combinations of collapse-free theories with disjoint sets of function symbols, *in*: J.H. Siekmann (Ed.): *8th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **230** (1986) 431-449, Springer-Verlag, Berlin, Heidelberg, New York.
48. J.S. Vitter & R.A. Simons: New classes for parallel complexity: a study of unification and other complete problems for P, *IEEE Trans. Comput.* **C-35** (1986) 403-417.
49. C. Walther: A classification of many sorted unification problems, *in*: J.H. Siekmann (Ed.): *8th International Conference on Automated Deduction*, Lect. Notes Comp. Sci. **230** (1986) 525-537, Springer-Verlag, Berlin, Heidelberg, New York.
50. K. Yelick: Combining unification algorithms for confined regular equational theories, *in*: J.-P. Jouannaud (Ed.): *Rewriting Techniques and Applications*, Lect. Notes Comp. Sci. **202** (1985) 365-380, Springer-Verlag, Berlin, Heidelberg, New York.

The Relation Between Two Patterns with Comparable Languages

Gilberto Filé

C.N.R.S., U.E.R. de Mathématiques et Informatique
Université de Bordeaux I, 351 Cours de la Liberation
33405 Talence Cedex, France

A pattern is a string consisting of terminals and variables. The language defined by a pattern is the set of terminal strings obtained by substituting (uniformly) terminal strings to its variables. A pattern simulates another pattern when its language includes that of the other one.

If q simulates p, one may intuitively think that there must be a substitution that, applied to q, produces p. This hypothesis is considered under different assumptions. The main result says that it is true only for very restricted patterns (with variables only) and only when erasing substitutions are considered. The relation between two patterns is studied also in the case that the languages they produce are equal.

1. Introduction

A pattern is a word consisting of terminal symbols and of variables. The language defined by a pattern is the set of strings obtained by substituting consistently terminal strings to all its variables. Patterns were introduced in [1], see also [3], in the context of inductive inference. We consider patterns independently of this application. In the study of patterns it is natural to consider the following problem *PD*: for any two patterns p and q decide whether the language of p includes that of q. Angulin [1] has left the decidability of *PD* as an open question. How would one attack such a question? Intuitively, the following *hypothesis H* seems reasonable and, if verified, would immediately give a decision method for *PD*:

H: If the language of q includes the one of p, then there must be a substitution ϕ such that $\phi(q) = p$.

Unfortunately, in [1] it is shown that *H* is false in the case that one considers only nonerasing substitutions in the definition of language of a pattern. In this paper we study whether *H* holds at least in some restricted case. Namely, the following cases are considered:

(i) also erasing substitutions are allowed,

(ii) only pure patterns are considered, i.e., patterns that contain only variables.

This gives us the four cases shown in Figure 1. Correspondingly, one has the four problems PD_1–PD_4 and the four hypothesis H_1–H_4. Only H_4 is known to be false; we study the remaining three cases.

	erasing	nonerasing
pure patterns	H_1	H_2
any pattern	H_3	H_4

Figure 1.

The first result that we obtain is, that H_1 is true. After this we want to verify whether the conditions of pure patterns and erasing substitutions are both necessary. This is indeed the case. Relatively simple counterexamples suffice to show that both H_2 and H_3 are false.

Therefore we only have a decision method for the inclusion of pattern languages in one of the four cases. Clearly, this does not imply that the other problems are undecidable. However, they are difficult problems: PD_4 is shown to be **NP**-hard in [1] and it is easy to modify this proof to show that the same is true for PD_3.

The paper is organized as follows. First, the necessary definitions are given in Section 2. In Section 3 we show that H_1 is true and in Section 4 that H_2 and H_3 are false. In Section 5 the relation between two patterns defining equal languages is studied. The paper is closed by a short conclusion in which some open problems are pointed out (Section 6).

2. Preliminaries

For any set S, $|S|$ is the number of elements of S and for any string s, $|s|$ is its length. A is a finite set of terminal symbols; $A = \{a,b,c,...\}$. $V = \{x_1,x_2,x_3,...\}$ is a set of variables. A *pattern* p is a word in $(A \cup V)^*$. $Var(p) = \{x \mid x$ is in V and appears in $p\}$; $Term(p) = \{a \mid a \in A$ and a appears in $p\}$. A pattern is *pure* if it contains only variables. A substitution σ is a function $\sigma: V \to (A \cup V)^*$. A substitution σ is *nonerasing* if, for every x in V, $\sigma(x) \neq \lambda$. A substitution is said to be a *variable renaming* if it defines a bijection from V to V. The *language generated by a pattern* p is the set $L(p) = \{w \mid w \in A^*$ and $w = \sigma(p)$ for some substitution $\sigma\}$. The set of all terminal strings that can be generated from p by means of nonerasing substitutions only is denoted by $LN(p)$.

For a pattern p, the i-th position of p, $1 \leqslant i \leqslant |p|$, is denoted by $<p,i>$. If the symbol occurring in $<p,i>$ is x then $<p,i>$ is an *occurrence* of x in p. When the pattern under consideration is clear from the context, a position $<p,i>$ is denoted with i only. For $x \in Var(p)$, the sequence of occurrences of x in p is denoted by $Occ(p,x)$ and is the sequence $<i_1,...,i_h>$ such that $1 \leqslant i_1 < i_2 < ... < i_h \leqslant |p|$ and such that $i_1,...,i_h$ are all and the only occurrences of x

in p.

As already explained in the Introduction, see also Figure 1, we want to show the truth or the falsity of the following four hypothesis H_1 to H_4:

Given any two pure patterns p and q,

H_1: $L(p) \subseteq L(q) \Longrightarrow$ there is a substitution σ such that $\sigma(q) = p$.

H_2: $LN(p) \subseteq LN(q) \Longrightarrow$ there is a nonerasing substitution σ such that $\sigma(q) = p$.

The hypothesis H_3 and H_4 are obtained from H_1 and H_2, respectively, by dropping the hypothesis that p and q are pure.

The falsity of H_4 has been shown in [1] by means of the following counterexample. Let $A = \{0,1\}$,

$$p = 0x\,10xx\,1 \quad \text{and} \quad q = xxy.$$

Similar counterexamples can be found for any finite A; see [1].

It is important to remark the role of the size of the terminal alphabet A for the problems under consideration. On the one hand, if $|A| = 1$, then it is easy to show that H_1 to H_4 are all false. For instance, the following counterexample suffices for showing that H_1 and H_2 are false:

$$p = xyyx \quad \text{and} \quad q = xx.$$

On the other hand, if $|A| \geqslant |Var(p)| + |Term(p)|$ then H_1 to H_4 are trivially verified: substitute each variable of p with a distinct symbol of A that is not in $Term(p)$, let w be the word obtained, since $L(p) \subseteq L(q)$ there is a substitution σ such that $\sigma(q) = w$; this σ trivially gives a substitution σ' such that $\sigma'(q) = p$. Thus, when considering two patterns p and q we will always assume that $2 \leqslant |A| < |Var(p)|$.

3. The First Hypothesis Is True

The goal of this section is to show the following theorem.

Theorem 1. *For an alphabet A containing at least two symbols, H_1 is true.* □

The proof of the theorem is quite long and it is split in several lemmas. Throughout the rest of the section the following notation is used.

Notation. p and q are patterns such that $L(p) \subseteq L(q)$; $k = |Var(p)|$, $k' = |Var(q)|$, $n = |p|$, $m = |q|$. □

The idea of the proof of Theorem 1 is that of defining a substitution π that associates to each variable of p a word that has "nothing" in common with the words of the other variables. Through π we obtain an effect similar to that of having an alphabet A such that $|A| \geqslant |Var(p)|$; see the observations at the end of Section 1.

Substitution π. The notation introduced above is used. Fix an arbitrary total order among the variables of p, i.e., fix a bijection $ord: Var(p) \to [1,k]$. For each $x \in Var(p)$, π is as follows: let $A = \{a,b\}$ and $ord(x) = i$,

$\pi(x) = as_1as_2...as_La$, where $L = 6mk$ and $s_j = b^{(i-1)L+j}$, $j \in [1,L]$.

A subword $ab^t a$ of $\pi(x)$ is called a *module* of $\pi(x)$.

In what follows π' is a substitution such that $\pi'(q) = \pi(p)$. Such a substitution exists because $L(p) \subseteq L(q)$. □

The reason for making π depend on L (and thus on q) is technical and it will become clear in Lemmas 1 and 3 below. The following is an important property of π.

Property (*). For any $x \in Var(p)$ consider a decomposition $\pi(x) = \alpha w \beta$, where w contains at least a module of $\pi(x)$. There is no other decomposition $\alpha' w \beta'$ of $\pi(x)$ where $\alpha \neq \alpha'$. □

If $i \in Occ(p,x)$ then with $\pi(<p,i>)$ we will denote $\pi(x)$. Similarly, for π'. This notation is extended to sequences of positions as follows: $\pi(p,i,i+1,...,i+h)$ denotes $\pi(<p,i>)...\pi(<p,i+k>)$.

We define the following two relations:

(1) A position $<p,i>$ is *simulated* by the positions $<q,j>,...,<q,j+h>$ if,

(a) $\pi'(q,1,...,j-1)$ is a prefix of $\pi(p,1,...,i-1)$ and,

(b) $\pi'(q,1,...,j+h)$ contains $\pi(p,1,...,i)$ as a prefix.

With $Issim(i)$ we denote the sequence $<j,...,j+h>$.

(2) A position $<q,j>$ *simulates* the positions $<p,i>,...,<p,i+h>$ when $Issim(i-1)$ (if it exists) does not contain j, $Issim(i),...,Issim(i+h)$ all contain j, and $Issim(i+h+1)$ does not.

The sequence $<i,...,i+h>$ is denoted by $Sim(j)$.

It is useful to be able to be more precise about what simulates what: we want to specify also what part of a string is simulated.

Consider two positions $<p,i>$ and $<q,j>$ such that the first is simulated by the second one. It is easy to understand that in this case $\pi(<p,i>)$ and $\pi'(<q,j>)$ must have a common substring w. Figure 2 shows one possible situation of the simulation of $<p,i>$ by $<q,j>$. Obviously, there are other cases, but for each the above statement remains true. Assume that the substring w produced by both $<p,i>$ and $<q,j>$ starts and ends in the positions h_1 and h_2 of $\pi(<p,i>)$, i.e., $\pi(<p,i>) = \alpha w \beta$, where $|\alpha| = h_1 - 1$ and $|w| = h_2 - h_1 + 1$. In this case we say that $<q,j>$ *simulates* $<p,i>$ *from* h_1 *to* h_2. In case the string w contains at least one module of $\pi(<p,i>)$, one says that $<q,j>$ is *principal* for $<p,i>$. If this is the case, $j \in Occ(q,y)$ and $i \in Occ(p,x)$ then y is said to be *principal* for x.

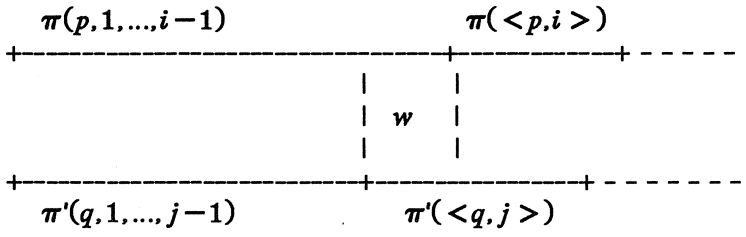

Figure 2.

In what follows we will prove three lemmas that will enable us to prove Theorem 1. Before doing this let us describe intuitively the line of thought that is followed. What we want to do is the following:

First, in Lemma 1, it is shown that each position $<p,i>$ is simulated by at least one position $<q,j>$ that is principal for it, (clearly, $j \in Issim(i)$).

In Lemma 2 it is shown that if an occurrence $<q,j>$ of y is principal for an occurrence $<p,i>$ of x, then every other occurrence of y must be principal for some other occurrence of x.

Finally, in Lemma 3 we show that each position $i \in [1,n]$ of p can "choose" a position $<q,j>$ of $Issim(i)$ that is principal for $<p,i>$, such that the following holds: let $<p,i>$ and $<q,j>$ be occurrences of x and y, respectively; since $<p,i>$ has chosen $<q,j>$, every other occurrence $<p,i'>$ of x such that an occurrence $<q,j'>$ of y is in $Issim(i')$, chooses $<q,j'>$.

Once this is shown, it is easy to construct a substitution ϕ such that $\phi(q) = p$ (thus showing the theorem):

(i) for all variables y of q that are never chosen in the above process, $\phi(y) = \lambda$,

(ii) for every other variable z of q, consider an occurrence $<q,j>$ of z and let $<i_h,...,i_{h+l}>$ be the elements of $Sim(j)$ that have chosen $<q,j>$; if $x_0,...,x_l$ are the variables occurring in $i_h,...,i_{h+l}$, respectively, then $\phi(z) = x_0...x_l$.

Lemma 1. *For every $i \in [1,n]$, $Issim(i)$ contains at least one element j such that $<q,j>$ is principal for $<p,i>$.*

Proof: A variable $y \in Var(q)$ that is principal for no variable of p, is such that $|\pi'(y)| \leqslant 2(kL+1)$. In fact, $\pi'(y)$ can have the forms, $b^t aab^{t'}$ or $b^t ab^{t'}$, where t and t' are at most kL; see the definition of π. Now, since $|q| = m$, $Issim(i)$ is at most $<1,...,m>$; in this case, the string that q can generate for simulating $\pi(<p,i>)$ has length at most $2m(kL+1)$. This cannot be sufficient because the length of $\pi(<p,i>)$ is as follows: let $r = ord(x)$, where x is the variable in $<p,i>$, then

$$|\pi(x)| = (L+1) + \sum_{j=1}^{L} \left|(r-1)L+j\right| = (L+1) + (r-1)L^2 + \frac{L(L+1)}{2}.$$

Because $|\pi(x)|$ depends on the square of L, it is easy to prove that $|\pi(x)|>2m(kL+1)$. Recall that $L = 6mk$; it suffices to consider the second term only (the first may be equal to zero if $r = 1$): we want to show that,

$$\frac{L(L+1)}{2} > 2m(kL + 1).$$

This is true if $L^2>4m(kL+1)$, now $L^2>5mkL>4m(kL+1)$. Thus the lemma is true. □

Lemma 2. *Consider two positions $<p,i>$ and $<q,j>$ such that the second simulates the first from h_1 to h_2 and it is principal for it. Let $j \in Occ(q,y) = <j_1,..,j_h>$ and $i \in Occ(p,x)$; for every $f \in [1,h]$, there is an element i' of $Occ(p,x)$ such that $<q,j_f>$ simulates it from h_1 to h_2 and is principal for it.*

Proof: By definition of π, only occurrences of x produce in π a module of $\pi(x)$. Hence, if y is principal for x, every occurrence of y in q must participate to the simulation of an occurrence of x. This, together with Property (*) shows the lemma. □

The following concept is very important for the sequel of the proof.

Definition. Let $x \in Var(p)$ and $Occ(p,x) = <i_1,..,i_h>$. A *choice* for x is a sequence $C_x = <j_1,..,j_h>$ of positions of q such that the following two conditions are satisfied:

(1) $j_r \in Issim(i_r)$ and $<q,j_r>$ is principal for $<p,i_r>$.

(2) Let y be the variable in position $<q,j_r>$ and assume that $<q,j_r>$ simulates $<p,i_r>$ from h_1 to h_2; for any other i_z, $z \in [1,h]$, such that $<p,i_z>$ is simulated from h_1 to h_2 by an occurrence $<p,j>$ of y, it must be that j_z is equal to j. □

The second point of the above definition may appear mysterious. Its goal is explained intuitively as follows. From a choice for each variable of p we intend to construct the substitution ϕ such that $\phi(q) = p$. To this end we need that once a simulation task (e.g., simulate $<p,i_r>$) is given to one occurrence of y, (e.g., $<q,j_r>$) that same task must be assigned to every other occurrence of y (Thus, $<q,j_z>$ must simulate $<p,i_z>$). Intuitively, one can require this condition because of Lemma 2; the formal proof is given in the following lemma.

Lemma 3. *For each variable $x \in Var(p)$ there is a choice for x.*

Proof: Let $Occ(p,x) = <i_1,..,i_h>$ and $H = |\pi(x)|$. For each $f \in [1,H]$, $Cut(f)$ is the sequence $<j_1,..,j_h>$ such that for every $r \in [1,h]$, $<q,j_r>$ simulates $<p,i_r>$ from h_1 to h_2 and $h_1 \leqslant f \leqslant h_2$. For proving the lemma it suffices to show that there is at least one f such that for each $r \in [1,h]$, $<q,j_r>$ is principal for $<p,i_r>$. That such a $Cut(f)$ is a choice for x is shown as follows.

Cut (f) satisfies trivially condition (1) of the definition of choice. It satisfies also condition (2) because otherwise the following would be true: *Cut* (f) contains two elements i_{l1} and i_{l2} of $Occ(p,x)$ such that,

(i) $<q,j_{l1}>$ simulates $<p,i_{l1}>$ from h_1 to h_2; let $j_{l1} \in Occ(q,y)$,

(ii) $<q,j_{l2}>$ simulates $<p,i_{l2}>$ from h_1' to h_2' and j_{l2} is not in $Occ(q,y)$,

(iii) there is an occurrence $<q,j>$ of y that simulates $<p,i_{l2}>$ from h_1 to h_2.

It is easy to see that this cannot be true because f is both in $[h_1,h_2]$ and in $[h_1',h_2']$ and hence, if (ii) and (iii) would be true at the same time, the f-th symbol of $<p,i_{l2}>$ would be "simulated twice".

An f, such that *Cut* (f) has the property specified above, exists because, otherwise, the non principal variables of q should generate more than H symbols and in the proof of Lemma 1 we have shown that this is not possible. □

We are finally in the condition of proving Theorem 1. For this proof we need the following notation. Consider a variable $x \in Var(p)$, let $Occ(p,x) = <i_1,...,i_h>$ and let $C_x = <j_1,...,j_h>$ be a choice for it; let $<p,i>$ be an occurrence of x, i.e., $i = i_r$ for some $r \in [1,h]$, then with $C_x(<p,i>)$ we denote the element j_r of C_x. Intuitively, $C_x(<p,i>)$ is the position in q that has been chosen for simulating $<p,i>$.

Proof of Theorem 1: Let for $x \in Var(p)$, C_x be a choice for x. The definition of the substitution ϕ such that $\phi(q) = p$ is as follows:

Definition of ϕ. For each $y \in Var(q)$ one needs first to fix the notation (a):

(a) Consider any occurrence $<q,j>$ of y and let $S = <<p,i>,...,<p,i+h>>$ be all the positions that have chosen $<q,j>$; formally, S is the maximal sequence of positions of p such that, for each $r \in [i,i+h]$, if x is the variable occurring in $<p,r>$, then $C_x(<p,r>) = j$.

Now, if S is empty then $\phi(y) = \lambda$, otherwise, if $x_0,...,x_h$ are the variables of p occurring in the positions $i,...,i+h$ of p, then $\phi(y) = x_0,...,x_h$. □

Notice that S consists of contiguous positions: this is the case because if $<q,j>$ is chosen by $<p,r>$ and $<p,r+2>$ then it is the only principal position of $<p,r+1>$ and hence, it must be chosen by $<p,r+1>$.

Since for defining $\phi(y)$ just any occurrence of y is taken, the reader may wonder whether the above definition characterizes a unique substitution. This is the case because of the following reason (*):

(*) If an occurrence $<q,j>$ of y simulates an occurrence $<p,i>$ of x from h_1 to h_2, then, by Lemma 2, every other occurrence $<q,j'>$ of y must simulate from h_1 to h_2 some other occurrence

$<p,i'>$ of x. By point (2) of the definition of choice, if $C_x(<p,i>) = j$, then $C_x(<p,i'>) = j'$. Hence, considering j or j' for defining $\phi(y)$ is precisely the same.

It remains to show that $\phi(q) = p$. To this end remark that p can be cut into h pieces, $h \geqslant 1$,

$$<1,...,i(1)>, <i(1)+1,...,i(2)>, ..., <i(h-1)+1,...,i(h)>$$

such that every position in each piece has chosen the same position of q (Each piece is like the sequence S in the definition of ϕ above). Let j_r be the position of q that is chosen by the r-th piece, $r \in [1,h]$. For obtaining the desired result, it suffices to observe that the definition of ϕ and reason (*) imply the following two points:

(1) The positions $<j_1,...,j_h>$ are all and the only positions in q of the variables y such that $\phi(y) \neq \lambda$,

(2) If y is the variable in $<q,j_r>$, $r \in [1,h]$, $\phi(y)$ is equal to the sequence of variables corresponding to the positions in the r-th piece of p, i.e., $<i(r-1)+1,...,i(r)>$; we assume that $i(0) = 0$. □

This result gives an exponential test for the inclusion of the languages of two pure patterns under erasing substitutions.

4. Hypotheses 2 and 3 Are False

These negative results are easier to present than the first one because it suffices to give a counterexample for each of them.

Counterexample for H_2. (Pure patterns and nonerasing substitutions) Let the terminal alphabet be $A = \{a,b\}$.

$$p = xyzwkmr \quad \text{and} \quad q = xyzyw.$$

$L(q)$ contains all words of length at least five and that can be decomposed into, $w_1w_2w_3w_2w_4$, such that all w_i are non empty. For showing that $L(q) \supsetneq L(p)$ observe that, if L_5 is the set of all words of length at least five on A, any $w \in L_5$ can be decomposed in, $w_1w_2w_3w_2w_4$, where w_1 and w_4 may be empty. Since $|p| = 7$, every word $w \in L(p)$ can be decomposed into $w_1w_2w_3$, where $w_2 \in L_5$ and w_1 and w_3 are not empty. Hence, $w \in L(q)$. It is evident that no nonerasing substitution ϕ exists such that $\phi(q) = p$.

It is not difficult to generalize this example to a larger alphabet A. □

Counterexample for H_3. (Any pattern and erasing substitutions) The case that $A = \{a,b\}$ is very simple:

$$p = xaybz \quad \text{and} \quad q = xaby.$$

Clearly, there is no erasing substitution ϕ such that $\phi(q) = p$, but $L(p) = L(q)$: they both contain all words in A^* containing ab. This example is due to Codognet [2].

We are not able to generalize this example to larger alphabets. A quite different counterexample is needed if $A = \{a,b,c\}$. For simplicity

we write p and q using the extra symbol $\not\in$ to denote the string abc,

$p = \not\in aa\ \not\in ba\ \not\in ca\ \not\in\not\in ab\ \not\in bb\ \not\in cb\ \not\in\not\in yayb$

$q = x\not\in y\not\in z\not\in w\not\in r\not\in kykw$

Let us show that $L(p) \subseteq L(q)$. Intuitively, the idea is that k of q cannot produce both ya and yb and hence, it must be "helped" by y and w, but y and w cannot be just a and b; they are at least strings of length 2. The last character of the string generated by y (under any substitution) can be a, b, or c, thus y must have the possibility of becoming, according to the need, aa, ba and ca, whereas w must be able to become ab, bb, and cb. This can be done by varying accordingly the values of the variables x, z, and r. More formally, consider any substitution σ and let the last character of $\sigma(y)$ be, for instance, c. Then one can define a substitution σ' such that $\sigma'(q) = \sigma(p)$ as follows:

$\sigma'(x) = \not\in aa\ \not\in ba$

$\sigma'(y) = ca$

$\sigma'(z) = \not\in ab\ \not\in bb$

$\sigma'(w) = cb$

$\sigma'(r) = \lambda$

$\sigma'(k) = \sigma(y)$ of which the last letter has been deleted.

Assume now that there is a substitution ϕ such that $\phi(q) = p$. Any such ϕ must satisfy the condition that $\phi(kykw) = yayb$ and hence, $\phi(k) = y$, $\phi(y) = a$, and $\phi(w) = b$. But, considering q, one sees that this is possible only if the first part of p contains $\not\in a\not\in$ and $\not\in b\not\in$. If $\not\in = abc$ this is not possible. □

Observe that these negative results do not imply the undecidability of the inclusion of pattern languages in the conditions of H_2 and H_3. However, they seem to imply that any method for deciding these problems will not be simple. In [1] it is proved that whether $L(p) \subseteq L(q)$ for any patterns p and q under nonerasing substitutions is NP-hard. It is simple to modify this proof for showing the NP-hardness of the problem also in the case that erasing substitutions are considered. This proof is not included here because it is a straightforward modification of the one of [1].

5. On Pattern Equivalence

Based on the results of the previous sections, one may say that, in general, the condition $L(p) \subseteq L(q)$ is not sufficient for showing a strict relationship between p and q. It is natural to wonder whether the condition that $L(p) = L(q)$ would then be strong enough.

Angulin in [1] shows the following result (a).

(a) For any two patterns p and q, $LN(p) = LN(q)$ if and only if p and q are equal modulo a variable renaming.

The proof of this result uses the (obvious) fact that if $LN(p) = LN(q)$ then $|p| = |q|$. Therefore, this proof breaks down if erasing substitutions are considered. In that case the following results can be shown:

(b) If p and q are pure patterns, then $L(p) = L(q)$ if and only if there are two substitutions σ and γ such that $\sigma(p) = q$ and $\gamma(q) = p$.

(c) For any two patterns p and q and for an alphabet A containing at least three symbols, if $L(p) = L(q)$, then p and q must be as follows:

$$p = w_1\alpha_1 w_2 \ldots w_k \alpha_k w_{k+1}$$

$$q = w_1\beta_1 w_2 \ldots w_k \beta_k w_{k+1}$$

where, for each $i \in [1,k+1]$, w_i is in A^* and for each $i \in [1,k]$, α_i and β_i are in $Var(p)^+$ and in $Var(q)^+$, respectively. When two patterns respect the above condition they are said to have the same structure.

Point (b) is an immediate consequence of the fact that H_1 is true. Point (c) is somehow a weaker version of (a). Point (c) can be proved, roughly, as follows (This proof was suggested by [2]). First, remark that the hypothesis that A contains more than two symbols is necessary: the first counterexample for H_3, where $A = \{a,b\}$, contradicts (c). Consider two patterns p and q such that $L(p) = L(q)$. It is easy to see that if $t(p)$ and $t(q)$ denote the terminal strings obtained from p and q by deleting the variables, then $t(p) = t(q)$. Assume that p and q contradict (c). This means that the following situation (or the symmetric one) takes place:

$$p = \Omega w \alpha_1 w' \Omega' \quad \text{and} \quad q = \Pi w w' \Pi'.$$

Assume that this is the left-most such situation. Let a be a symbol in A that is different from the last symbol of w and from the first one of w'. Let σ be the substitution sending every variable of p to a. There is no σ' such that $\sigma'(q) = \sigma(p)$. Assume, in fact, that such σ' exists. Since σ sends all variables of p to a, by the fact that $t(p) = t(q)$, σ' must do the same. From the assumption that Ω and Π respect point (c), it follows that Ωw and Πw contain the same number (at least one) of symbols different from a; σ and σ' must be such that these symbols occur in corresponding places of $\sigma(p)$ and $\sigma'(q)$. Thus, in particular, $|\sigma(\Omega w)| = |\sigma'(\Pi w)| = k$. Observe now that the $k+1$-th symbol of $\sigma(p)$ is a, whereas the $k+1$-th symbol of $\sigma'(q)$ is the first symbol of w' that is different from a by construction. Hence, $\sigma'(q) \neq \sigma(p)$. □

6. Conclusions and Open Problems

We have studied the problem whether, for two patterns p and q, the fact that the language of q includes the one of p implies the existence of a substitution σ such that $\sigma(q) = p$. This is true only in the case that p and q are pure patterns and that erasing substitutions are considered. Thus, only in this case we have an exponential method for deciding the inclusion of pattern languages.

The stronger hypothesis that $LN(p) = LN(q)$ implies the equality (modulo renaming) of p and q, whereas, under the hypothesis that $L(p) = L(q)$, we are able to prove only the equality of the structures of p and q, see point (c) in the previous section.

Several problems must still be answered:

(1) Can a stronger result than that of point (c) of the previous section be shown for any two patterns p and q such that $L(p) = L(q)$?

(2) Are there methods for deciding the inclusion of two pattern languages when the two patterns are not pure or if one considers nonerasing substitutions?

(3) In the case of erasing substitutions, is it possible to define a set of rules for transforming a given pattern into one of minimal length still defining the same language?

Acknowledgements. I would like to thank Christian Codognet and Michel Billaud of the University of Bordeaux for stimulating conversations and helpful discussions.

References

1. D. Angulin: Finding patterns common to a set of strings. Proc. 11-th ACM Symp. on Theory of Comp. Atlanta, Ga. (1979) 130-141.
2. C. Codognet: Personal communication (1986).
3. T. Shinohara: Polynomial time inference of pattern languages and its application. Proc. 7-th IBM Symp. on Math. Found. of Comp. Sci. (1982).

Attributed Abstract Program Trees

Henk Alblas and Frans J. Faase

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Traditionally, an attribute grammar is presented as a context-free grammar which is augmented with attributes and attribute evaluation rules. This makes attribute grammars a suitable means for the specification of the semantics of programming languages in the context of derivation trees. For the specification of semantic integrity constraints in the context of abstract program trees the concept of attribute grammars has to be re-defined. For this purpose, a language for the specification of context-free tree grammars is defined. This language is extended to an attribute tree grammar specification language.

1. Introduction

In the classical theory [7] attribute grammars form an extension of the context-free grammar framework in the sense that information is associated with programming language constructs by attaching attributes to the grammar symbols representing these constructs. Each attribute has a set of possible values. Attribute values are defined by attribute evaluation rules associated with the productions of the context-free grammar.

The attributes associated with a grammar symbol are divided into two disjoint classes, the synthesized attributes and the inherited attributes. The attribute evaluation rules associated with a production define the synthesized attributes attached to the grammar symbol on the left-hand side and the inherited attributes attached to the grammar symbols on the right-hand side of the production.

A non-ambiguous context-free grammar assigns a single derivation tree to each sentence. The values of the synthesized attributes at a node of a derivation tree and the inherited attributes at its immediate descendents are defined by the attribute evaluation rules associated with the production applied at that node. The value of a synthesized attribute of the parent is computed from the values of the attributes at its children and (possibly) other attributes of the parent itself. The value of an inherited attribute of a child is computed from the values of attributes at its parent and its siblings and (possibly) other attributes of the child itself.

Generally speaking, a synthesized attribute attached to a tree node contains information concerning the subtree at that node. This

attribute therefore contains information from the terminal string derived from the nonterminal symbol labeling that node. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which the construct appears.

The traditional way of thinking about attribute grammars is in terms of derivation trees. First, the parser generates a derivation tree for a given program. Next, the attribute evaluator computes the values of the attribute instances attached to the nodes of the derivation tree by executing the attribute evaluation rules associated with these attribute instances.

In this research we consider attribute grammars for abstract program trees in which nonterminal symbols are no longer used and where operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the derivation tree. Another reduction found in abstract program trees is the elimination of chain productions. This allows an abstract program tree to be viewed as a compact and simplified representation of a derivation tree, where each operator is represented by an interior node whose children represent the arguments of that operator and where all redundant information needed for the syntactical analysis has been deleted. In other words, an abstract program tree is a non-redundant representation of the hierarchical structure of a source program. In this paper we want to describe abstract program trees as a separate concept, not as an adaptation of context-free grammars. We also do not discuss the conversion of derivation trees of a context-free grammar to abstract program trees, nor do we discuss how context-free grammars can be transformed into abstract program trees as described in [10].

Essentially the compilation process consists of an analysis phase and a synthesis phase. The result of the analysis phase (i.e., lexical and syntactic analysis) is a tree (possibly in a linearized form), which expresses the structure of the source program. Attributes can be attached to the nodes of the tree to carry semantic information. Semantic analysis (e.g., type checking) can be expressed by attribute evaluation rules and semantic conditions. The aim of the synthesis phase is the inclusion of necessary constraint checks (e.g., array bound checks) and the translation of the control structures and the data structures of the source program into the instructions and the storage locations of the target machine. Our ultimate goal is to specify these translations by a stepwise application of tree transformations, starting from the structures of the source program and ending with the structures of the target machine.

Tree transformations also form a suitable means for the specification of compiler optimizations. These transformations replace complicated and non-efficient tree structures by equivalent but simpler and more efficient tree structures.

For the specification of tree transformations, both for the purpose of translations and for optimizations, the classical attribute grammar framework has to be extended with conditional tree transformation rules [2,6,11]. The predicates on attribute values (carrying context information) may be used to enable the application of these transformations.

So, we are interested in non-redundant program trees, which can be used as a concept to define the information flow of the associated program and which can also be considered as an object to be operated on. In this paper we restrict ourselves to the description of the structure and the attribution of abstract program trees. We will deal with tree transformations in a future paper.

This paper is organized as follows. Section 2 provides an introduction to the basic concepts of the classical attribute grammar framework, based on context-free grammars. A language for the specification of abstract program trees is given in Section 3. In Section 4 this language is extended to an attribute grammar specification language. Concluding remarks are made in Section 5.

2. Classical Attribute Grammars

In the classical theory [7] an attribute grammar AG is based on a context-free grammar G which is augmented with attributes and attribute evaluation rules.

The underlying grammar G is a 4-tuple (V_N, V_T, P, S). The finite sets V_N of nonterminal and V_T of terminal symbols form the vocabulary $V = V_N \cup V_T$. P is the set of productions and $S \in V_N$ is the start symbol, which does not appear in the right part of any production. The grammar G is reduced in the sense that each nonterminal symbol is reachable from the start symbol and can generate a string which contains no nonterminal symbols.

Each symbol $X \in V$ has a finite set $A(X)$ of attributes, partitioned into two disjoint subsets $I(X)$ and $S(X)$ of inherited and synthesized attributes, respectively.

Let P consist of r productions, numbered from 1 to r and let the p-th production be

$$X_{p0} \rightarrow X_{p1} X_{p2} \cdots X_{pn_p}$$

where $n_p \geqslant 0$, $X_{p0} \in V_N$ and $X_{pk} \in V$ for $1 \leqslant k \leqslant n_p$.

Production p is said to have the attribute occurrence (a,p,k) if $a \in A(X_{pk})$. The set of attribute occurrences of production p will be denoted by $AO(p)$. This set can be partitioned into two disjoint sets of defined occurrences and used occurrences denoted by $DO(p)$ and $UO(p)$, respectively.

These subsets are defined as

$$DO(p) = \{(s,p,0) \mid s \in S(X_{p0})\} \cup \{(i,p,k) \mid i \in I(X_{pk}) \wedge 1 \leqslant k \leqslant n_p\}$$

$$UO(p) = AO(p) - DO(p) = \{(i,p,0) \mid i \in I(X_{p0})\} \cup \{(s,p,k) \mid s \in S(X_{pk}) \wedge 1 \leqslant k \leqslant n_p\}$$

Associated with each production p is a set of attribute evaluation rules which specify how to compute the values of the attribute occurrences in $DO(p)$. The evaluation rule defining attribute occurrence (a,p,k) has the form

$$(a,p,k) := f((a_1,p,k_1),(a_2,p,k_2), \cdots ,(a_m,p,k_m))$$

where $(a,p,k) \in DO(p)$, f is a total function and $(a_j,p,k_j) \in AO(p)$ for $1 \leqslant j \leqslant m$.

An attribute grammar is said to be in normal form if the extra condition $(a_j,p,k_j) \in UO(p)$ holds for $1 \leqslant j \leqslant m$. It is easy to transform every attribute evaluation rule (by a sequence of transformations) such that only attribute occurrences in $UO(p)$ appear as arguments of f.

For each sentence of G a derivation tree exists. The nodes of the tree are labeled with symbols from V. At each inner node a production $p: X_{p0} \rightarrow X_{p1} X_{p2} \cdots X_{pn_p}$ is applied, such that the node is labeled with X_{p0} and its sons with $X_{p1}, X_{p2}, \cdots , X_{pn_p}$.

Given a derivation tree, instances of attributes are attached to the nodes in the following way: if node N is labeled with grammar symbol X, then for each attribute $a \in A(X)$ an instance of a is attached to node N. An attribute instance a of node N will be denoted by a of N.

Let N_0 be a node, p the production at N_0 and $N_1, N_2, \cdots , N_{n_p}$ its sons from left to right, respectively. An attribute evaluation instruction

$$a \text{ of } N_k := f(a_1 \text{ of } N_{k_1}, a_2 \text{ of } N_{k_2}, \cdots , a_m \text{ of } N_{k_m})$$

is associated with attribute instance a of N_k if the attribute evaluation rule

$$(a,p,k) := f((a_1,p,k_1),(a_2,p,k_2), \cdots ,(a_m,p,k_m))$$

is associated with production p.

The task of an attribute evaluator is to compute the values of all attribute instances attached to the derivation tree by executing their associated evaluation instructions. In general the order of evaluation is unimportant, with the only restriction that an attribute evaluation instruction cannot be executed before the values of its arguments are

available. Initially the values of all attribute instances attached to the derivation tree are undefined, with the exception of the inherited attribute instances attached to the root (containing information concerning the environment of the program) and the synthesized attribute instances attached to the leaves (determined by the parser).

At each step we choose an attribute instance whose value can be computed. The evaluation process continues until all attribute instances in the derivation tree are defined or until none of the remaining attribute instances can be evaluated.

An attribute grammar is circular if a derivation tree exists for which it is not possible to evaluate all attribute instances.

Several methods have been developed to evaluate the semantic attributes within the derivation tree of a program. An overview is given in [4].

3. Abstract Program Trees

In this section we present a mechanism to define abstract program trees. One could think of defining an abstract program tree by a transformation applied to a derivation tree. This is, however, not a suitable approach as we want the definition mechanism of abstract program trees to be a framework that can easily be extended to an attribute grammar-like definition. Moreover, we view an abstract program tree as an intermediate structure subject to further transformations. The concept of a derivation tree is therefore not a good starting point. Instead, we shall use a completely different model, namely graphs. Starting with graphs we will make a number of restrictions which will finally bring us to a grammar for the specification of abstract program trees.

3.1. Starting with Graphs

First of all we restrict our graphs to be directed and we assume every graph includes one node from which all other nodes can be reached when following the arcs. We will call this distinguished node the root node, because for trees it can be considered as the node on which the whole tree stands (or from which it hangs).

Secondly, we will color the nodes and the arcs (using graph terminology) and put some restrictions on the coloring. In our terminology we would refer to typing and labeling. The idea is to associate with every node a type and to define node types by a type rule that determines the number of outgoing arcs, their labeling (or distinguishing colors) and the permitted node types pointed to by these arcs.

We will now have a closer look at the type system that we are going to use. For each node type a (possibly empty) set of pairs could be given determining the number of arcs. Each pair consists of a

unique label and a non-empty set of allowed node types.
This leads to

$$
\begin{aligned}
&RULE(node\_type)\\
&= \quad \{ (label_1, \{node\_type_{1,1}, \cdots, node\_type_{1,m_1}\})\\
&\quad\ , \ \cdots\\
&\quad\ , (label_n, \{node\_type_{n,1}, \cdots, node\_type_{n,m_n}\})\\
&\quad\ \}
\end{aligned}
$$

where $n \geqslant 0$, $m_i > 0$ for $1 \leqslant i \leqslant n$, and $label_i \neq label_j$ for $1 \leqslant i < j \leqslant n$.

Now a family of graphs can be defined by a 3-tuple (N,L,R), where N is the set of node types, L is the set of labels and R is the set of type rules with a single type rule for every node type. Each member of this family is a typed graph in which all the nodes have the right number of arcs with the right labels, and the right node types at the end of the arcs.

3.2. Towards an Abstract Program Tree Grammar

Having constructed a language for the definition of graphs one could think of an extension of the classical attribute grammar framework from trees to graphs. However, such an extension introduces complications which go far beyond the scope of this paper. For this reason in this paper we restrict ourselves to trees only.

The above-mentioned type system applied to trees defines for every node type a fixed number of sons. The type of the father node puts some restrictions on the types of the son nodes. Every tree node can be reached from the root node through a unique path. The root node is considered as the highest node from which all other nodes hang.

We require a strict ordering of the sons of a node, which is the same for all the nodes of the same type. Such an ordering is needed for the definition of certain tree traversal strategies (e.g., especially for the pass-oriented strategies [1]).

In Section 3.1 the node types allowed at a certain arc have been written as a set. In practical applications, these sets are often similar. We therefore introduce an abbreviation mechanism called classes for these sets. A class will be defined as a subset of the node types. Having defined classes, trees can be defined by so-called tree rules, which are similar to the node type definitions in the previous section, but differ in that the labels have a fixed order, and the sets are replaced by a class or a node type. The labels will be called partnames, because they indicate a part of the tree under a node.

Tree rules are of the form

$$
\begin{aligned}
&a\_node\_type\\
&\quad \Longrightarrow \quad partname_1 : element_1\\
&\quad\ , \quad \cdots\\
&\quad\ , \quad partname_n : element_n \ .
\end{aligned}
$$

where $n \geqslant 0$, $element_i \in node\_types \cup classes$ for $1 \leqslant i \leqslant n$ and $partname_i \neq partname_j$ for $1 \leqslant i < j \leqslant n$.

In our formalism we shall also expand the class definitions. In most applications it is possible to divide the members of a certain class into groups that have the same properties, i.e., they have tree rules that are similar if we look at the allowed node types at their parts. This observation leads to two extensions of the previous definition.

We first introduce a hierarchical class definition by allowing classes to have other classes as their members. A class definition is of the form

$$a\_class = \{element_1, \cdots, element_n\}$$

where $n > 0$ and $element_i \in node\_types \cup classes$ for $1 \leqslant i \leqslant n$.
The introduction of hierarchical class definitions requires some restrictions. We exclude recursive class definitions, as for example

$$class\_A = \{class\_B, class\_A, node\_type\_N\}$$
$$class\_B = \{class\_A, node\_type\_M\}$$

We now introduce the concept of the closure over the class definitions. We shall use the function $class(C)$ to denote the set of members of class C. Likewise we shall use the function $clos\_class(C)$, for the closure over the member-of-class relation that will return all the members of $class(C)$ together with the $clos\_class$ of all the classes in $class(C)$.

Although recursive class definitions are not allowed, there remains a kind of ambiguity, as illustrated by the example

$$class\_A = \{class\_B, class\_C\}$$
$$class\_B = \{node\_type\_N, node\_type\_M\}$$
$$class\_C = \{node\_type\_N, node\_type\_K\}$$

In this example $node\_type\_N$ is a member of $clos\_class(class\_A)$, but in the case of an instance of $class\_A$ in an abstract program tree, where $node\_type\_N$ is selected, it is not clear whether $node\_type\_N$ is a member of $class\_B$ or $class\_C$. The question whether we will allow these ambiguous class definitions, is further dealt with in Section 4.1.

Secondly, we make the extension that tree rules may also be associated with a class. This means that in addition to a $node\_type$, we will also allow a class as the left-hand side of a tree rule. The tree rule associated with the class C holds also for all the elements in $clos\_class(C)$.

In the previous section we decided to associate exactly one tree rule with each node type. With respect to classes this rule requires that once a tree rule has been written for a class, no tree rule may be written for a member (or a member of a member, etc) of the class.

Also, if an element is a member of more than one class, it is not possible that more than one of these classes has an associated tree rule.

3.3. Formal Definition of Abstract Program Trees

In this section the concept of an Abstract Program Tree Grammar (APTG) will be defined formally. An APTG can be defined as a 5-tuple (V_N, V_C, TR, CD, R). The finite set V_N of node types and V_C of classes form the set of elements $V = V_N \cup V_C$. TR is the set of tree rules and CD is the set of class definitions. R is the root element which may be either a class or a node type.

The members of TR will be of the form

$$V_0 \Longrightarrow P_1 : V_1, \cdots, P_n : V_n \ .$$

where $n \geqslant 0$, $V_i \in V$ for $0 \leqslant i \leqslant n$, P_i is a partname and $P_i \neq P_j$ for $1 \leqslant i < j \leqslant n$.

The members of CD will be of the form

$$C = \{V_1, \cdots, V_n\}$$

where $C \in V_C$, $n > 0$ and $V_i \in V$ for $1 \leqslant i \leqslant n$. For each $C \in V_C$ there is exactly one member in CD which has C as its left-hand side. All the members of CD together should not include recursive class definitions. TR and CD combined should not define more than one tree rule for each node type of the grammar.

We will define the function $class(e:V)$ in the context of CD as

$class(e:V)$
= **if** $e \in V_N$
then $\emptyset$
else $\{V_1, \cdots, V_n\}$
where $'e = \{V_1, \cdots, V_n\}' \in CD$
fi

We will now define the set of abstract program trees defined by an APTG. We do not talk about derivations here, thus this cannot be viewed as an extension from string to tree grammars. Before we can do this we have to choose a representation for trees. We use the representation where every subtree is represented by the node type of its root, followed by the sequence of representations of its subtrees (in the same order as they are hanging in the tree), enclosed in the brackets "<" and ">". The brackets may be omitted if a node type has no sons. Each node type name with its associated brackets represents one instance of a node, with that node type, and the links (or arcs) to its subtrees. Because we have only one tree rule associated with every node type, this representation will be complete. The following example depicts the tree representing the arithmetic expression 3*(4+5)

mul_op < *num* , *plus_op* < *num* , *num* > >

In this representation the values of the numbers are not represented.

We first define a function *Tree* that returns all subtrees for a certain root element as

$$
\begin{aligned}
&Tree(V_N, V_C, TR, CD, s \in V) \\
= \quad &\{ N < t_1, \cdots, t_n > \\
&\mid N \in V_N \cap clos\_class\_and(s) \\
&\wedge\ 'V_0 \Longrightarrow P_1{:}V_1, \cdots, P_n{:}V_n' \in TR \\
&\wedge\ N \in clos\_class\_and(V_0) \\
&\wedge\ \forall 0 \leqslant i \leqslant n \\
&\qquad (t_i \in Tree(V_N, V_C, TR, CD, V_i)) \\
&\} \\
&\textbf{where}\ clos\_class\_and(s \in V) \\
&\qquad = \{s\} \cup \bigcup_{e \in class(s)} clos\_class\_and(e)
\end{aligned}
$$

Using this definition we can define the function *Tree* that for a given APTG G yields the set of all trees that are defined by it as

$$Tree(G{:}APTG) = Tree(V_N(G), V_C(G), TR(G), CD(G), R(G)) .$$

Above we have essentially presented a language to describe APT's, and defined the set of APT's that are defined by a given APTG. This language shows certain similarities with the Interface Description Language IDL [8,9]. A more restricted formalism for the description of abstract program trees is presented in [3].

4. Attributed Abstract Program Trees

For the specification of the information flow in abstract program trees we will augment our abstract program tree grammars with attributes and attribute evaluation rules in a similar way as classical attribute grammars have evolved from context-free grammars. However, for abstract program tree grammars this extension turns out to be far more complex.

4.1. How to Add Attributes

Abstract program trees are assumed to be decorated with attributes in the same way as attributes are attached to the nodes of a derivation tree in a classical attribute grammar implementation. As for classical attribute grammars, the attribute instances attached to an abstract program tree can be partitioned into inherited and synthesized attributes, according to the way they carry and receive their information to and from other attribute instances. The simplest way of defining the attributes of abstract program trees is to associate with each node type $E \in V_N$ a finite set $A(E)$ of attributes, partitioned into two disjoint subsets $I(E)$ and $S(E)$ of inherited and synthesized attributes, respectively.

Given a tree rule, the problem occurs of how to identify the attribute occurrences of this tree rule, as both the left-hand side and the right-hand side may contain classes which do not have attributes. In the rest of this paper we shall identify the elements of a tree rule by their partnames, and introduce # as the partname for the left-hand side element.

We distinguish two ways to identify an attribute occurrence of a tree rule $V_0 \Longrightarrow P_1{:}V_1, \cdots, P_n{:}V_n$. The first way is to write it as *attr* of $P_i.N$, where $0 \leqslant i \leqslant n$, $P_0 = \#$, $attr \in A(N)$ and $N \in V_N \cup clos\_class\_and(V_i)$. The second way is to write it as a sequence of elements expressing the hierarchical structure of classes containing classes, in the following way: *attr* of $P_i.E_1.\cdots.E_m$, where $attr \in A(E_m)$, $E_m \in V_N$, $m \geqslant 1$, $E_1 = V_i$, $E_j \in V_C$ and $E_{j+1} \in class(E_j)$ for $1 \leqslant j < m$.

At first sight the second method seems to be unnecessarily complicated or appears to use redundant information. This is true if we require the inclusion of node types in classes to be non-ambiguous; cf. 3.2. If classes are allowed to be ambiguous we can ask the question whether attributes at node types, ambiguously included in a class, have to be considered as different. In the latter case, we need to distinguish such attributes, and the only way to do this is to use the second method. We shall use the first method in the rest of this paper, and leave open the question of how to handle ambiguous classes.

The set of attribute occurrences associated with the tree rule of element E will be denoted by $AO(E)$. This set can be partitioned into two disjoint sets of defined occurrences and used occurrences, denoted by $DO(E)$ and $UO(E)$, respectively.

These subsets are defined as

$$\begin{aligned} DO(E) &= \{\, s \text{ of } P_0.N \mid s \in S(N) \,\} \\ &\quad \cup \{\, i \text{ of } P_i.N \mid i \in I(N) \wedge 1 \leqslant k \leqslant n \,\} \\ UO(E) &= AO(E) - DO(E) \\ &= \{\, i \text{ of } P_0.N \mid i \in I(N) \,\} \\ &\quad \cup \{\, s \text{ of } P_i.N \mid s \in S(N) \wedge 1 \leqslant k \leqslant n \,\}. \end{aligned}$$

We have defined above how to address attribute instances in the tree by attribute occurrences, and we have defined the sets of attribute occurrences associated with a tree rule. The set of attribute instances attached to a concrete node in a tree and its sons, is generally only a subset of the attribute occurrences associated with the tree rule applied at this node. This is because attributes are attached to node types, and for a given tree rule, classes can be involved in both sides of the rule.

4.2. The Attribute Evaluation Rules

In the same way as for attribute grammars we associate with each tree rule a set of attribute evaluation rules which specify how to compute the values of the attribute occurrences in $DO(E)$, where E is the element in the left part of the tree rule. Only the evaluation rules for those attributes that are attached to a certain node in a tree are applied. But for the right-hand side of the rule some problems may arise. We cannot know whether an attribute occurrence is available at a certain position in the tree. Take for example the tree rule

plus ⟹ *left* : *expression*,
right : *expression*.

together with the class rule

expression = {*constant*, *plus*}

and an attribute *value* ∈ *A* (*constant*). At a node typed *plus* we cannot with certainty refer to *value* of *left.constant* because its identity may not be of type *constant* but of type *plus*.

To solve this problem we need a mechanism to find out which node type is applied at a position in the tree, where a tree rule has a class. We do this by introducing case-expressions on the partnames. Referring to the above example, we could for example write

case *left* **of**
constant : *value* **of** *left.constant* ;
plus : 0
esac

to express that we want the value of attribute *value* if the actual node at *left* is of type *constant*, and otherwise the value 0. In the case of hierarchical class definitions we need nested case-expressions. We shall now define the expressions which form the right-hand side of the attribute evaluation rules. We distinguish three different constructs. Firstly an attribute occurrence, secondly a general function format in which the arguments are again expressions, and thirdly the case-expression. We could describe the syntax using

expr ::= *attr* **of** *partname*
| $f(expr_1, \cdots, expr_n)$
| **case** *partname* **of**
E_1 : $expr_1$;
...
E_m : $expr_m$
esac

We need a number of semantic conditions. To every subexpression a context can be assigned that specifies the binding of elements to partnames. In the context of the whole expression the binding is specified

by the tree rule. For an attribute occurrence *attr* of *partname*, *attr* of *partname.N* has to be a member of $UO(E)$, where N is the node type that is bound to partname *partname*, and E the element in the left part of the tree rule. The context of the arguments of a function will be the same as the context of the whole function-expression. For a case expression the following restrictions should be imposed. If we assume C to be the element bound to partname *partname*, then $C \in V_C$, $class(C) = \{E_1, \cdots, E_m\}$ and $E_i \neq E_j$ for $1 \leqslant i < j \leqslant m$. The context of each subexpression $expr_i$ has to be such that E_i will be bound to *partname*, and the rest of the context will be the same as the context of the whole case-expression.

We can now define the attribute evaluation rules to be of the form

attr of *partname.N* = *expr*

where *attr* of *partname.N* $\in DO(E)$ and *expr* is a correct expression as described above.

We could also introduce for reasons of orthogonality a case mechanism for the left-hand side of the evaluation rules. With these case constructions it is possible to combine several rules together that have the same partname for the left-hand side. Furthermore, if we make the extension that we can combine cases with the same expression into one case, then this can lead to a reduction on the size of the rules. However, these extensions are merely syntactic sugar.

4.3. Attributes Attached to Classes

In the foregoing discussion we attached attributes to the node types of the grammar only. In practical applications it often occurs that the same attributes are associated with all the members of a class. This leads to case-expressions which have the same expression for all the cases. To solve this problem, we allow an attribute to be associated with a class if it is associated with all the members of that class. Of course, this rule applies recursively over the hierarchy of the class definitions. This has implications to the definition of $AO(E)$, $UO(E)$ and $DO(E)$. We can now write *attr* of *partname.E'*, where $E' \in V$ instead of $E' \in V_N$. We can also weaken the semantic restriction on the attribute occurrences in the expression on the right-hand side of an evaluation rule. For the attribute occurrence *attr* of *partname*, *attr* of *partname.E'* has to be a member of $UO(E)$ where the element bound to the partname *partname* is a member of *clos_class_and* (E').

4.4. How to Evaluate Attributes

Just as with traditional attribute grammars we have to define how the attributes are evaluated for a given tree. For each tree $t \in Tree(G)$, instances of attributes are attached to the nodes in the following way:

if a node n is of node type N, then for each attribute $a \in A(E)$ where $N \in clos\_class(E)$, an instance of a is attached to node n. An attribute instance a of node n will be denoted by a **of** n.

Let n_0 be a node with node type N_0 and let $n_1, n_2, \cdots, n_m$ be its sons with node types $N_1, N_2, \cdots, N_m$ respectively. An attribute evaluation instruction

$$a \text{ of } n_k := f(a_1 \text{ of } n_{k_1}, a_2 \text{ of } n_{k_2}, \cdots, a_m \text{ of } n_{k_l})$$

is associated with the instances of a **of** n_k, which is extracted from the evaluation rule of a **of** $P_k.N_k$, where a_i **of** n_{k_i} represent the instances that are used. Because we now know which node types are applied at the different partnames with the tree rule of node type N_0, we can replace every **case** P_i **of** $\cdots E$: *expr* $\cdots$ **esac** in the evaluation rule of a **of** $P_k.N_k$ by *expr* where $N_i \in clos\_class(E)$. Thus the function f representing the applied evaluation rule, depends only on the attribute values attached to the nodes.

The task of an attribute evaluator is to compute the values of all attribute instances attached to the nodes of a tree by executing their associated evaluation instructions, in the same manner as for traditional attribute grammars.

5. Conclusions

In this paper we have demonstrated how to describe abstract program trees with a tree grammar and we have shown that it is possible to add attributes to the definition of that grammar.

A more detailed description of this approach can be found in [5]. A similar solution is presented in [9]. The attribution of abstract program trees is also discussed in [6] and [11].

In our research project on compiler-compilers we have implemented an attribute evaluator generator that given an APTG generates a PASCAL program to evaluate the attributes in any APT of that grammar. In this work we encountered a number of problems in producing efficient code. These implementation problems involved making non-trivial choices in the storage of the attributes associated with the classes in a tree. This work will be reported in a future paper.

Our current research includes the description of transformations of abstract program trees for the purpose of program optimization, and the generation of programs that can perform these transformations while keeping attribute values consistent. It should also be possible to generate transformations from one grammar to another, for example in the code generation phase of a compiler.

Acknowledgement. We are grateful to Albert Nymeyer who helped in the preparation of this paper.

References

1. H. Alblas: A characterization of attribute evaluation in passes, *Acta Inform.* **16** (1981) 427-464.
2. H. Alblas: Incremental simple multi-pass attribute evaluation, *Proc. NGI-SION Symposium* **4** (1986) 319-342.
3. F.L. DeRemer & R. Jullig: Tree-affix dendrogrammars for languages and compilers, *in:* Semantics-directed compiler generation, Lect. Notes Comp. Sci. **94** (1980) 300-319, Springer-Verlag, Berlin - Heidelberg - New York.
4. J. Engelfriet: Attribute Grammars: Attribute evaluation methods, *in:* B. Lorho (Ed.): *Methods and Tools for Compiler Construction*, Cambridge University Press, (1984) 103-138.
5. F.J. Faase: Een attribuut evaluator generator, Masters thesis, Dept. of Computer Science, University of Twente, Enschede, The Netherlands, (1986).
6. I. Glasner, U. Möncke & R. Wilhelm: OPTRAN, a language for the specification of program transformations, *Informatik-Fachberichte* **34,** (1980) 125-142, Springer-Verlag, Berlin - Heidelberg - New York.
7. D.E. Knuth: Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127-145, Correction in: *Math. System Theory* **5** (1971) 95-96.
8. J.R. Nestor, W.A. Wulf & D.A. Lamb: IDL-Interface Description Language, Technical Report, Dept. of Computer Science, Carnegie Mellon University (1981).
9. J.R. Nestor, B. Mishra, W.L. Scherlis & W.A. Wulf: Extensions to attribute grammars, Technical Report TL 83-36, Tartan Laboratories Inc., Pittsburgh (1983).
10. H. van Thienen: Automatic Generation of Abstract Grammars, Memorandum INF-87-19, Dept. of Computer Science, University of Twente, Enschede, The Netherlands, (1987).
11. R. Wilhelm: Computation and use of data flow information in optimizing compilers, *Acta Inform.* **12** (1979) 209-225.

Program Generation through Symbolic Processing

J.A.van Hulzen

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Computer algebra systems can be useful when attempting to automatize mathematics. One can use these facilities to assist in the construction of programs for numerical purposes, i.e., to assist in making the problem dependent parts of the software needed to solve a problem numerically. We discuss aspects of the symbolic-numeric interface required to accomplish this. Special attention is given to program generation aspects and to code optimization.

1. Introduction

Powerful computing resources are available today on a personal basis and for limited costs. It is therefore predictable that the use of personal computers to perform scientific computations will considerably increase. This in turn will enlarge the interest in a further and better integration of various mathematical software tools, such as computer algebra features and numeric and graphic facilities. It is therefore also expectable that computer algebra will slowly shift from a seemingly exotic and expensive hobby to an essential toolkit in problem solving, assuming adequate symbolic-numeric and symbolic-graphic interfaces are created. Computer algebra systems require dynamic development and dynamic storage of the mathematical expressions they allow to elaborate. But a language like FORTRAN, frequently employed to solve problems numerically, is used in a compile, load and execute fashion. The solution strategy is often based on the use of library subroutines in a problem defined context. A computer algebra system can be used for the construction of the mathematical expressions, which together define a specific problem and how to utilize the library facilities for its solution. Hence a symbolic-numeric interface is needed to transport information from one world to another, from a symbolic processing environment to a numeric scene. This, of course, must be worth the effort, i.e., the information to be transported must be extensive. In fact so extensive, that producing it by hand is not only error prone and impractical, but almost impossible. The symbolic-numeric interface will therefore ideally cover possibilities for program construction, code optimization and (a priori) error analysis (features). We discuss some of the aspects of such an interface, which are needed for

program generation. The type of generation we are interested in concentrates on "easy" construction of efficient and reliable programs. But such a discussion needs as a prerequisite some insight in the overall functioning of a computer algebra system. We hope to provide this knowledge in Section 2, using REDUCE to illustrate our assertions. Relevant aspects of and approaches to a symbolic-numeric interface are then presented in Section 3. Section 4 is dedicated to program generation. We mention some ideas about code optimization, intended for the production of more reliable and more efficient programs in Section 5, before some conclusions are given. Our contributions to the development of a symbolic-numeric interface are, in fact, realized as extensions of REDUCE. This is an additional reason to pay attention to REDUCE.

2. Computer Algebra

The differentiation programs of Kahrimanian [35] and Nolan [44], presented in 1953, are often considered as the first attempts to employ a digital computer to perform formal mathematical operations. We now know a rich diversity of computer algebra systems [63]. Some of these systems are frequently and routinely used to assist in solving non-trivial problems in science and engineering [2,13,48]. Well known are MACSYMA [48], MAPLE [16], muMATH [53,58], REDUCE [21] and SCRATCHPAD [34]. This list is certainly not exhaustive. We only mentioned some of the intended general purpose systems, which are either widely used, like REDUCE, or have a noteworthy design, like SCRATCHPAD. Introductory surveys of computer algebra are given in [47,70]. Recent summaries of the state of the art can be found in [11,15,19]. The mathematical capabilities of the better systems of today are of course strongly correlated to the early successes of computer algebra in such areas as integration, celestial mechanics, general relativity and quantum electro dynamics. These applications tended to shape the classes of mathematical expressions, to be formulated and manipulated in the various systems. Polynomial and rational function algebra was considered as a basic requirement. All of the well-known elementary transcendental functions, naturally entering in the description of our (approximate) models of physical reality, were and are considered as intriguing objects. A classification of computer algebra systems can be based on the class of mathematical expressions they allow to operate on. The impact of such a system is largely related to the class of transformations, which it allows to perform on its expressions, either automatically or via user control. Examples of such transformations are differentiation, integration and substitution. Portability, maintenance and ergonomic aspects, such as ease of interactive use, comprehensibility of output and performance, are additional criteria for judging such a system.

The mathematical criteria are strongly related to the quality of the algebraic simplification algorithms implemented in the system.

Simplification was once qualified as the most pervasive process in algebraic manipulation [42]. Much of the controversy around it is due to differences between the desires of a user and a designer, because the notion of simplicity is not context-free. Simplification has two aspects. An important issue is to be able to obtain an equivalent but simpler representation of a mathematical object, either internally or externally. Another aspect deals with the question how to compute a unique representation for equivalent objects. Finding equivalent but simpler objects requires an effective procedure $S : T \to T$, where T is a class of mathematical objects, such that for all t in T holds that $S(t) \sim t$ and $S(t) \leqslant t$, if $\sim$ is an equivalence relation on T and if " $\leqslant$ " connotes "simplicity". Obtaining a unique representation requires, in addition, that for all s,t in T holds $s \sim t \implies S(s) = S(t)$. Hence S is meant to single out a unique representative for each equivalence class. $S(t)$ is therefore called the canonical form of t. However, it is proven that a canonical simplifier can not always be found for an equivalence relation on a given set of (mathematical) objects [12,14]. Therefore — in practice — weaker notions are employed, such as zero-equivalence and regular simplification. Zero-equivalence can be defined when a given set of expressions contains a zero-element 0. We then call $S : T \to T$ a zero-equivalence (or normal) simplifier if for $\sim$ on T holds that for all t in T: $S(t) \sim t$ and $t \sim 0 \implies S(t) = S(0)$. Regular simplification is used in the context of expressions involving transcendental functions. It guarantees that transcendentals occurring in an expression are algebraically independent, a requirement which is for instance needed in the design of symbolic integration facilities, based on the Risch-Norman algorithm [45,46]. Simplification can be used as a ("political") instrument to produce a classification of computer algebra systems, as once done by Moses [42]:

- Radical systems can handle a single well defined class of expressions (polynomials, rational functions, for instance), by using a canonical simplifier to get all expressions into their internal canonical form. This implies that the task of the manipulating algorithms is well defined. But it can lead to inefficiencies. On input a user can present an expression as a string over a certain alphabet in any desired, syntactically correct form. This string is just one possible external representation of one internally unique object, being the representative of a whole equivalence class. To obtain such a unique object we need a set of rules, defining term ordering via degree ordering or — alternatively — ordering of the (irreducible) factors of an expression, in combination with some lexicographical ordering of the variable-symbols, occurring in the alphabet. This can imply that the output, being a reflection of the internal ordering, can surprise a user.
- New Left systems arose in response to some of the difficulties with radical a systems, such as caused by automatic expansion

(think of $(x+y)^{1000}$) or factorization (for instance $x^{1000}-y^{1000}$). Expansion is brought under user control. Such systems usually can handle a wide variety of expressions with greater ease by using labels for non-rational (sub)expressions. REDUCE is such a system.

- Liberal systems give more freedom to a user and are therefore in general slower than new left systems.
- Catholic systems, finally, can use more than one internal representation and know different approaches to simplification. They tend to be large. A well-known example is MACSYMA.

Most computer algebra systems are interactive. The system reaction, an output expression, is of course just one of many possible visual representations of an internally stored expression. Other striking aspects of the use of such a system are time and space requirements. Intermediate expression swell is a well-known phenomenon. It can be caused by temporary fill-in. Factorization for instance requires expansion. Differentiation is another example of possible intermediate expression explosion. Since the purpose of computing can be qualified as an attempt to increase insight, it is obvious that we are interested in obtaining the most simple form of an expression. This is often also the shortest representation of an expression. Hearn, who designed and implemented most of REDUCE, has been studying these problems since he started making this system. He recently [29,30] gave a nice classification of simplification approaches, when considered as expression structuring activities:

- Structure preserving techniques are concerned with maintaining structure in an expression as long as possible in a given computation.
- Structure determining techniques cover attempts to induce structure on otherwise unstructured expressions.
- Structure reducing techniques are those which can be employed to reduce an expression using a set of side relations.
- Structure displaying techniques allow to present the output in a form that makes its structure more apparent to the user.

Structure preserving techniques are based on the reasonable presumption that the initial formulation of most scientific problems has a natural structure. Most simplifiers are based on this structure preservation philosophy. For instance taking an expression like

$$(x+1)^2-2*x$$

we immediate see that it has the simpler form

$$x^2+1\,.$$

We want our algebra system to produce this result as well. Input

expansion easily allows to get this result, if we collect terms of equal degree and employ ordering considerations. However, a form like

$$(x + 1)^{100} + 1$$

can better not be expanded at all. Brown [7] was the first who observed that more flexibility was needed, against the price of dropping a canonical representation. He proposed to guarantee a normal form by representing a polynomial as a product of expanded factors in the form :

$$\prod_i \Big(\sum_{n_i=0}^{N_i} u_{n_i} x^{n_i} \Big)^{k_i}$$

Simplification is straightforward. When multiplying polynomials, given in such a form, one simply maintains the existing factor structure. For addition one starts collecting equal factors, before adding the expanded remaining portions to produce a new factor. Hearn implemented similar facilities in REDUCE. An implication is the need to allow two internal forms, i.e., an expanded as well as a factored form. The user operates by default with expanded forms, thus using a canonical simplifier. He can employ a factored form on request, implying that the non-expanded form construction is based on normal simplification. But this does not always result in a factored form. Internally a comparison is always made between the two alternatives. The shortest is stored. But how? In REDUCE a recursive polynomial definition is used [27]. The system is implemented using Standard LISP [41], to guarantee a degree of portability. Thus the internal representation is always in the form of lists. The recursive definition implies that a polynomial is stored as a pair consisting of a leading term and a reductum, formed by the remaining terms of the polynomial, ordered in some system dependent way, with (of course) the possibility of user influence. A leading term is considered to be a pair again. This time formed by a leading coefficient and a leading power. The coefficient can again be a polynomial. The leading power also consists of a pair, now formed by a main variable and its leading degree. The leaves of this binary tree are either integer coefficients, non-zero integral powers or variables, of which the ordering can be determined either via the object list, or by user defined alternatives. To resolve the problem of undesirable expansions, like for

$$(x + 1)^{100} + 1 ,$$

Hearn generalized the variable-concept [28]. In stead of the notion variable REDUCE utilizes the kernel-concept. A kernel can either be a variable in the traditional sense or a polynomial. So in the above given example $(x + 1)$ acts like a variable and expansion can thus be avoided. Once the parser knows of transcendental functions, like sine and cosine, lists of the form (sine argument), for instance, can also be used

as a kernel. And again the argument can be a recursively defined polynomial. The REDUCE simplifier assumes all input to be the quotient of two polynomials (again a pair). When the input is really a polynomial the denominator-part of this so called Standard Quotient is simply 1. In summary:

<Standard Quotient>	::=	<Numerator> / <Denominator>
<Numerator>	::=	<form>
<Denominator>	::=	<polynomial>
<form>	::=	| <polynomial>
<polynomial>	::=	<integer> |
		<Leading Term> + <Reductum>
<Reductum>	::=	<form>
<Leading Term>	::=	<Leading Power> *
		<Leading Coefficient>
<Leading Coefficient>	::=	<polynomial>
<Leading Power>	::=	<Kernel> ↑ <Leading Degree>
<Leading Degree>	::=	<nonzero positive integer>
<Kernel>	::=	<variable> | <polynomial> |
		<operator> (<list of operands>)

In addition it ought to be mentioned that the rich output repertoire of REDUCE can assist a user in influencing the visual version of the internal representation of the result of a computation, always being the transformation of an expression. Worth mentioning are tools to change the variable precedence or to display a partly factored form. Another facility which allows to modify output is formed by certain structure displaying commands, as mentioned by Hearn. The expression, subjected to such a command, is cut into obvious pieces which are renamed and separately shown. The renaming allows to list repeatedly occurring subexpressions only once.

The main reason to explain the overall functioning of REDUCE in some detail is to simplify our discussion of the symbolic-numeric interface. It might be illustrative to give

Example 2.1. Let us assume that we are interested in the determinant DM of the symmetric matrix

$$M = \begin{bmatrix} t_0 & t_1 & t_2 \\ t_1 & t_3 & 0 \\ t_2 & 0 & t_4 \end{bmatrix}$$

So obviously we have

$$DM = t_0 * t_3 * t_4 - t_1^2 * t_4 - t_2^2 * t_3.$$

Let us now assume that the entries of M have the following values (This matrix was derived in the course of research reported in [3]):

```
                  2                            2                    2
M(1,1) :=  - ((9*P *M30 + J30Y - J30Z)*SIN(Q3)  - (18*M30 + M10)*P  -

                                  2
              18*COS(Q3)*COS(Q2)*P *M30 - J30Y - J10Y)

                            2                            2
M(2,1) := M(1,2) :=  - ((9*P *M30 + J30Y - J30Z)*SIN(Q3)  - 9*COS(Q3)*

                       2           2
              COS(Q2)*P *M30 - 9*P *M30 - J30Y)

                                                  2
M(3,1) := M(1,3) :=  - 9*SIN(Q3)*SIN(Q2)*P *M30

                  2                            2      2
M(2,2) :=  - ((9*P *M30 + J30Y - J30Z)*SIN(Q3)  - 9*P *M30 - J30Y)

M(3,2) := M(2,3) := 0

              2
M(3,3) := 9*P *M30 + J30X
```

Neglecting the above given structure and using the facility REDUCE offers to compute the determinant of a given matrix, can lead to a number of different visualizations of one and the same object.

1. The result is presented in expanded form.

```
          4        2  6    3             4        2  4    2
729*SIN(Q3) *SIN(Q2) *P *M30  + 81*SIN(Q3) *SIN(Q2) *P *M30 *J30Y - 81

        4        2  4    2                  2        2  6    3
*SIN(Q3) *SIN(Q2) *P *M30 *J30Z - 729*SIN(Q3) *SIN(Q2) *P *M30  - 81*

       2        2  4    2                  2  6    3            2  6
SIN(Q3) *SIN(Q2) *P *M30 *J30Y - 729*SIN(Q3) *P *M30  - 81*SIN(Q3) *P

    2                2  4    2                2  4    2
*M30 *M10 - 81*SIN(Q3) *P *M30 *J30Y + 81*SIN(Q3) *P *M30 *J30Z - 81*

       2  4    2                2  4    2                2  4
SIN(Q3) *P *M30 *J1OY - 81*SIN(Q3) *P *M30 *J30X - 9*SIN(Q3) *P *M30*

                    2  4                              2  4
J30Y*M10 + 9*SIN(Q3) *P *M30*J30Z*M10 - 9*SIN(Q3) *P *M30*M10*J30X - 9

       2  2                          2  2                          2
*SIN(Q3) *P *M30*J30Y*J1OY - 9*SIN(Q3) *P *M30*J30Y*J30X + 9*SIN(Q3) *

 2                          2  2                          2  2
P *M30*J30Z*J1OY + 9*SIN(Q3) *P *M30*J30Z*J30X - 9*SIN(Q3) *P *M30*

                    2  2                          2  2
J1OY*J30X - SIN(Q3) *P *J30Y*M10*J30X + SIN(Q3) *P *J30Z*M10*J30X -

       2                          2                          2
SIN(Q3) *J30Y*J1OY*J30X + SIN(Q3) *J30Z*J1OY*J30X - 729*COS(Q3) *

       2  6    3             2        2  4    2             6    3
COS(Q2) *P *M30  - 81*COS(Q3) *COS(Q2) *P *M30 *J30X + 729*P *M30  +

    6    2           4    2           4    2           4    2
81*P *M30 *M10 + 81*P *M30 *J30Y + 81*P *M30 *J1OY + 81*P *M30 *J30X

      4                   4                   2                   2
 + 9*P *M30*J30Y*M10 + 9*P *M30*M10*J30X + 9*P *M30*J30Y*J1OY + 9*P *

                   2                   2
M30*J30Y*J30X + 9*P *M30*J1OY*J30X + P *J30Y*M10*J30X + J30Y*J1OY*J30X
```

2. We turn off the expansion and get a normal form.

```
     2                                  2                    2
((9*P *M30 + J30Y - J30Z)*SIN(Q3)  - (18*M30 + M10)*P  - 18*COS(Q3)*

 COS(Q2)*P *M30 - J30Y - J1OY)*((9*P *M30 + J30Y - J30Z)*SIN(Q3)  - 9*

    2                  2
   P *M30 - J30Y)*(9*P *M30 + J30X) -

     2                                  2                    2
((9*P *M30 + J30Y - J30Z)*SIN(Q3)  - 9*COS(Q3)*COS(Q2)*P *M30 - 9*P *

          2     2                             2
 M30 - J30Y) *(9*P *M30 + J30X) + 81*((9*P *M30 + J30Y - J30Z)*

     2     2                          2        2  4    2
   SIN(Q3)  - 9*P *M30 - J30Y)*SIN(Q3) *SIN(Q2) *P *M30
```

3. We use the possiblily to get the structure of the determinant displayed for the unexpanded form of *DM*, and now denoted by S7:

```
S7
   WHERE
      S7 := S3*S4*S5 - S6**2*S5 + 81*S4*SIN(Q3)**2*SIN(Q2)**2*P**4*M30**2

      S6 := S1*SIN(Q3)**2 - 9*COS(Q3)*COS(Q2)*P**2*M30 - 9*P**2*M30 - J30Y

      S5 := 9*P**2*M30 + J30X

      S4 := S1*SIN(Q3)**2 - 9*P**2*M30 - J30Y

      S3 := S1*SIN(Q3)**2 - S2*P**2 - 18*COS(Q3)*COS(Q2)*P**2*M30 - J30Y -
            J1OY

      S2 := 18*M30 + M10

      S1 := 9*P**2*M30 + J30Y - J30Z
```

4. Finally we display this *DM*-structure in FORTRAN-notation.

```
      S1=9*P**2*M30+J30Y-J30Z
      S2=18*M30+M10
      S3=S1*SIN(Q3)**2-S2*P**2-18*COS(Q3)*COS(Q2)*P**2*M30-
     . J30Y-J1OY
      S4=S1*SIN(Q3)**2-9*P**2*M30-J30Y
      S5=9*P**2*M30+J30X
      S6=S1*SIN(Q3)**2-9*COS(Q3)*COS(Q2)*P**2*M30-9*P**2*
     . M30-J30Y
      S7=S3*S4*S5-S6**2*S5+81*S4*SIN(Q3)**2*SIN(Q2)**2*P**4
     . *M30**2
      S=S7
```

None of these forms is as compact as the originally given one using the t_i's. The conclusion is that much room for improvement of output presentation exists and that the results, although easily obtained, can be far from optimal, especially when a numerical value for *DM* is required for a given set of input values for the different variables occurring in *DM*.

In a numerical setting methods for solving systems of linear equations and determinant calculations are polynomial time-bounded operations, both in time and space. In a computer algebraic setting however, the algorithms show an exponential behaviour [31], although we have to remark that in such a setting problem size is always moderate in comparison with "numerical" problems. This limited size is related to core consumption during intermediate stages in the computations and to storage requirements for the final result. This example is quite illustrative. It clearly suggests what might happen when the matrix-size is enlarged and expansion is not turned off, for instance. □

The example also serves to stress that simplification, although algorithmic in nature, is not context-free. One has to try to avoid

undesirable side effects quite carefully. This is the main reason Hearn began considering the possibility of using structure determining techniques, i.e., heuristic tools to find structure in an expression, which otherwise would remain unchanged. Hearn's presumption is that many physical problems have enough structure to allow user-controlled regrouping, based on expansion or factorization and applied at some lower levels inside an expression, and using knowledge about the "weighted", physical meaning of the various variables used to built the given expression. What can be done is considering an expression to be a (multivariate) polynomial in (a) certain variable(s), factorize its coefficients or searching the different terms in these coefficients for common subexpressions to be factored out. These regrouping techniques can lead to remarkable compressions as is for instance shown by the following

Example 2.2. We show the effect of compression when applied on the expanded form of *DM*, taken from Example 2.1.1. Application of the same compression command on the unexpanded version of *DM*, as shown in Example 2.1.2, does not lead to an improvement.

```
- (((9*((J30Y - J30Z)*(J1OY + J30X) + J1OY*J30X)*M30 + J30Y*M10*J30X
                                                    2
     - J30Z*M10*J30X)*P  + 9*(9*(J30Y - J30Z + J1OY + J30X)*M30 + (
                                                  4                2
         J30Y - J30Z + J30X)*M10)*P *M30 + 81*(9*P *M30 + J30Y)*
          2  4    2                          6    2
    SIN(Q2) *P *M30  + 81*(9*M30 + M10)*P *M30  + J30Y*J1OY*J30X -
                                      2
    J30Z*J1OY*J30X)*SIN(Q3)  - (9*((J1OY + J30X)*J30Y + J1OY*J30X)*
                                  2                          2
      M30 + J30Y*M10*J30X)*P  - 81*((J30Y - J30Z) + 9*P *M30)*
       4        2  4    2
    SIN(Q3) *SIN(Q2) *P *M30  - 9*(9*(J30Y + J1OY + J30X)*M30 + (J30Y
                                4                2                    2
         + J30X)*M10)*P *M30 + 81*(9*P *M30 + J30X)*COS(Q3) *
          2  4    2                          6    2
    COS(Q2) *P *M30  - 81*(9*M30 + M10)*P *M30  - J30Y*J1OY*J30X)
```

Also quite illustrative is the result of performing some compression experiments on the expression EXPR, given below. It shows why it is important that algebra systems can be used interactively and it stresses again that simplification is not context-free.

```
on exp $

expon := expr;

                  2              2  2            2                  2
EXPON := 2*C*D*E  - 2*C*D*F *G  - 4*C*D*F *G*H*K - 4*C*D*F *G*H*L - 2*

           2  2  2           2  2              2  2  2       2
         C*D*F *H *K  - 4*C*D*F *H *K*L - 2*C*D*F *H *L  + F*G  + 2*F*

                                 2  2          2           2  2
         G*H*K + 2*F*G*H*L + F*H *K  + 2*F*H *K*L + F*H *L

TIME: 833 MS

off exp$

expoff := expr;

                    2                2  2                    2  2    2
EXPOFF :=  - (2*(((K  + 2*K*L + L )*H  + 2*(K + L)*G*H + G )*F  - E )*

                 2                2  2                    2
             C*D - ((K  + 2*K*L + L )*H  + 2*(K + L)*G*H + G )*F)

TIME: 901 MS

nfac expoff,c,f,(lfactr);

                                2                  2
 - (((K + L)*H + G) *(2*C*D*F - 1)*F - 2*C*D*E )

TIME: 1700 MS

nfac expoff,c,lfactr,(f);

                          2  2    2                2
 - (2*(((K + L)*H + G) *F  - E )*C*D - ((K + L)*H + G) *F)

TIME: 1904 MS
```

Citing Hearn [26], the present simplification algorithms in REDUCE, being used when the expansion is turned off, are a "moving target". This is mainly due to the fact that he is experimenting with the above indicated compression facilities, since the first experimental facilities were made by Hulshof, when visiting Hearn. Details about these facilities, as implemented for REDUCE, can be found in [33]. □

The above introduced structure-determining techniques can contribute to a reduction of the arithmetic complexity of an expression, which is needed in further numerical calculations. Another structure-determining technique — certainly in Hearn's view — is formed by the code optimization techniques, which are discussed in Section 5. They are based on reduction in arithmetic occurring in a given (set of) expression(s) by heuristically searching for common (sub)expressions. Hearn hopes that such heuristic techniques can be made instrumental for algorithmic methods to assist in a further reduction of the structure of an expression to a more simple form. The key idea is, that such common subexpression searches can lead to information about possibly

occurring side relations, which can be used for such a reduction. Another interesting thought of Hearn is to use a Gröbner base algorithm to assist in determining if these candidate side relations are consistent, by investigating their algebraic interrelations [9,10,29,30]. We only made these last remarks to underline that both heuristics and algorithmics have an important role to play in future developments in computer algebra, directed towards improving the quality of the output, certainly also when needed for further numeric work.

We gave a capsule view of some of the output features of computer algebra systems. The main intention in doing so is to provide a view on or perhaps a feeling for the rich variety of output possibilities — and thus of unwished inefficiencies — allowed by algebra systems. Although illustrated by REDUCE, similar remarks can be made for other computer algebra systems. These considerations also play a role in the next section.

3. The Symbolic-Numeric Interface

Aspects of the symbolic-numeric interface are discussed in some detail by Brown and Hearn [8] and complementary to them by Ng [43]. The apparent need for such an interface suggests, as already indicated in the introduction, that certain communication problems exist, related to information exchange between computer algebra systems and programming facilities, more specifically designed for numerical purposes. Brown and Hearn distinguished two problem sources: Numerical evaluation of symbolic results and Hybrid problems. The latter category demands for solution methods which are a mixture of numeric and symbolic techniques, implying that at some stage numerical evaluation of symbolic results might be needed as well. For numerical evaluation one can choose between, say interpretative evaluation, using a computer algebra system for both symbolic and numeric processing, and generation of arithmetic statements in an existing language for numeric processing. Both alternatives have certain drawbacks and implications.

Interpretation might be convenient for "one shot" applications (citing Ng [43]), if big float facilities, such as Sasaki's package [54], can be used and if problem size is moderate. Kanada and Sasaki [36] found their Standard LISP-package to be half as fast as Brent's well-known FORTRAN package [5], if they guarantee portability. Steele [56] and Pitman [49] came to similar conclusions concerning the use of MACSYMA for numerical evaluation. Pitman made FORTRAN to LISP translation facilities, thus creating the possibility of using the IMSL (International Mathematics and Statistics Library) in a LISP context. A drawback might however be that error analysis, and thus control over the precision of the big float calculations, is still left to the user.

When only differentiation is needed one can use instead of a computer algebra system special software tools, which in addition allow to utilize interval arithmetic to obtain reliable results [18,37,51,52]. These tools essentially use augmented FORTRAN or PASCAL compilers, which allow to produce subprograms, defining derivatives, created by making use of expression flow graphs, reflecting some form of intermediate 3-address code [1]. These approaches, however, do not provide simplification and thus can severely suffer from inefficiencies or limit the applicability to problems of moderate size.

The alternative – generation of arithmetic statements – is not perfect either. Many computer algebra systems offer users the possibility to obtain output in the form of assignment statements in FORTRAN notation. If the user decides to employ such a facility the obvious intend is to construct in some way or another complete programs and/or subroutines, which contain this arithmetic in some meaningful order. We discuss in this context strategies, which have been developed to assist users in producing such code in the next section.

Expression size might be an additional problem. Applications and application strategies, as for instance described by Cook [17], Smit and van Hulzen [55], Steinberg and Roache [57], Van den Heuvel et al. [60] and Wang et al. [65,66,67] clearly illustrate that computer algebra systems have to be used carefully. Often a form of lazy evaluation is employed to reduce or delay simplification activities. These applications illustrate Hearn's warning [29] that we have to learn to deal effectively with structure, which for instance might have been imposed by symmetry or by additional physical knowledge. Wang [65,66] recently showed how profitable this can be for the generation of finite element analysis software. Hearn also stated, as explained in the previous section, that the output we obtain is just one of a large number of possible representations and that structure determining techniques, such as code optimization, to be discussed in Section 5, can have a dramatic influence on reducing the arithmetic reflected by computer algebra output. This is also illustrated in most of the just mentioned papers on applications.

Once we are able to effectively generate efficient code for performing numerical computations it would be quite helpful if we are also able to guarantee the reliability of these calculations. In view of the existence of multiple precision floating point arithmetic packages it might be attractive to employ the power of a computer algebra system to determine, prior to the actual computations, how the precision has to be chosen during (parts of) the real computations, as to avoid unnecessary loss of significant digits. Our experiments with REDUCE and using Sasaki's big float package suggest that, in principle, this is possible [32]. The augmented compiler approach concentrates on creating limited symbolic facilities in a numerical context, as to allow to perform reliable computations, requiring at some stage derivatives,

without looking at the efficiency of the production of these derivatives. The creation of symbolic-numeric interfaces is still in development and has not yet resulted in completed facilities, which can be utilized to obtain reliable results.

The above outlined aspects for the construction of programs for numerical purposes are obviously related to the more traditional sequential view of programming and program execution. However, we believe that our ongoing research, based on variations and deviations of this theme, will lead the to development of similar facilities for vector- and parallel architectures, slowly on entering the market. In addition we — at least — indicated, that the already available tools do not cover the whole spectrum of instruments needed to automatize the process of solving a problem reliably.

4. Program Generation

The only tool for program generation, until recently provided by computer algebra systems such as REDUCE and MACSYMA, was the facility to switch from normal to FORTRAN-coded output. Hence to produce complete and executable FORTRAN programs directly from these systems was not possible. This left the user with the necessity to shorten output-expressions whenever required, to meet the limitations given by the 20-lines rule in FORTRAN, and to find a way (text editing or the use of write statements) to complete the FORTRAN program. The first packages to assist in this programming task are MACTRAN [69] and VAXTRAN [39]. MACTRAN, running under MACSYMA, allows to construct complete FORTRAN subroutines based on user-supplied template files. Such a file contains an outline of a FORTRAN program, the so-called passive parts of the file, and active parts, consisting of MACSYMA commands. MACTRAN processes such a file by simply copying the passive parts on the actual output file and by executing the active portions, which of course ought to result in meaningful arithmetic assignment statements in FORTRAN notation on the same output file. Hence the passive parts of such a template file define in fact the control structure of the FORTRAN program or subroutine which ought to be produced in this way. VAXTRAN, implemented in Franz LISP to run under VAXIMA, is similar to and based on MACTRAN. IN addition to MACTRAN it provides a more general interface between symbolic and numerical computing techniques. Although VAXTRAN compiles generated code from VAXIMA, using an augmented compiler, and interfaces the resulting compiled code to make it callable directly from VAXIMA, it still relies on the MACSYMA FORTRAN switch only. This might effectively limit its use to moderate problems.

More recently GENTRAN [68], a code GENeration and TRANslation package became available, originally implemented in Franz LISP to

run under VAXIMA. Although specifically created to generate RATFOR-subprograms for use with an existing FORTRAN-based finite element package [66], it has the flexibility required to handle most code generation applications. A second more recent version of GENTRAN is written in RLISP to run under REDUCE [24,25]. This version transforms REDUCE prefix forms into formatted FORTRAN, RATFOR or C code. GENTRAN does not only allow generation of arithmetic expressions or assignment statements, but also of control structures, subprogram headings and type declarations. A consequence of this is that template file processing, although possible in GENTRAN, is not longer required under all circumstances. This implies that a user can generate complete (sub)programs for numerical purposes through a series of interactive (REDUCE or MACSYMA) commands. Besides a variety of flexible file handling commands, also allowing recursively performed template file processing, GENTRAN has some additional facilities which are notably interesting for the generation of numerical code: automatic expression segmentation and suppression of simplification through the generation of temporary variables. The latter facility is for instance quite attractive, as we show below in Example 4.1, to produce efficient code for the determinant DM, introduced in Example 2.1.

GENTRAN provides very powerful tools for the construction of efficient programs for numerical purposes, certainly when combined with code optimization facilities, to be discussed in the next section. We therefore give a short survey of the essentials of GENTRAN and conclude this section with an illustrative example. GENTRAN, viewed as a REDUCE extension, contains code generation and file handling commands, mode switches and global variables, all of which are accessible from both the algebraic and symbolic mode of REDUCE. The algebraic mode is the normal user interface with the system, while the symbolic mode — in fact a LISP-like system level — is meant for system modification and extension. Hence when the package is loaded, REDUCE can be considered to be brought in a new state. All REDUCE commands preceded by the keyword GENTRAN are now processed according to the GENTRAN rules. After conversion of the command into REDUCE prefix form it is transformed into formatted FORTRAN, RATFOR or C code, depending on the value of the global variable GENTRANLANG!*. The whole transformation process is done in three stages: Between in pre- and postprocessing the translation phase is performed. During this phase either the prefix forms are translated into semantically equivalent code strings in the target language or an error message is generated. In addition subprogram headings, declarations and the like are produced. Hence, prior to translation, REDUCE evaluations have to be performed. They are actually done during the preprocessing phase. Although this strategy is similar to processing passive and active parts of template files, noteworthy differences exist.

The passive parts of a template file ought to consist of syntactically correct code strings in the target language. GENTRAN accepts translatable REDUCE commands. The active parts can be dealt with in different ways. Partly or full evaluation is under user control, either in algebraic or in symbolic mode, through some simple facilities.

For instance EVAL EXP, where EXP is any REDUCE expression or statement, causes EXP to be evaluated before translation takes place. So, assuming F stands for

$$2*x^2 - 5*x + 6$$

and GENTRANLANG!* has the value 'FORTRAN, the command

```
GENTRAN Q := EVAL(F) / EVAL(DF(F,X)) $
```

will result in

```
Q=(2*X**2-5*X+6)/(4*X-5)
```

GENTRAN also has three additional assignment operators, being :=:, ::= and ::=:. These operators are constructed out of the usual REDUCE assignment operator := by adding (an) extra colon(s). If the extra ":" is given on the left it means that the indices occurring in the matrix or array element of the left hand side have to be evaluated before translation is carried out. An extra colon to the right means that the right hand side has to be evaluated before translation into the target language is performed. So if M(2,2) := A and if M(3,3) := B then the command

```
FOR j := 2:3 DO GENTRAN M(j,j) ::=: j*M(j,j)$
```

will result in, again assuming FORTRAN is the target language,

```
M(2,2)=2*A
M(3,3)=3*B
```

During the translation phase prefix forms of those arithmetic statements which are longer than a specified length can be replaced by equivalent sequences, which assign subexpression values to temporary variables whose values are gradually combined. This is simply achieved by assigning values to the globals MAXEXPPRINTLEN!* and FORTLINELEN!* (or RAT- or CLINELEN!*) and by turning on the GENTRANSEG switch. This segmentation requires a facility to generate temporary variable names, which can be stored in a symbol table, like other names, to guarantee to obtain adequate declarations. A combination of these name generation facilities and the use of the special GENTRAN features for evaluation provides a powerful tool for effectively reducing simplification. Through the command VAR :=

TEMPVAR()$ a temporary variable name is assigned to VAR. The command MARKVAR VAR$ serves to further protect VAR for a too early reuse as temporary variable name. Thus it guarantees that the atom VAR can represent a significant value until further notice. Therefore the commands

```
VAR := TEMPVAR()$
MARKVAR VAR$
M(1,3) := VAR$
GENTRAN EVAL(VAR) := M(1,3)$
```

result in the REDUCE setting

```
M(1,3) = T0
```

and in the FORTRAN assignment

```
      T0=M(1,3)
```

assuming the value of VAR is T0. Observe that M(1,3) is assigned a new value in the REDUCE-context, while T0=M(1,3) is only an output string. If all matrix entries are treated similarly the code to be produced for the computation of a determinant or an inverse matrix can be made much more efficient. We show the effect of these nice facilities, applied on the matrix M, introduced in Section 2, in

Example 4.1.

```
 M(1,1)=-(9*SIN(Q3)**2*P**2*M30)-(SIN(Q3)**2*J30Y)+SIN(Q3)**2*
. J30Z+18*COS(Q3)*COS(Q2)*P**2*M30+18*P**2*M30+P**2*M10+J30Y+J1OY
 M(1,2)=-(9*SIN(Q3)**2*P**2*M30)-(SIN(Q3)**2*J30Y)+SIN(Q3)**2*
. J30Z+9*COS(Q3)*COS(Q2)*P**2*M30+9*P**2*M30+J30Y
 M(1,3)=-(9*SIN(Q3)*SIN(Q2)*P**2*M30)
 M(2,2)=-(9*SIN(Q3)**2*P**2*M30)-(SIN(Q3)**2*J30Y)+SIN(Q3)**2*
. J30Z+9*P**2*M30+J30Y
 M(2,3)=0
 M(3,3)=9*P**2*M30+J30X
 T0=M(1,1)
 T1=M(1,2)
 T2=M(1,3)
 T3=M(2,2)
 T4=M(3,3)
 DM=T0*T3*T4-(T1**2*T4)-(T2**2*T3)
```

The above given piece of FORTRAN code is the result of executing the following mixture of GENTRAN and REDUCE commands :

```
GENTRANLANG!* := 'FORTRAN $
FORTLINELEN!* := 70 $
GENTRANOUT "M.OUT" $

for j:=1:3 do
  for k:=j:3 do
    GENTRAN M(j,k) ::=: M(j,k) $

for j:=1:3 do
  for k:=j:3 do
    if M(j,k) neq 0 then
      << VAR := TEMPVAR() $
         MARKVAR VAR $
         M(j,k) := VAR $
         M(k,j) := VAR $
         GENTRAN EVAL(VAR) := M(EVAL(j),EVAL(k))
      >> $

GENTRAN DM :=: det(M) $

GENTRANSHUT "M.OUT"$
```

It is obvious that the possibility of generating temporary variables after having saved all relevant information — the real values of the matrix elements — can in principle lead to an enormous efficiency increase by computing — in fact only — the skeletal structure of the determinant. □

Further examples can for instance be found in [24]. Further details about GENTRAN are given in [22,23,24].

5. Code Optimization

The level of sophistication of computer algebra systems easily allows generation of output of a size which is far beyond human understanding. The examples in the previous sections show that structure displaying techniques eventually combined with compression methods, based on heuristics, can largely improve the compactness and comprehensibility of output expressions. But when producing sets of output expressions, which are going to form the arithmetic parts of programs for numerical purposes, tools for reducing the computational complexity of such sets can be attractive. It is however only relevant to consider reducing this complexity if the arithmetic is extensive or when the solution strategy requires repetitions of identical sequences of arithmetic operations. The programs resulting from attempts to solve such, computationally intensive, problems do not meet Knuth's conclusion that for an average FORTRAN program the extension of the compiler, with features for optimization of the arithmetic, is overdone [38]. This might explain why optimization of arithmetic code can not be qualified as a popular research area.

We recall that we do not concentrate on average programs. The elements of our sets of expressions are viewed as definitions of computational processes. Hence such a set can be seen as a block of straight

line code. The arithmetic complexity is defined as the number of elementary arithmetic operations required to obtain the results of these computations for a set of permissible inputs. This view can be refined by associating weights with the various elementary operations. These weights reflect computational costs. Attempts to optimize the description of such basic blocks can be considered as techniques for minimizing, or at least reducing, the arithmetic complexity of the given sets of expressions. However, a reduction is only possible when redundancy occurs. Redundancy is a needless form of repetition, i.e. the presence of common (sub)expressions. This is certainly true for the traditional sequential view on program execution. Other architectures demand for other notions, like "not sufficiently vectorized in a reasonable way" or "insufficiently decomposed in (sub)sets which can be processed in parallel without causing deadlock problems when combining the results to obtain the final answers". We only consider here the traditional sequential processes.

When attempting to minimize arithmetic, defined through expressions producible with a computer algebra system, we ought to know what the structure of these expressions can be before we are able to design methods allowing to discover eventually existing redundancy. As suggested in Section 2 a user can produce almost everything, efficient or not, of almost arbitrary size and depth of nesting. Therefore the design of algorithms for searching for common (sub)expressions (cse's for short) ought to be based on the presumption that the elements of the input set of which the description ought to be optimized, can have an arbitrary structure. As a consequence such algorithms ought to be designed to allow finding cse's of an equally arbitrary structure. The representation of cse's inside a set of expressions is certainly normal, if we presume the commutative, associative and distributive laws to hold. With the REDUCE Standard Quotient form for expressions in mind, we can describe expressions, what ever their structure might be, in a prefix notation, as pairs of the form (operator . list of operands). Here "operator" stands for PLUS, TIMES or "something else". PLUS and TIMES, denote the usual commutative operations of addition and multiplication, respectively. Hence any desirable permutation of the elements of the "lists of operands" will in principle be allowed, when the operator is PLUS or TIMES. The lists of operands are again formed by such expressions. When for a while excluding the "something else" alternative, the expressions are multivariate polynomials over $\mathbb{Z}$. Such a polynomial can be viewed as a sum (product) of primitive and/or composite terms (factors). We call a term primitive if it is an integer, a variable or an integer multiple of a variable. These primitives form together an (eventually empty) linear expression. Hence the composite terms are products. It depends on the ordering considerations of the algebra system, where the primitive and composite terms are located in the "list of operands". A primitive factor is a constant, a variable or a

power of a variable. Hence a product of primitive factors is simply a monomial. The composite factors are obviously sums. Every polynomial can be thought of as being built up by linear expressions and monomials only, what ever its (un)nested structure might be. When searching for cse's we — in principle — use these "primitive" information carriers, linear expressions and monomials. As soon as a new cse is found, it can be replaced by a new, system selected, variable name, assuming its description is added to the set of expressions. To obtain a correct basic block we ought to assume our set of expressions to be (partly) ordered. Every cse-description has to be inserted correctly in this sequence, i.e., before its first occurrence. When replacing a cse by a new variable it might happen that composite terms or factors collapse and become a primitive. Hence when basing the search for cse's on primitives the overall process becomes obviously iterative.

In contrast to the usual dag models for the representation of arithmetic expressions [1], we employ, following Breuer [6,61,62], sparse extendible matrices, albeit in a slightly more sophisticated way. We therefore have to (re)parse the internally stored list representations of the elements of our sets of expressions, multivariate polynomials, in more transparent and multi-accessible matrix structures. The columns of the matrices are associated with the variables and the rows with the (sub)expressions. Although merged in practice, we store the linear expressions and the monomials in separate matrices, which are interconnected via hierarchic information about the structure of the expressions, subjected to our cse-search. When a variable occurs in a linear (sub)expression its coefficient is stored as matrix entry. Similarly the exponent is stored when the variable occurs in a monomial. Therefore the validity of the commutative law ought to be presumed. To be able to retrieve the original structure of the expressions involved in the search, additional information about the hierarchy ought to be associated with the rows of the matrices. What kind of additional information is of importance? Of interest is a list of so-called children, i.e., a list of indices of rows where the descriptions of the composite terms or factors of the present row are stored. Also important is a name field, used to store the name associated with an expression or if we are dealing with a subexpression the index of the so-called father of this subexpression. We further mention an operator field and an ordered list of cse-indicators, allowing to obtain correct evaluation sequences, when translating the result of a cse-search into, for instance, FORTRAN code. The operator field is not only important for distinguishing between PLUS and TIMES, because we use merged structures internally, but also for effectively using the "something else" alternative. These "strange" operators are "removed" so as to get back to the multivariate polynomial scheme. We again distinguish between primitives and composites. When all elements in the "list of operands" are primitives, i.e., constants and/or variables, the pair (operator . list of

operands), a kernel in the REDUCE-setting, is replaced by a new variable, such that all identical primitive kernels share the same name. Kernels with (partly) composite operands are treated like sums. The operator field has a different value and the searches for identical operands are slightly different since commutativity is not longer valid. To avoid complicating our discussion we further neglect the "something else" alternative. Let us now try to visualize the data structures we employ temporarily to obtain an optimized version of a set of expressions via

Example 5.1. Let us assume to have as set of input expressions:

```
                                    4  6  5
E1 := (2*A + 4*B + 3*C)*A *C *D

                                    2  4  3
E2 := (4*A + 6*C + 5*D)*A *B *C
```

This set is — when oversimplifying reality — parsed and stored in the following way:

```
Sumscheme :

   !  A  B  C  D!Far
--------------------
  1!  2  4  3   ! 0
  3!  4     6  5! 2
--------------------

Productscheme :

   !  A  C  D  B!Far
--------------------
  0!  4  6  5   ! E1
  2!  2  3     4! E2
--------------------
```

More detailed examples are given in [62]. A cse-search will result in:

```
S0 := 2*A + 3*C

        2  3
S1 := A *C

         2  5
E1 := S1 *D *(S0 + 4*B)

           4
E2 := S1*B *(2*S0 + 5*D)
```

This set is reconstructed from the matrices, resulting from the cse-search:

```
Sumscheme :

   ! S0  A  B  C  D!Far
-----------------------
  1!  1     4     ! 0
  3!  2           5! 2
  4!     2     3   ! S0
-----------------------

Productscheme :

   ! S1  A  C  D  B!Far
-----------------------
  0!  2        5   ! E1
  2!  1           4! E2
  5!     2  3      ! S1
-----------------------
```

So initially cse's are either linear expressions or monomials. To discover them the integer matrices are heuristically searched for submatrices of rank 1 of maximal size. A basic scan is used, which can be qualified as "test whether the determinant of a (2,2)-matrix of non-zero entries is zero". Its use is based on information about row weights, which allow to locate completely dense submatrices. The row-weight is a reflection of the arithmetic complexity of the primitive defined by that row. Further details are given in [61,62]. Since we want to reduce the arithmetic complexity, say the pair AC = (np,nm), a cse-detection ought to contribute to a reduction of the number of additions (np) and/or the number of multiplications (nm). This is only possible if a cse occurs at least twice and contains at least one addition and/or multiplication. Other less detailed criteria are conceivable. Another category of cse's is formed by repeatedly occurring constant multiples of variables and by single powers, delivering addition chain problems. This category can be enlarged during the optimization process. This can be illustrated by

Example 5.2. The result shown in Example 5.1 is in fact intermediate. The real result, given by the present version of the Optimizer, is:

```
Number of operations in the input is:

Number of (+,-)-operations : 4
Number of (*)-operations : 12
Number of integer exponentiations : 6
Number of other operations : 0

S0 := 2*A + 3*C
S3 := A*A
S7 := C*C
S4 := C*S7
S1 := S3*S4
S2 := S1*S1
S8 := D*D
S7 := S8*S8
S5 := D*S7
E1 := S2*S5*(S0 + 4*B)
S7 := B*B
S6 := S7*S7
E2 := S1*S6*(2*S0 + 5*D)

Number of operations after optimization is:

Number of (+,-)-operations : 3
Number of (*)-operations : 19
Number of integer exponentiations : 0
Number of other operations : 0
```

First a repeated search for cse's with at least two operands is performed. Then the optimization is completed with a finishing touch.

The first step consists of four subsearches:

1— Application of the commutative law when looking for linear (sub)sums and (sub)monomials, respectively. The strategy is based on an extension of Breuer's grow factor algorithm. Cse's are replaced by new names and their description is added to the matrix, implying that composite operands can be reduced to (new) primitives.

2— A kernel search, to discover if composite kernels can now be viewed as primitives, followed by update operations.

3— Merging activities based on the assumption that composites are possibly reducible to primitives, i.e., a composite factor, defined in the sum matrix and reduced to a primitive, can be migrated, in its new form, to the product scheme and visa versa.

4— Application of the distributive law, i.e., replacement of an expression like a*b + a*c by a*(b + c) by adequate information migration.

Although the basic scans are always performed on primitive structures the cse's can have an arbitrary complex structure, because information is continuously migrated through the matrices.

The finishing touch consists of factoring out contents of integer coefficients in sums, detection of repeatedly occurring integer multiples of variables and addition chain operations so as to replace all exponentiations by multiplication sequences. This phase is characterized by one

row (or one column) operations, in contrast to the first, where mainly completely dense submatrices are examined.

Example 5.3. The determinant *DM* of the matrix *M*, introduced in Section 2, can be computed quite efficiently, when using the possibility of introducing temporary variables, the t_i's, via simple GENTRAN-commands. This is even more striking when the inverse matrix is required; see [24]. However we can further reduce the arithmetic complexity by optimizing the set of expressions formed by the different entries of *M*, leading to:

```
Number of operations in the input(T0,T1,T2,T3 and T4) is:

Number of (+,-)-operations : 17
Number of (*)-operations : 29
Number of integer exponentiations : 13
Number of other operations : 9

S0 := SIN(Q3)
S8 := S0*S0
S1 := COS(Q3)
S2 := COS(Q2)
S7 := P*P
S5 := S7*M30
S4 := S5*S1*S2
S6 :=  - J30Y + J30Z
S13 := 9*S5
S10 :=  - S13 + S6
S11 := S10*S8
T0 := S11 + 18*S4 + J30Y + J1OY + S7*(18*M30 + M10)
S9 := S13 + J30Y
T3 := S11 + S9
T1 := T3 + 9*S4
S3 := SIN(Q2)
T2 :=  - S13*S0*S3
T4 := S13 + J30X


Number of operations after optimization is:

Number of (+,-)-operations : 11
Number of (*)-operations : 13
Number of integer exponentiations : 0
Number of other operations : 4
```

Optimization of the various forms of *DM*, introduced earlier, leads to different results, as shown in Table 5.1. Observe that the arithmetic complexity of the optimized version of the expanded form of *DM* is compatible with the arithmetic complexity of the not optimized version of the unexpanded form of *DM*. It is obvious that a slight increase of the size of *M*, without increasing the complexity of its entries, will result in more drastic differences. □

As stated before expression size can impose problems. Our Optimizer allows to handle extensive input piece wise, but such that the results of previous optimization activities are taken into account on user request. This offers a possibility to handle expressions via partitioning. The user interface is simple. Only a few commands are needed, in combination with a number of mode switches and flag settings to influence output-notation or to obtain additional information

Form of *DM*	Status	Number of operations			
		+,−	*	↑	other
expanded	input	36	82	30	6
	output	27	51	-	4
compressed, after expansion	input	35	63	23	6
	output	27	42	-	4
unexpanded	input	24	40	21	10
	output	13	19	-	4

Table 5.1.

about the optimization process. Details will be given in [64].

6. Some Conclusions

Ideally, as said before, the symbolic-numeric interface ought to provide user friendly facilities to allow to produce efficient and reliable numerical programs in a natural way. This requires a further integration of program generation and code optimization facilities and an extension of optimization techniques as to be able to optimize structured programs in stead of only local blocks of straight line code. A priori error analysis for such programs is an additional need and more far reaching than the present possibilities. It would also be of tremendous help if the Optimizer could be extended with a module which allows to discover automatically patterns in the problem formulation, which, for instance, are due to symmetries. This is certainly useful if extensive differentiation or integration of the code, to be optimized, is an additional need. We expect to witness such extensions during the coming years. In fact, our work in progress covers some of these items. We, not only work on variations related to program construction for vector and parallel architectures, but also on improvements of the symbolic-numeric interface and certain aspects of simplification. Worth mentioning are :

- Bottom-up structure recognition facilities [20], to be used to develop methods, which allow to discover symmetries, for instance. Such algorithms can also be helpful in improving differentiation procedures.
- A combined use of unification and simplification. We created already an environment to perform experiments in a REDUCE context [50].
- Improvements of the symbolic-numeric interface by investigating classes of problems, which can largely profit from such facilities for their solution. The design and implementation of programs for user friendly generation of Jacobians and Hessians, which ought to allow to simply connect their output with NAG library routines, learned that a further integration of a package like

GENTRAN with our Optimizer is not too complicated and certainly most profitable [59].

- Thus far we limited the Optimizer activities to expressions with integer coefficients and exponents. But expressions over other domains are conceivable [4], implying that an extension of the Optimizer ought to be considered.

Acknowledgements. Over the past years I had many valuable and pleasant discussions about the above indicated aspects of computer algebra. I like to mention especially Johan de Boer, Barbara Gates, Victor Goldman, Tony Hearn, Ben Hulshof, Arthur Postmus, Jaap Smit, Pim van den Heuvel and Paul Wang. The continued interest of Leo Verbeek has been most stimulating.

References

1. A.V. Aho, R. Sethi & J.D. Ullman: *Compilers — Principles, Techniques, and Tools* (1986), Addison-Wesley, Reading, Mass.
2. A.T. Balaban: Symbolic computation in chemistry, *in:* B. Buchberger (Ed.): *Proceedings EUROCAL'85*, Vol. 1, Lect. Notes Comp. Sci. **203** (1985) 68-79, Springer-Verlag, Berlin - Heidelberg - New York.
3. A.M. Bos & M.L.J. Tiernego: Formula manipulation in the bond graph modeling of large mechanical systems, *J. Franklin Inst.* **319** (1985) 51-65.
4. R.J. Bradford, A.C. Hearn, J.A. Padget & E. Schrüfer: Enlarging the REDUCE domain of computation, *in:* B.W. Char (Ed.): *Proceedings SYMSAC '86* (1986) 100-106, Assoc. Comput. Mach., New York.
5. R. Brent: A FORTRAN multiple-precision arithmetic package (1976), Computer Centre, Australian National University, Canberra, Australia.
6. M.A. Breuer: Generation of optimal code for expression via factorization, *Comm. Assoc. Comput. Mach.* **12** (1969) 333-340.
7. W.S. Brown: On computing with factored rational expressions, *ACM SIGSAM Bulletin* **31** (1974) 27-34.
8. W.S. Brown & A.C. Hearn: Application of symbolic mathematical computations, *Comput. Phys. Comm.* **17** (1979) 207-215.
9. B. Buchberger: Gröbner bases: an algorithmic method in polynomial ideal theory, *in:* N.K. Bose (Ed.): *Recent Trends in Multidimensional Systems Theory*, Chapter 6 (1985), Reidel, Dordrecht.
10. B. Buchberger: History and basic features of the critical-pair/completion procedure, *J. Symbolic Comput.* **3** (1987) 3-38.

11. B. Buchberger, G.E. Collins & R. Loos (Eds.): *Computer Algebra, Computing* Supplementum **4** (1982), Springer-Verlag, Wien.

12. B. Buchberger & R. Loos: Algebraic simplification, in [11] (1982) 11-44.

13. J. Calmet & J.A. van Hulzen: Computer algebra applications, in [11] (1982) 245-258.

14. B.F. Caviness: On canonical forms and simplification, *J. Assoc. Comput. Mach.* **17** (1970) 385-396.

15. B.F. Caviness: Computer algebra: past and future, *J. Symbolic Comput.* **2** (1986) 217-236.

16. B.W. Char, G.J. Fee, K.O. Geddes, G.H. Gonnet & M.B. Monogan: A tutorial introduction to MAPLE, *J. Symbolic Comput.* **2** (1986) 179-200.

17. G.O. Cook, Jr.: Development of a Magnetohydrodynamic Code for Axisymmetric High-β Plasmas with Complex Magnetic Fields, Ph.D. Thesis (1982), Lawrence Livermore Nat. Lab., Cal.

18. A.A.M. Cuyt & L.B. Rall: Computational implementation of the multivariate Halley method for solving nonlinear systems of equations, *ACM Trans. Math. Software* **11** (1985) 20-36.

19. J. Davenport, Y. Siret & E. Tournier: *Calcul Formel* (1987), Masson, Paris.

20. J.K. de Boer: Bottom-up Structure Recognition, MA Thesis (1986), University of Twente, Enschede, The Netherlands.

21. J. Fitch: Solving algebraic problems with REDUCE, *J. Symbolic Comput.* **1** (1985) 211-228.

22. B.L. Gates: GENTRAN user's manual — REDUCE version, Memorandum INF-85-11 (1985), University of Twente, Enschede, The Netherlands.

23. B.L. Gates: GENTRAN design and implementation — REDUCE version, Memorandum INF-85-12 (1985), University of Twente, Enschede, The Netherlands.

24. B.L. Gates: GENTRAN: An automatic code generation facility for REDUCE, *ACM SIGSAM Bulletin* **75** (1985) 24-42.

25. B.L. Gates: A numerical code generation facility for REDUCE, *in:* B.W. Char (Ed.): *Proceedings SYMSAC '86*, (1986) 94-99, Assoc. Comput. Mach., New York.

26. A.C. Hearn: Private communication.

27. A.C. Hearn: REDUCE 2: a system and language for algebraic manipulation, *in:* S.R. Petrick (Ed.): *Proceedings SYMSAM* 2 (1971) 128-133, Assoc. Comput. Mach., New York.

28. A.C. Hearn: The structure of algebraic computations, *in:* A. Visconti (Ed.): *Proceedings of the Fourth Colloquium on*

Advanced Methods in Theoretical Physics (1977) 1-15, St. Maximin, France.

29. A.C. Hearn: Structure: the key to improved algebraic computation, *in: Proceedings RSYMSAC* (1984), Wako-shi, Saitama, Japan.
30. A.C. Hearn: Optimal evaluation of algebraic expressions, *in:* J. Calmet (Ed.): *Proceedings AAECC-3*, Lect. Notes Comp. Sci. **229** (1986) 392-403, Springer-Verlag, Berlin - Heidelberg - New York.
31. E. Horowitz & S. Sahni: On computing the exact determinant of matrices with polynomial entries, *J. Assoc. Comput. Mach.* **22** (1975) 38-50.
32. B.J.A. Hulshof & J.A. van Hulzen: Automatic error cumulation control, *in:* J. Fitch (Ed.): *Proceedings EUROSAM '84* , Lect. Notes Comp. Sci. **174** (1984) 260-271, Springer-Verlag, Berlin - Heidelberg - New York..
33. B.J.A. Hulshof & J.A. van Hulzen: An expression compression package for REDUCE. In preparation.
34. R.D. Jenks: A Primer: 11 keys to new SCRATCHPAD, *in:* J. Fitch (Ed.): *Proceedings EUROSAM '84*, Lect. Notes Comp. Sci. **174** (1984) 123-147, Springer-Verlag, Berlin - Heidelberg - New York.
35. H.G. Kahrimanian: Analytic Differentiation by Computer, MA Thesis (1953), Temple University, Philadelphia, Pa.
36. Y. Kanada & T. Sasaki: LISP based "big float" system is not slow, *ACM SIGSAM Bulletin* **58** (1981) 13-19.
37. G. Kedem: Automatic differentiation of computer programs, *ACM Trans. Math. Software* **6** (1980) 150-165.
38. D.E. Knuth: An empirical study of FORTRAN programs, *Software Practice and Experience* **1** (1970) 105-133.
39. D.H. Lanam: An algebraic front-end for the production and use of numeric programs, *in:* P.S. Wang (Ed.): *Proceedings SYMSAC '81*, (1981) 223-227, Assoc. Comput. Mach., New York.
40. M.A.H. MacCallum: Algebraic computing in relativity, Report TAU 86-04 (1986), Queen Mary College, University of London.
41. J. Marti, A.C. Hearn, M.L. Griss & C. Griss: Standard LISP report, *ACM SIGSAM Bulletin* **53** (1980) 23-43.
42. J. Moses: Algebraic simplification: a guide for the perplexed, *Comm. Assoc. Comput. Mach.* **14** (1971)548-560.
43. E.W. Ng: Symbolic-numeric interface: a review, *in:* E.W. Ng (Ed.): *Proceedings EUROSAM '79*, Lect. Notes Comp. Sci. **72** (1979) 330-345, Springer-Verlag, Berlin - Heidelberg - New York.
44. J. Nolan: Analytic Differentiation on a Digital Computer, MA Thesis (1953), Math. Dept., M.I.T., Cambridge, Mass.

45. A.C. Norman: Integration in finite terms, in [11] (1982) 57-70.

46. A.C. Norman: Computing in transcendental extensions, in [11] (1982) 169-172.

47. R. Pavelle, M. Rothstein & J. Fitch: Computer algebra, *Scientific American* (1981) 102-113.

48. R. Pavelle & P.S. Wang: MACSYMA from F to G, *J. Symbolic Comput.* **1** (1985) 69-100.

49. K.M. Pitman: A FORTRAN → LISP translator, *in:* V.E. Lewis (Ed.): *Proceedings 1979 MACSYMA User's Conference* (1979) 200-214, M.I.T., Cambridge, Mass.

50. A.G. Postmus: Design of a HyperLisp Interpreter with Prolog Features, MA Thesis (1987), University Twente, Enschede, The Netherlands.

51. L.B. Rall: *Automatic Differentiation: Techniques and Applications*, Lect. Notes Comp. Sci.**120** (1981), Springer-Verlag, Berlin - Heidelberg - New York.

52. L.B. Rall: Differentiation in PASCAL-SC: type GRADIENT, *ACM Trans. Math. Software* **10** (1984) 161-184.

53. A. Rich & D.R. Stoutemyer: Capabilities of the muMATH-79 computer algebra system for the INTEL-8080 microprocessor, *in:* E.W. Ng (Ed.): *Proceedings EUROSAM '79*, Lect. Notes Comp. Sci. **72** (1979) 241-248, Springer-Verlag, Berlin - Heidelberg - New York.

54. T. Sasaki: An arbitrary precision real arithmetic package in REDUCE, *in:* E.W. Ng (Ed.): *Proceedings EUROSAM '79* , Lect. Notes Comp. Sci. **72** (1979) 358-368, Springer-Verlag, Berlin - Heidelberg - New York.

55. J. Smit & J.A. van Hulzen: Symbolic-numeric methods in microwave technology, *in*: J. Calmet (Ed.): *Proceedings EUROCAM '82*, Lect. Notes Comp. Sci. **162** (1982) 281-288, Springer-Verlag, Berlin - Heidelberg - New York.

56. G.L. Steele, Jr.: Fast arithmetic in MACLISP, *Proceedings 1977 MACSYMA User's Conference* (1977) 215-224, NASA-CP2012.

57. S. Steinberg & P. Roache: Using VAXIMA to write FORTRAN code, *in:* V.E. Golden (Ed.): *Proceedings 1984 MACSYMA User's Conference* (1984) 1-22, Gen. Elect. Schenectady, New York.

58. D.R. Stoutemyer: A preview of the next IBM-PC version of muMath, *in:* B. Buchberger (Ed.): *Proceedings EUROCAL '85*, Vol. 1, Lect. Notes Comp. Sci. **203** (1985) 33-44, Springer-Verlag, Berlin - Heidelberg - New York.

59. P. van den Heuvel: Aspects of Program Generation Related to Automatic Differentiation, MA Thesis (1986), University Twente, Enschede, The Netherlands.

60. P. van den Heuvel, J.A. van Hulzen & V.V. Goldman: Automatic generation of FORTRAN-coded Jacobians and Hessians, *in:* J.H. Davenport (Ed.): *Proceedings EUROCAL '87*, To appear.

61. J.A. van Hulzen: Breuer's grow factor algorithm in computer algebra, *in:* P.S. Wang (Ed.): *Proceedings SYMSAC '81* (1981) 100-104, Assoc. Comput. Mach., New York.

62. J.A. van Hulzen: Code optimization of multivariate polynomial schemes: a pragmatic approach, *in:* J.A. van Hulzen (ed.): *Proceedings EUROCAL '83*, Lect. Notes Comp. Sci. **162** (1983) 286-300, Springer-Verlag, Berlin - Heidelberg - New York.

63. J.A. van Hulzen & J. Calmet: Computer algebra systems, in [11] (1982) 221-243.

64. J.A. van Hulzen & B.J.A. Hulshof: A code optimization package for REDUCE. In preparation.

65. P.S. Wang: Taking advantage of symmetry in the automatic generation of numerical programs for finite element analysis, *in:* B.F. Caviness (Ed.): *Proceedings EUROCAL '85, Vol. 2*, Lect. Notes Comp. Sci. **204** (1985) 572-582, Springer-Verlag, Berlin - Heidelberg - New York.

66. P.S. Wang: FINGER: a symbolic system for automatic generation of numerical programs in finite element analysis, *J. Symbolic Comput.* **2** (1986) 305-316.

67. P.S. Wang, T.Y.P. Chang & J.A. van Hulzen: Code generation and optimization for finite element Analysis, *in:* J. Fitch (Ed.): *Proceeding EUROSAM '84*, Lect. Notes Comp. Sci. **174** (1984) 237-247, Springer-Verlag, Berlin - Heidelberg - New York.

68. P.S. Wang & B.L. Gates: A LISP-based RATFOR code generator, *in:* V.E. Golden (Ed.): *Proceedings 1984 MACSYMA User's Conference*, (1984) 319-329, Gen. Electr., Schenectady, New York.

69. M.C. Wirth: On the Automation of Computational Physics, Ph.D. Thesis (1980), Lawrence Livermore Nat. Lab., Ca.

70. D.Y.Y. Yun & D.R. Stoutemyer: Symbolic mathematical computation, *in:* J. Belzer, et al. (Eds.): *Encyclopedia of Computer Science and Technology*, Vol. **15** (1980) 235-310, Marcel Dekker, New York - Basel.

Non-Monotonic Reasoning in Man and Machine

Edward Hoenkamp

Psychological Laboratory, University of Nijmegen
Montessorilaan 3, Nijmegen, The Netherlands

In traditional systems of deductive logic adding an axiom gives rise to new theorems, i.e., the set of theorems grows monotonically with the set of axioms. The study of non-monotonic reasoning started with the recognition that, in contrast, people often retract earlier conclusions on the basis of new input. This intuition encouraged the exploration of different extensions to logic. The present paper evolves in three steps: (1) It reviews extensions to monotonic logics, (2) It brings to bear psychological findings showing that the extensions fail where our intuition falls short: Inferences are often remarkably unresponsive to new input even if the original basis for the inferences is discredited, and (3) It presents a model that accounts for this more accurate view of human retraction behavior.

1. Introduction

Life is potentially full of surprises. If we want to act at all, we have to do so on the basis of often incomplete, vague, inconsistent or corrupted information. Yet, compared to the quality of the information, people fare remarkably well under such circumstances. Two important mechanisms enable us to do so. One is the use of *defaults*, i.e., knowledge of what is usually or typically the case; the other is *belief revision*, i.e., the retraction of earlier conclusions to reflect perceived changes in the environment or acquisition of new information. Modeling this aspect of common-sense reasoning has been a challenge to Artificial Intelligence (AI) since its early days. So far two main avenues have been pursued to cope with this challenge: a formal one, trying to extend the logic, and a technical one, attempting to endow computer programs with methods to revise their databases. These topics will be treated first. Then this paper opens a third avenue, shedding light on the issue by looking at how people actually behave when confronted with situations where belief revision is called for. To accomplish this, well-documented psychological experiments are modeled using existing AI methods. It will be apparent that both psychology and AI may benefit from this approach.

Part A: A Logical Excursion

2. Ignoring the Impossible vs. Ignoring the Possible.

In making decisions under mundane circumstances, there are simply too many things to consider that might be relevant. At the same time it is hardly possible to foresee even a fraction of the consequences of these decisions. (To be sure, considering the not-so-obvious may make a scientist successful, but here we must realize how rare these happy moments are.) Indeed, people are very good at ignoring possibilities. In contrast, people are well aware of impossibilities, and reject these with facility. The latter has its analogue in artificial intelligence, where techniques for rejecting impossibilities are widespread. Depth-first search is a very systematic way of doing this. Others are alpha-beta pruning and unification. Techniques for ignoring possibilities are much harder to come by (and most are even covered in this paper). One case is the technique of parsimonious reasoning known as *circumscription*. It has been observed (e.g. [19]) that when trying to solve a problem, one assumes that only the relationships mentioned are relevant. For example in solving the missionaries and cannibals problem, one usually does not discuss the presence of oars, absence of bridges, leakages etc. McCarthy [19] formulated a second order inference rule that produces axiom schemata from sets of (first order) axioms. It can be shown that the models for the new set of axioms are minimal in those for the original one. Informally speaking, they contain as few objects as possible (e.g., no bridges or planes).

Circumscription has gone through various incarnations (predicate, joint, formula, prioritized, and point-wise circumscription; see e.g., [19,20,18]), but a simple example may show the gist of this technique. Let A be a set of axioms stated in a language containing a predicate P. The circumscription schema for $P(x)$ in A is:

$$[A(Q) \,\&\, (x)[Q(x) \rightarrow P(x)]] \rightarrow (x)[P(x) \rightarrow Q(x)].$$

As an example, suppose we receive a postcard from Italy showing a leaning tower. Of course we jump to the conclusion that this is a picture of *the* Leaning Tower. Let us summarize the appearance of the Leaning Tower (say of an old, leaning tower) with the axiom:

$$\mathit{old\text{-}and\text{-}leaning}(\mathit{Leaning\text{-}Tower}).$$

Now, circumscribing this predicate in the set of axioms (in this case there is only one element) yields:

$$[Q(\mathit{Leaning\text{-}Tower}) \,\&\, (x)[Q(x) \rightarrow \mathit{old\text{-}and\text{-}leaning}(x)]] \rightarrow (x)[\mathit{old\text{-}and\text{-}leaning}(x) \rightarrow Q(x)].$$

Now we substitute the only known instance of leaning towers we have by taking $Q(x)$ to be "$x = \mathit{Leaning\text{-}Tower}$". One can easily verify that the first and second conjunct become true, so that by modus ponens we infer:

$$(x)[old\text{-}and\text{-}leaning(x) \rightarrow x = Leaning\text{-}Tower];$$

i.e., every old and leaning tower we see we will recognize as the Leaning Tower; the desired result. If later we discover that we saw a picture of a maquette, the conclusion is no longer true, and indeed it cannot be derived anymore. Although circumscription has been categorized as a fairly successful approach to non-monotonic reasoning, this categorization may be disputed on several grounds. First, as in the example above, it does not allow one to retract a conclusion constructively once it has been drawn. Second, Doyle [9] shows that the intent of circumscription is the same as that of "implicit definability" from monotonic logic. Third, circumscription may generate minimal models subsuming implausible ones ([13], I will come back to this). In short, it clearly formalizes one aspect of common-sense reasoning, namely that of jumping to conclusions. It may not, however grasp the intuitive idea of non-monotonic reasoning.

3. Non-Monotonic Theories

3.1. Why Non-Monotonic Theories?

The paradigmatic (and unfortunately somewhat overworked) example of non-monotonic reasoning begins with one axiom, namely that birds can fly. If told that Tweety is a bird, we may infer that Tweety can fly. Formalized:

$$(x)\ Bird(x) \rightarrow CanFly(x) \tag{1}$$

$$Bird(Tweety) \tag{2}$$

so by universal specialization and modus ponens:

$$CanFly(Tweety) \tag{3}$$

If, however, we subsequently learn that Tweety is an ostrich we have to retract our earlier conclusion. It seems that (1) should be replaced by (1'):

$$(x)\ Bird(x) \ \&\ \neg Ostrich(x) \rightarrow CanFly(x) \tag{1'}$$

to avoid drawing conclusion (3). Why then, do we need a non-monotonic formalism in the first place? The point is that (1') would make it impossible to infer (3) right from the start, which is contrary to what we actually did. So instead of questioning the axioms (not to mention that (1') should be amended for penguins, emus, kiwis etc.), one could reflect on the inference rules. In fact there are more options that have been explored, falling broadly in two categories: extensions to the logic, and meta-devices.

3.2. Extensions To First Order Logics

One way to extend the logic has been to extend the components of the theory with default rules. *Default reasoning* is a way of drawing conclusions in the absence of certain data, by virtue of these data being absent. In this way, (3) can be inferred by default. To take another example: as long as no-one has come to tell you that your car has been stolen, you will confidently walk to the parking lot after your work. A *default rule* is typically of the form:

$$\bullet \textit{ If } A, \textit{ and it is not known that } B, \textit{ then conclude } C. \qquad \text{(d1)}$$

For an example of the formalization of such a rule refer to [27], which gives a complete proof theory for a large class of generally occurring defaults.

A common instance of (d1), where C is $\neg B$, is the rule "If it is not known that B, then conclude $\neg B$". This is known as the *closed-world* assumption. It says that relationships not explicit in the database do not hold. But what exactly do we mean by "it is not known that B"? Suppose (d1) is used by a database manager. A database most often encodes only a fraction of its knowledge explicitly, for much data can easily be computed on demand (or is rarely used). A rule to capture this could be:

$$\bullet \textit{ If } A, \textit{ and it cannot be proved that } B, \textit{ then conclude } C. \text{(d2)}$$

This rule subsumes a mirror-image of the closed-world assumption, namely where B is $\neg C$. This is an important special case which has been studied by McDermott and Doyle [22]. They extend a first order (monotonic) logic by introducing a "non-monotonic modality" **M** in the language, and add the rule "infer **M***p* from the inability to infer $\neg p$." For example (1) would be replaced by

$$(x)\, \mathit{Bird}(x) \,\&\, \mathbf{M}\ \mathit{CanFly}(x) \rightarrow \mathit{CanFly}(x) \qquad (1'')$$

McDermott and Doyle prove that the non-monotonic predicate calculus thus obtained is complete and that the non-monotonic sentential calculus is decidable. The modal systems T, S4 and S5 can also be turned into non-monotonic theories by interpreting their modal M ("possible") as **M** above.[1] For proof procedures, semantics, and completeness results see [21].

These desirable formal properties aside, let me mention a commonality of these and other extensions: first, the definitions for "provable" and "consistent" are not constructive [5] and second, non-monotonic modals (such as **M** above) may not completely capture their intended meaning [24]. To finesse, in McDermott's approach **M** is part of the language, whereas in Reiter's approach it is a useful marker in

[1] The sharp-eyed reader will notice that the rule for **M** is ill-formed, but see [21].

non-monotonic inference rules.

3.3. Meta-Devices

Above we applied rule (d1) to the example where you were about to head for your car after work. Now try to apply rule (d2) instead. It could mean that you stay in your office pondering the myriad of circumstances that may have conspired for the car not to be where you left it. Obviously, when people engage in deliberation they stop after a limited amount of time. In the case of a machine using predicate calculus the rule is definitely useless, since this system is undecidable. For real applications, as for people, the following rule therefore seems more plausible: "If A cannot be proved within some allocated amount of time, then conclude B." A generalization of this rule has been implemented in the KRL system [4,35] as:

- *If all resources are exhausted before A is proved, assume A is false.*

The rule applies when a number of inference processes occur simultaneously (e.g., as in human perception [26]). With this rule we have come to a point where one moves outside the logical theory. Thus it may become impossible to make formal statements about the behavior of such a system. Fortunately there are ways to introduce a meta-device without loss of rigor. A classic example can be found in the work of Weyhrauch [34], who provides a general framework in which non-monotonic inference rules are part of a first-order meta-theory having the object theory as its domain. A third meta-system I want to mention actually comprises a whole class of so-called *reason maintenance systems*, which will be studied in the next section.

3.4. Reason Maintenance Systems

Suppose we add the following axiom to (1) and (2):

$$(x)\ CanFly(x) \rightarrow HasWings(x) \tag{4}$$

so that in addition to (3) also

$$HasWings(Tweety) \tag{5}$$

is inferred. If after this it is stated that Tweety is an ostrich, not only (3) must be retracted, but also everything that depends on (3), in this case (5). But an ostrich has wings after all, say because:

$$(x)\ Bird(x) \rightarrow HasWings(x) \tag{6}$$

so that (5) must not be retracted. We may want to replace (4) by a non-monotonic rule since, although planes have wings too, Superman does not. It will be clear from the examples that in a more serious application most of the time would be spent propagating changes through the database. This is particularly the case for artificial

intelligence systems, where most problem solvers are forced to reason with inconsistent or incomplete information. How, then, can this problem be solved?

In the example above, the difficulty was to know which conclusions were affected in order to know which to retract. We had to retrace the inferences mentally. This would have been much easier had we remembered for each conclusion the inferences on which it was based. This observation is half of the solution that a reason maintenance system presents. Fikes [11] kept track of derivations in the STRIPS system, and Stallman and Sussman [30] used a similar technique for electronic circuit analysis. A full solution was first proposed by Doyle [6] with his TMS (for Truth Maintenance System). TMS is used in relation to a database, to which it is invisible. Every assertion entering the data base is represented by a node. A record is kept of the dependence of nodes on inferential steps, i.e., the *justifications* of a node. This way the inference steps can be retraced to maintain consistency in a system. An assertion that is believed is called IN.

An assertion that depends on the fact that another assertion is not believed (i.e., is OUT), is called an *assumption*. For example, one assumes that a bird flies as long as there is no reason to believe that it is an ostrich or a penguin (a so-called OUT-*justifier*). A justification is valid if all its IN-justifiers are IN and all its OUT-justifiers are OUT. A node is IN if it has at least one valid justification. Now it is easily seen how a new assertion will affect the network. Believing an assertion that was not believed before means that the node representing it will come IN. Every node that has this node as OUT-justifier will now go OUT, propagating onwards. (Later in this paper I will give an elaborate concrete example.) This so called data-dependency backtracking takes place on the initiative of the database manager, and the same component is responsible for deciding what is inconsistent.

Two classes of reason maintenance systems can be distinguished in the literature, depending on what is recorded during the inference process. The justification-based systems, of which TMS is an example, record with each assertion the assertions that directly originated it. Assumption-based systems (e.g., de Kleer's ATMS, [16]) record the hypotheses (i.e., the non-derived assertions) that ultimately originated it. The latter systems have many advantages over the former in terms of efficiency and they have been implemented in some advanced expert systems. TMS in its pure form has the advantage that the database manager can read off the dependencies to explain to the user why an assertion is believed. The two techniques have been combined to produce a control strategy that is more efficient than either one alone [17].

3.5. Reasoned Assumptions

What has been said so far about reason maintenance and non-monotonic logic can be unified nicely using the concept of a *reason*. The reason for believing C in the presence of A and in the absence of B will be denoted as: $A \parallel B \mid\!\!- C$; cf. [7]. This is called a simple reason, where A, B and C can be interpreted as sets of potential beliefs. In a reason maintenance system A and B represent the IN- and OUT-justifiers for C. In a non-monotonic logic, a reason represents the default reason if we take B to be of the form $\neg C$ (the set containing the negations of the elements in C). There are two other special cases worth mentioning: If B is empty ($\emptyset$), the inference from A to C is like an ordinary implication. If B has the form "*Defeated* (R)", the reason R is called *defeasible*, i.e., use of the reason has been ruled out. Now, given a set of reasons, what collection of beliefs do we want to admit as justified by these reasons? In the monotonic case (all of the B's are empty) the deductive closure would be appropriate. Indeed, for the non-monotonic case a closure can be defined analogously. An example may illustrate what will happen. Suppose we adopt the reasons that, unless there is evidence to the contrary, Quakers are pacifists, and Republicans are not. I.e.,

$$Quaker(x) \parallel \neg Pacifist(x) \mid\!\!- Pacifist(x)$$

$$Republican(x) \parallel Pacifist(x) \mid\!\!- \neg Pacifist(x)$$

(omitting the curly set brackets for the moment). Let us further instantiate one individual N (e.g., Nixon), who is both a Republican and a Quaker. On the basis of these axioms two sets can be derived:

$$\{Quaker(N), Republican(N), Pacifist(N)\}$$

$$\{Quaker(N), Republican(N), \neg Pacifist(N)\}.$$

This shows that the closure in this case has two fixed points; they are called the *admissible extensions* of the reasons. It is important to realize that there is no good reason to prefer one over the other. No logical one, that is. In reality people will have a preference, but that will be on psychological grounds. With this observation we have arrived at an appropriate point to stop discussing the logical approaches to non-monotonic reasoning and start studying how people actually behave in what theoretically may be a choice situation. The source for this study is a set of psychological experiments.

Part B: A Psychological Excursion

4. A Case for Non-Monotonic Reasoning in Humans: "Debriefing" after Deception Experiments

A speaker at a conference may be heartened afterwards by someone from the audience who congratulates him for his interesting and clear exposition. If later he finds that this person mistook him for a potential referee, his self-esteem may decline again somewhat but it will probably not sink all the way back to its original level. In contrast to the AI examples of belief revision, people are often reluctant to adjust their opinion after the original evidence is discredited. This phenomenon has received special attention in connection with psychological experiments in which subjects are deceived about the true nature of the setting, and are later "debriefed" about the manipulation. The dramatic observations in Milgram's [23] study are a case in point. In this experiment, a subject, thinking he participates in an experiment on the effects of punishment on learning, has to administer increasingly more intense shocks to another subject who is actually a stooge.[2] After the experiment the subject must be convinced that the information was fraudulent, i.e., he must be *dehoaxed*. Sometimes, the subject's feelings about himself (e.g., due to having behaved unethically) must be altered. This aspect concerns the *subject's behavior*, which cannot be refuted. Dehoaxing on the other hand, concerns the *experimenter's deception*. For this aspect, conclusions accepted earlier during the experiment can be disproved.[3] Therefore, in the context of non-monotonic reasoning the dehoaxing aspect of debriefing is the more appropriate one to study.

4.1. Experiments on Dehoaxing Per Se

The aspect of dehoaxing would be difficult to isolate from the experiments, since these differ greatly in the nature and degree of deception. Fortunately enough, many experiments have been conducted on dehoaxing per se, employing different designs, different domains, and varying degrees of external validity. A sample of representative studies is summarized in Table 1. As the paradigmatic example I will use an experiment by Ross, Lepper, and Hubbard [28], which was very carefully designed and has been replicated many times. In it, the subject was presented with cards containing pairs of suicide notes. She was told that one note in each pair was genuine, the other bogus, and

[2] The reported traumatic effects on the subjects evoked considerable concern about experiments that may involve harmful after-effects [3], and more attention came to be paid to debriefing [15,31].

[3] E.g., when electrodes have been attached, dehoaxing can be achieved by showing the subject that the wires did not lead anywhere, or as in Milgram's case by a friendly reconciliation with the "victim".

Source	Domain	Deception	Debriefing information	Perseverance after discrediting
Valins 1966	S watches slides taken from Playboy Magazine	heartbeat audible feedback, changing rate with some slides	sound tape was prerecorded	S prefers "reinforced" slides
Walster et al. 1967	S fills out a "Social Aptitude Achievement Test"	high (vs. low) scores are reported	no such test exists	S rates herself as similar to person with high (low) score
Holmes 1973	Instructions on tape informs S he will receive electric shocks during subsequent period	(no shocks are administered)	experimenter interrupts and tells electrode is fake	arousal remains
Ross et al. 1975	S discriminates authentic from unauthentic suicide notes	report of success (vs. failure)	ratings were prepared in advance	S rates herself according to original feedback
Anderson et al. 1980	S examines relationship between risk-taking and success as firefighter	data suggestive of positive (vs. negative) relationship	data on ability are manufactured	S perseveres in estimates for new cases
Jennings et al. 1981	A blood drive: S has to persuade two other students to participate	one reacts positive (vs. uninterested), other doesn't pick up	both confederates, other "apparently failed to keep line free".	S predicts next ten calls will be a success (vs. failure)
Caretta et al. 1982	1972 voters for Nixon were selected (vs. McGovern voters)	(not applicable)	Watergate hearings	Nixon voters retain positive feelings

Table 1. A representative selection of experiments on debriefing, with an approximate account of the setup. "Vs." indicates, where applicable, the success vs. failure manipulation. "S" refers to the subject.

she was asked to indicate the genuine one. In addition she was informed about the average score in a pretest. The subject received false feedback indicating success or failure after each card. After completion of the task she was informed that the feedback had been determined prior to the experiment, and that it was not related to her actual performance (this was called *outcome debriefing*). Nevertheless, the greater the apparent initial success, the higher she estimated her scores for past and future performances. In short, subjects showed a

substantial perseverance of the initial, erroneous impressions. Only after the process underlying the perseverance was explicitly discussed was the initial perception abandoned (the *process debriefing*).

Ross et al.'s explanation for the phenomenon has essentially two parts. The first part stems from the literature on attribution theory: An individual who witnesses a surprising (or extreme) outcome generates (searches for) confirmatory evidence capable of explaining the observed outcome. Second, if the original evidence for the outcome is removed, these antecedents may survive to give independent evidence for the outcome. For example, a subject may attribute her success on the discrimination task to the fact that she was once personally acquainted with a suicide victim.

It may be argued that Ross et al.'s experiment does not in itself prove the presence of self-generated confirmation-biased evidence. Independent support for its presence has been found in various ways, however. For example, enhancing the possibility of producing such evidence increases perseverance [1]. An even stronger test is to prevent the subject from engaging in explanations. This has been done using interference; e.g., in a task similar to that of Ross et al.'s experiment, Fleming and Arrowood [12] made subjects count backwards from 200 by 3 between feedback and outcome debriefing. In a variant of Valins' [32] heartbeat feedback experiment (see Table 1), Barefoot and Straub [2] reduced the exposure time of the slides substantially. In both cases no perseverance effect could be established.

5. A Model for the Process of Debriefing

5.1. Debriefing Modeled Using TMS

To introduce the model for Ross et al.'s experiment I will use TMS, the reason maintenance technique discussed earlier. The debriefing experiment is depicted in Table 2 with the different stadia in terms of TMS. Node **b** represents that the experimenter's assertions are believed as long as it is not believed that the other person is lying. If the latter belief comes IN, the assumption will go OUT (is not believed anymore).

The subject starts out with no particular beliefs about the task. When the experimenter says the subject has performed well (**c**), she infers that this is the case (**b**). From this she generalizes to the belief that she is generally good at recognizing real suicide notes (**a**). This can be probed, e.g., by asking a subject how she would score in the future, or how she thinks she compares to other subjects. At the same time she generates confirmatory evidence (**e**), which comes IN. This evidence itself is an additional justification for belief **a**.

The debriefing takes place by informing the subject about the deception **d**. Since **b** depends on **d** being OUT, **b** goes OUT when **d**

BELIEF	dependencies		beginning of experiment IN	OUT	after feedback IN	OUT	after outcome debriefing IN	OUT	after process debriefing IN	OUT
a. I am good at this kind of task	b,e		o		o		o			o
b. I performed well on this task	c	d	o		o			o		o
c. E said I performed well			o		o		o		o	
d. E provided bogus information			o			o	o		o	
e. [self-generated confirmatory evidence]	c		o		o		o			o

Table 2. The debriefing experiment by Ross et al. The columns labeled "dependencies" show how beliefs depend on other beliefs. The dots represent the status of the assertion on the same line.

comes IN. But when asked, the subject will still believe **a**, on the basis of the independent support **e**. The process debriefing consists of an elaborate discussion of the perseverance phenomenon itself. The subject becomes aware of the self-generated confirmatory evidence she used, and leaves this out of the argument, i.e., **e** goes OUT, and as a consequence **a** goes out as well.

There is more to say about factors that are conducive to belief perseverance; see e.g., [29]. Keeping things simple however, consider a variation of the experiment by Ross et al. One could start the system with **d** IN. In other words, the subject is told in advance that feedback will not be genuine. What will happen? We will come back to this after I have taken a closer look at the states of belief involved in the experiment.

5.2. States of Belief as Admissible Extensions

The model developed so far describes the intended behavior (i.e., in the Ross et al.'s experiment) by showing how the subject gets from one state of belief to another. To ensure that the system represents the intended model, however, it must also rule out behavior not found in (or falsified by) the experiments. A way to find this out is by examining what belief states the system is capable of generating. To this let us describe the dependencies from Table 2 as a set of reasons **R** (indexed by consequent):

$$\mathbf{R} = \{r_{a1}, r_{a2}, r_b, r_c, r_d, r_e\}$$

with

$$r_{a1} = b \parallel \varnothing \parallel\!\!- a \qquad r_{a2} = e \parallel \varnothing \parallel\!\!- a \qquad r_b = c \parallel \mathrm{d} \parallel\!\!- b$$
$$r_c = \varnothing \parallel \varnothing \parallel\!\!- c \qquad r_d = \varnothing \parallel \varnothing \parallel\!\!- d \qquad r_e = c \parallel \varnothing \parallel\!\!- e$$

Computing the closure (e.g., [10]) $\mathbf{R}^*$ of **R** gives two extensions:

$$\mathbf{R}_1^* = \mathbf{R} \cup \{a,b,c,e\}$$

$$\mathbf{R}_2^* = \mathbf{R} \cup \{a,c,d,e\}$$

which are precisely the statements believed before and after debriefing. Now, where does the process debriefing come in? Statement a perseveres via r_c, r_e and r_{a2}. At least one of these reasons is apparently attacked by E (the experimenter). Reason r_c cannot be refuted since c is a fact. So, by discussing the perseverance process itself, the experimenter either defeats r_e, or r_{a2}. Let us first assume the former. This can be formalized by rewriting r_e (and **R** changing accordingly):

$$r_{e1} = c \parallel \mathit{Defeated}(r_{e1}) \parallel\!\!- e$$

$$r_{e2} = \emptyset \parallel \emptyset \parallel\!\!- \mathit{Defeated}(r_{e1})$$

Now, in addition to $\mathbf{R}_1^*$ and $\mathbf{R}_2^*$, two new extensions result:

$$\mathbf{R}_3^* = \mathbf{R} \cup \{a,b,c,\mathit{Defeated}(r_{e1})\}$$

$$\mathbf{R}_4^* = \mathbf{R} \cup \{d,c,\mathit{Defeated}(r_{e1})\}.$$

where $\mathbf{R}_4^*$ gives the belief state after process debriefing. $\mathbf{R}_3^*$ shows the efficacy of the process debriefing, i.e., as measured by the subject's prediction of her future performance on a similar experiment. Another interpretation is that a subject may be forewarned not to generate confirmatory evidence, i.e., to have r_{e2} ready in advance. A natural setting where this could occur is the courtroom. Indeed, in such a situation subjects are much easier to debrief. (E.g., in an experiment by Hatvany and Strack [14] two civil court cases were staged, in which the credibility of the key witness was manipulated.) Independent support to propose r_{e1} and r_{e2} stem from the experiments with an interference task. The interference effectively blocks the generation of confirmatory evidence, or formally, defeats r_{e1}. In this case $\mathbf{R}_3^*$ and $\mathbf{R}_4^*$ represent the states of belief before and after outcome debriefing in the interference task. Now r_{e1} and r_{e2} have been sufficiently justified, it remains to discuss the role of r_{a2}. It could be that this reason is defeated during process debriefing, although this cannot be ascertained on the basis of the experimental evidence currently available. In any case, it can be formalized in a manner analogous to our treatment of r_e above.

5.3. The Model is Neither too Weak Nor too Strong

Since the model proposed above is based on an existing formalism for non-monotonic reasoning, I want to relate it to a criticism that has been advanced concerning such formalisms. Recently, Hanks and McDermott [13] questioned whether these formalisms produce the expected results. They provide axioms for a simple problem (the "Yale shooting scenario") and show that a well-established technique (c.q. predicate-circumscription) produces not only the intended extension, but in addition one that is counter-intuitive. Now, whereas Hanks and McDermott could have chosen between attacking either the axioms or the inference technique, they chose the latter. For this reason, in the section above I generated all the extensions of the proposed axioms for

the Ross et al. experiment, and checked if they indeed belonged to the states of belief I wanted to model. They did. So the model is guaranteed neither too weak, nor too strong in generating states of belief. Yet, a moment of reflection will show that this is not enough to ensure the same holds for the intended behavior, i.e., for the *sequence* of states. To see this, suppose in the Ross et al. experiment the subject is briefed *in advance* that the feedback will not be genuine. That is, we start in Table 2 with **d** IN. Following through the experiment we will see that the same behavior ensues as before. In other words, the subject believes she performs well on the experiment even knowing beforehand that the feedback is bogus. This surely runs counter to our intuition. A similar reasoning as in the "shooting scenario" therefore leads us to believe that our model, as defined by **R**, is too weak (it predicts unintended behavior). Yet, let us stay in the vein of this paper, and see if the prediction can be tested. In fact this has been done already by Wegner, Coulton and Wenzlaff [33] who briefed the subjects in advance with the same words that were used by Ross et al. during debriefing. They found the same perseverance phenomenon, on the basis of which they rejected the theory of Ross et al., and formulated a principle of *transparency of denial*. This principle basically says that when people encounter denied information, that information is available despite the denial. However, in their experiment Wegner et al. tell the subject in advance that the information she *will* obtain is false, i.e., the information is not available at that time. In my opinion it is not necessary to introduce a new principle. Using our terminology, Wegner et al. seem to think they defeat reason r_{e1}, whereas in fact they produce **d**, so that **b** cannot be derived but **c** can. Whichever may be the case, the experiment confirms the counter-intuitive behavior predicted by our model.

6. Degree of Belief

After the painstaking analysis of the debriefing phenomenon, I would like to add one last refinement to the model. Since in the dehoaxing experiments subjects are often asked to rate their beliefs in certain statements, it is justified to ask how such a measure can be modeled.

6.1. Iterated Retraction

Let me go back for a moment to the paradigmatic example of Tweety. Suppose the first thing one learns about Tweety is that it is an ostrich:

Tweety is an ostrich, Tweety is a bird, Tweety cannot fly. (7)

And there is nothing more to infer, nor to retract. To be sure this won't happen, let me start with another Tweety, who is a canary. So (since canaries are birds and birds can fly, canaries can fly) we get:

Tweety is a canary, Tweety is a bird, Tweety can fly. (8)

But suppose we learn that

Tweety broke his wing

upon which we have to retract (8), and infer (7) again. Some time later we come to know that

this happened last summer

and we retract (7) again. Later we learn that

it happened again last night

(and so on).

By now, most people would be inclined to hedge their bets. The belief that Tweety can fly turns out to be graded, and this degree of belief drops when the original assumption is reinstated. It is very tempting to introduce some scheme for evidential reasoning at this point, probabilistic, possibilistic, or otherwise. The problem is: where do these numbers come from? In expert systems it is customary to have numbers attached to derivations through which certainties of data-items are propagated, but there is little theoretical justification for this. Some researchers have tried to fight this ad hoc approach. For example, Nilsson [25] introduces a probabilistic logic based on a possible world semantics. Broadly speaking, he takes as the probability of a sentence the proportion of the possible worlds in which the sentence is true. This resembles Doyle's [7] suggestion that these numbers (e.g., subjective probabilities) do not enter into the computation directly but into the *observation* of the computation.

For the degree of belief in a statement, I take a strictly decreasing function on the size of extensions. That is, extensions of equal size are equally likely, larger ones are less likely. Here is an example of such a function DB for an extension E in the set of extensions $\mathbf{R}^*$ (cf. [8]):

$$DB(E\,|\mathbf{R}^*) = 2^{-|E|}/N$$

where N normalizes the function's range to [0,1]. For the DB of a statement **a** with respect to the underlying set of reasons, a sensible choice could be to sum the DBs of the extensions that contain it:

$$DB(a\,|\mathbf{R}^*) = \sum_{a \in E \in \mathbf{R}^*} DB(E\,|\mathbf{R}^*)$$

During the processing of the example about the canary in the beginning of this section, there was a change in degree of belief upon every new input. To model a similar change in Ross et al.'s experiment, I propose to compute the DB with respect to the extensions the perceiver is aware of at the moment she has to do the rating. This is the topic of the last section.

6.2. Change in Degree of Belief During the Experiment

Let us apply the last section to Ross et al.'s experiment. In the situation after outcome debriefing, the subject has produced $\mathbf{R}_1^*$ and $\mathbf{R}_2^*$, which are of equal size. Thus, since statement **a** ("I am good at this kind of task") is in both extensions, it has 1 as degree of belief. After process debriefing, $\mathbf{R}_4^*$ is produced, in which **a** is not believed, so that the belief for **a** will drop. As a final example and to make my proposal more vulnerable to experimental verification, let me speculate about an experiment that to my knowledge has not yet been conducted. After going through the original experiment one could debrief the debriefing, i.e., inform the subject that the feedback had been genuine all along. If we amend the rules accordingly the end result is that the new extension contains **b** but not **a** (which is no longer supported by any self-generated confirmatory evidence). To get an indication of the degree of belief I used the belief function *DB* defined in the last section. It should be very clear that numbers produced this way can only give an indication of trend. At this stage there can be little evidence as to what extensions the subjects will actually perceive or how large these sets might be. For example, in the model I lumped together all confirmatory evidence, whereas subjects, when asked, may generate several reasons for success or failure [12]. In addition the *DB* function is one of the simplest one can think of. With these reservations, some results can be reported: (1) The values for **a**; the rated ability at the task. The simulated numbers show the same trend found in all the debriefing experiments. After the last stage of the imaginary experiment the value levels off, i.e., the subject will consider herself no better or worse than average. (2) The value for **e**, the confirmatory evidence. This value declines after process debriefing, as it should, but there is a notable residue. After the last step of the imaginary experiment the value reduces to almost zero. (3) One may wonder why the subject should continue trusting the experimenter after the first deception. The values for **d** ("*E* provides mock information") reflect a steeply ascending suspicion on behalf of the subject. Although in the past great care has been taken to improve the face value of the dehoaxing, the simulation seems to make a further investigation into this factor worthwhile.

7. Conclusions

An overview was presented of a lively area in the foundations of artificial intelligence, namely that of non-monotonic reasoning. As it turns out, the formal approaches proposed to date fail to explain why people prefer particular extensions in the set of extensions allowed by the postulated logical theories. The current paper proposed a process model that describes how such preferences come into existence and how they change. The model also describes formally why people cling to their initial beliefs more strongly than appears warranted. The new

results prove the value of focusing on how people actually behave in a wide variety of experimental settings where change of belief is called for.

Acknowledgements: I am grateful to Anthony Jameson, Eric Meyer and Peter Shell for discussions about the work reported.

References

1. C. Anderson, M. Lepper & L. Ross: Perseverance of social theories: The role of explanation in the persistence of discredited information, *Journal of Personality and Social Psychology* **39** (1980) 1037-1049.
2. J. Barefoot & R. Straub: Opportunity for information search and the effect of false heart-rate feedback, *Journal of Personality and Social Psychology* **17** (1971) 154-157.
3. D. Baumrind: Some thoughts on ethics of research: After reading Milgram's "Behavioral study of obedience", *American Psychologist* **19** (1964) 421-423.
4. D. Bobrow & T. Winograd: An overview of KRL, a Knowledge Representation Language, *Cognitive Science* **1** (1977) 3-46.
5. P. Cohen & E. Feigenbaum (Eds.): *The Handbook of Artificial Intelligence*, Volume 3 (1982), Chapter II-E, William Kaufmann, Los Altos, Ca.
6. J. Doyle: A truth maintenance system, *Artificial Intelligence* **12** (1979) 231-272.
7. J. Doyle: Some theories of reasoned assumptions: An essay in rational psychology, Report (1982), Department of Computer Science, Carnegie-Mellon University.
8. J. Doyle: Methodological simplicity in experts system construction, *AI Magazine* **4** (1983) Nr.2, 39-43.
9. J. Doyle: Circumscription and implicit definability, *Journal of Automated Reasoning* **1** (1985) 391-405.
10. D. Etherington: Formalizing non-monotonic reasoning systems, *Artificial Intelligence* **31** (1987) 41-85.
11. R. Fikes: Deductive retrieval mechanisms for state description models, *Proceedings of IJCAI-75* (1975) 99-106.
12. J. Fleming & A. Arrowood: Information processing and the perseverance of discredited self-perception, *Personality and Social Psychology Bulletin* **5** (1979) 201-205.
13. S. Hanks & D. McDermott: Default reasoning, non-monotonic logics, and the frame problem, *Proceedings of AAAI-86* (1986) 328-333, Morgan Kaufmann, Los Altos, Ca.

14. N. Hatvany & F. Strack: The impact of a discredited key witness, *Journal of Applied Social Psychology* **10** (1980) 490-509.

15. D. Holmes: Debriefing after psychological experiments, I. Effectiveness of postdeception dehoaxing, II. Effectiveness of postexperimental desensitizing, *American Psychologist* **31** (1976) 858-875.

16. J. de Kleer: Choices without backtracking, *Proceedings of AAAI-84* (1984) 79-85, Morgan Kaufmann, Los Altos, Ca.

17. J. de Kleer & B. Williams: Back to backtracking: Controlling the ATMS, *Proceedings of AAAI-86* (1986) 910-917, Morgan Kaufmann, Los Altos, Ca.

18. V. Lifschitz: Point-wise circumscription: Preliminary report, *Proceedings of AAAI-86* (1986) 406-410, Morgan Kaufmann, Los Altos, Ca.

19. J. McCarthy: Addendum: Circumscription and other non-monotonic formalisms, *Artificial Intelligence* **13** (1980) 171-172.

20. J. McCarthy: Applications of circumscription to formalizing common-sense knowledge, *Artificial Intelligence* **28** (1986) 89-116.

21. D. McDermott: Non-monotonic logic II: Non-monotonic modal theories, *J. Assoc. Comput. Mach.* **29** (1982) 33-57.

22. D. McDermott & J. Doyle: Non-Monotonic logic I, *Artificial Intelligence* **13** (1980) 41-72.

23. S. Milgram: Behavioral study of obedience, *Journal of Abnormal and Social Psychology* **67** (1963) 371-378.

24. R. Moore: Semantical considerations on non-monotonic logic, *Artificial Intelligence* **25** (1985) 75-94.

25. N. Nilsson: Probabilistic logic. *Artificial Intelligence* **28** (1986) 71-87.

26. D. Norman & D. Bobrow: On data-limited and resource-limited processes, *Cognitive Psychology* **7** (1975) 44-64.

27. R. Reiter: A logic for default reasoning, *Artificial Intelligence* **13** (1980) 81-132.

28. L. Ross, M. Lepper & M. Hubbard: Perseverance in self-perception and social perception: Biased attributional processes in the debriefing paradigm, *Journal of Personality and Social Psychology* **32** (1975) 880-892.

29. Y. Schul & E. Burnstein: When discounting fails: Conditions under which individuals use discredited information in making a judgment, *Journal of Personality and Social Psychology* **49** (1985) 894-903.

30. R. Stallman & G. Sussman: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit

analysis, *Artificial Intelligence* **9** (1977) 135-196.

31. D. Ullman & T. Jackson: Researchers' ethical conscience: Debriefing from 1960-1980, *American Psychologist* **37** (1982) 972-973.
32. S. Valins: Cognitive effects of false heart-rate feedback, *Journal of Personality and Social Psychology* **4** (1966) 400-408.
33. D. Wegner, G. Coulton & R. Wenzlaff: The transparency of denial: Briefing in the debriefing paradigm, *Journal of Personality and Social Psychology* **49** (1985) 338-346.
34. R. Weyhrauch: Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence* **13** (1980) 133-170.
35. T. Winograd: Extended inference modes in reasoning by computer systems, *Artificial Intelligence* **13** (1980) 5-26.

MATHEMATICAL CENTRE TRACTS

1 T. van der Walt. *Fixed and almost fixed points.* 1963.

2 A.R. Bloemena. *Sampling from a graph.* 1964.

3 G. de Leve. *Generalized Markovian decision processes, part I: model and method.* 1964.

4 G. de Leve. *Generalized Markovian decision processes, part II: probabilistic background.* 1964.

5 G. de Leve, H.C. Tijms, P.J. Weeda. *Generalized Markovian decision processes, applications.* 1970.

6 M.A. Maurice. *Compact ordered spaces.* 1964.

7 W.R. van Zwet. *Convex transformations of random variables.* 1964.

8 J.A. Zonneveld. *Automatic numerical integration.* 1964.

9 P.C. Baayen. *Universal morphisms.* 1964.

10 E.M. de Jager. *Applications of distributions in mathematical physics.* 1964.

11 A.B. Paalman-de Miranda. *Topological semigroups.* 1964.

12 J.A.Th.M. van Berckel, H. Brandt Corstius, R.J. Mokken, A. van Wijngaarden. *Formal properties of newspaper Dutch.* 1965.

13 H.A. Lauwerier. *Asymptotic expansions.* 1966, out of print; replaced by MCT 54.

14 H.A. Lauwerier. *Calculus of variations in mathematical physics.* 1966.

15 R. Doornbos. *Slippage tests.* 1966.

16 J.W. de Bakker. *Formal definition of programming languages with an application to the definition of ALGOL 60.* 1967.

17 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 1.* 1968.

18 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 2.* 1968.

19 J. van der Slot. *Some properties related to compactness.* 1968.

20 P.J. van der Houwen. *Finite difference methods for solving partial differential equations.* 1968.

21 E. Wattel. *The compactness operator in set theory and topology.* 1968.

22 T.J. Dekker. *ALGOL 60 procedures in numerical algebra, part 1.* 1968.

23 T.J. Dekker, W. Hoffmann. *ALGOL 60 procedures in numerical algebra, part 2.* 1968.

24 J.W. de Bakker. *Recursive procedures.* 1971.

25 E.R. Paërl. *Representations of the Lorentz group and projective geometry.* 1969.

26 European Meeting 1968. *Selected statistical papers, part I.* 1968.

27 European Meeting 1968. *Selected statistical papers, part II.* 1968.

28 J. Oosterhoff. *Combination of one-sided statistical tests.* 1969.

29 J. Verhoeff. *Error detecting decimal codes.* 1969.

30 H. Brandt Corstius. *Exercises in computational linguistics.* 1970.

31 W. Molenaar. *Approximations to the Poisson, binomial and hypergeometric distribution functions.* 1970.

32 L. de Haan. *On regular variation and its application to the weak convergence of sample extremes.* 1970.

33 F.W. Steutel. *Preservation of infinite divisibility under mixing and related topics.* 1970.

34 I. Juhász, A. Verbeek, N.S. Kroonenberg. *Cardinal functions in topology.* 1971.

35 M.H. van Emden. *An analysis of complexity.* 1971.

36 J. Grasman. *On the birth of boundary layers.* 1971.

37 J.W. de Bakker, G.A. Blaauw, A.J.W. Duijvestijn, E.W. Dijkstra, P.J. van der Houwen, G.A.M. Kamsteeg-Kemper, F.E.J. Kruseman Aretz, W.L. van der Poel, J.P. Schaap-Kruseman, M.V. Wilkes, G. Zoutendijk. *MC-25 Informatica Symposium.* 1971.

38 W.A. Verloren van Themaat. *Automatic analysis of Dutch compound words.* 1972.

39 H. Bavinck. *Jacobi series and approximation.* 1972.

40 H.C. Tijms. *Analysis of (s,S) inventory models.* 1972.

41 A. Verbeek. *Superextensions of topological spaces.* 1972.

42 W. Vervaat. *Success epochs in Bernoulli trials (with applications in number theory).* 1972.

43 F.H. Ruymgaart. *Asymptotic theory of rank tests for independence.* 1973.

44 H. Bart. *Meromorphic operator valued functions.* 1973.

45 A.A. Balkema. *Monotone transformations and limit laws.* 1973.

46 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 1: the language.* 1973.

47 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 2: the compiler.* 1973.

48 F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn. *An ALGOL 60 compiler in ALGOL 60, text of the MC-compiler for the EL-X8.* 1973.

49 H. Kok. *Connected orderable spaces.* 1974.

50 A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (eds.). *Revised report on the algorithmic language ALGOL 68.* 1976.

51 A. Hordijk. *Dynamic programming and Markov potential theory.* 1974.

52 P.C. Baayen (ed.). *Topological structures.* 1974.

53 M.J. Faber. *Metrizability in generalized ordered spaces.* 1974.

54 H.A. Lauwerier. *Asymptotic analysis, part 1.* 1974.

55 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 1: theory of designs, finite geometry and coding theory.* 1974.

56 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry.* 1974.

57 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 3: combinatorial group theory.* 1974.

58 W. Albers. *Asymptotic expansions and the deficiency concept in statistics.* 1975.

59 J.L. Mijnheer. *Sample path properties of stable processes.* 1975.

60 F. Göbel. *Queueing models involving buffers.* 1975.

63 J.W. de Bakker (ed.). *Foundations of computer science.* 1975.

64 W.J. de Schipper. *Symmetric closed categories.* 1975.

65 J. de Vries. *Topological transformation groups, 1: a categorical approach.* 1975.

66 H.G.J. Pijls. *Logically convex algebras in spectral theory and eigenfunction expansions.* 1976.

68 P.P.N. de Groen. *Singularly perturbed differential operators of second order.* 1976.

69 J.K. Lenstra. *Sequencing by enumerative methods.* 1977.

70 W.P. de Roever, Jr. *Recursive program schemes: semantics and proof theory.* 1976.

71 J.A.E.E. van Nunen. *Contracting Markov decision processes.* 1976.

72 J.K.M. Jansen. *Simple periodic and non-periodic Lamé functions and their applications in the theory of conical waveguides.* 1977.

73 D.M.R. Leivant. *Absoluteness of intuitionistic logic.* 1979.

74 H.J.J. te Riele. *A theoretical and computational study of generalized aliquot sequences.* 1976.

75 A.E. Brouwer. *Treelike spaces and related connected topological spaces.* 1977.

76 M. Rem. *Associons and the closure statement.* 1976.

77 W.C.M. Kallenberg. *Asymptotic optimality of likelihood ratio tests in exponential families.* 1978.

78 E. de Jonge, A.C.M. van Rooij. *Introduction to Riesz spaces.* 1977.

79 M.C.A. van Zuijlen. *Emperical distributions and rank statistics.* 1977.

80 P.W. Hemker. *A numerical study of stiff two-point boundary problems.* 1977.

81 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 1.* 1976.

82 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 2.* 1976.

83 L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* 1979.

84 H.L.L. Busard. *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?), books vii-xii.* 1977.

85 J. van Mill. *Supercompactness and Wallman spaces.* 1977.

86 S.G. van der Meulen, M. Veldhorst. *Torrix I, a programming system for operations on vectors and matrices over arbitrary fields and of variable size.* 1978.

88 A. Schrijver. *Matroids and linking systems.* 1977.

89 J.W. de Roever. *Complex Fourier transformation and analytic functionals with unbounded carriers.* 1978.

90 L.P.J. Groenewegen. *Characterization of optimal strategies in dynamic games.* 1981.

91 J.M. Geysel. *Transcendence in fields of positive characteristic.* 1979.

92 P.J. Weeda. *Finite generalized Markov programming.* 1979.

93 H.C. Tijms, J. Wessels (eds.). *Markov decision theory.* 1977.

94 A. Bijlsma. *Simultaneous approximations in transcendental number theory.* 1978.

95 K.M. van Hee. *Bayesian control of Markov chains.* 1978.

96 P.M.B. Vitányi. *Lindenmayer systems: structure, languages, and growth functions.* 1980.

97 A. Federgruen. *Markovian control problems; functional equations and algorithms.* 1984.

98 R. Geel. *Singular perturbations of hyperbolic type.* 1978.

99 J.K. Lenstra, A.H.G. Rinnooy Kan, P. van Emde Boas (eds.). *Interfaces between computer science and operations research.* 1978.

100 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1.* 1979.

101 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2.* 1979.

102 D. van Dulst. *Reflexive and superreflexive Banach spaces.* 1978.

103 K. van Harn. *Classifying infinitely divisible distributions by functional equations.* 1978.

104 J.M. van Wouwe. *Go-spaces and generalizations of metrizability.* 1979.

105 R. Helmers. *Edgeworth expansions for linear combinations of order statistics.* 1982.

106 A. Schrijver (ed.). *Packing and covering in combinatorics.* 1979.

107 C. den Heijer. *The numerical solution of nonlinear operator equations by imbedding methods.* 1979.

108 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 1.* 1979.

109 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 2.* 1979.

110 J.C. van Vliet. *ALGOL 68 transput, part I: historical review and discussion of the implementation model.* 1979.

111 J.C. van Vliet. *ALGOL 68 transput, part II: an implementation model.* 1979.

112 H.C.P. Berbee. *Random walks with stationary increments and renewal theory.* 1979.

113 T.A.B. Snijders. *Asymptotic optimality theory for testing problems with restricted alternatives.* 1979.

114 A.J.E.M. Janssen. *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes.* 1979.

115 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 1.* 1979.

116 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 2.* 1979.

117 P.J.M. Kallenberg. *Branching processes with continuous state space.* 1979.

118 P. Groeneboom. *Large deviations and asymptotic efficiencies.* 1980.

119 F.J. Peters. *Sparse matrices and substructures, with a novel implementation of finite element algorithms.* 1980.

120 W.P.M. de Ruyter. *On the asymptotic analysis of large-scale ocean circulation.* 1980.

121 W.H. Haemers. *Eigenvalue techniques in design and graph theory.* 1980.

122 J.C.P. Bus. *Numerical solution of systems of nonlinear equations.* 1980.

123 I. Yuhász. *Cardinal functions in topology - ten years later.* 1980.

124 R.D. Gill. *Censoring and stochastic integrals.* 1980.

125 R. Eising. *2-D systems, an algebraic approach.* 1980.

126 G. van der Hoek. *Reduction methods in nonlinear programming.* 1980.

127 J.W. Klop. *Combinatory reduction systems.* 1980.

128 A.J.J. Talman. *Variable dimension fixed point algorithms and triangulations.* 1980.

129 G. van der Laan. *Simplicial fixed point algorithms.* 1980.

130 P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint, A.H. Veen. *ILP: intermediate language for pictures.* 1980.

131 R.J.R. Back. *Correctness preserving program refinements: proof theory and applications.* 1980.

132 H.M. Mulder. *The interval function of a graph.* 1980.

133 C.A.J. Klaassen. *Statistical performance of location estimators.* 1981.

134 J.C. van Vliet, H. Wupper (eds.). *Proceedings international conference on ALGOL 68.* 1981.

135 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part I.* 1981.

136 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part II.* 1981.

137 J. Telgen. *Redundancy and linear programs.* 1981.

138 H.A. Lauwerier. *Mathematical models of epidemics.* 1981.

139 J. van der Wal. *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games.* 1981.

140 J.H. van Geldrop. *A mathematical theory of pure exchange economies without the no-critical-point hypothesis.* 1981.

141 G.E. Welters. *Abel-Jacobi isogenies for certain types of Fano threefolds.* 1981.

142 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 1.* 1981.

143 J.M. Schumacher. *Dynamic feedback in finite- and infinite-dimensional linear systems.* 1981.

144 P. Eijgenraam. *The solution of initial value problems using interval arithmetic; formulation and analysis of an algorithm.* 1981.

145 A.J. Brentjes. *Multi-dimensional continued fraction algorithms.* 1981.

146 C.V.M. van der Mee. *Semigroup and factorization methods in transport theory.* 1981.

147 H.H. Tigelaar. *Identification and informative sample size.* 1982.

148 L.C.M. Kallenberg. *Linear programming and finite Markovian control problems.* 1983.

149 C.B. Huijsmans, M.A. Kaashoek, W.A.J. Luxemburg, W.K. Vietsch (eds.). *From A to Z, proceedings of a symposium in honour of A.C. Zaanen.* 1982.

150 M. Veldhorst. *An analysis of sparse matrix storage schemes.* 1982.

151 R.J.M.M. Does. *Higher order asymptotics for simple linear rank statistics.* 1982.

152 G.F. van der Hoeven. *Projections of lawless sequences.* 1982.

153 J.P.C. Blanc. *Application of the theory of boundary value problems in the analysis of a queueing model with paired services.* 1982.

154 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part I.* 1982.

155 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part II.* 1982.

156 P.M.G. Apers. *Query processing and data allocation in distributed database systems.* 1983.

157 H.A.W.M. Kneppers. *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic.* 1983.

158 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 1.* 1983.

159 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 2.* 1983.

160 A. Rezus. *Abstract AUTOMATH.* 1983.

161 G.F. Helminck. *Eisenstein series on the metaplectic group, an algebraic approach.* 1983.

162 J.J. Dik. *Tests for preference.* 1983.

163 H. Schippers. *Multiple grid methods for equations of the second kind with applications in fluid mechanics.* 1983.

164 F.A. van der Duyn Schouten. *Markov decision processes with continuous time parameter.* 1983.

165 P.C.T. van der Hoeven. *On point processes.* 1983.

166 H.B.M. Jonkers. *Abstraction, specification and implementation techniques, with an application to garbage collection.* 1983.

167 W.H.M. Zijm. *Nonnegative matrices in dynamic programming.* 1983.

168 J.H. Evertse. *Upper bounds for the numbers of solutions of diophantine equations.* 1983.

169 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 2.* 1983.

CWI TRACTS

1 D.H.J. Epema. *Surfaces with canonical hyperplane sections.* 1984.

2 J.J. Dijkstra. *Fake topological Hilbert spaces and characterizations of dimension in terms of negligibility.* 1984.

3 A.J. van der Schaft. *System theoretic descriptions of physical systems.* 1984.

4 J. Koene. *Minimal cost flow in processing networks, a primal approach.* 1984.

5 B. Hoogenboom. *Intertwining functions on compact Lie groups.* 1984.

6 A.P.W. Böhm. *Dataflow computation.* 1984.

7 A. Blokhuis. *Few-distance sets.* 1984.

8 M.H. van Hoorn. *Algorithms and approximations for queueing systems.* 1984.

9 C.P.J. Koymans. *Models of the lambda calculus.* 1984.

10 C.G. van der Laan, N.M. Temme. *Calculation of special functions: the gamma function, the exponential integrals and error-like functions.* 1984.

11 N.M. van Dijk. *Controlled Markov processes; time-discretization.* 1984.

12 W.H. Hundsdorfer. *The numerical solution of nonlinear stiff initial value problems: an analysis of one step methods.* 1985.

13 D. Grune. *On the design of ALEPH.* 1985.

14 J.G.F. Thiemann. *Analytic spaces and dynamic programming: a measure theoretic approach.* 1985.

15 F.J. van der Linden. *Euclidean rings with two infinite primes.* 1985.

16 R.J.P. Groothuizen. *Mixed elliptic-hyperbolic partial differential operators: a case-study in Fourier integral operators.* 1985.

17 H.M.M. ten Eikelder. *Symmetries for dynamical and Hamiltonian systems.* 1985.

18 A.D.M. Kester. *Some large deviation results in statistics.* 1985.

19 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 1: Philosophy, framework, computer science.* 1986.

20 B.F. Schriever. *Order dependence.* 1986.

21 D.P. van der Vecht. *Inequalities for stopped Brownian motion.* 1986.

22 J.C.S.P. van der Woude. *Topological dynamix.* 1986.

23 A.F. Monna. *Methods, concepts and ideas in mathematics: aspects of an evolution.* 1986.

24 J.C.M. Baeten. *Filters and ultrafilters over definable subsets of admissible ordinals.* 1986.

25 A.W.J. Kolen. *Tree network and planar rectilinear location theory.* 1986.

26 A.H. Veen. *The misconstrued semicolon: Reconciling imperative languages and dataflow machines.* 1986.

27 A.J.M. van Engelen. *Homogeneous zero-dimensional absolute Borel sets.* 1986.

28 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 2: Applications to natural language.* 1986.

29 H.L. Trentelman. *Almost invariant subspaces and high gain feedback.* 1986.

30 A.G. de Kok. *Production-inventory control models: approximations and algorithms.* 1987.

31 E.E.M. van Berkum. *Optimal paired comparison designs for factorial experiments.* 1987.

32 J.H.J. Einmahl. *Multivariate empirical processes.* 1987.

33 O.J. Vrieze. *Stochastic games with finite state and action spaces.* 1987.

34 P.H.M. Kersten. *Infinitesimal symmetries: a computational approach.* 1987.

35 M.L. Eaton. *Lectures on topics in probability inequalities.* 1987.

36 A.H.P. van der Burgh, R.M.M. Mattheij (eds.). *Proceedings of the first international conference on industrial and applied mathematics (ICIAM 87).* 1987.

37 L. Stougie. *Design and analysis of algorithms for stochastic integer programming.* 1987.

38 J.B.G. Frenk. *On Banach algebras, renewal measures and regenerative processes.* 1987.

39 H.J.M. Peters, O.J. Vrieze (eds.). *Surveys in game theory and related topics.* 1987.

40 J.L. Geluk, L. de Haan. *Regular variation, extensions and Tauberian theorems.* 1987.

41 Sape J. Mullender (ed.). *The Amoeba distributed operating system: Selected papers 1984-1987.* 1987.

42 P.R.J. Asveld, A. Nijholt (eds.). *Essays on concepts, formalisms, and tools.* 1987.