

Conversions of λ -terms and combinators

Alexander Lourier

Department of Cryptology and Discrete Mathematics
Moscow Engineering Physics Institute
Kashirskoye Shosse, 31
Moscow, 115409, Russia
aml@rulezz.ru

Abstract

This paper describes architecture of the compiler from lambda-terms to combinators and combinatorial evaluator, performing the reverse task. New recursive data structures were developed for handling syntax tree, allowing various optimization techniques to be applied. Complete implementation of the algorithms covered in the paper is available at <http://www.rulezz.ru/cl/compiling/ceval.tar.gz>. The purpose of the development is to help in designing of the new optimizing compilers for applicative programming languages. Students of the Computer Science departments should have a tool to check their knowledge of the combinatory logics basics and a research tool to automate some operations like graph reduction and compilation lambda-terms to the combinatory code.

Keywords: combinatory logic, lambda-calculus, compilation, combinators, parser, compiler, interpreter, optimization, functional programming.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CSIT copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Institute for Contemporary Education JMSUICE. To copy otherwise, or to republish, requires a fee and/or special permission from the JMSUICE.

**Proceedings of the Workshop on Computer Science and Information Technologies CSIT'2001
Ufa, Yangantau, Russia, 2001**

1 Introduction

Compiler and interpreter share the common data structures – syntax tree, representing the expression. Functions to parse expression to the tree and restoring expression from tree were made for these structures. For details, look at Section 3.1.

Compilation of the lambda-term to combinatory representation is described in the Section 3.3. Optimization techniques from that section give a great performance boost - look at the Figure 1. As you can see, the algorithm with optimization gives a better performance on the long combinators. Execution time becomes a polynomial function, rather than exponential in the not optimized case.

The program offers user an ability to check his knowledge in the combinatory logic and lambda-calculus and to use it in their scientific work. The program can give users tasks, for example to do not only 'x, y, z' lambda-term compilation (which is boring for students), but 'convert' one english word to another (if every letter of the words is a variable), for example:

$$\lambda void.odd = K(S(KK)(S(S(KS) \\ (S(S(KS)(S(KK)I))(KI)))(KI)))$$

This is not only useful, but also interesting for students.

2 Theory

As well known, any lambda-term can be represented as a combinatory term. Process of conversion from lambda-term representation to combinatory representation can be automated and it can simplify the work of a programmer. It is enough to have a very limited set of combinators (named basis) to compile a lambda-term of any complexity. For example using only I , K

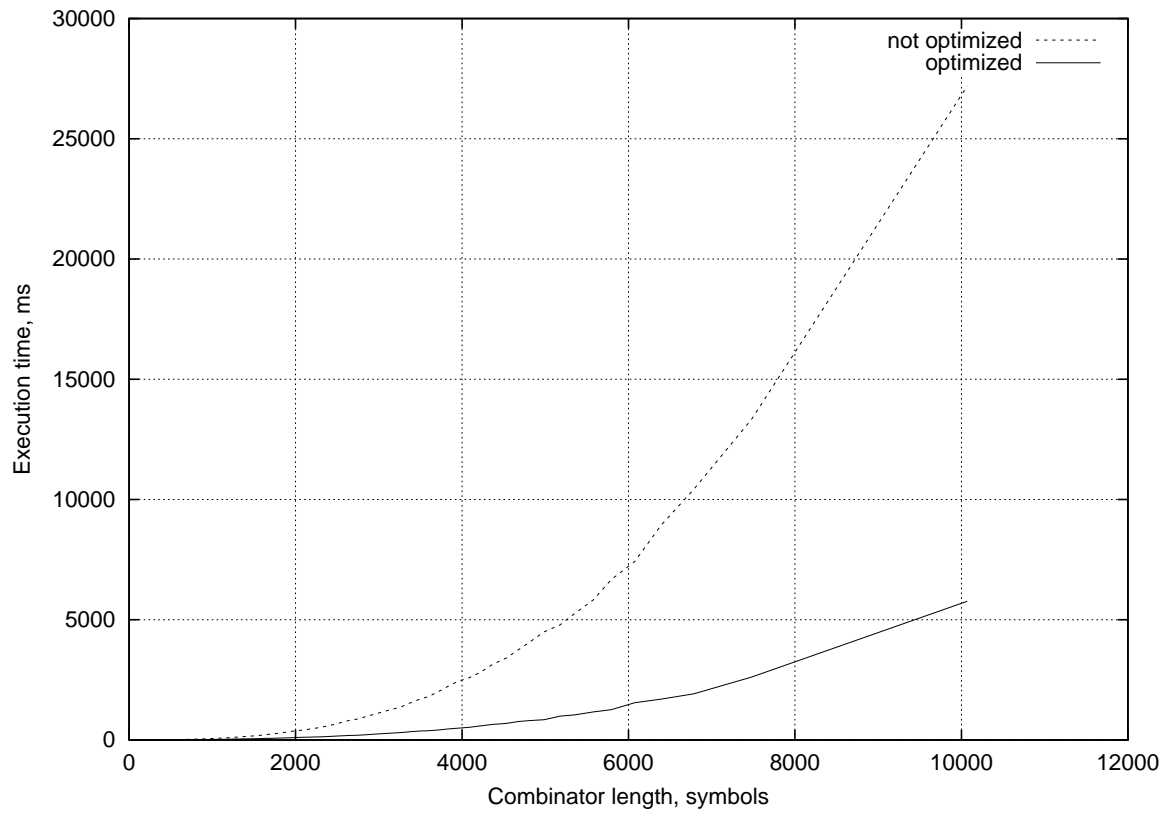


Figure 1: Performance of the optimized and not optimized algorithms

and S combinators (for details look at [1]). Furthermore ‘ I ’ can be represented as ‘ SKK ’, so only S and K are necessary.

$$\begin{aligned} I &= \lambda x.x \\ K &= \lambda xy.x \\ S &= \lambda xyz.xz(yz) \end{aligned} \quad (1)$$

Here are the rules of conversion of lambda-term to combinatory form (compiling into $\langle I, K, S \rangle$ -basis) (taken from [2]):

$$\begin{aligned} \lambda x.x &= I \\ \lambda x.P &= KP, \text{ if } x \notin P \\ \lambda x.PQ &= S(\lambda x.P)(\lambda x.Q) \end{aligned} \quad (2)$$

This is an example how to convert *void* to *odd* (every letter of the word represents a variable):

$$\begin{aligned} \lambda \text{void.odd} &= (\lambda v.(\lambda o.(\lambda i.(\lambda d.\text{odd}))) \\ &= (\lambda v.(\lambda o.(\lambda i.S(\lambda d.od)(\lambda d.d)))) \\ &= (\lambda v.(\lambda o.(\lambda i.S(S(\lambda d.o)(\lambda d.d))(\lambda d.d)))) \\ &= (\lambda v.(\lambda o.(\lambda i.S(S(Ko)(\lambda d.d))(\lambda d.d)))) \\ &= (\lambda v.(\lambda o.(\lambda i.S(S(Ko)I)(\lambda d.d)))) \\ &= (\lambda v.(\lambda o.(\lambda i.S(S(Ko)II))) \\ &= (\lambda v.(\lambda o.K(S(S(Ko)II))) \\ &= (\lambda v.S(\lambda o.K)(\lambda o.S(S(Ko)II))) \\ &= (\lambda v.S(KK)(\lambda o.S(S(Ko)II))) \\ &= (\lambda v.S(KK)(S(\lambda o.S(S(Ko)I))(\lambda o.I))) \\ &= (\lambda v.S(KK)(S(S(\lambda o.S)(\lambda o. \mapsto \\ &\quad \mapsto S(Ko)I))(\lambda o.I))) \\ &= (\lambda v.S(KK)(S(S(KS) \mapsto \\ &\quad \mapsto (\lambda o.S(Ko)I))(\lambda o.I))) \\ &= (\lambda v.S(KK)(S(S(KS)(S \mapsto \\ &\quad \mapsto (\lambda o.S(Ko))(\lambda o.I)))(\lambda o.I))) \\ &\dots \\ &= K(S(KK)(S(S(KS)(S(S(KS) \mapsto \\ &\quad \mapsto (S(KK)I))(KI)))(KI))) \end{aligned}$$

The program can do also the reverse operation – graph reduction:

$$\begin{aligned} Ia &= a \\ Kab &= a \\ Sabc &= ac(bc) \end{aligned} \quad (3)$$

This is an example how to reduce the $K(S(KK)(S(S(KS)(S(S(KS)(S(KK)I))(KI)))(KI)))$ combinator. ‘ \triangleright ’ is the relation of reduction. Let’s transfer 4 actual parameters – v, o, i and d :

$$\begin{aligned} K(S(KK)(S(S(KS)(S(S(KS)(S(KK)I)) \mapsto \\ \mapsto (KI)))(KI)))\text{void} \triangleright \end{aligned}$$

$$\begin{aligned} &\triangleright S(KK)(S(S(KS)(S(S(KS)(S(KK)I)) \mapsto \\ &\quad \mapsto (KI)))(KI))oid \triangleright \\ &\triangleright KKo(S(S(KS)(S(S(KS)(S(KK)I)) \mapsto \\ &\quad \mapsto (KI)))(KI)o)id \triangleright \\ &\triangleright K(S(S(KS)(S(S(KS)(S(KK)I)) \mapsto \\ &\quad \mapsto (KI)))(KI)o)id \triangleright \\ &\triangleright S(S(KS)(S(S(KS)(S(KK)I))(KI)))(KI)od \triangleright \\ &\triangleright S(KS)(S(S(KS)(S(KK)I))(KI)o(KIo)d) \triangleright \\ &\triangleright KSo(S(S(KS)(S(KK)I))(KI)o(KIo)d) \triangleright \\ &\triangleright S(S(S(KS)(S(KK)I))(KI)o(KIo)d) \triangleright \\ &\triangleright S(S(KS)(S(KK)I))(KI)od(KIo)d \triangleright \\ &\triangleright S(KS)(S(KK)I)o(KIo)d(KIo)d \triangleright \\ &\triangleright KSo(S(KK)Io)(KIo)d(KIo)d \triangleright \\ &\triangleright S(S(KK)Io)(KIo)d(KIo)d \triangleright \\ &\triangleright S(KK)Iod(KIo)d(KIo)d \triangleright \\ &\triangleright KKo(Io)d(KIo)d(KIo)d \triangleright \\ &\triangleright K(Io)d(KIo)d(KIo)d \triangleright \\ &\triangleright Io(KIo)d(KIo)d \triangleright \\ &\triangleright o(KIo)d(KIo)d \triangleright \\ &\triangleright o(Id)(Id) \triangleright \\ &\triangleright odd \end{aligned}$$

3 Algorithm

3.1 Syntax analysis

First of all our program parses user input and builds syntax tree, as shown in Figure 2.

Current pointer of `syntax_tree` is pointed to the first token in the line. Than parser scans the line supplied and looks for letters, points, combinators and other objects. Once it finds variable or combinator, it creates new `syntax_tree` structure with type `T_VARIABLE` or `T_OBJECT` respectively and attaches it to the end of list (after the current position). Current position is shifted to the new created element (see Figure 3). If program finds an opening bracket, than it searches for the matching closing bracket and calls itself recursively for the internal string. If program finds symbol ‘ L ’ (representing λ), it recognizes it as lambda-term beginning and tries to fetch bound variables and to build the respective `syntax_tree` structure. Every `syntax_tree` structure has a ‘next’ pointer, which in the most cases represents the order of applications, and pointers `d1` and `d2`, that mean different expressions depending on the ‘type’: for `type=T_BRACKETS`, `d1` means expression in the brackets, for `type=T_LAMBDA`, `d1` is the list of bound variables and `d2` is the body of the lambda-term. As you can see, the only case, when ‘next’ pointer does not mean application is the list of bound variables in the lambda-term.

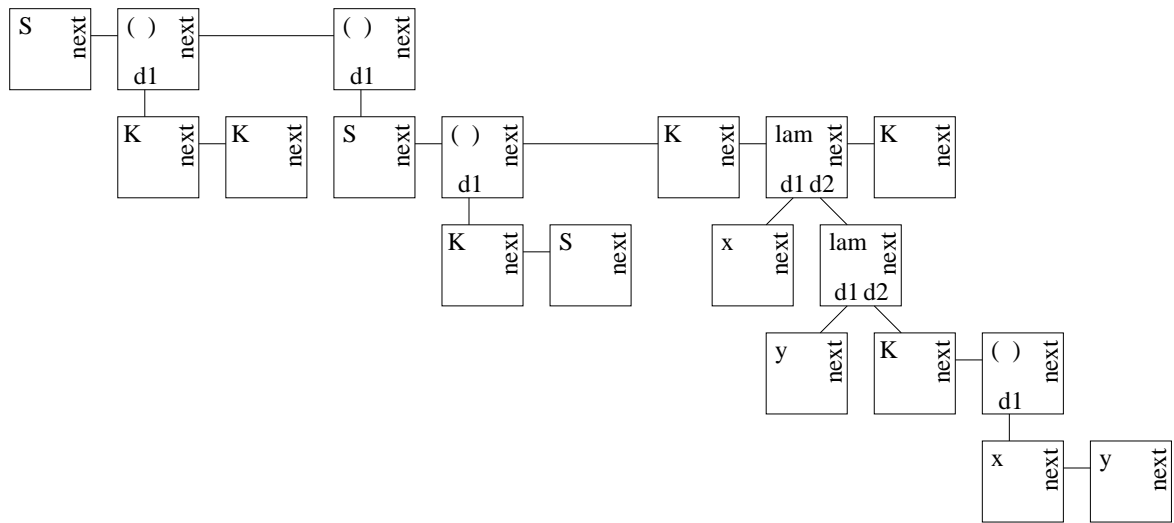


Figure 2: Syntax tree of $S(KK)(S(KS)K(\lambda xy.K(xy))K)$

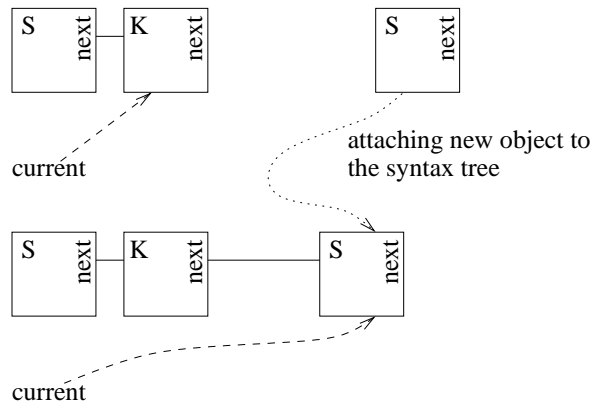


Figure 3: Appending a new combinator to the syntax tree

3.2 Reduction

Programs checks the first object in the tree – if it is a known combinator, it tries to fetch an appropriate number of arguments to this combinator. If it succeeds, it creates duplicate subtrees, deletes unused subtrees and changes next pointers of some objects according to rules (3). For example, how it is done for the S combinator, see Figure 4. There you can see, that first of all three arguments for the S combinator are disconnected from the tree. Then third argument is duplicated, brackets object is created and finally all five objects are linked in the new order.

Then program recursively traverses all braces in the expression and repeats the process for all expressions in the brackets, for instance:

$$S(Sxyz) \triangleright S(xz(yz)) \quad (4)$$

This process repeats until no reduction can be made.

3.3 Lambda-term compilation

First of all, program finds the innermost lambda-expression, then checks the internal expression and determines the rule (2) to apply. Then the lambda-term is replaced by the equivalent combinatory expression and the process repeats. After all conversions are made, process goes to the outer level and continues compilation.

Experiments with the program showed, that too big expressions (with length of resulting combinators near 100000 symbols) are compiled very slowly. To improve performance we decided to replace constant combinators (that will never be changed up to the end of calculation) with numeric substitutors. When program encounters completely constant expression, it searches it's cache to find existing number for this constant. In case it finds required combinator, number is inserted into expression on the place of the entire constant. If constant is not found in the cache, it is appended to the end of cache. This optimization greatly speeds up the whole process. Example of such calculation is shown below:

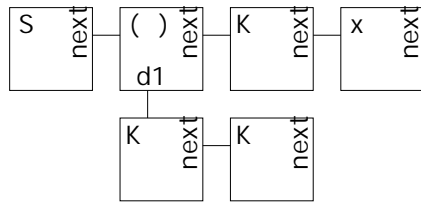
$$\begin{aligned}
& (\lambda v.(\lambda o.(\lambda i.S(\lambda d.od)(\lambda d.d)))) = \\
& = (\lambda v.(\lambda o.(\lambda i.S(S(\lambda d.o)(\lambda d.d))(\lambda d.d)))) = \\
& = (\lambda v.(\lambda o.(\lambda i.S(S(Ko)(\lambda d.d))(\lambda d.d)))) = \\
& = (\lambda v.(\lambda o.(\lambda i.S(S(Ko)I)(\lambda d.d)))) = \\
& = (\lambda v.(\lambda o.(\lambda i.S(S(Ko)I)I))) = \\
& = (\lambda v.(\lambda o.K(S(S(Ko)I)I))) = \\
& = (\lambda v.S(\lambda o.K)(\lambda o.S(S(Ko)I)I)) = \\
& = (\lambda v.S(KK)(\lambda o.S(S(Ko)I)I)) = \\
& = (\lambda v.S\{00000\}(S(\lambda o.S(S(Ko)I)) \mapsto \\
& \quad \mapsto (\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S(\lambda o.S) \mapsto \\
& \quad \mapsto (\lambda o.S(Ko)I))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S(KS)(\lambda o.S(Ko)I)) \mapsto \\
& \quad \mapsto (\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S(\lambda o.S(Ko)) \mapsto \\
& \quad \mapsto (\lambda o.I)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S(S(\lambda o.S) \mapsto \\
& \quad \mapsto (\lambda o.Ko))(\lambda o.I)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S(S(KS)(\lambda o.Ko)) \mapsto \\
& \quad \mapsto (\lambda o.I)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S(S\{00001\} \mapsto \\
& \quad \mapsto (S(\lambda o.K)(\lambda o.o)))(\lambda o.I)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S(S\{00001\} \mapsto \\
& \quad \mapsto (S(KK)(\lambda o.o)))(\lambda o.I)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S(S\{00001\} \mapsto \\
& \quad \mapsto (S\{00000\}I)))(\lambda o.I)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S(S\{00001\}(S\{00003\} \mapsto \\
& \quad \mapsto (KI)))(\lambda o.I))) = \\
& = (\lambda v.S\{00000\}(S\{00006\}(KI))) = \\
& = K(S\{00000\}\{00007\})
\end{aligned}$$

Resulting constant cache after this calculation:

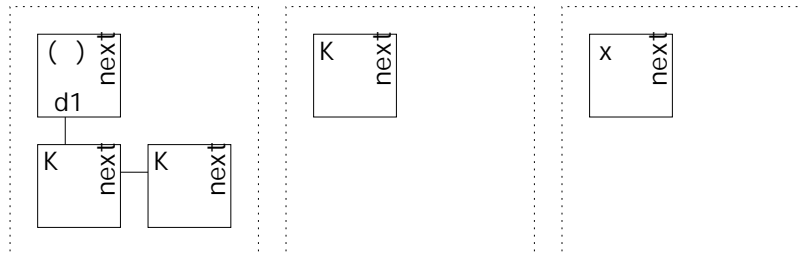
$$\begin{aligned}
\{00000\} & = KK \\
\{00001\} & = KS \\
\{00002\} & = S\{00000\}I \\
\{00003\} & = S\{00001\}\{00002\} \\
\{00004\} & = KI \\
\{00005\} & = S\{00003\}\{00004\} \\
\{00006\} & = S\{00001\}\{00005\} \\
\{00007\} & = S\{00006\}\{00004\} \\
\{00008\} & = S\{00000\}\{00007\}
\end{aligned}$$

And finally, these constants are inserted to the resulting combinator:

original structure:



fetching 3 arguments for S:



creating new brackets object, making copy of x and linking objects in th

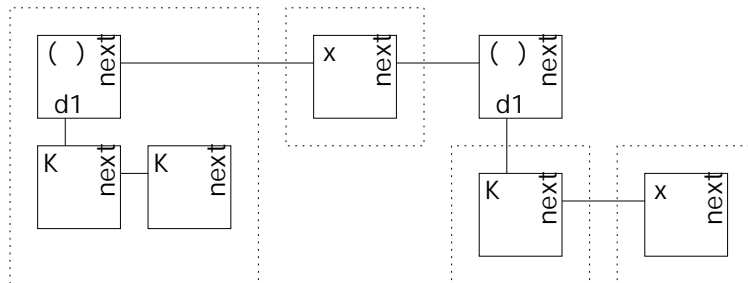


Figure 4: Reduction of the *S* combinator

$$\begin{aligned}
& K(S\{00000\}\{00007\}) = \\
& = K(S(KK)\{00007\}) = \\
& = K(S(KK)(S\{00006\}\{00004\})) = \\
& = K(S(KK)(S(S\{00001\}\{00005\})\{00004\})) = \\
& = K(S(KK)(S(S\{00001\}\{00005\})(KI))) = \\
& = K(S(KK)(S(S(KS)\{00005\})(KI))) = \\
& = K(S(KK)(S(S(KS)(S\{00003\} \mapsto \\
& \quad \mapsto \{00004\}))(KI))) = \\
& = K(S(KK)(S(S(KS)(S(S\{00001\} \mapsto \\
& \quad \mapsto \{00002\})\{00004\}))(KI))) = \\
& = K(S(KK)(S(S(KS)(S(S\{00001\} \mapsto \\
& \quad \mapsto \{00002\}))(KI))(KI))) = \\
& = K(S(KK)(S(S(KS)(S(S(KS)\{00002\} \mapsto \\
& \quad \mapsto (KI))(KI))) = \\
& = K(S(KK)(S(S(KS)(S(S(KS) \mapsto \\
& \quad \mapsto (S\{00000\}I))(KI))(KI))) = \\
& = K(S(KK)(S(S(KS)(S(S(KS) \mapsto \\
& \quad \mapsto (S(KK)I))(KI))(KI)))
\end{aligned}$$

4 Task generation

To test students ability to perform lambda-conversion and compilation, program can dynamically generate tasks for them. The program uses dictionary and randomly chooses a word with the quite small length (3 – 4 letters) and searches the word, containing all letters, that present in the first one. For example:

λpage.peg
λrudy.dry
λtack.act
λbye.eye
λlink.ink

5 Applications

Algorithms implemented in this program can be successfully used in educational purposes (to teach and test students on the subject of combinatory logic), in scientific ones (as an auxiliary tool to discover various properties of combinators and lambda-terms) and in programming (to perform automatic optimization of programs, remaining the equivalence to the source code).

6 Conclusion

In this paper the main principles of building parser, compiler and interpreter of simple applicative language were shown. New concept of optimization was put forward. In other works, for example in Bologna Optimal Higher-order Machine [3] optimization is achieved via sharing memory between several lambda-expressions,

manipulating with pointers and usage counters. This approach is junctioned with garbage creation and its subsequent collection. Method presented in this paper involves static tables with all frequent constant combinatory expressions being enumerated. And they are inserted to the syntax tree as atoms. It greatly improves performance of the compiler and interpreter.

This program is available at <http://www.rulezz.ru/cl/compiling/ceval.tar.gz>.

7 Appendix

Syntax parser and reduction algorithms were implemented in C language. CGI interface and task generation were written in Perl. Computer involved for calculations was Pentium III-555, 128 Mb RAM.

References

- [1] Wolfengagen V. E., Goltseva L. V. Applicative calculations on the basis of combinators and λ -calculus. Moscow, 1992
- [2] Wolfengagen V. E. Combinatory logic in programming. Moscow, 1994
- [3] Andrea Asperti, Cecilia Giovannetti, Andrea Naletto The Bologna Optimal Higher-order Machine, Department of Computer Science, University of Bologna, Bologna, 1995
- [4] Manfred Schmidt-Schauß and Michael Huber, Fachbereich Informatik, Frankfurt, 2000