Rewriting with Layout

M.G.J. van den Brand
Mark.van.den.Brand@cwi.nl
CWI
Kruislaan 413, NL-1098 SJ
Amsterdam, The Netherlands

J.J. Vinju
Jurgen.Vinju@cwi.nl
CWI
Kruislaan 413, NL-1098 SJ
Amsterdam, The Netherlands

15th December 2000

Abstract

Rewriting technology has proved to be an adequate and powerful mechanism to perform source code transformations. These transformations can not only be efficiently implemented using rewriting technology, but it also provides a firmer grip on the source code syntax. However, an important shortcoming of rewriting technology is that source code comments and layout are lost during rewriting. We propose "rewriting with layout" to solve this problem. We present a rewriting algorithm that keeps the layout of sub-terms that are not rewritten, and reuses the layout occurring in the right-hand side of the rewrite rules.

1 Introduction

Rewriting technology has proved to be an adequate and powerful mechanism to tackle all kinds of problems in the field of software renovation. Software renovation is to bring existing source code up to date with new requirements. One of the techniques applied in this field is source code transformation. Source code transformations are simple syntactic transformations which are mainly implemented using string replacement technology.

Such transformations can also conveniently be implemented using rewriting technology. Using rewriting technology is safer because it provides a firmer grip on source code syntax. An important shortcoming of rewriting technology is that source code comments and layout are lost during rewriting. It is possible to retrieve comments and layout, but this complicates the specification of such a source code transformation tool because every rule has to take comments and layout explicitly into consideration. This is unlike string replacement technology, where comments are just strings like any other part of the source code. We propose *rewriting with layout* to solve this problem, and try to make rewriting technology an attractive alternative to conventional software maintenance tooling.

We present a rewriting algorithm that retains the layout[1] of sub-terms that are not rewritten and reuses the layout occurring in the right-hand side of the rewrite rules. Even sub-terms that are reused in the normal form will have their original layout. However a certain amount of layout is still lost when a rule is applied to a certain node in the tree. We offer some ideas to solve this when discussing future work.

1.1 Source code transformations

Maintenance programmers frequently use string replacement tools to automate their software maintenance tasks. For example, they use the regular expressions available in scripting languages like Perl [14] to perform all kinds of syntactical transformations. Naturally, such source code transformations must be precise. But regular string matching alone is not powerful enough to recognize all kinds of syntactical structures commonly found in programming languages. This lack of power is usually solved by extensive explicit programming, or not at all. Some string replacement languages, like SNOBOL [11], would provide the maintenance programmer with low level context-free matching. But these powerful languages are hardly used. Probably because they are too "low level" to offer practical solutions to software renovation problems.

There is quite a difference between source code transformations on one hand and general program transformations [13] on the other hand. Source code transformations deal with automatic syntactical transformations. They automate software maintenance tasks, but they do not automate any correctness proof of the adaptations to the source code. Whereas general program transformations usually require user interaction and involve proving that the transformations are sound and complete.

As opposed to string rewriting, term rewriting technology is a different approach to implement source code transformations. The source code is fully parsed given the context-free grammar of the language, the term representation is transformed according to a set of powerful rules and the result is unparsed to obtain source code again. We use an algebraic specification formalism, ASF+SDF [10], based on term rewriting. Due to recent improvements of its compilation techniques [5] and term representation [4], ASF+SDF can now be applied to *industry sized* problems [6]. For example, COBOL renovation factories have been implemented and used [8].

The rewrite rules in ASF+SDF use concrete syntax. They are defined on the actual syntax of the source language, not some abstract representation. Because the rules are applied to structured terms instead of strings, complex syntactical structures can be grasped by a single rule. A very important feature of rewrite rules is that they abstract from the arbitrary layout of the source code, ignoring even source code comments. This is all in favor of simplicity. It can be expected that software maintenance tooling can be developed with more confidence and less effort using such rewriting technology. But first the rather practical issue

[1] We will use the term "layout" to indicate the layout as well as source code comments.

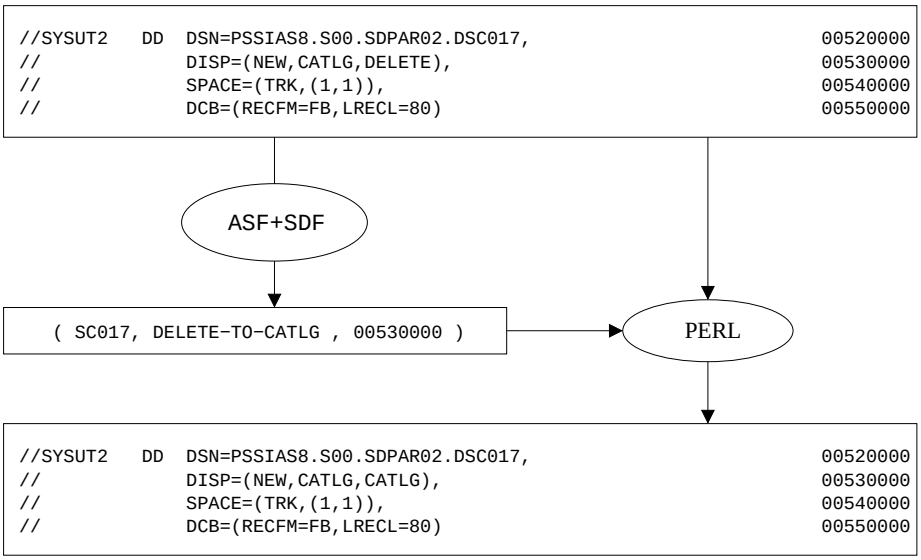

Figure 1: Sample input and output of a source code transformation in JCL. The `DELETE` keyword is be replaced by `CATLG`, but in a specific context.

of loosing the layout needs to be solved. For no maintenance programmer will consider using rewriting technology and loose all of his source code comments during a large maintenance job.

1.2 Example

The following example of a source code transformation shows the importance of considering layout while rewriting. This example is a part of a recent reverse engineering project of JCL scripts[2] in cooperation with a Dutch software house.

The interpretation of JCL scripts is sensitive to their particular layout, which is an unfortunate but not uncommon language property. There are numerous examples of languages that depend on layout, e.g. COBOL and Haskell. As an adequate solution to this problem, the source code transformation was performed in two steps:

1. A rewriting language, ASF+SDF, was used to reduce a JCL script to a list of instructions that indicate precisely where to modify the script.

2. The list of instructions was interpreted by a Perl script that used regular string replacements to implement them.

The above situation is depicted in Figure 1 for a specific JCL instruction. Obviously, this effective combination of term rewriting and string replacement is

[2]JCL stands for Job Control Language and is mainly used in combination with COBOL programs on IBM mainframes.

neither an attractive nor a generic solution to the problem of transforming layout sensitive source code. It would be preferable to encode the entire transformation in the rewriting language.

1.3 Overview

To explore the subject, we have developed an ASF+SDF interpreter that retains layout. Apart from the actual rewriting algorithm, there are two important prerequisites to the idea of a layout preserving rewriter. Firstly, the parser should produce trees in which the layout is preserved in some way. Secondly, rewriting must be performed on a term representation that also contains all layout, for instance the parse tree directly.

In Section 2 we will introduce the term format. In Section 3 we discuss our layout preserving algorithm for ASF+SDF. Section 4 describes some benchmark figures. We compare the performance of the layout preserving rewriter with the original ASF+SDF rewriter, and with two other interpreted rewriting systems: ELAN [3] and Maude [9]. Finally, in Section 5 we draw some conclusions and present directions for future work.

2 Term format

One of the features of rewriting technology is that it automatically abstracts from layout. To implement this abstraction, usually layout information is just not included in the term format. So, it is common practice to have a very concise tree representation to represent terms that have to be rewritten. Typical examples of these concise formats are REF [2] used within the ELAN system [3], and μASF used within the ASF+SDF compiler [5].

We have been exploring the other direction: using parse trees as term format for rewriting. These parse trees contain all information encountered during parsing, e.g. layout, keywords, application of syntax production rules, etc. Although all this information seems redundant for rewriting itself, it is of importance to the entire transformation process from input to output. In the following two sections we describe our parse trees and the generic term data-type it is based on briefly.

2.1 ATERM data-type

Our representation of parse trees is based on a generic abstract data-type called ATERM [4]. The corresponding libraries for this ATERM format have a number of important properties. One of the most important properties is that the ATERM library ensures a maximal sharing of terms, each term is unique. This property results in a memory and execution time efficient run-time behavior. Maximal sharing proofs to be especially beneficial when applied to our parse tree format, which is rather compressible.

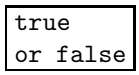

Figure 2: A term over the Booleans. An ASF+SDF specification of the Booleans can be found in Figure 4.

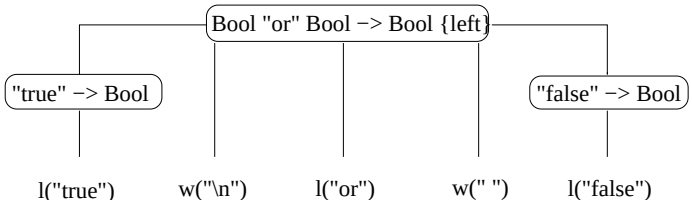

Figure 3: Parse tree of the term in Figure 2. Applications of productions are boxed. There are layout leafs (`w`) and lexical leafs (`l`).

A striking consequence of the maximal sharing is that term equality can be implemented as pointer equality. A negative effect of sharing is that the ATERM library allows only a functional manipulation of terms. This means that destructive updates on terms can only be implemented by rebuilding the updated term from scratch. But the ATERM library is time efficient nevertheless (see Section 4).

2.2 Parse trees

Based on the ATERM format we are able to define a simple format to represent trees. A parse tree is a collection of nodes consisting of an application of a production rule (context-free or lexical syntax rule) to a number of arguments. An argument is a node representing an application of another production rule, a lexical symbol, a variable, a literal (keyword), or layout. Except for the applications all other nodes are leafs of the parse tree.

There is a picture of a parse tree in Figure 3. Parse trees are built in such a way that for a left to right depth first traversal of the leafs every two contiguous lexical nodes have a layout node in between. In this manner, all possible internal layout is encoded in the parse tree.

3 Rewriting

Rewriting technology is an adequate technology to implement source code transformations. Losing the original layout including comments is a severe drawback. We will discuss an adaptation of the rewriter used within the ASF+SDF Meta-Environment [12, 7] which preserves as much layout as possible. First we will briefly introduce the reader to the semantics of our rewriting formalism ASF [1]. Then we discuss the interpreter and the adapted interpreter in detail.

```
module Booleans

imports Layout

exports
  sorts Bool
  context-free syntax
    "true"          -> Bool
    "false"         -> Bool
    Bool "or" Bool -> Bool {left}
  variables
    "Bool"[0-9]* -> Bool

equations
 [or-1] true or Bool = true
 [or-2] false or Bool = Bool
```

Figure 4: Simple equations for `or`-operator

3.1 ASF

ASF is based on an innermost reduction strategy over terms. Rules are expressed as *conditional equations*. The left- and right-hand side of equations are terms in *concrete* syntax (user-defined syntax), augmented with sorted variables. The right-hand side of an equation introduces no new variables (or uninstantiated variables), the left-hand side of an equation may not consist of a single variable. Equations can be marked as *default*. These equations are only applied if no other equations could successfully be applied.

Equations can have conditions. There are positive and negative conditions. Conditions are evaluated one after the other in a fixed order. Only if the evaluation of a condition is successful the rest of the conditions is evaluated. A positive condition checks the syntactical equality of the left- and right-hand side after normalization of both sides. One side of a positive condition may introduce new variables, this side is not normalized but matched with the term of the normalized side. A negative condition checks the syntactical inequality of both sides after normalization. No new variables may be introduced in negative conditions. An equation can be applied if the left-hand side matches and *all* conditions are successfully evaluated.

Another characteristic feature of ASF is list matching. List matching (also called associative matching) enables the specification writer to manipulate elements of a list in a concise manner. The manipulation of the list elements is performed via so-called list patterns, in such a list pattern the individual list elements can be addressed or sublists can be matched via list variables. List matching may involve backtracking. However, the backtracking is restricted to the scope of the rewrite rule in which the list pattern occurs. The possible matches are strictly ordered to enforce deterministic and finite behavior.

```
module BoolList

imports Booleans
exports
  sort List
  context-free syntax
    "[" {Bool ";"}* "]" -> List
  variables
    "Bool*"[0-9]* -> {Bool ";"}*

equations
 [set-1] [ Bool*1 ; Bool ; Bool*2 ; Bool ; Bool*3 ] =
           [ Bool*1 ; Bool ; Bool*2 ; Bool*3 ]
```

Figure 5: An extension of the Boolean syntax and an equation with list variables.

ASF+SDF supports two variants of lists: lists without separators and lists with separators. A '`*`' indicates zero or more elements and a '`+`' indicates that the list should contain at least one element. For example: `A*` is a list of zero or more elements of sort `A` and `{B ";"}+` represents a list of at least one element of sort `B`. The `B` elements are separated by semicolons.

An example of a very simple ASF+SDF specification is presented in Figure 4, the equations that define the semantics of the `or` operator are specified. Note that the equations are labelled with tags. These labels have no semantics. The ASF+SDF specification in Figure 5 demonstrates the use of list matching in an equation. This equation will remove all double occurring `Bool` terms from the `List`. For more elaborate ASF+SDF examples we refer to [10].

3.2 Normalization of terms

Given an ASF+SDF specification and some term to be normalized, this term can be rewritten by interpreting the ASF equations as rewrite rules. One approach is to compile these equations to C functions [5]. We do this to optimize *batch* performance of rewrite systems. On the other hand, a small interpreter facilitates *interactive* development of rewrite systems. To explore the subject of rewriting with layout, we have chosen to extend the ASF+SDF interpreter. First we will discuss an interpreter which ignores layout completely in both the term and the equations. Thereafter, we discuss the extension to an interpreter that retains layout in a specific manner.

The original ASF+SDF interpreter takes as input slightly modified parse trees of both the term and the set of equations. We explicitly do not use some abstract term representation. The modification consists of stripping the parse trees of both the equations and the term of all layout nodes. This is an easy and efficient implementation of abstraction from layout. The efficiency benefit is due to ATERM library on which the parse trees are built (Section 2.1). Without the

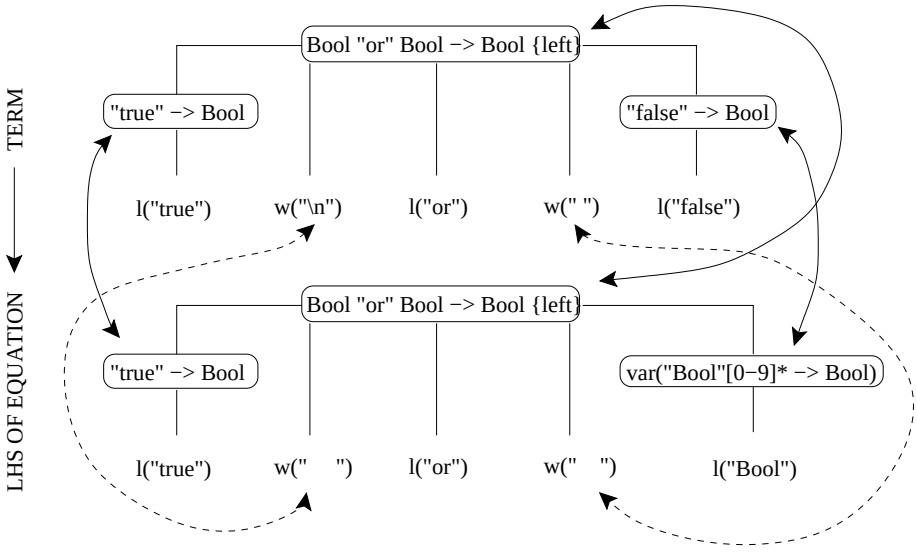

Figure 6: Matching the parsed term in Figure 2 to the parsed left-hand side of rule `[or-1]` in Figure 4. The dotted lines depict matches of layout leafs.

layout nodes and with maximal sharing, term equality can be decided by single pointer comparison.

The rewriter operates on these stripped parse trees as an ordinary rewriting engine. Based on the outermost function symbol of some sub-term the appropriate set of rewrite rules (with the same outermost function symbol in the left-hand side) is selected. If the left-hand side matches with respect to the arguments, variables are instantiated by this match. Then the conditions are evaluated one-by-one using the instantiated variables. Along with the evaluation of the conditions new variables are instantiated. Note that due to the innermost reduction strategy we know that the arguments of any sub-term will be in normal form. If the evaluation of *all* conditions is successful, the reduct is built by instantiating the variables in the right-hand side of an equation.

For the moment we are experimenting with a rather simple and straightforward rewriting technique with layout. We modified our rewriter in such a way that it is no longer necessary to strip the layout for the parse trees. Firstly, to implement abstraction from layout, term equality can no longer be implemented as pointer equality. An almost full traversal of both trees is now needed to decide term equality. In Section 4 we will show what performance penalty is paid now that we need to look deeper for equality modulo layout.

Secondly, the matching algorithm with layout is depicted in Figure 6. Two parse trees are matched by comparing their top node. If their pointers are equal, we can stop early. If only the productions are equal, the match continues recursively. Variable productions match any subtree of the same top sort and the match instantiates them immediately. Of course lexicals should always

be exactly equal. But any two compared layout nodes always match, which implements abstraction from layout.

Thirdly, the layout occurring in the right-hand side of a successful equation is just left in the normal form. Which effectively means that it is inserted in the reduct. In this way the specification writer can influence the layout of the constructed normal form by formatting the right-hand side of an equation manually. A negative consequence of this approach is that all rewritten terms will loose their original layout. But the values of the instantiated variables (subterms) do retain their original layout. Ideas to possibly save even more layout from the original term will be offered in Section 5.

3.3 List matching

The interpreter implements list matching by means of backtracking. Given a term representing a list and a list pattern all possible matches are tried one after the other until the first successful match including a successful evaluation of all conditions. Backtracking takes only place if more than one list variable occurs in the list pattern.

List matching also needs some adaptation when dealing with layout. There are layout nodes between every consecutive element (or separator) in a list. When constructing a sublist to instantiate a list variable these layout nodes have to be incorporated as well.

Some special care must be taken when constructing a term containing a list variable. If such list variable is instantiated with a sublist consisting of zero elements, the layout occurring before and/or after this list variable must be adapted to ensure the resulting term is well formed with respect to layout again.

Suppose we want to normalize the term `[true; true]` given the specification presented in Figure 5. The left-hand side of rule `set-1` matches with this term resulting in the following variable substitutions: *Bool*1* = ε, *Bool* = `true`, *Bool*2* = ε, *Bool* = `true`, and *Bool*3* = ε, where ε represents the empty list. A naive substitution of the variables in the right-hand side of `set-1` would result in: `[ ; true ; ; ]`. The interpreter checks whether a list variable represents an empty list and does not include the redundant separators and layout. In our example the resulting term will be `[true]`.

3.4 Discussion

The impact of introducing rewriting with layout in our interpreter was rather small. The pre- and post-processing of the parse trees for the equations and term had to be adapted. The matching algorithm was modified to implement abstraction from layout.

The impact on the efficiency of the interpreter will be discussed in Section 4, given a few benchmarks the performance of our interpreters with and without layout preserving are compared. In order to get an impression of the overall

performance we have considered the interpreters of ELAN [3] and Maude [9] as well.

4 Performance

How much does this rewriting with layout cost? There are three issues which may have a negative influence on the performance:

- Matching becomes more expensive because the layout nodes are still in the tree and have to be matched.
- Matching is more expensive because all terms have to be fully inspected.

In order to get insight in the *relative* performance of rewriting with layout we compare the time and memory usage of the classical ASF interpreter and the layout preserving interpreter. Furthermore, we have run the benchmarks on interpreters of other rule based systems, like ELAN [3] and Maude [9] as well to provide the reader with a better context. Note that for higher execution speed both the ELAN system and the ASF+SDF Meta-Environment also provide compilers [3].

We have used two simple benchmarks based on the symbolic evaluation of expressions 2^n **mod** 17: `evalsym` and `evaltree`. These benchmarks have been used before for the analysis of compiled rewriting systems in [5]. We reuse them for they isolate the core rewriting algorithm from other language features. All measurements have been performed on a 450 MHz Intel Pentium III with 256 MB of memory and a 500 MB swap disk.

The `evalsym` benchmark The `evalsym` benchmarks computes 2^n **mod** 17 in a memory efficient manner. Its memory complexity is in $O(1)$ for all benchmarked interpreters. From this benchmark we obviously try to learn what the consequences of rewriting with layout are for time efficiency.

The results for this benchmark are in Figure 7. The different implementations of rewrite systems all show the same time and memory complexity behavior. We pay a structural 50% time penalty for rewriting with layout. But this does not change the relative speed to the other systems much. The ASF+SDF system still runs about as fast as the ELAN system.

The `evaltree` benchmark The `evaltree` algorithm generates a huge amount of terms. Real world source code transformations usually involve enormous terms and therefor scaling up is an important aspect to source code transformation. So in this benchmark we focus on the space complexity behavior of rewriting with layout.

The results for this benchmark are in Figures 8 and 9. The ASF+SDF system uses a constant amount of memory, while the other systems show exponential

[3] See [5] for details on compilation of ASF+SDF

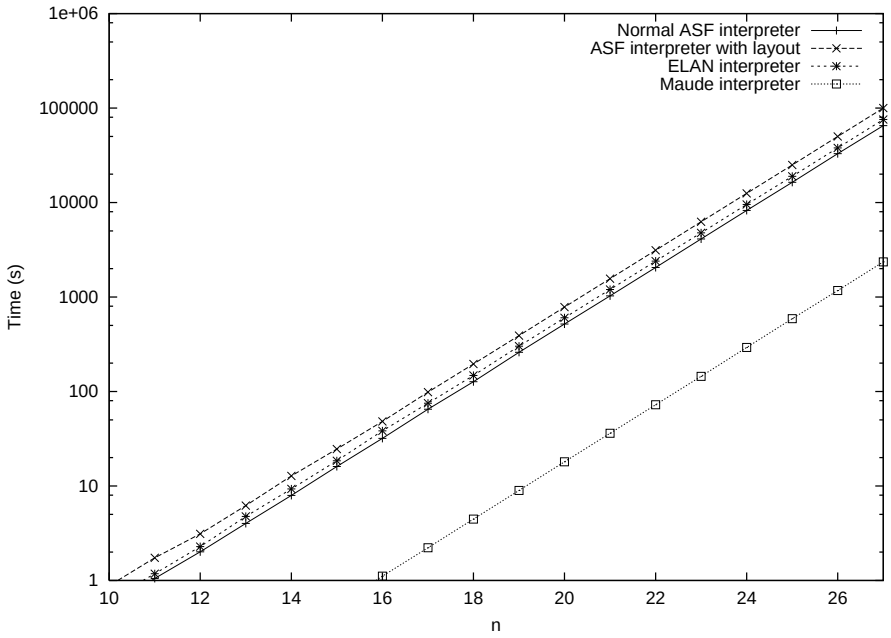

Figure 7: Timing results for the `evalsym` benchmark.

growth in memory usage. This is due to maximal sharing. Obviously, any extra memory allocated for layout is insignificant. Again, we pay a structural 50% time penalty for reducing with layout and the relative speed is not affected significantly.

5 Conclusions and future work

We presented an adaptation of the standard ASF+SDF interpreter in order to deal with rewriting with layout. This form of rewriting proves to be very useful with the field of reverse engineering and more specific in the realm of source code transformations. We showed that due to the fact that our interpreter is based on an intermediate format to represent terms and equations, which is very close to parse trees, only a very small modification was needed to obtain this functionality. This intermediate format is based on the ATERM library which provides maximal sharing.

The only concern is the computational overhead, because by keeping the layout information in the intermediate representation the advantage of maximal sharing is lost. By means of some small benchmarks we showed that rewriting with layout is about 50% slower than rewriting without layout, but the amount of used memory is not affected by this modification.

At the moment we have performed only a very straightforward modification

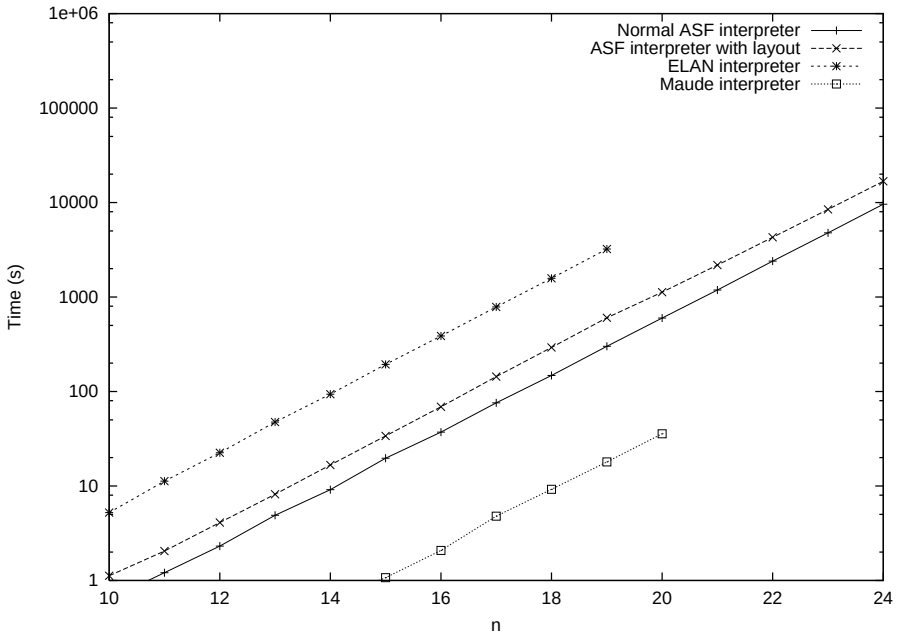

Figure 8: Timing results for the `evaltree` benchmark. The lines stop where the system runs out of memory.

of the rewriter. Given this framework it is interesting to perform experiments with matching on layout and/or reusing the layout occurring in a term in the right-hand side of a rule. In Figure 10 we give an example of how matching on layout could be specified in ASF+SDF.

Finally, compilation of rewriting with layout is needed to bring it to the industrial application domain. The runtime term format of compiled rewriting systems need to have layout encoded. And the compilation scheme needs to generate code to implement abstraction from layout.

References

[1] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[2] P. Borovanský, S. Jamoussi, P.-E. Moreau, and Ch. Ringeissen. Handling ELAN Rewrite Programs via an Exchange Format. In *Proc. of [?]*, 1998.

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *1st Intl. Workshop on Rewriting Logic and its Applica-*

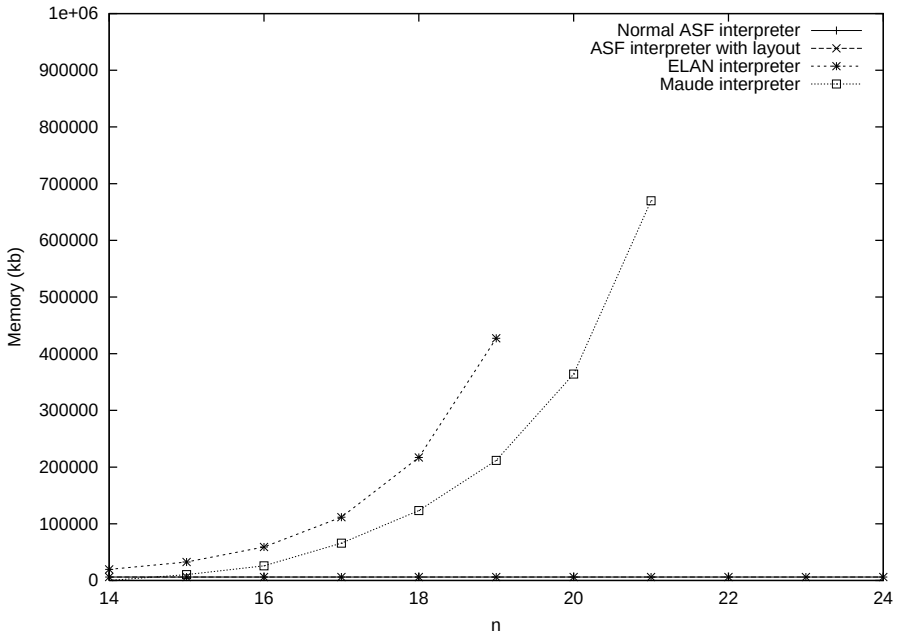

Figure 9: Memory profiling results for the `evaltree` benchmark. The lines stop where the system runs out of memory.

tions, Electronic Notes in Theoretical Computer Science. Elsevier Sciences, 1996.

[4] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[5] M.G.J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC '99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[6] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications, WRLA 98*, 1998.

[7] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and implementation of a new asf+sdf meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, 1997. Springer/British Computer Society.

```
module Booleans

imports Layout

exports
  sorts Bool
  context-free syntax
    "true"              -> Bool
    "false"             -> Bool
    Bool "or" Bool      -> Bool {left}
    LAYOUT "++" LAYOUT -> LAYOUT {left}
  variables
    "Bool"[0-9]* -> Bool
    "Ws"[0-9]*   -> LAYOUT

equations
 [or-1] true Ws1 or Ws2 Bool = true Ws1++Ws2
 [or-2] false Ws1 or Ws2 Bool = Ws1++Ws2 Bool
```

Figure 10: Simple equations for `or`-operator with matching on layout. The user-defined `++` operator concatenates layout.

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36:209–266, 2000.

[9] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *1st Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Sciences, 1996.

[10] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[11] J.F. Gimpel. *Algorithms in SNOBOL4*. John Wiley & Sons, 1976.

[12] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[13] H.A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer-Verlag, 1990.

[14] L. Wall. *Practical Extraction and Report Language*. O' Reilly. `http://www.perl.com/pub/doc/manual/html/pod/perl.html`.