Universität Karlsruhe (TH)
Institut für
Programmstrukturen
und Datenorganisation
Lehrstuhl Professor Goos

A Modification to BURS in Codegeneration

Tech Report Nr. 2003-12

ISSN 1432-7864

Boris J. H. Boesler

boesler@ipd.info.uni-karlsruhe.de

June 30, 2003

Contents

Chapter 1

Introduction

In compiler construction, the code generation is a very difficult task. Therefore it is divided into the three sub tasks: code selection, instruction scheduling and register allocation. Finding the optimal solution for each task is NP-complete. Several efficient techniques were developed to reduce the amount of calculations to generate code. For code selection, there are two commonly used techniques: Bottom Up Pattern Matcher (BUPM) and Bottom Up Rewrite Systems (BURS)[9]. A Bottom Up Rewrite System is more powerful than a Bottom Up Pattern Matcher. Variables can be used in patterns to match any term and a term can be rewritten by another term. For instance the commutativity of the +-operator can be formalized by a rewrite rule: $+(X, Y) \to +(Y, X)$.

In chapter 3, we give a short introduction of BURS and show what problems arise, in chapter 4 we present a modification to solve these problems. The results are presented in chapter 5.

Chapter 2

Related Work

The origin of current code-generation techniques are LR and tree grammars. Graham and Glanville introduced LR grammars in [11], while tree grammars are used in a big variety [3][1][5]. Pelegrí-Llopart[9] and Emmelmann[4] introduced term rewrite systems to generate code. [9] was reformulated by Nymeyer and Katoen[7][6][8] to give some intuitive definitions of the complex theory of BURS. In addition they divided BURS into two parts: the calculation of *all* rewrite sequences and searching for the most inexpensive one. We use and extend their work.

Chapter 3

Introduction of BURS

First we give a short introduction of BURS. Then we explain the problems we encountered.

3.1 BURS in short

A BURS is have a *costed rewrite system* $((\Sigma, V), R, C)$ with an alphabet Σ, a finite set of variables V, a finite set R of rules and a cost function C. Such a system rewrites a term t. Every sub-term in t can be identified by its position $p \in Pos(t)$, denoted $t|_p$. A rewrite step is the application of a rewrite rule r to a sub-term of t at position p ($\langle r, t|p \rangle$). A rewrite sequence $S(t)$ is a sequence of rewrite rules which are applied to a term t. A *local rewrite sequence* $L(t|_p)$ is a label for the sub-term $t|_p$, where the local rewrite sequence will be applied. This does not define that a rule of such a local rewrite sequence rewrites $t|_p$ itself, it is sufficient if it rewrites any sub-term of $t|_p$. In a *decoration* $D(t)$ each sub-term in t is labeled. The sequence $S_D(t)$ for a decoration $D(t)$ is defined by a post-fix order of labels. If $S_D(t)$ can rewrite a term t for some goal term g then the *inputs of a decoration* $I_D(t)$ is defined as

$$I_D(t) = \begin{cases} t & \text{if} \quad t \in \Sigma_0 \\ a(t'_1, \ldots, t'_n) & \text{if} \quad t = a(t_1, \ldots, t_n) \end{cases}$$

where $I_D(t_i) \overset{L_D(t_i)}{\Longrightarrow} t'_i, \quad$ for $1 \leq i \leq n$. The *outputs of a decoration* $O_D(t)$ are defined as $O_D(t) = t'$ with $I_D(t) \overset{L_D(t)}{\Longrightarrow} t'$.

The amount of possible rewrite sequences can be reduced by eliminating redundant rewrite sequences. Two decorations $D(t)$ and $D'(t)$ are equivalent, $D(t) \equiv D'(t)$, if they are permutations of each other. In a *normal-form decoration* $NF(t)$ for the term t every local rewrite sequence has rewrite steps of the form $\langle r, \epsilon \rangle$. This means that every sub-term $t|_p$ is labeled with rewrite steps which will be applied to it - they can not be moved to a sub-term of $t|_p$. In a *strong normal-form* $SNF(t)$, t is in normal-form and all rewrite steps with variables are at the latest possible position in a sequence. The normal-form and the strong normal-form are used to reduce the number of rewrite sequences. See [7] for more details.

3.2 Number of Rewrite-Sequences

The algorithm in [7] generates decorations in two passes:

1. generate *all possible* rewrite sequences for *all possible* goal terms, then
2. use the *given* goal term to delete all rewrite sequences, which are not required (*trimming*).

Example 3.1 *For the expression* $+(c, c)$ *and the rules* $r_1 : c \to b$ *and* $r_2 : +(b, b) \to b$*, each leaf has the rewrite sequences* $\{\langle c, \epsilon, c\rangle, \langle c, r_1, b\rangle\}$ *and the* $+$ *operator has five rewrite sequences* $\{\langle +(c, c), \epsilon, +(c, c)\rangle, \langle +(b, c), \epsilon, +(b, c)\rangle, \langle +(c, b), \epsilon, +(c, b)\rangle, \langle +(b, b), \epsilon, +(b, b)\rangle$ *and* $\langle +(b, b), r_2, b\rangle\}$ *before trimming.*

Only three rewrite sequences in the example are useful. The other six are deleted by trimming after all rewrite sequences for all operators have been calculated. We will give a lower bound and an approximation of an upper bound for the number of rewrite sequences before trimming for a term.

Example 3.1 shows, that the algorithm generates a lot more rewrite sequences before trimming, than are needed. So, we are interested in the number of rewrite sequences before trimming. (This is the maximum size of set $W(t)$ in algorithm in [7].) We can give a lower bound and an approximation of the upper bound.

3.3 Lower Bound

We can give a lower bound for the number of rewrite sequences.

Theorem 1 *The lower bound of number of rewrite sequences for a term is defined by the following situation, which is an optimistic situation.*

- *the term is a list of unary operators* o *and one leaf* l

- *there is only one rule that matches an operator o in the list and one rule that matches the leaf l.*

If the list has h nodes we can show that the algorithm produces $\Omega(h^2)$ rewrite sequences before trimming by calculating the number of rewrite sequences $t(h)$ for an operator o_h at height h and the two rules

- $r_1 : l \rightarrow u$
- $r_2 : o(u) \rightarrow u$

The algorithm generates two rewrite sequences for the leaf: $\langle l, \epsilon, l \rangle$ (a copy of the original term) and one rewrite sequence $\langle l, r_1, u \rangle$ for the matching rule r_1. For the operator o_h at height h in the list we find that it has $h+1$ rewrite sequences: h rewrite sequences of the form $\langle o_h(u_i), \epsilon, o_h(u_i) \rangle$, where u_i is the result of i-th rewrite sequence of the kid of o_h and one rewrite sequence $\langle o(u), r_2, u \rangle$ for the matching rule r_2.
Proof: For a leaf in the list $t(1) = 2$ as shown. At height h an operator has $t(h) = h + 1$ rewrite sequences. At height $h + 1$ with operator o_{h+1} the algorithm generates $t(h)$ rewrite sequences of the form $\langle o_{h+1}(u_i), \epsilon, o_{h+1}(u_i) \rangle, 1 \leq i \leq t(h)$, and one rewrite sequence $\langle o(u), r_2, u \rangle$. So the algorithm generates $t(h) + 1$ rewrite sequences at height $h + 1$.

$$t(h + 1) = t(h) + 1 = (h + 1) + 1$$

■

In general, $t(h) = h + 1$ for $h \geq 1$ rewrite sequences will be generated for every node. The sum $s(h)$ of all triples in the list is:

$$s(h) = \sum_{i=1}^{h} t(i) = \sum_{i=1}^{h} (i + 1) = \sum_{i=1}^{h} i + \sum_{i=1}^{h} 1 = \frac{h}{2}(h + 1) + h \in \Omega(h^2)$$

The generation of $\Omega(h^2)$ rewrite sequences requires at least time $\Omega(h^2)$. We can assume that the program representation is more complex than a list and that there is more than one rule per node that matches. Therefore $\Omega(h^s)$ is a lower bound.

3.4 Upper Bound

We can make an estimation of the number of rewrite sequences in general before trimming.

Theorem 2 *We can make some assumptions, which are reasonable:*

- *There are k rules that match an operator.*
- *The term is a m-ary tree.*
- *The term is a balanced tree.*

The number of rewrite sequences $t(h)$ of a node at height h is the number of all possible permutations of the m operands plus k for the matching rules.

$$t(h) = \begin{cases} 1+k & \text{if} \quad h = 1 (\text{see example 3.1}) \\ t(h-1)^m + k \geq (k+1)^{(m^{h-1})} & \text{if} \quad h > 1 \end{cases}$$

Proof: For a term with height $h = 1$, we get $t(1) \geq (k+1)^{(m^{1-1})} = (k+1)^{m^0} = (k+1)$. For height $h+1$, we find

$$\begin{aligned} t(h+1) &= t(h)^m + k \geq ((k+1)^{(m^{h-1})})^m + k = (k+1)^{(m*m^{h-1})} + k \\ &\geq (k+1)^{(m^h)} \end{aligned}$$

■

The number $s(h)$ of all rewrite sequences at height h is the sum $s(h-1)$ of m operands and the number of rewrite sequences at height h.

$$\begin{aligned} s(h) &= \sum_{i=1}^{m} s(h-1) + t(h) = m * s(h-1) + t(h) = \sum_{i=1}^{h} m^{h-i} t(i) \\ &\geq \sum_{i=1}^{h} m^{h-i} (k+1)^{(m^{i-1})} \end{aligned}$$

For a binary tree ($m = 2$), where only one rule matches each node ($k = 1$), we find $t(h) = 2^{(2^{(h-1)})}$ and $s(h) = \sum_{i=1}^{h} 2^{h-i}(2)^{(2^{i-1})} = \sum_{i=1}^{h} (2)^{(2^{i-1}+h-i)}$. In this simple example the algorithm uses at least an exponential amount of time and memory to calculate all rewrite sequences.

3.5 Analyzing

The algorithm in [7] is feasible, if the intermediate representation is in a tree-form and if the given term in this tree-form is a small expression. The time and memory consumption to generate code depends on the height of the given expression.

Chapter 4

Our Solution

We observe that most sequences are produced though there is no rule to match them. Our extension uses the set of given rules to calculate the set of *local goal terms* for each operator in the term. If a rewrite sequence produces a result which is not in this set, then the rewrite sequence is deleted.

For a given balanced binary tree ($m = 2$) and a rule, that matches at every node ($k = 1$), only one rewrite sequence will be left per node (the one that matches) before trimming. If we assume, that we have a binary balanced term with height $h = 6$, then there are 63 rewrite sequences only, instead of at least $t(6) = 211309439856 \geq 2^{32}$.

The idea to avoid the production of useless rewrite sequences (chapter 3.2) is to make sure, that the operands produce sequences which can be used by the operator. E.g., in example 3.1, there is a rule $+(b, b) \rightarrow b$. So each operand of a +-operator should have rewrite sequences which produce a term b and nothing else, because there is no rule to match any other results. We know all rules, so we can use this context to make some precalculations.

4.1 Basic Definitions

For term rewrite systems, we give the additional definitions:

Definition 3 (Operator) *The function* $op : T_\Sigma(V) \rightarrow \Sigma \cup V$ *returns the operator of the root of term* t*.*

Definition 4 (Local goal term) *A* local goal term *is a term, which is the result of a rewrite sequence, such that it can be used as an input term for a rewrite step.*

Definition 5 (Set of local goal terms) *Given a term rewrite system* $((\Sigma, V), R)$. *The* set of local goal terms lg(t) *for the term* t *is*

$$lg(t|_\epsilon) = \{t \in T_\Sigma(V) \mid t \textit{ is global goal term of the root}\}$$

For the term $t = a(t_1, \ldots, t_n)$*, the set of local goal terms for the operand* t_i *is defined by*

$$\begin{aligned} lg(t_i) \;&=\; \{rt_i \mid \forall\, a(rt_1, \ldots, rt_n) \to t' \in R \wedge 1 \le i \le n \wedge op(rt_i) \notin V\} \\ &\cup\; \{lt_i \mid \forall\, \hat{a}(lt_1, \ldots, lt_n) \in lg(t|_\epsilon) \wedge 1 \le i \le n \wedge op(lt_i) \notin V\} \\ &\cup\; \begin{cases} lg(t|_\epsilon) & \textit{if} \left\{ \begin{array}{l} \exists r : t' \to t'' \in R : op(t'|_\epsilon) = a \\ \wedge VP(t'') = \{\epsilon\} \wedge i \in VP(t') \wedge op(t''|_\epsilon) = op(t'_i) \end{array} \right. \\ \emptyset & \textit{else} \end{cases} \\ &\cup\; \begin{cases} \{t_i\} & \textit{if} \left\{ \begin{array}{l} \exists r : t' \to t'' \in R : op(t'|_\epsilon) = a \\ \wedge i \in VP(t') \wedge \nexists j \in VP(t'') : op(t'|_i) = op(t''|_j) \end{array} \right. \\ \emptyset & \textit{else} \end{cases} \\ &\cup\; \bigcup_j lg((t \xrightarrow{r_j} \hat{t}_j)|_{k''}) \forall r_j : t'_j \to t''_j \mid \begin{cases} op(t'_j) = a \\ \wedge \exists k'' \in VP(t''_j) \wedge k' \in VP(t'_j) : \\ op(t''_j|_{k''}) = op(t'_j|_{k'}) \\ \to lg(t''_j) = lg(t) \end{cases} \end{aligned}$$

The definition 5 has 6 parts. The first part defines the set of local goals for the root. The other 5 parts define the set of local goals for the operands t_i. We give short examples for these 5 cases:

1. for the term $t = +(c, c)$ and the rule $+(r, c) \to r$, r is passed as local goal to the left operand and c to the right operand of t

2. if $+(r, c) \in lg(t)$ then r is passed as a local goal term to the left operand and c to the right operand of t

3. for the term $t = +(c, c)$ and the rule $+(X, 0) \to X$, the set $lg(t)$ is passed as a set of local goals to the left operand of t, 0 is passed as a local goal to the right operand (case 1)

4. for the term $t = *(c, c)$ and the rule $*(X, 0) \to 0$, $c = t|_1$ is passed as local goal to the left operand of t, 0 is passed as a local goal to the right operand (case 1)

5. for the term $t = +(c, r)$ and the rules $r_1 : +(X, Y) \to +(Y, X), r_2 : +(r, c) \to r$ the term $\hat{t} = +(c, r)$ is created by applying the rule r_1 to t, the set of local goals of $\hat{t}|_\epsilon$ is set to $lg(t)$ and the local goals must be calculated, rule r_2 passes r as local goal to the left side and c to the right side of $\hat{t}$

Local goal terms are not limited to terms with just one single node. Definition 5 checks all rules to calculate local goal terms for the operands and then passes the operands of a local goal term as local goal terms to the operands, too. The rule $+(a, *(b, c)) \to d$ the generates a local goal term $*(b, c)$ for the second operand of the +-operator (case 1). The local goal terms b and c are passed down then by the case 2.

As a side effect, we can detect whether a term can be rewritten or not: if the set of local goal terms is empty, then the term can not be rewritten with the given rules.

With this definition, we reformulate the inputs of a decoration $I_D(t)$

Definition 6 *(Inputs of a decoration) Let $D(t) \in SNF(t)$ such that for some given goal term g, $t \overset{S_D(t)}{\Longrightarrow} g$. For each sub-term t' of t, the possible inputs, denoted $I_D(t')$, are defined as follows:*

$$I_D(t) = \begin{cases} t & \text{if} \ \ t \in \Sigma_0 \wedge t \in lg(t) \\ a(t'_1, \dots, t'_n) & \text{if} \ \ t = a(t_1, \dots, t_n) \wedge \\ & \quad\ \ t'_i \in lg(t_i) \ \text{for} \ 1 \le i \le n \end{cases}$$

where $I_D(t_i) \overset{L_D(t_i)}{\Longrightarrow} t'_i, \quad$ for $1 \le i \le n$.

The extension of the definition about the input of a decoration reduces the number of rewrite sequences tremendously, because now rewrite sequences are not generated for all possible goal terms, but for valid ones only.

But variables must appear as kids, they can not appear somewhere deep in a local goal term. E.g., it is impossible to apply the rule $r_1 : Conv(Add(X, Y)) \to Add(X, Y)$ to the tree $t = Conv(Const_{Val})$, because the positions for X and Y do not exist in t. At a first glance it might be enough to check if the positions exist, but that is not enough. If there is a rule $r_2 : Const_{val} -> Add(Const_{val-1}, Const_1)$, then the rules r_2 and r_1 could be applied, but there are no correct local goals for the $Const$ node. In such a case concrete nodes must replace the variables in the goal term.

In example 3.1, the set of local goal terms for the leaf c is $\{b\}$. The only rewrite sequence, that is not deleted at the leaf is $\langle c, r_1, b \rangle$. Consequently, the algorithm can produce two sequences only at the +-operator: $\langle +(b, b), \epsilon, +(b, b) \rangle$ and $\langle +(b, b), r_2, b \rangle$. The first one will be removed, if the term $+(b, b)$ is not a global goal term.

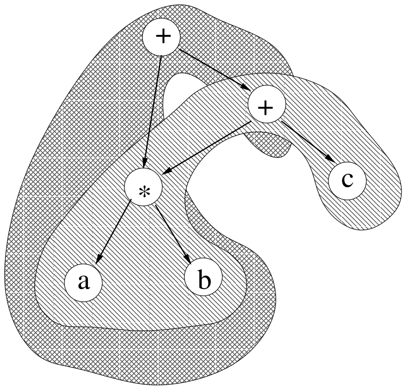

Figure 4.1: Common subexpressions matched by two patterns

4.2 Local Goal Terms in a DAG

For terms, the above term rewrite system is sufficient. The generation of local goal terms in a DAG for common subexpressions is not complex. An operator o of the term $o(t_1, \ldots, t_n)$ calculates the set of local goal terms $lg_o(t_i), 1 \leq i \leq n$, for its operand t_i. If F is the set of operators, which share t_i as common subexpression, we can define

$$lg(t) = \bigcap_{o \in F} lg_o(t)$$

By using the intersection, we assure that a common subexpression produces rewrite sequences, which can be used by all of its operators. Furthermore, this assures that we do not have to split graphs into trees. E.g., the example in [10] page 44f (see figure 4.1) can be solved as expected with two instructions, if there are the rules $a \rightarrow r, b \rightarrow r, c \rightarrow r$ and $+(*(r, r), r)$. This is possible because the rewrite sequence $\langle *(r, r), \epsilon, *(r, r)\rangle$ is returned by the $*$-operator. The rule for a multiply-and-add instruction can be applied twice for the common subexpression.

Chapter 5

Results

5.1 A small example

For the running example $t = +(0, +(c, c))$ in [7] we get the following numbers of rewrite sequences for every position p in t (see table 5.1). With local goals less rewrite sequences are generated.

position	¬trim	trim	local goals, ¬trim
$t\|_1$	4	3	4
$t\|_{2\cdot 1}$	3	3	3
$t\|_{2\cdot 2}$	3	3	3
$t\|_2$	21	6	6
$t\|_\epsilon$	137	3	3

Table 5.1: Number of local rewrite sequences in $t = +(0, +(c, c))$

5.2 More practical examples

We implemented the modified version of BURS and used it in the Java-compiler `jack`. It generates a *Static Single Assigment* (SSA) representation in graph-form[1]. Naturally these graphs have a larger number of nodes than an expression tree. Our BURS implementation rewrites the graph to low-level C-code.

We analyzed four Java programs[2]. 177 of the generated 213 methods have 35 nodes or less. All other methods have up to 312 nodes (see figure 5.1).

[1]Using BURS on a general graph form is not discussed here. See [2] for more details.
[2]Sieve.java, Queens.java, QuickSort.java, HeapSort.java

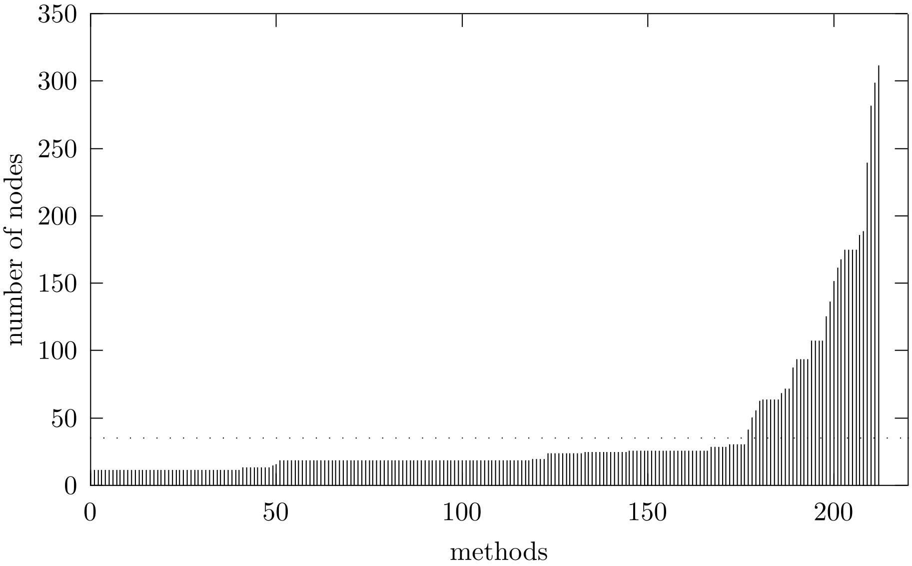

Figure 5.1: Number of nodes for 213 methods

213 methods were generated, mostly empty initializer methods. In 179 methods the maximal expression-height is less then 20. All other methods have a maximal height varying from 20 to 69 (see figure 5.2).

As explained in chapter 3.4 we found out that we would get approximately $2^{2^{70}}$ rewrite sequences in these cases.

As a consequence, we are interested in the number of rewrite sequences (see figure 5.3). Even for a graph with 312 nodes there are only 513 rewrite sequences after trimming. From prior experiments [2], we know that a graph with more than 30 nodes is too complex for code generation. With our extension we can handle essentially larger graphs.

Furthermore we compiled the Java version of a test-program (figure 5.4) with `jack`, the C version with `gcc` and compared the run-times. The run-times of the compiled C version does not vary very much, no matter what optimizations are used. In contrast, the run-time of the generated low-level C-code from the Java version depends on the optimizations used by the C compiler. The last two columns show, that our generated low-level C-code (compiled by `gcc`) performs better (table 5.2).

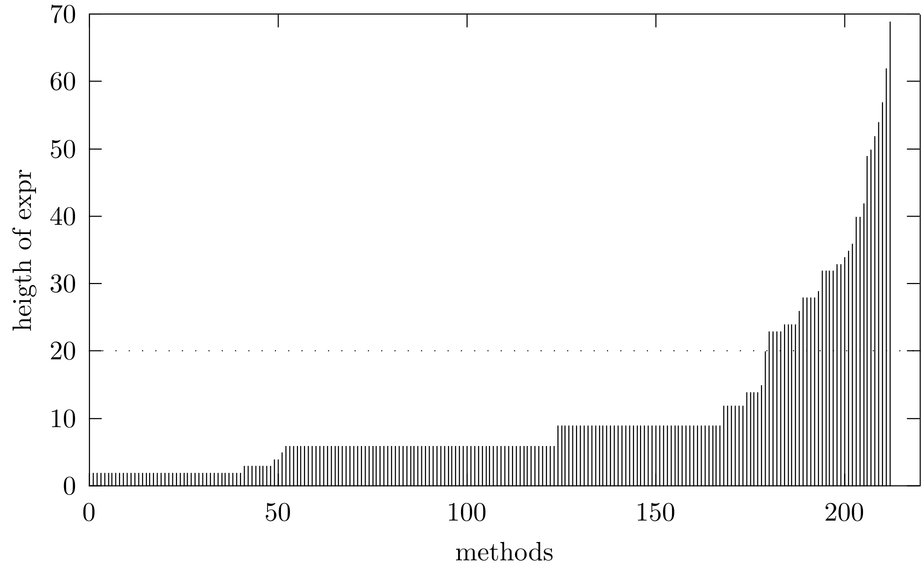

Figure 5.2: Maximal height of expressions for 213 methods

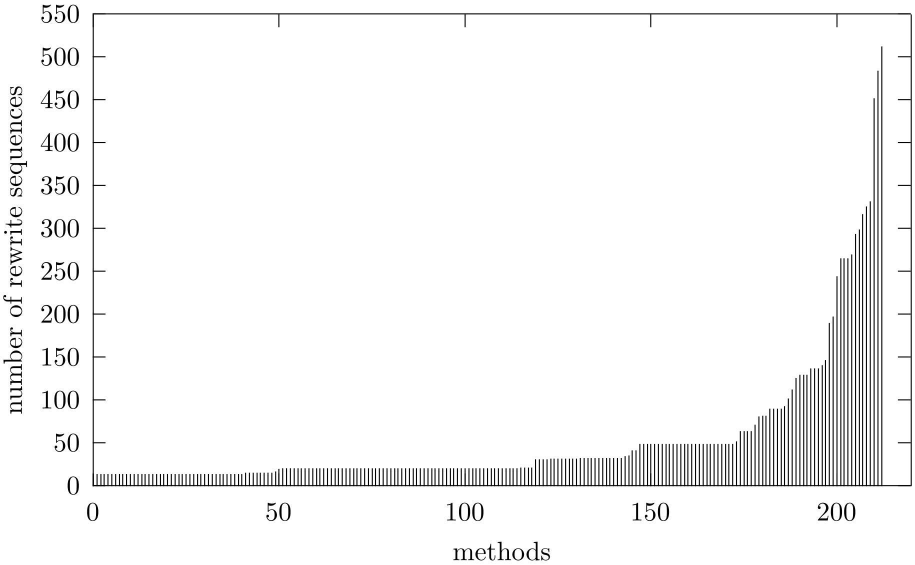

Figure 5.3: Number of rewrite sequences for 213 methods

```
i := 1; j := 1;
while i <= 10000 do
  a := i; j := 1;
  while j <= i do
    a := i; b := j;
    while a /= b do
       if a > b then a := a - b;
                else b := b - a;
       endif;
    endwhile;
    j := j + 1;
  endwhile;
  i := i + 1;
endwhile;
```

Figure 5.4: Pseudo code of test-program

compiler	-O0	-O2	-O5
gcc 2.96	32.96	30.21	30.92
jack	114.90	27.83	26.09

Table 5.2: Run-times in seconds of Java version of test-program

Chapter 6

Summary

We have shown that the unmodified BURS algorithm rewrites small terms, but it is not sufficient for large terms or an intermediate representation in graph form. We modified it by analyzing the given rules in the term rewrite system and introducing local goal terms. With these, we changed the definition of the inputs of a decoration. This small extension enables the algorithm to deal with essentially large terms. Furthermore we can apply this algorithm to DAGs instead of terms.

Bibliography

[1] A. Balachandran, Dhananjay M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[2] Boris J. H. Boesler. Codeerzeugung aus Abhängigkeitsgraphen. Diplomarbeit, Universität Karlsruhe, June 1998.

[3] Helmut Emmelmann. BEG - A back end generator - user manual. Arbeitspapier nr. 420, GMD, 1989.

[4] Helmut Emmelmann. *Codegenerierung mit regulär gesteuerter Termersetzung*. PhD thesis, Universität Karlsruhe, 1994.

[5] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[6] J.-P. Katoen and A. Nymeyer. The systematic development of a pattern-matching algorithm using term rewrite systems. *Proceedings of CATS'97*, 1997.

[7] A. Nymeyer and J.-P. Katoen. Code generation based on formal burs theory and heuristic search. pages 597–635, 1997.

[8] Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code generation = A* + BURS. In *Computational Complexity*, pages 160–176, 1996.

[9] E. Pelegrí-Llopart. *Rewrite systems, pattern matching and code generation*. PhD thesis, University of California, Berkley, 1987.

[10] Peter Mahrwedel Rainer Leupers. *Retargetable Compiler Technology For Embedded Systems, Tools and Applications*. Kluwer academic publishers, 2001.

[11] Glanville RS and Graham SL. A new method for compiler code generation. *Proc. of the 5th ACM Symp. on Principles of Programming Languages*, pages 231–240, 1978.