TWO TREE PATTERN MATCHERS FOR CODE SELECTION

Beatrix Weisgerber, Reinhard Wilhelm*

FB 10 - Informatik
Universität des Saarlandes
D - 6600 Saarbrücken
Federal Republic of Germany

ABSTRACT

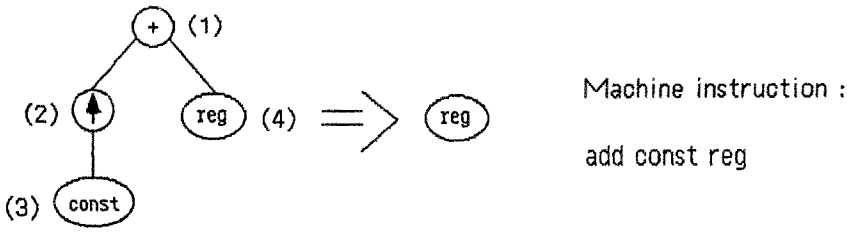
A bottom up- and a top down pattern matching algorithm for code selection are presented. The setting is the same as in [AhGa84]. First all covers of the intermediate representation (IR) are computed, and cheapest ones are determined by dynamic programming. Then code selection proper is performed. While Graham-Glanville-like code generators ([GlGr78],[Glan77]) use dynamic targeting, i.e. by selecting appropriate productions at reduction time, and while Aho- Ganapathi shift the targeting task to the semantic attributes and functions, the two algorithms presented in this paper use static targeting. Targeting rules, i.e. rules with patterns of depth 1, are simulated in the states of the recognizing automata. Therefore, all covers found for an IR are adequate as far as no semantic constraints are concerned. The bottom up approach suffers from the theoretical worst case complexity, i.e. the (static) size of the automata may grow exponentially with the size of the machine description. The top down approach has a linear (static) size of the automaton, but a dynamic size, i.e. the size of additional data structures, of |IR| * |machine description|. The bottom up approach has been implemented as a modification of the OPTRAN bottom up pattern matcher generator [Weis83].

1. Introduction

In recent years, there has been done a considerable amount of research on the generative approach to code generation. A survey is given in [Lune83]. This approach automates the generation of code generators and facilitates retargeting. In the area of code selection, works based on pattern matching has been proved to deliver reasonable results ([Glan77], [Henr84]). For a comparison of different approaches on an algebraic basis see [GiSc88] [Gieg88].

In the pattern matching approach, machine instructions are selected according to patterns of the intermediate representation (IR). Each (input) pattern models a target machine instruction. A second (result) pattern describes, where the result of the corresponding instruction will be available. Leaves in the input pattern and the single node result pattern correspond to registers, memory cells etc. Input pattern, result pattern, machine instruction and a cost statement estimating the amount of memory and machine cycles necessary for the machine instruction, form a kind of a "annotated" reduction rule. See figure 1.

present address: Standard Elektrik Lorenz AG, Forschungszentrum, Lorenzstr. 10, D-7000 Stuttgart 40

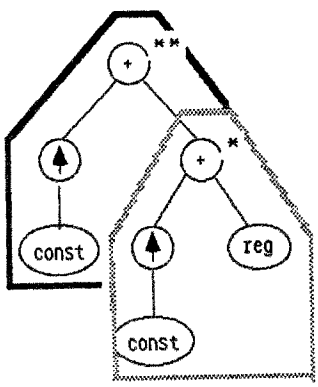


Add the content of a memory cell, the address of which is given by "const" to the content of the register "reg". The result is written into the register "reg". The sub-patterns of the left-hand side pattern are numbered for later reference.

Figure 1

For every given IR tree, a pattern driven code generator has to find at least one sequence of instructions corresponding to the tree. Such sequences are issued as indicated by covers of the IR tree by input patterns. A cover is found by plastering the tree with input patterns, such that every node of the tree has to be covered by exactly one node of one pattern and such that the patterns "fit together". This means, that the result pattern of the input pattern is or can be transferred to a leaf of another input pattern. The problem corresponds to the targeting problem: the result of the corresponding machine instruction has to be in the place where the next instruction expects it to be. An example is given in figure 2.

On the other hand, addressing modes may be factorized by special rules. Then, each result pattern of such rules symbolizes a certain addressing mode (indirection, indexing etc.) which can be used within other patterns without explicitly repeating the possible huge address pattern. "Fitting together" means in this context insuring that the appropriate addressing mode can be used.



*The pattern of figure 1 matches at node * and delivers a result 'reg', so that the pattern matches again at node **. Every node of the original tree is covered by exactly one pattern. The result of the first reduction produces a leaf of the second pattern.*

Figure 2

A tree pattern matcher finding covers of trees should simulate the "pasting" process. This means in terms of rewrite systems to simulate the reduction of the input pattern by the result pattern. If the input pattern of a rule matches a subtree, this subtree is replaced by the result pattern. The process continues until no more pattern matches and only one node results. See figure 3.

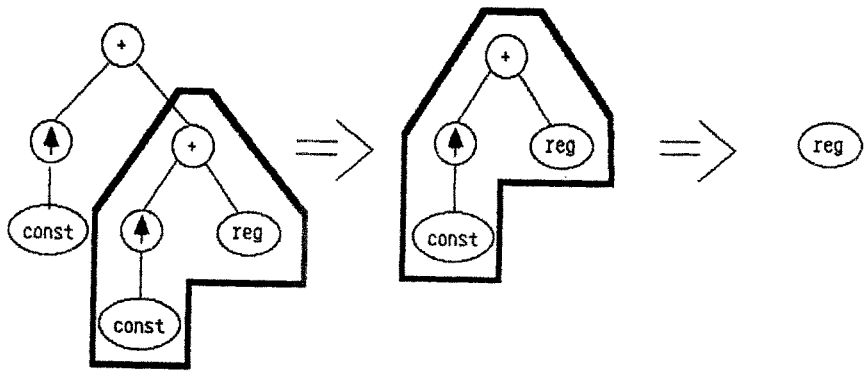


Figure 3

In general, there is more than one cover for a tree by real (non RISC) machine descriptions. A code generator should be able to select one with minimal costs, i.e. sum of the costs for the instruction sequence issued.

The Graham/Glanville approach ([Glan77], [GlGr78]) reduced the problem of tree pattern matching to the problem of recognizing words of a (string) language. The IR tree is linearized according to a preorder traversal. Each reduction rule is represented by a production of a context-free grammar.

Example: REG -> + ↑ const REG,

A LR-type parser is generated from this grammar. Each reduction in the parsing process corresponds to a tree reduction. Certain heuristics and backtrack mechanisms had to be built in to find a cover, because the grammars are ambiguous in general. But there is a left bias due to the preorder traversal and the employed heuristics. Early selection decisions and the one pass approach preclude the generation of even locally optimal code.

The approach of Ganapathi/Fischer [GaFi82] provides further attribution mechanisms and formulation of predicates for more flexible conflict resolution.

As [Henr84] indicates, the context-free approach to the problem has been driven to its inherent limits. Tree pattern matching technology promises to overcome these limitations. However, tree analysers like those of Kron [Kron75] or Hoffman/O'Donnell [HoDo82] cannot be used unchanged for this purpose. They can determine whether and where in a tree patterns from a set of patterns match, but they don't simulate the reduction process as described above.

Aho/Ganapathi [AhGa85] propose a solution for this problem. The leaves of their patterns and the result patterns are parameter nodes which match every tree. So, they can use conventional pattern match automata. A suitable attribution has to provide the targeting task. These computations give rise to additional costs at run time.

The approach presented in this paper will overcome this problem by adapting the well-known tree pattern matching automata to simulate the reduction process. For every given tree, the automata will provide all possible covers according to a set of reduction rules. Dynamic programming ([AhJo76]) performed in parallel will then find a locally cheapest code sequence.

2. The bottom up tree analyser

Bottom up tree analysers have the great advantage, that their analysis time costs are linear in the number of the nodes of the subject tree. The bottom up tree analyser of Kron and Hoffmann/O'Donnell works as follows: With every node in the tree a set of subpatterns is associated which match the subtree at this node. These match sets are computed bottom up.

The match set of a node can be determined, if all subtrees of this node are already analysed. A subpattern matches at a node if and only if its root has the same label as the node and if every child pattern matches the corresponding child tree of the node. The latter fact can be determined by inspecting the match sets of the children. If the child pattern can be found in the match set of the corresponding child tree, then the child pattern matches the child tree.

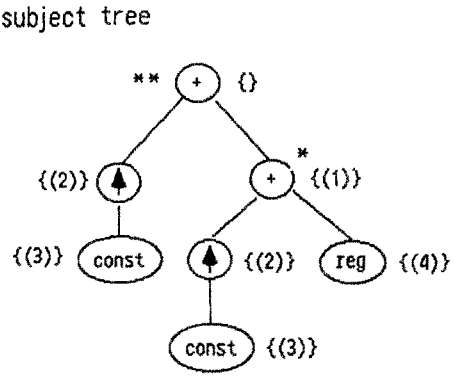


Figure 4

In figure 4 the subject tree to be analysed is shown together with its match sets. At node *, the full pattern matches. This fact is represented by associating match set {(1)} with *. The root label of the pattern is the label of the node, at the first child the first subtree of the pattern ((2)) matches and at the second child the second subtree of the pattern ((4)) matches. We can see, that at node ** no pattern matches, because at its second child no match can be found for subpattern <reg>. This subpattern would be produced by reduction at node *. We can simulate this rewrite rule. Every time, a full pattern occurs in a match set, all occurrences of the result pattern (as input subpatterns) are added to the match set. Figure 5 shows the result for our example.

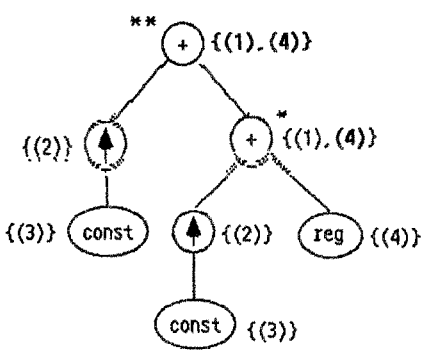


Figure 5

At node *, we now have the additional information that subpattern (4) matches. This has the effect, that the main pattern is recognized at node **. Each match set now reflects the actual situation of the tree and additionally each situation which could arise by reductions of the tree.

An example for more complex rules as shown in figure 6 is given in figure 7.

At this point, we can observe several properties of the analyser. Chains of so-called transfer rules (rules having an input pattern consisting of a single node, modelling for example

transformation rules (represented by the top node of the left–hand side pattern)

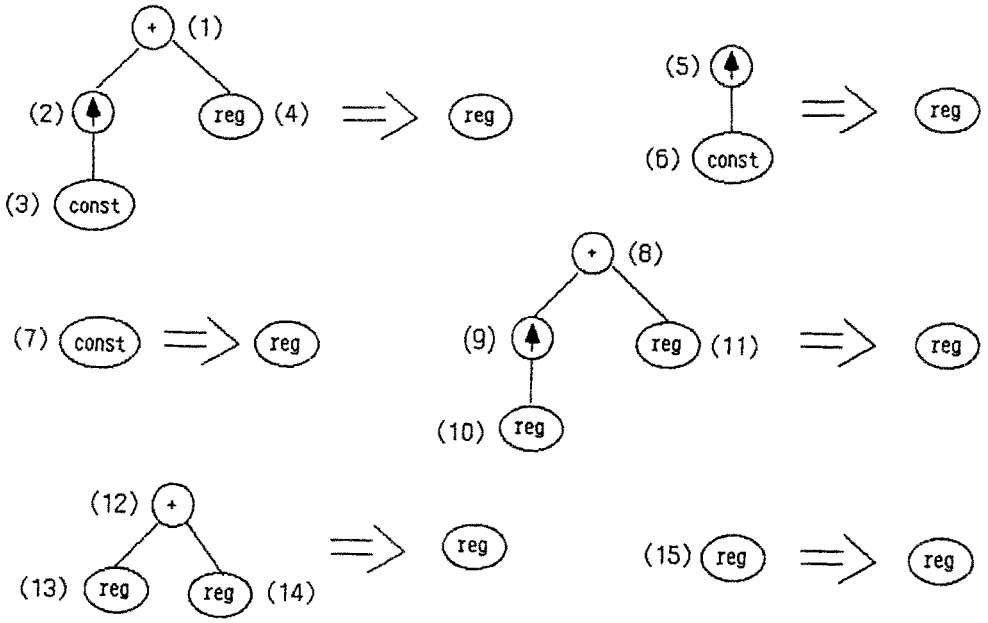


Figure 6 subject tree

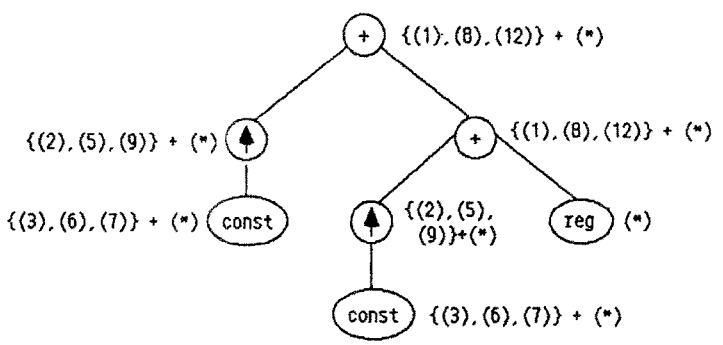


Figure 7 (*) = {(4),(10),(11); (13),(14),(15)}

register-to-register transfers or memory-to-register transfers or factorizing rules) are enclosed in one match set. If the input pattern of a transfer rule is contained in a match set, then at the same time the input subpatterns having the same label as the result pattern of the rule can be added to this match set. See for example rule (7) and (15). If (7) is contained in a match set, then set (*) can be added immediately. For rule (15) nothing is added, because the results are already considered. The effects of such chains of transfer rules are computed simultaneously. There are no problems with possible loops in transfer rule chains. In the Graham/Glanville approach, such possible infinite reduction sequences have to be detected and broken explicitly.

The second observation concerns the kind of information provided: The simulating analyser only states, that certain patterns match, if certain transformation rules are applied at certain subtrees or possibly no transformation rule is applied. But the information about the different covers found is not immediately available. The approach of Möncke [Moen85] provides more exact information in this respect, but it will probably produce larger automata.

Thus, we have to collect further information at run time to determine an actual instruction sequence corresponding to the covers found by the automaton. A possible solution is to construct the graph representing all covers for an individual tree. An example for such a graph, called history graph, is shown in figure 8. Reduction rules are associated with those nodes in the tree to which they can be applied. If a reduction at node n depends on other reductions at descendent nodes, then an edge is drawn from the history graph of the descendent node to the history graph of n. This shows which result of which reduction is responsible for the existence of an appropriate leaf of another pattern.

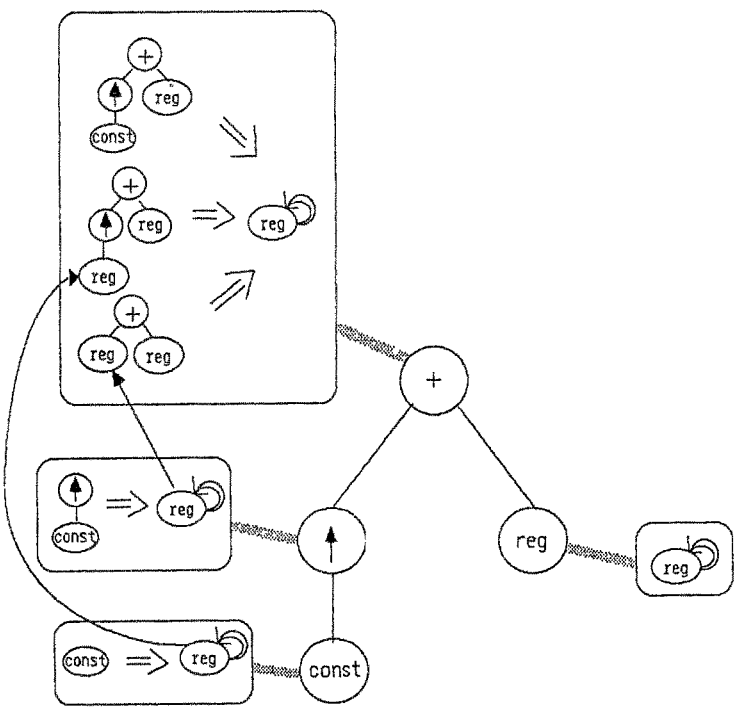


Figure 8

3. The generator of the bottom up analyser

Naturally, it is possible to compute the match sets at analysis time. But it will be much more efficient to precompute them as the states of a finite tree automaton from the machine description. We are then able to compute encoded match sets and the associated partial history graphs for IR trees by means of this automaton. Pure analysis costs are linear in the number of nodes of the subject tree to be analysed.

Hoffmann/O'Donnell [HoDo82] developed a generator for conventional tree analysers producing a n-dimensional matrix M[op] for every operator op, where n is the rank of op. Each entry $M[op][f_1,...f_n]$ supplies the match set for a tree node labelled op, where every child i, from 1 to n, is already analysed by match set f_i. Realistic pattern sets will result in huge

sparse matrices, i.e. most entries will denote the empty match set. Table compression methods will produce reasonable automata representation [MWW85].

Kron's generator [Kron75] (see also [Weis83]) produces finite bottom up tree automata. Match sets (or states in the finite automata terminology) are computed for a subtree by successively regarding the root label (op), the match set of the first child (f_1), the match set of the second child (f_2),...and the match set of the last child (f_n). *init*[*op*] yields the start state containing all subpatterns having root label op. Let the rank of the operator op be greater than zero. Then $trans(init[op],(1,f_1)) = z_1$ denotes the state containing all the subpatterns of $init[op] = z_0$ which agree with the tree in the root label and the first child. All those subpatterns of z_0, whose first child cannot be found in f_1, are not contained in z_1. $trans(z_{i-1},(i,f_i)) = z_i$ encodes the subpatterns of z_i which match the tree regarding the root label and the match sets $f_1,...f_i$ of the children 1,...,i. Compared to z_{i-1}, all those subpatterns are removed, whose child is not contained in f_i. At last, the final state $z_n = trans(z_{n-1},(n,f_n))$ denotes the match set of patterns agreeing with the whole subtree. An example of a tree analysis process on the basis of the rule set of figure 6 is given in figure 9. (There is no simulation of reductions in this figure!).

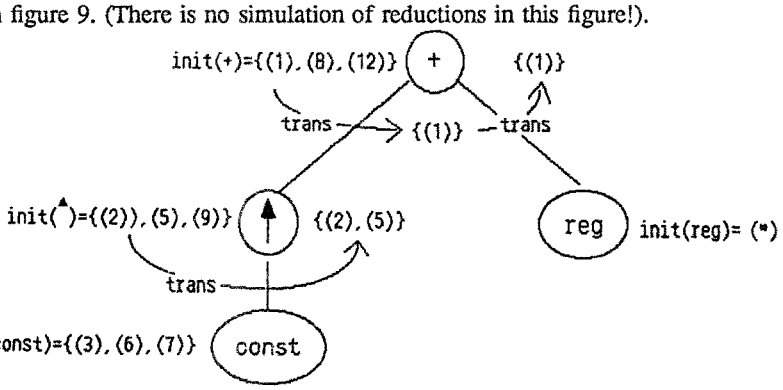


Figure 9

The automaton for this pattern set is shown in figure 10. The following convention is used:

init[*op*] = z_0 is represented as $op \dashrightarrow z_0$

The transition from state z_{i-1} to z_i by f_i at the ith child, expressed by $trans(z_{i-1},(i,f_i)) = z_i$, is represented as

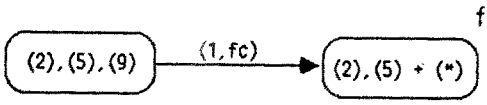
$$z_{i-1} \xrightarrow{(i,f_i)} z_i$$

Final states are enclosed in bold face boxes.

Transitions which are not explicitly mentioned lead to the empty state. Every transition out of the empty state leads back to the empty state.

We will now simulate the reduction. As already mentioned in the previous chapter, the first step is to insert into every final state containing a full pattern (i) all those subpatterns which could be matched as result of the reduction by (i). In our example, all subpatterns with root label "reg" (set (*)) are added to every final state that contains a full pattern of a rule with a result pattern "reg". The result of this first step is shown in figure 11.

If such a new final state f is involved in a transition of the kind $trans(z,(i,f)) = z'$, then it may be possible that z' should contain additional subpatterns according to the definition of transitions. The simulation of the reduction may cause certain subpatterns to be no longer eliminated on the transition from z to z'. Look at the following transition in the subautomaton of ↑:



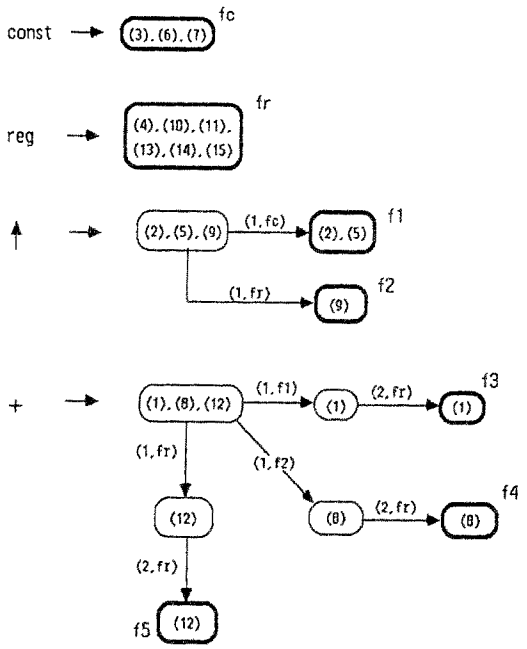


Figure 10

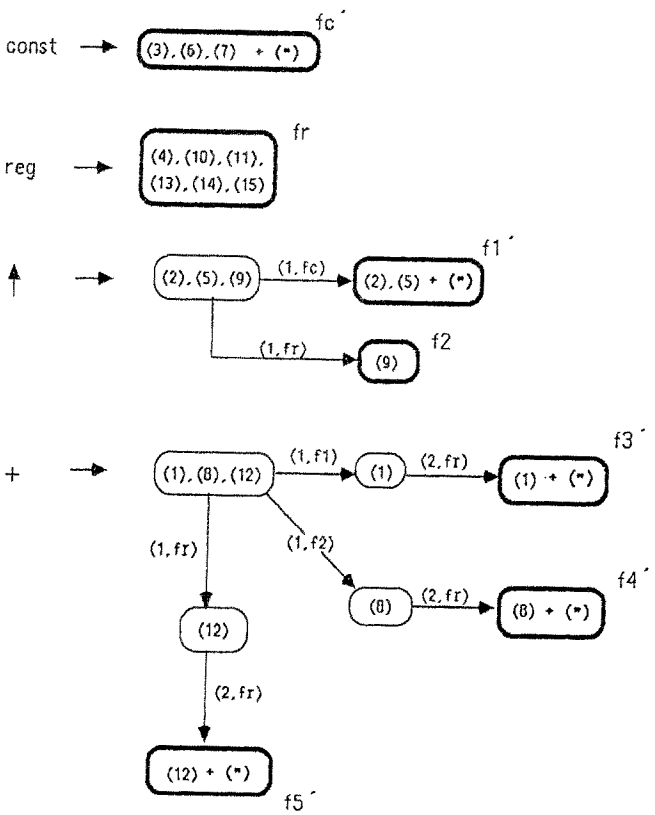
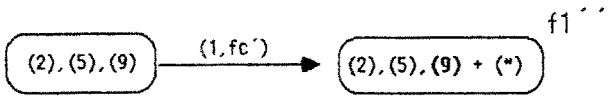


Figure 11

After the transition state *fc* has changed to *fc'*, by simulating the reduction <const> ==> <reg>, pattern (9) will be a possible match, too. Pattern (9) is no longer eliminated by the transition to f_1'.



The effects produced in this way by the final states changed in the first step, yield the automaton of figure 12. Remember that some transitions to the empty state not shown in the previous figure are also changed.

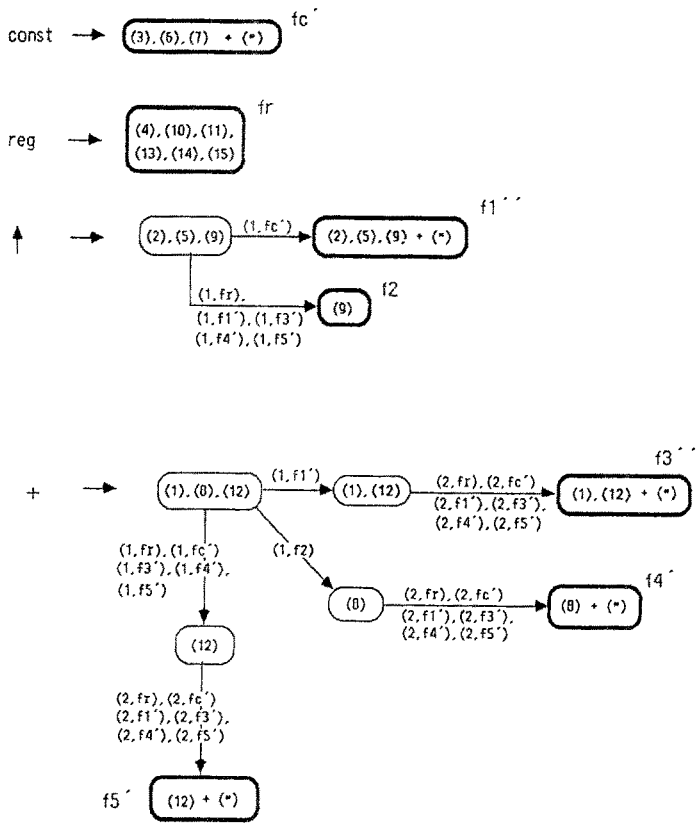


Figure 12

After each step an automaton is produced using transitions on final states of the previous automaton, that is, final states which may exist no longer in the current automaton. For example, f_1' of figure 12 has changed to f_1''. The process continues until the set of final states is no longer changed. The process terminates because during each step a state f_j (existing before the step) may only produce final states f_j', satisfying $f_j \subseteq f_j'$, and because only a finite number of combinations of subpatterns exists.

The final result for our example is shown in figure 13.

The major disadvantage of the bottom up automaton generation is the exponential worst case behaviour (see [HoDo82]). But examples using realistic patterns have shown good results. Preliminary experiments on the basis of small rule sets didn't support our fears that the

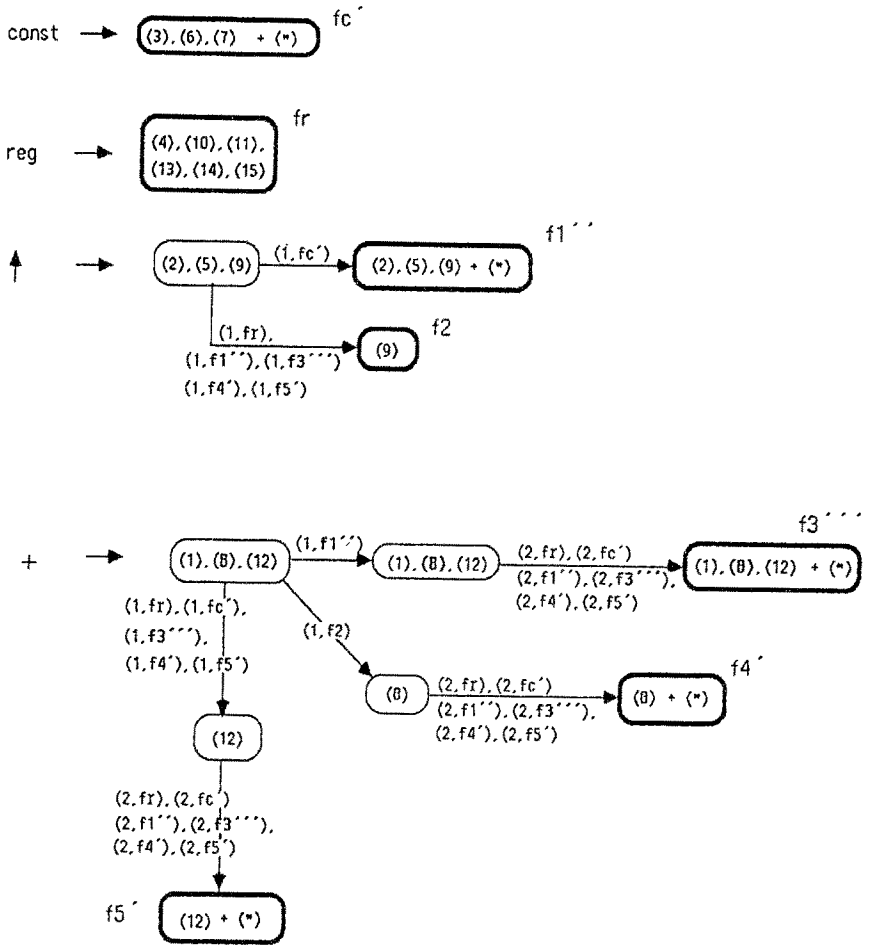


Figure 13

simulating automaton becomes significantly larger than the non-simulating one (the worst case behaviour of the two kinds of automata is naturally the same). There were even rule sets where the number of states (when identifying states containing the same subpatterns) decreased and only the number of transitions to non-empty states increased slightly . Further experiments using realistic machine descriptions will show, whether these good results will be confirmed.

Finally, we have to discuss the generation of the partial history graphs for every final state. The iteration process makes this generation obvious: In the basic (non-simulating) automaton a partial history graph is associated to each final state describing the reduction rules of its full patterns. If during an iteration step a new final state f' is produced on the basis of a final state f, then f' takes over the partial history graph of f and the reduction rules corresponding to newly added patterns are integrated. If an application of a reduction rule may induce an application of a chain of further reduction rules, then they are linked together. Possible cycles may be represented. Figure 14 shows an example.

The construction of the history graph can be prepared at generation time in the following way. There is a static number of leaves resulting from previous reductions in each (sub)pattern in a state. Therefore a static assignment of positions in a vector can be computed for all the

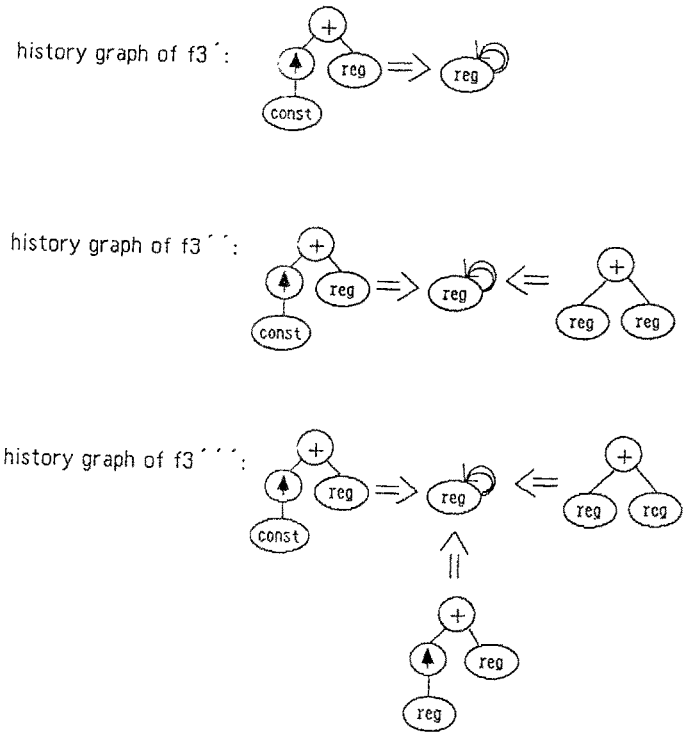


Figure 14

leaves in all the (sub)patterns in a state. For each transition of the pattern matcher a program (or a table entry) can be generated, which uses these address assignmentsto transport the root addresses of partial history graphs to the point of next reduction. Precomputations on the basis of the rule costs will decrease the graphs for example by shortening rule chains.

4. The top down analyser

The most prominent property of the top down analysis automaton of Hoffmann/O'Donnell is its small generation cost. It takes time linear in the number of subpatterns. On the other hand the analysis time costs are significantly higher than those of the bottom up automaton: the worst-case behaviour is O(number of nodes of the subject tree * number of subpatterns).

A top down analyser tries to recognize paths from the root to the leaves of patterns within a given subject tree. The paths are represented as strings of labels and child positions. For example, pattern number (1) of figure 6 can be described by the strings

+ 1 ↑ 1 const

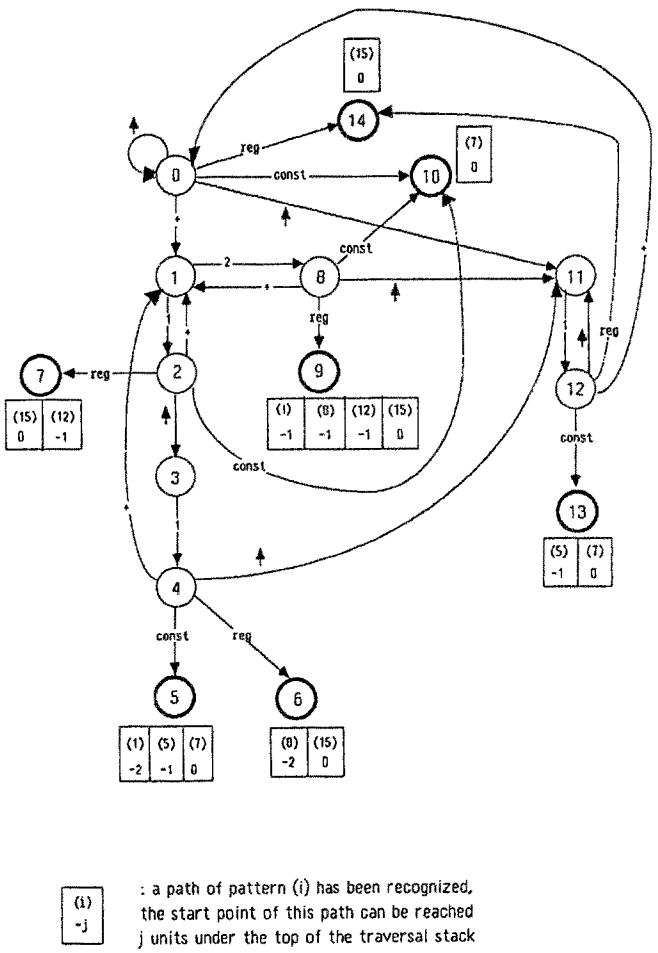
and + 2 reg

The task of matching tree patterns within a tree is reduced to the problem of matching all strings of tree patterns and recording successfully matched strings of patterns at nodes in the subject tree.

At generation time a finite automaton is build up on the basis of the pattern strings (see Hoffmann/O'Donnell using the principles of Aho/Corasick [AhCo75]).

A top down automaton for the patterns of figure 6 is shown in figure 15. A subject tree is analysed in preorder traversal. The transitions of the automaton are controlled by the label of the visited subject tree node and by the position of the visited child when descending the tree.

Entering a final state (represented by bold face boxes) means that a full path of a pattern matches. This fact has to be reported to the start node of the path. If a traversal stack is used at run time, we can precompute the relative distance on the stack, where we can find the start node of a path when entering a final state. If we have a match vector for every node with one counter for each full pattern (initialized by 0), then a match of a path of a full pattern (i) increases the counter (i) of the match vector at the start node of the path. A counter equal to the number of different paths of (i) indicates a match of the pattern.



(i) -j : a path of pattern (i) has been recognized, the start point of this path can be reached j units under the top of the traversal stack

Figure 15

Figure 16 shows an analysis example using the automaton of figure 15. At each node we show the match vector and the transitions made by the node labels.

This top down analyser will now be modified to simulate the reduction process. A problem with the top down analyser is that the state computed for a node considers no information at all about the subtree at this node. The state is only determined by the upper context. On the other hand, a reduction process using only patterns of the sort described above is inherently bottom up. Thus, the information about the result of the transformation process is only computed when the subtree is completely traversed. Thus, we have to visit each node a second time in postorder. At this time, the match vector of the node informs us about which reduction

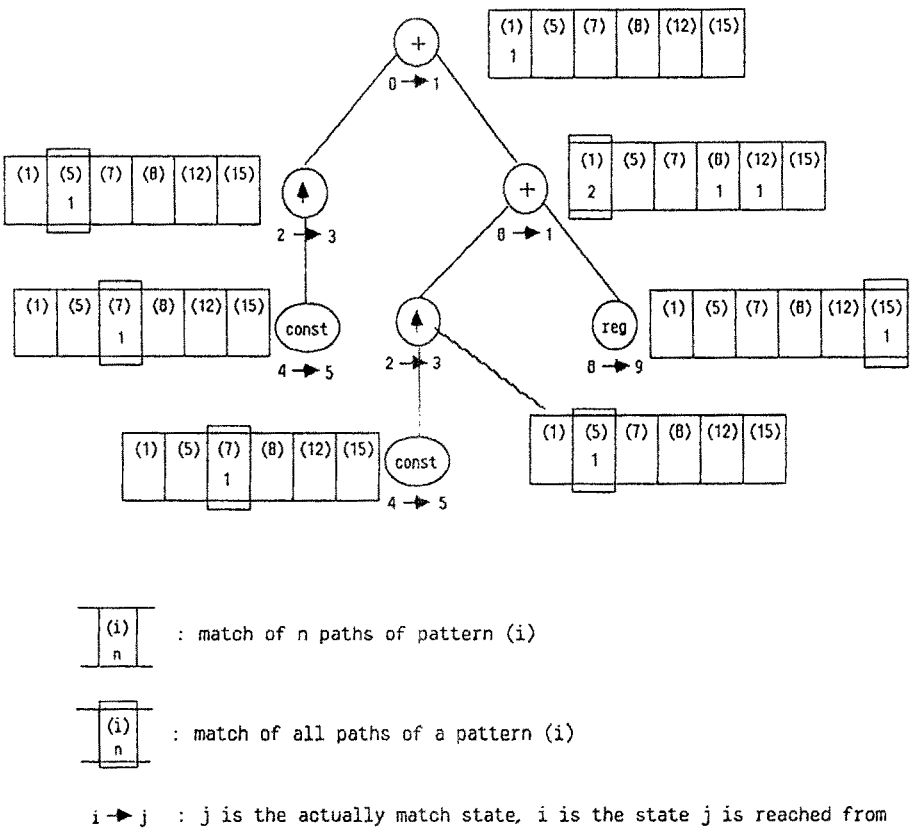


Figure 16

rules are applicable and which result they produce. Say, a rule having a result pattern <op> can be applied at node n. This means that a path leading to n would continue with the label op, if this rule would be applied at n. If we remember the state z entered before the node label is taken into account, the transition entered from this state z following the edge labelled op supplies a state containing all information about possible matches basing on the simulated application.

If this new transition produced additional matches at the node itself, then we have to consider the results of those rules again. The effects of such chains can be precomputed for each non-transfer rule and possible cycles may be broken.

Figure 17 shows the simulation for the example of figure 16.

For *each* label produced by the simulation of a (chain of) reduction rule(s) an additional transition has to be made. Each additional match has to be reported to an ancestor node. This means an additional cost at analysis time which is significantly higher than the run time of the simulating bottom up automaton.

The simulation process of the bottom up automaton only increases the generation costs, not the run time costs (not mentioning the possible higher access costs induced by larger tables). The top down simulation method only affects the run time behaviour. The generator of the bottom up analyzer as programmed by the first author is being integrated into the PCC-environment by Nicole Leitges with support from SIEMENS AG München.

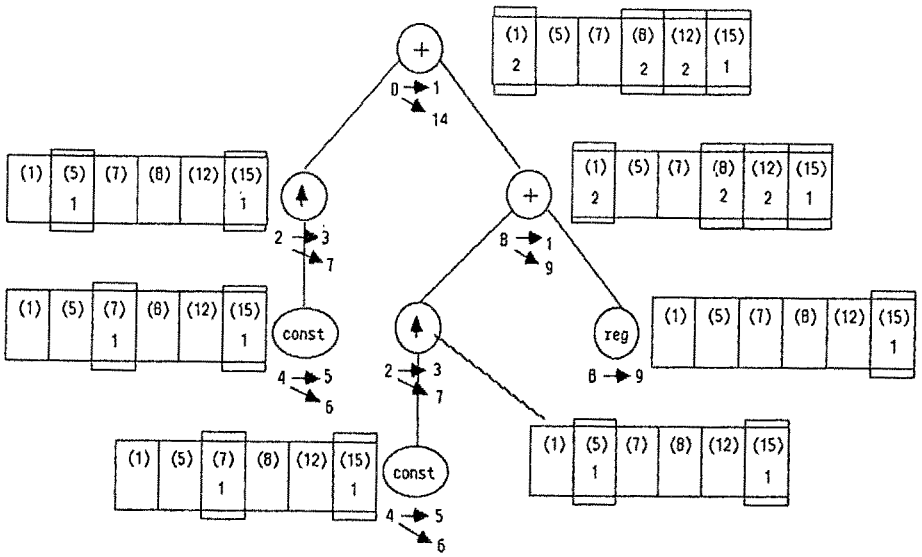


Figure 17

5. Conclusion

A bottom up and a top down pattern matcher for code selection have been presented. Both pattern matcher first compute all possible covers for an IR tree, then a cheapest cover can be selected by dynamic programming performed in parallel. The bottom up pattern matcher performs static targeting: all transfer rules are simulated in the states of a reduction simulating automaton. The top down pattern matcher performs dynamic targeting by making several additional transitions instead of a single one. But both approaches treat the problem on the pure syntactic level.

The bottom up automaton may grow exponentially with the size of the machine description. The generation cost for the top down automaton is linear. On the other hand, the analysis time performance of the bottom up automaton is much better than that of the top down automaton. Experiments will have to reveal the advantages for realistic machine descriptions.

6. References

[AhCo75] A.V. Aho, M.J.Corasick, Efficient String Matching: An Aid to Bibliographic Search, CACM, June 1975, Vol. 18, Nr. 6

[AhJo76] A.V. Aho, S.C. Johnson, Optimal Code Generation for Expression Trees, JACM, Vol. 23, No 3, July 1976

[AhGa84] A.V. Aho, M. Ganapathi, Efficient Tree Pattern Matching: An Aid to Code Generation, POPL 1985

[GaFi82] M. Ganapathi, C.N. Fischer, Description-Driven Code Generation Using Attribute Grammars, POPL 1982

[GiSc88] R. Giegerich, K. Schmal, Code Selection Techniques: Pattern Matching, Tree Parsing, and Inversion of Derivors, Proceedings of ESOP'88, LNCS 300, Springer Verlag 1988

[Gieg88] R. Giegerich, Code Selection by Inversion of Order-Sorted Derivors, to appear in Theoretical Computer Science, North Holland

[Glan77] R.S. Glanville, A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers, PhD Dissertation, University of California, Berkley, December 1977

[GlGr78] R.S. Glanville, S.L. Graham, A New Method for Compiler Code Generation, POPL 1978

[Henr84] R.R. Henry, Graham Glanville Code Generators, PhD Dissertation, University of California, Berkley, 1984

[HoDo82] D.M. Hoffman, M.J. O'Donnell, Pattern Matching in Trees, JACM 29,1, 1982

[Kron75] H. Kron, Tree Templates and Subtree Transformational Grammars, PhD Dissertation, University of California, Santa Cruz, 1975

[Lune83] H. Lunell, Code Generator Writing Systems, Software System Research Center, Linköping, Sweden, 1983

[Moen85] U. Möncke, Generierung von Systemen zur Transformation attributiertern Operatorbäume: Komponenten des Systems und Mechanismen der Generierung, Dissertation, Universität des Saarlandes, Saarbrücken, 1985

[MWW85] U. Möncke, B. Weisgerber, R. Wilhelm, Generative Support for Transformational Programming, ESPRIT Technical Week, 1985

[Weis83] B. Weisgerber, Attributierte Transformationsgrammatiken: Die Baumanalyse und Untersuchungen zu Transformationsstrategien, Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1983