How to Uniformly Specify Program Analysis and Transformation with Graph Rewrite Systems

Uwe Aßmann

INRIA Rocquencourt
Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France
Uwe.Assmann@inria.fr*

Abstract. Implementing program optimizers is a task which swallows an enourmous amount of man-power. To reduce development time a simple and practial specification method is highly desirable. Such a method should comprise both program analysis and transformation. However, although several frameworks for program analysis exist, none of them can be used for analysis and transformation uniformly. This paper presents such a method. For program analysis we use a simple variant of graph rewrite systems (*edge addition rewrite systems*). For program transformation we apply more complex graph rewrite systems. Our specification method has been implemented prototypically in the optimizer generator OPTIMIX. OPTIMIX works with arbitrary intermediate languages and generates real-life program analyses and transformations. We demonstrate this by several examples and measurements.

Keywords: Program analysis, program transformation, optimization, specification, graph rewrite systems

1 Introduction

This paper presents a uniform specification method for program analysis and transformation. It is based on graph rewrite systems (GRS), can be used for arbitrary intermediate languages (also abstract syntax trees), and leads to efficiently executing optimizer parts. The major idea behind it is the insight that the intermediate representations in optimizers are *graphs* which are constructed and manipulated by *graph transformations*. Thus it is quite natural to use graph rewrite systems to specify both problems of program analysis and transformation.

With our specification method the process of writing an optimizer is divided into four phases. First a data model for the graphs — intermediate code as well

* This work has been supported by Esprit project No. 5399 COMPARE. Most of this work has been done while the author was at Universität Karlsruhe IPD, Vincenz-Prießnitz-Str. 3, 76128 Karlsruhe, Germany

as analysis information — has to be developed. Secondly, program analysis has to be specified by graph rewrite systems. For this we supply a special variant of graph rewrite systems, *edge addition rewrite systems* (EARS). EARS are very simple graph rewrite systems because they only allow the addition of edges to graphs and do not remove any existing parts. Thirdly, program transformations can be specified by more general graph rewrite systems which allow deletion of edges, and insertion and deletion of nodes. Finally, after having described the optimization abstractly, the representations of the graph classes can be changed in order to speed up the generated algorithms. The user also can feed the generator with assertions so that the generator can apply index structures (dictionaries). We will show that this is an essential part to achieve effiently executing optimizer parts.

The structure of the paper is shaped along this process. We will present a running example, the elimination of partial redundancies (lazy code motion) [KRS94]. It comprises syntactic equivalence of intermediate expressions (section 4.1), global data flow analysis (section 4.2), and a transformation phase (section 5). Additionally we present some figures concerning the efficiency of the generated algorithms. We conclude the paper with a section about related work.

To demonstrate the validity of the specification method the optimizer generator OPTIMIX and several optimizer parts have been developed within the compiler framework CoSy [AAvS94]. All generated components work on the *common COMPARE medium intermediate representation*, the intermediate language CCMIR. The CCMIR serves as common platform for frontends in C, Fortran-90, Modula-2, Modula-P [VH92], and soon C++ (`http://www.ace.nl`).

2 The method

The central idea of our specification method is to regard the optimization process as a sequence of applications of graph rewrite systems (Figure 1). We start with a graph given by the frontend which is the abstract syntax tree or the intermediate code sequence. First we have to perform program analysis. This is normally done by several EARS, executed one after the other. These may comprise one or several data-flow analyses. After that the transforming graph rewrite systems can be applied.

The advantages of this scheme are the following. First we have a uniform view on the intermediate language and the analysis information: everything is represented as graphs, and all actions are done by graph rewriting. Second, instead of describing everything with one big graph rewrite system (which could be possible) we prefer to write a sequence of smaller ones. Larger graph rewrite systems tend to loose confluence or termination. With a suitable separation one can isolate non-confluency or even resolve non-termination. Third, for smaller graph rewrite systems we know how to generate good code because smaller rewrite systems in general have a lower *order* than bigger ones: often linear or quadratic algorithms can be achieved [Aß95b]. Forth, this division breaks the optimization task down into component problems which can be solved independently. If we

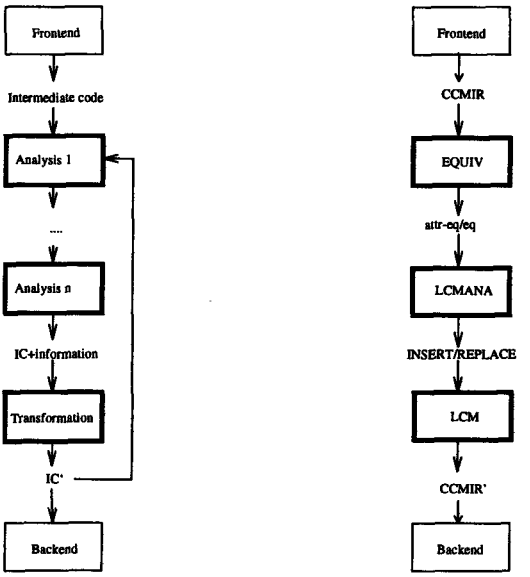


Fig. 1. Left: Optimization as a sequence of graph rewrite system applications. Thick boxes are generated phases. Right: our examples as an instance of the method

do not find a system for a certain task, or some developed system turns out to be too slow, or we can always substitute it by a hand-written algorithm. This is anyway the case for the frontend and the backend; they are implemented as usual.

In order to make this scheme work, all phases of the compiler have to use a uniform object handling package including uniform graph and set classes. In COMPARE this is solved by the data description language fSDL. It provides graph and set *functors* (template classes). From this a uniform access interface is generated [WKD94]. All hand-written and generated phases use this interface so that a uniform data access is given.

3 Designing the data model

Before developing graph rewrite systems for program analysis and transformations, we have to say how the intermediate graphs look like. This amounts to specifying a *data model* or *graph schema* [Sch90]. We have to layout the following:

1. model the graph node types: among these are intermediate code instructions such as expressions, statements, loops, procedures. Also already analyzed information can be encoded in nodes of graphs, e.g. definitions, or expression equivalence classes.

2. model the graph edge types (relations): First we have to model the information the frontend produces, e.g. expression tree pointering, or statement lists. Second we have to model all predicates we want to infer about the program, i.e. the relations among the objects of the program and the analysis. Examples are relations such as the classical flow dependencies, equvialence relations, calling relations, control flow relations.

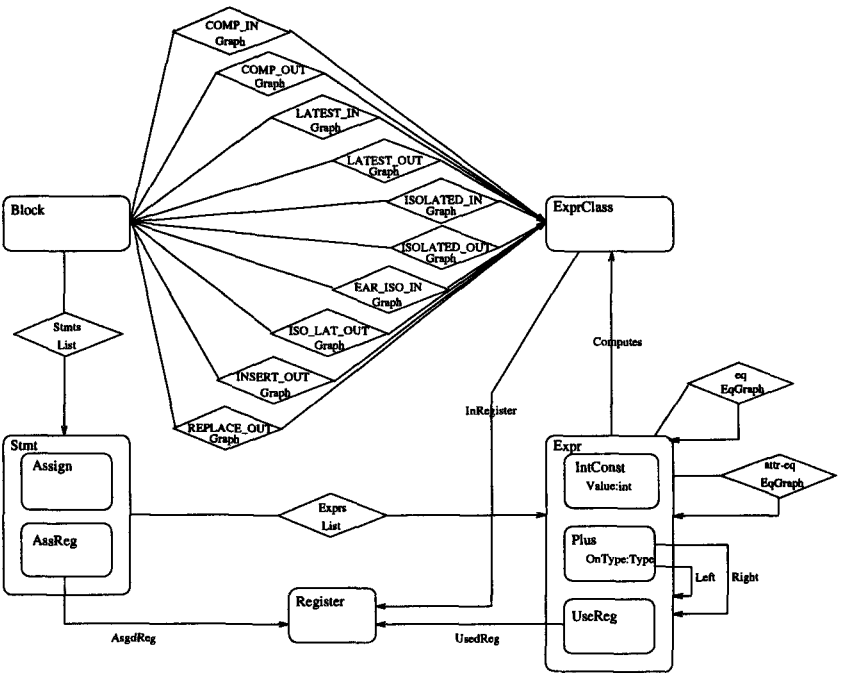


Fig. 2. Data model as higraph.

Our running example relies on the data model in figure 2. The diagram is a higraph, i.e. an extended ER-model [Har88]. Boxes denote entity types, box inclusion denotes inheritance. n:m-relations are depicted by diamond boxes, n:1-relations by simple arrows. Because we deal with directed graphs, we have to indicate source and target domains of the relations, which is done by arrows. For the example we assume some familiar object types, such as blocks, statements and expressions. Several of them are generated by the frontend (`Block`, `Assign`, `IntConst`, `Plus`) and form the basis for the optimization process. Others are generated by the optimizer: `AssReg`- and `UseReg`- objects denote assignments and uses of registers (`Register`), objects of type `ExprClass` denote syntactically equivalent expressions. Also the relations are only partially produced by the

frontend (**Stmts**, **Exprs**, **Left**, **Right**). All others are the results of optimizer components, i.e. computed by graph rewrite systems.

This data model can easily be transformed into a concrete data description language such as a fSDL [WKD94]. In such a language relations must be annotated by a concrete representation class. fSDL provides two kinds of representations: bidirectional relations (such as **Graph** or **EqGraph**), and directed relations (such as **Set** or **List**). The relations in our figure carry such annotations. Note that this choice does not influence the specification, but the generated code.

4 Specifying the analysis

4.1 Local analysis

One of the central ideas of our specification method is to describe the collection of information as simple graph rewrite systems which only add edges to graphs. One example for a local program analysis is the specification of syntactic equivalence of expression trees. This is the prerequisite for lazy code motion data-flow analysis. In intermediate languages such as the CCMIR we distinguish two kinds of expressions: those with operands (non-leaf expressions) and those without (leaf expressions). Among the first there are operators such as **Plus**, among the latter there are those such as **IntConst**.

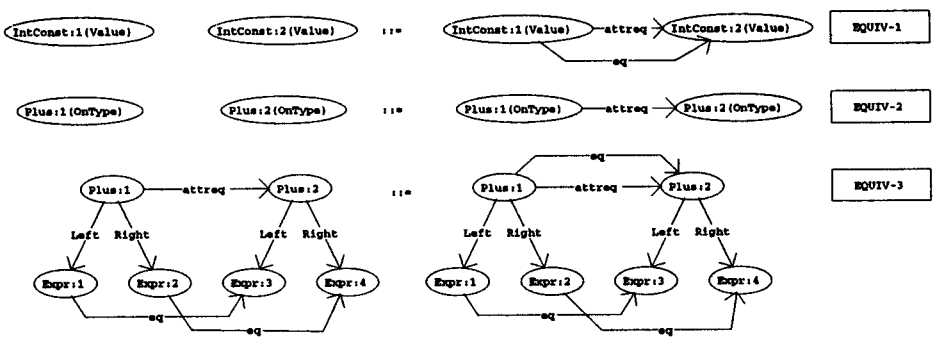


Fig. 3. EQUIV: Syntactic equivalence of expressions

Two arbitrary expressions are attribute-equivalent to each other if their attributes are the same. For **IntConst** expressions the value of the constant must be equal. For **Plus** expressions their field **OnType** must be equal. Two leaf expressions are equivalent if they are attribute-equivalent. Two non-leaf expressions are equivalent if they are attribute-equivalent and all their sons are equivalent. For **IntConst** and **Plus** these conditions are expressed by the EARS in Figure 3. For

a complete specification of syntactical tree equivalence of expressions we have to add similar rules for all subtypes of expressions.

Applying the rules of EQUIV to the intermediate representation of a procedure yields a relation **eq** which is a partition of all expressions and denotes the syntactical equivalence relation over expressions. With another simple graph rewrite system we can generate for each partition a representant node `ExprClass`. This object type will be used in the global data-flow analysis. Due to the lack of space we will not show this here.

4.2 Global data-flow analysis

With EARS it is also possible to specify global data-flow analysis. If the EARS is recursive, the rules have to be applied until a *fixpoint* is reached. This is similar to finding a fixpoint in a distributive data-flow framework. We demonstrate this by some equations from the equation system for lazy code motion ([KRS94], p. 1138). This data-flow analysis relates the blocks of the program to its expression classes, i.e. the syntactical equivalent expressions. If we encode this with graphs, the nodes are the blocks and the expression classes, whereas the edges represent the data-flow sets. Due to the lack of space we will only present the equation system for `ISOLATED`, as well as the non-recursive equation system for `INSERT/REPLACE_OUT` which determines the places of transformations at the end of blocks (Figure 4). We also do not show the initialization of the system, which can easily be written down with graph rewrite rules, too.

The rules mean the following: an expression must be inserted at the end of a block, if is latest and not isolated there. An expression is isolated at the exit of a block if it is either earliest or isolated at the entry of all successor blocks. Thus isolatedness at the exit of a block is expressed by recursive rules, defined over the successors of the block, which makes the fixpoint evaluation necessary. An expression is to be replaced at the end of a block if it is computed there and not isolated and latest.

For these equations we have to use two extensions of EARS, namely negation and universal quantifiers [Aß95a]. If an edge in a left hand side is marked with `NOT` no edge of this type is allowed to occur between the corresponding redex nodes. If a left hand side node is allquantified, then redexes must exist for all graph nodes in a neighbor set matched by an ingoing left hand side edge. In order to achieve a sound semantics for the negation we have to stratify the rules so that they still yield a unique normal form [Aß95a]. As strata exactly the rule groups (equation systems) result which are mentioned in [KRS94]. Here they are coalesced only for the purpose of demonstration. Also the given equations relate almost one-to-one to the equations in [KRS94]. Thus, writing a specification for global data-flow analysis, and generating an executable algorithm with OPTIMIX can be done in very short time. Actually this part seems to be the most easy part of the whole optimization process: take an arbitrary equation system, and type it in.

In this paper we consider only intraprocedural analysis. For interprocedural analysis special specification or evaluation strategies have to be applied to reach

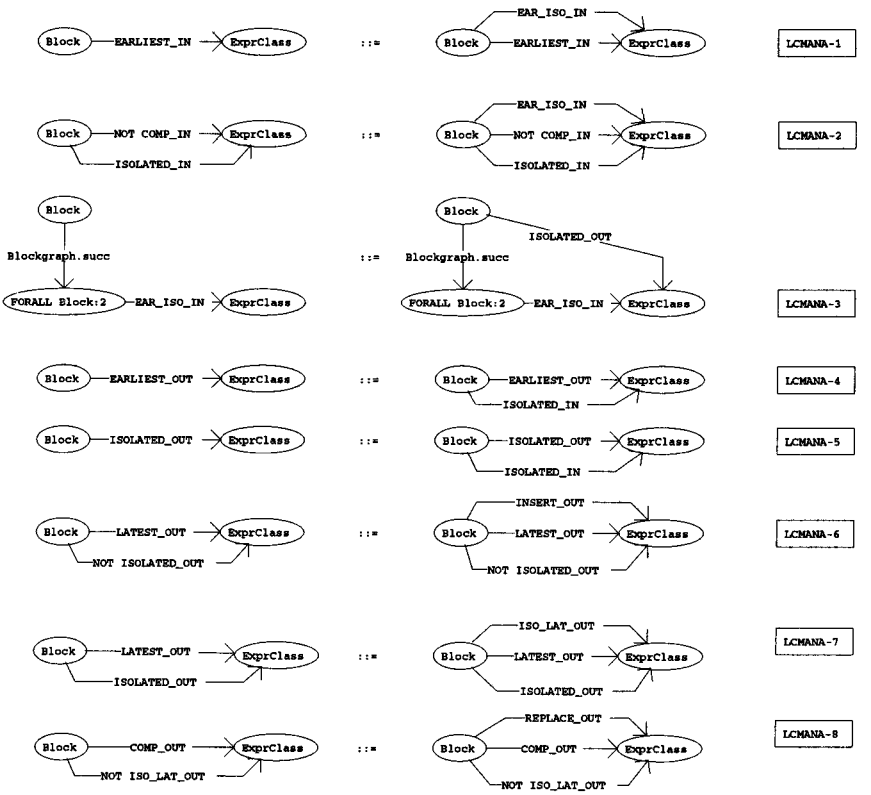


Fig. 4. LCMANA: Data-flow analysis for lazy code motion: isolatedness, earliestness, insert-out, replace-out. LCMANA-1 – LCMANA-5 make up the first rule group, LCMANA-6–8 the second

the same preciseness. As EARS model distributed data-flow frameworks over finite powersets there are several methods which can be applied [RHS95].

5 Specifying transformations

If we allow edge deletions, node additions, and node deletions, we can specify program transformations by deleting and adding objects from and to the intermediate representation. Our running example is continued by specifying some transformation rules for lazy code motion (Figure 5). We regard only the necessary relations for transformation at the end of blocks (`INSERT_OUT`, `REPLACE_OUT`). The first rule inserts an expression at the end of a block. No nodes are deleted; only new statements for creating the expression value and

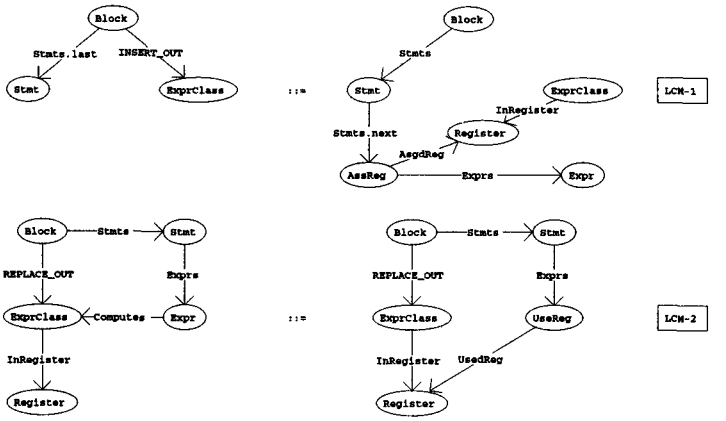


Fig. 5. LCM: Insert and replace of an expression at the exit of a basic block

storing it into a register are created. Edge `INSERT_OUT` is deleted in order to prevent that LCM-1 is applied again. LCM-2 also deletes nodes, because it has to replace a computing instruction of an expression class to a `UseReg` instruction which uses the already computed value from a register. In order to arrive at a complete lazy code motion transformation similar rules for the beginning of the blocks must be written.

With arbitrary additions and deletions we may loose termination and also (strong) confluence. Of course termination is absolutely necessary. [Aß95a] presents two termination criteria which are statically decidable:

1. Termination by *edge addition*. In the case of EARS the rewrite process stops when the added relations are complete. This can be carried over to general graph rewrite systems, if each rule adds an edge of such a *terminating relation* and no other rule is allowed to delete these edges again. Then the system terminates because the terminating relation is complete.
2. Termination by *redex deletion*: each rule deletes nodes or edges of types which are not added either. This means that although the rules may add nodes and edges of other types they do not produce any redex again. Thus after deletion of all initial redexes termination follows. This is the case for LCM: LCM-1 subtracts the edge `INSERT_OUT` and LCM-2 the edge `Computes`. Thus it terminates, although it adds statements and expressions.

These termination criteria are very useful if a graph rewrite system has to perform a finite number of actions depending on the size of the manipulated graph. This is the case for many code optimizations, especially for code motion and replacement algorithms: first they compute the places where to transform and then they transform only once.

If a graph rewrite system is not confluent, it delivers a correct, but arbitrarily selected result. LCM cannot be proved to be confluent regarding its rules. However, because the redexes in the manipulated graph do not overlap, it is in deed confluent and delivers unique results.

6 Execution

6.1 Optimix meta-code-generation

In order to understand how the specifications given so far can be executed efficiently, we will give a short overview on a special (meta-)code-generation for graph rewrite systems [Aß95b], the *order evaluation*. This technique is based on the idea that we will find all redexes of a rule if we regard all possible permutations of source nodes of left hand sides and look up the rest of the redex by traversing neighbor sets. For that the *order* of the graph rewrite system has to be calculated, which is the maximal number of source nodes in left hand sides. If a graph rewrite system has order k, we also call it GRS(k).

For order evaluation we have to compute an *edge disjoint path cover* for each left hand side. This is a set of paths which cover the left hand side such that they intersect each other only at their end points. Then the problem of finding a redex for a given set of source nodes in the manipulated graph is reduced to a join of the path problems of the edge disjoint path cover. Consider Figure 6 which contains a path cover of LCM-2. For this rule OPTIMIX generates the code in Figure 7. The outer loops (1) and (2) enumerate the path problem of path 1. The inner loops (3) and (4) enumerate path 2. The paths are joined with a nested-loop-join under the join condition $c = c2$, because we need to find the intersection of their enumerated objects.

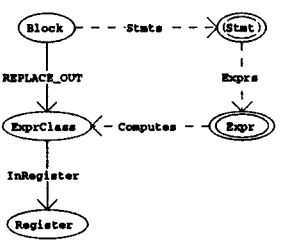


Fig. 6. A edge-disjoint path cover for LCM-2 with two paths. Path 1 is drawn with normal edges, path 2 with dotted edges

For non-recursive (recursive) graph rewrite systems the *order evaluation* has complexity $O(n^k e^{lp})(O(n^{k+2}e^{lp}))$, where l the length of the longest path of a path cover over all left hand sides, p the maximum number of paths in a path

Input: Node type Block. Relations Stmts,
Exprs, REPLACE_OUT, Computes, InRegister

Output: Modified intermediate code: expressions replaced by UseReg expressions

```
/* Enumerate path 1 */
forall b ∈ Block do (1)
    forall c ∈ b.REPLACE_OUT do (2)
        r ← c.InRegister; /* 1:1-relation */
        /* Join with path 2 */
        forall s ∈ b.Stmts do (3)
            forall e ∈ s.Exprs do (4)
                c2 ← e.Computes; /* n:1-relation */
                /* Join condition */
                if not c = c2 then
                    continue ;
                /* Redex (place for transformation) found. */
                .. Do the transformation ..
```

Fig. 7. Algorithm generated for LCM-2

cover, n the maximal number of nodes in a node type, and e the maximum out-degree of a node concerning an arbitrary relation. However, for concrete algorithms often better costs result because many relations are n:1 and often object domains are partitioned by graphs. Because the test for LCM-1 can be overlapped with the loops of algorithm 7, it can be used to solve LCM. LCM is non-recursive, because it does not create new redexes for itself. According to the order evaluation cost formula it has complexity $O(|\texttt{Block}|^1 e^6)$, because $k = 1, l = 3, p = 2$. Then **Computes** and **InRegister** are n:1- and 1:1-relations, respectively. Also expressions are partitioned over the blocks, i.e. loop (3) and (4) only loop once over all expressions of a procedure. Thus the generated algorithm has cost $O(|\texttt{Block}||\texttt{Expr}||\texttt{ExprClass}|)$. It is also clear that if other directions are chosen for the relations of the data model, a graph rewrite system with higher order may result. For our example we use already a reasonably good one, but in general this needs some thinking.

EQUIV has order 2, because the rules for attribute-equivalence contain isolates in the left hand sides. It is recursive. A theorem in [Aß95b] shows that because we deal with expression trees the fixpoint computation can be avoided. Also, **Left** and **Right** are n:1-relations. Thus EQUIV can be solved with order evaluation in $O(|\texttt{Expr}|^2 e^3)$ because $l = 1, p = 3$. However, e is here much smaller than |**Expr**| because the relations **eq** and **attreq** partition **Expr**. If we neglect their cardinality, we achieve a quadratic algorithm in the number of expressions.

LCMANA has order 1, because **Block** is the only source node type of all rules. Because it is recursive, we have to apply fixpoint computation. Because $l = 2, p = 2$, we have complexity $O(|\{\texttt{Block},\texttt{ExprClass}\}|^3 e^4)$. However, if we use a bitvector representation for the relations, the standard round-robin iteration for data-flow analysis results. Because bitvector union/intersection has linear effort in the number of expressions, a concrete algorithm has cost $O(|\texttt{Block}|^2|\texttt{ExprClass}| \times |\texttt{Block}||\texttt{ExprClass}|) = O(|\texttt{Block}|^3|\texttt{ExprClass}|^2)$. This rough estimate can be improved by taking the loop nesting of the control-flow graph into account [Hec77].

All terminating graph rewrite systems can be solved by order evaluation. For terminating and confluent systems it will deliver the unique normal form. For non-confluent systems it will deliver a correct, but arbitrarily selected result. Hence order evaluation provides a simple and uniform solution procedure for program optimization problems. Thus we are able not only to specify analysis and transformation uniformly, but also execute the specifications with a uniform execution mechanism.

6.2 Speeding up

The following sections present some numbers and shows how to speed up the execution of order evaluation. Within CoSy two optimizer configurations have been tested: a Modula-2 compiler with lazy code motion, and a C compiler with copy propagation.

Our experience is that by using graph rewrite systems the development time of an optimizer is greatly reduced. Our estimate is that about 50% savings are possible. For instance, equation systems such as LCMANA can be typed in in a few hours. If the generator generates correct code the correctness of the generated algorithm can be achieved quite quickly. Also in transformation the savings are enourmous: normally it is a tedious work to write algorithms which search for the transformation places. Using graph rewriting this is automatic.

Also the relation of generated and hand-written part in the optimizers is very interesting. Because OPTIMIX does not yet implement all possible features, still some parts are hand-written. Nevertheless, the constructed optimizer components consist of about 60-80% of generated code and the hand-written parts are not critical for the runtime of the optimizers. We estimate that with an industrial-strength implementation of OPTIMIX 90% of an optimizer can be generated without problems.

6.3 Effectiveness of generated optimization phases

The lazy code motion optimizer demonstrates that graph rewriting in deed can produce effective optimizers. The following table shows the execution times in seconds of some routines of the Stanford benchmark from Henessy and Nye. The optimizer achieves up to 32% speedup, although it is a prototypical implementation. Gcc and sun-cc are much more effective here. However, the quality of the optimized program code is a matter of the quality of the specification of the optimizer components. Thus much better results are possible by improving the specification, e.g. by including better alias information.

Routine	without LCM	with LCM	speedup in %	gcc -O4	sun-cc -O4
Queens	11.9	10.9	8	4.7	5.2
MatrixMult	12.6	8.5	32	2.9	5.9
Puzzle	4.9	4.0	18	1.4	1.8
Quicksort	8.3	7.4	13	2.8	6.1
Erastothenes	32.5	24.1	25	24.7	19.5

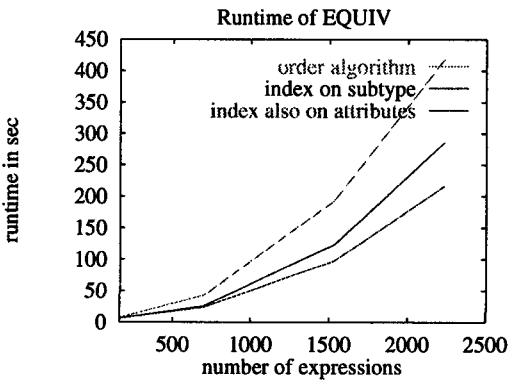
6.4 Overall optimization speed

In order to test the compilation speed of the generated components, the two compilers were compared running with and without optimization, compiling the Stanford benchmark. The table shows that the optimizers add about factor 3-7 to the runtime of the compilers. Clearly gcc and sun-cc are faster here, although they perform more optimizations. However, for generated optimizer parts the results are very good: OPTIMIX and the specified components are quite new implementations which leave a lot of possibilities for improvements. Also the lazy code motion optimizer performs already rather a computation-intensive optimization: it computes four data-flow equation fixpoints and syntactical expression equivalence. Again, with an industrial-strength implementation the velocity of the generated optimizer parts could very well reach that of hand-written parts.

Compiling Stanford benchmark	Slowdown factor
M-2, lazy code motion	7.2
C, copy propagation	2.9
gcc -O4	2.2
sun-cc -O4	3.1

6.5 Speeding up expression equivalence

[Aß95b] shows that an index structure can be used to speed up EQUIV. This structure maps attributes to nodes and simulates virtual edges between the two isolates of EQUIV-1/EQUIV-2. With this modified production EQUIV turns into an EARS(1) and its cost changes to $O(|\mathbf{Expr}|e^3)$. If the maximum number of elements in expression classes is small compared with the number of expressions, the runtime of EQUIV will be dominated by the cubic cost factor only for very large programs.



This assumption is supported by the diagram above. Here the runtime of a Modula-2 compiler including the EQUIV algorithm is measured. The upper curve shows clear quadratic runtime in the number of expressions of the program (order evaluation). The curve in the middle is measured with a hash index over

the expressions; the hash index function comprises the subtypes of the expression nodes. The lower curve additionally comprises attributes of the most frequent expressions. Although the behaviour is not linear in the number of expressions the cubic factor starts to dominate the runtime only when very large procedures are compiled. Thus using a good hash function doubles the speed of the compiler: the use of index structures is very important in practice.

6.6 Graph representations

We also measured the influence of the graph representations on the runtime of the Modula-2 compiler. For that the LCMANA component was run with a list-based and a bitvector-based implementation for the data-flow sets. Both representations can be exchanged by changing the concrete class of the relations in the fSDL data specification (and of course, by adapting any non-generated code). When compiling the Stanford benchmark, the bitvector implementation is about 6 times faster than the list-based implementation. The difference lies in the implementation of the union and intersection operation of neighbor sets: on bitvectors these operations are linear and on lists they are quadratic.

This shows that it is very important which data representations are chosen for the manipulated graphs. Thus the final phase of writing an optimizer with graph rewriting consists of selecting the right data representation. Fortunately this can be done by only changing the data model; the generator adjusts its code generation automatically. This reveals a unique strength of our method: the specification is independent of the representation of the graphs.

7 Related work

Sharlit [TH92] and PAG [AM95] are two tools for generating efficient data-flow analyses. With both tools users have to supply data structures in C for the lattice elements and also for flow functions. Thus exchange of implementations is not so easy. SPARE [Ven89] follows the same approach and is additionally tied to the Synthesizer Generator. Generation of data-flow analysis from modal logic specification stems from [Ste91]. Although the powerful modal operators allow very short specifications, an application in a real-life compiler is not known. All these tools allow for the specification of more complex lattices and flow functions. However, we believe that our method is much more intuitive for the average programmer because it relies on the more familiar concept of graphs.

Only few approaches are known which integrate transformations. SPECIFY [Koc92] additionally provides a proof language but was never implemented completely. GENESIS [WS90] allows very powerful transformation specifications. Preconditions can be specified in a variant of first-order logic. However, because fixpoint computations cannot be specified, generation of data-flow analysis is not possible. Also it seems that the code generation scheme is quite ad hoc.

Also the existing tools for graph rewrite systems could be used for generation of program optimizers. PROGRES [Sch90] is the most advanced. However,

it is designed for an interactive user environment and not for batch processing. Currently it does not allow for fixpoint computations and is tied to an underlying database, although it provides an excellent user interface. UBS systems [Dör95] provide a subclass of graph rewrite systems which can be handled more efficiently. However, the described implementation is still too slow for program optimization. OPTIMIX produces much faster algorithms: order evaluation of EQUIV (which only adds edges) performs at least $2300^2/400 = 13000$ redex tests/second[2]. Using an index the system is twice as fast: thus OPTIMIX should also provide one of the fastest existing tools to execute graph rewrite systems.

8 Outlook

We have presented in this paper a novel uniform specification method for program analysis and transformation. EARS can be used for analysis, terminating GRS for transformation. Both can be evaluated uniformly and efficiently with order evaluation. With this method for the first time complete optimizers can be specified. The prototypical tool OPTIMIX demonstrates that it is also possible to generate them.

EARS are equivalent to Datalog with binary predicates [Aß95a]. The idea that Datalog can been used to describe data-flow analysis has also been discovered by [Rep94]. However, the restriction to binary predicates makes it possible to use efficient graph search algorithms.

I would like to thank Jürgen Vollmer and Andreas Winter. As main implementors of the mentioned optimizer components and first users of OPTIMIX they forced me to spent a lot of nights in front of my machine; however, I believe, with nice results.

References

[Aß95a] Uwe Aßmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen.* PhD thesis, Universität Karlsruhe, Kaiserstr. 12, 76128 Karlsruhe, Germany, July 1995.

[Aß95b] Uwe Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, editor, *5th Workshop on Graph Grammars and Their Application To Computer Science*, to appear in Lecture Notes in Computer Science. Springer, 1995.

[AAvS94] M. Alt, U. Aßmann, and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In P. A. Fritzson, editor, *Compiler Construction, Lecture Notes in Computer Science 786*, pages 278–293. Springer Verlag, April 1994.

[AM95] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with pag. In A. Mycroft, editor, *Static Analysis Symposium*, volume to appear of *Lecture Notes in Computer Science, Spinger Verlag.* Springer Verlag, 1995.

[2] The measurements of EQUIV comprise the runtime of the frontend. For simplification a quadratic algorithm is assumed for the number of tests. In fact even more tests are done

[Dör95] Heiko Dörr. *Efficient Graph Rewriting and Its Implementation*, volume 922 of *Lecture Notes in Computer Science, Spinger Verlag.* Springer Verlag, 1995.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[Koc92] Gerd Kock. *Spezifikation und Verifikation von Optimierungsalgorithmen.* GMD Bericht 201, Universität Karlsruhe, 1992.

[KRS94] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *Transactions on Programming Languages and Systems*, 16(7), July 1994.

[Rep94] Thomas Reps. Solving Demand Versions of Interprocedural Analysis Problems. In P.A. Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 389–403, April 1994.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, volume 22, pages 49–61. ACM, January 1995.

[Sch90] A. Schürr. Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language. In *Graph-Theoretic Concepts in Computer Science*, volume 541 of *Lecture Notes in Computer Science*, pages 444–458. Springer Verlag, 1990.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. In *Proceedings of Theoretical Aspects of Computer Software (TACS)*, pages 346–364, 1991.

[TH92] S. W. K. Tjiang and J. L. Henessy. Sharlit – A tool for building optimizers. *SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[Ven89] G. A. Venkatesh. A Framework for Construction and Evaluation of High-Level Specifications for Program Analysis Techniques. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[VH92] Jürgen Vollmer and Ralf Hoffart. Modula-P, a language for parallel programming: Definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California*, pages 54–64. IEEE, IEEE Computer Society Press, April 1992.

[WKD94] H.R. Walters, J.F.Th. Kamperman, and T.B. Dinesh. An extensible language for the generation of parallel data manipulation and control packages. In P. Fritzson, editor, *Proceedings of the Poster Session of Compiler Construction*, number LiTH-IDA-R-94-11 in PELAB Research Report. Linköping University, 1994.

[WS90] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *ACM Conference on Principles and Practice of Parallel Programming (PPOPP)*, 1990.