Graph Rewrite Systems for Program Optimization

Uwe Assmann

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Graph Rewrite Systems For Program Optimization

Uwe Aßmann

N° 2955

August 1996

THÈME 2

ISSN 0249-6399

INRIA
ROCQUENCOURT

Graph Rewrite Systems For Program Optimization

Uwe Aßmann*

Thème 2 — Génie logiciel
et calcul symbolique
Projet OSCAR

Rapport de recherche no2955 — August 1996 — 39 pages

Abstract: This paper demonstrates how graph rewrite systems can be used to specify and generate program optimizations. For termination of the systems we develop some new rule-based criteria, defining *exhaustive graph rewrite systems*. We also define *stratification* of graph rewrite systems which automatically selects single normal forms for many non-deterministic systems. To illustrate the technology we specify parts of the lazy code motion optimization. For the resulting graph rewrite system classes a uniform evaluation algorithm is given. On the basis of this method the optimizer generator OPTIMIX has been developed. It is one of the first tools which can be used to specify analysis and transformation uniformly and we present compilation time results of generated optimizer components.

Key-words: Compiler generator, program optimization, specification, stratification, very high-level languages, visual programming

(Résumé : tsvp)

I would like to thank Martin Jourdan and my other colleagues from the group OSCAR at INRIA Rocquencourt. During my post-doc year they provided an excellent environment to write this paper. Berthold Hoffmann (University of Bremen) proof-read the paper, and proposed the term *exhaustive graph rewrite systems* to replace an earlier, more ugly name. Carol-Ina Trudel helped me to improve my English. An previous, much restricted version of this paper appeared in [Aß96]. This work has partly been supported by Esprit project No. 5399 COMPARE.

* Uwe.Assmann@inria.fr. From Nov. 96 on: University of Karlsruhe, Institut für Programm- und Datenstrukturen, Vincenz-Priessnitz-Str. 3, 76128 Karlsruhe, Germany, assmann@informatik.uni-karlsruhe.de

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Systèmes de réécriture de graphe pour l'optimisation de programmes

Résumé : Cet article explique comment les systèmes de réécriture de graphe peuvent être utilisés pour spécifier et générer des optimisations de programmes. Pour la terminaison des systèmes, nous avons développé de nouveaux critères sur les règles, en définissant des *systèmes de réécriture de graphe exhaustifs*. Nous avons également défini la *stratification* des systèmes de réécriture de graphe, qui permet de choisir automatiquement une unique forme normale pour de nombreux systèmes non-déterministes.

Pour illustrer cette technique, nous spécifions quelques étapes du problème d'optimisation du déplacement de code paresseux. Nous donnons pour les classes résultantes un algorithme uniforme d'évaluation. Le générateur d'optimiseurs OPTIMIX a été développé sur la base de cette méthode. C'est l'un des premiers outils qui traite uniformément des analyses et transformations. Nous présentons quelques résultats de temps de compilation des composants d'optimiseurs ainsi générés.

Mots-clé : Générateur de compilateurs, optimisation de programmes, spécification, stratification, langages de haut niveau, programmation visuelle

1 Introduction

Writing an optimizer for a programming language is an error-prone and tedious task. In order to shorten development time, to facilitate validation, and to improve maintainability, a methodology for specification and generation of optimizers is highly desirable. However, until now specification methods handled only subtasks of program optimization, or they led to optimizers which executed too slowly, or they were closed systems which were tied to a specific environment or intermediate language.

This paper demonstrates how graph rewriting can be used for specification and generation of program optimization. The central idea of our method is to regard all information in an optimizer as a set of graphs. Program objects, intermediate code instructions, and predicates over the entities of the program are all represented either as nodes or edges in these graphs. Program analysis and transformation are performed by graph rewriting. Typically, analyzing programs means enlarging graphs with new edges which represent the information, while code transformation means rewriting graphs by deleting and attaching subgraphs. Thus the optimization process is divided into a sequence of graph rewrite system applications, starting from the intermediate representation given by the front-end, and resulting in an intermediate graph which is handed over to the back-end for code generation.

However, when trying to specify program optimization with graph rewrite systems, we may easily specify systems that neither terminate nor yield unique normal forms. Thus our paper defines some new rule-based termination criteria, *termination by edge addition* and *termination by subtraction*. They result in the class of *exhaustive graph rewrite systems*. Furthermore, we develop *stratification* of graph rewrite systems, which automatically selects a *stratified normal form* for many non-deterministic systems. This technique is especially important for the specification of program optimization which requires negation, deletion and addition of arbitrary subgraphs. This often results in non-deterministic systems delivering non-expected results. Stratification selects normal forms which are rather natural because the selected derivations retain subgraphs which are otherwise deleted.

A practical method for generating optimizer components will be judged on three criteria. First it must be able to express a broad range of program analysis and transformation problems. We demonstrate this by specifying several parts of the *lazy code motion* analysis and transformation [KRS94]. Second the generated algorithms must run efficiently so that the components can be used in practical optimizers. Thus we have developed a uniform evaluation algorithm for the given rewrite systems, which works efficiently on directed sparse graphs and often results in linear or quadratic algorithms. Third the specification method must be flexible and provide an open system. Our method can handle arbitrary source and intermediate languages because the shape of the graphs is specified in a data model. The optimizer components can be generated in standard programming languages, and may easily be integrated with existing front-ends or back-ends.

The structure of the paper is as follows. First we demonstrate the use of graph rewrite systems by a short example, the collection of intermediate code expressions. This can be described by a very simple graph rewrite system, and we give an idea of which code can be generated to execute it. Section 2 defines our notion of graph rewriting and exhaustive graph rewrite systems. Section 3 defines the stratification of graph rewrite systems. Then we present important parts of the lazy code motion optimization, including syntactic equivalence of intermediate expressions (section 4.2), global data flow analysis (section 4.3), and the transformation phase (section 4.4). Section 5 deals with the uniform evaluation algorithm, states its cost for our examples, and gives a short overview of other evaluation methods. Section 6 describes practical experiences with our method. In order to demonstrate its validity the optimizer generator OPTIMIX has been developed. It is one of the first tools which can be used to specify program analysis *and* transformation. It has been used to generate several prototypical optimizer parts in the compiler framework CoSy (Esprit project COMPARE) [AAvS94], and we present some figures concerning their efficiency. We conclude the paper with a section about related work and an outlook on future work.

Figure 1: COLLECT-EXPRESSIONS: Collection of expressions from the statement lists

Input: Node domains `Proc`. Relations `Blocks`, `Stmts`, `Exprs`

Output: New relation `AllExprs`

```
/* Enumerate all objects of Proc, Block, Stmt, Expr */
forall p ∈ Proc do (1)
    forall b ∈ p.Blocks do (2)
        forall s ∈ b.Stmts do (3)
            forall e ∈ s.Exprs do (4)
                /* Redex found. */
                p.AllExprs += e;
```

Figure 2: An algorithm solving COLLECT-EXPRESSIONS

1.1 Motivation

This section presents a short example. We specify a graph rewrite system that attaches all expressions in the statement list of a procedure to the procedure object. The simple graph rewrite system in Figure 1 consists only of one rule: COLLECT-EXPRESSIONS-1. The figure contains the rule in two forms. To the left the original rewrite rule is drawn. If a rule only adds edges, we can also use a shorthand form, which draws the rule invariant part only once, and the added edges are dashed. This simplifies the specifications.

We assume that the intermediate representation provides the object types `Proc`, `Block`, `Stmt`, and `Expr`, as well as the directed relations `Blocks`, `Stmts`, and `Exprs`. COLLECT-EXPRESSIONS-1 attaches all expressions of all statements in a procedure into a directed relation `AllExprs`: if an edge `Blocks` exists between a procedure `Proc` and a block `Block`, and there is an edge `Stmts` from `Block` to a statement `Stmt`, and there is an edge `Expr` from `Stmt` to an expression `Expr`, then there will be an edge `AllExprs` from `Proc` to `Expr`. Thus the only action which COLLECT-EXPRESSIONS performs is to add edges of label `AllExprs` to the host graph.

Our aim is to generate optimizers and optimizer parts automatically from specifications. Thus, given this graph rewrite system, a practical optimizer generator should generate an executable procedure to be embedded into a compiler. OPTIMIX is such a tool, and Figure 2 contains the layout of the code it generates for COLLECT-EXPRESSIONS. For our example we assume that the given relations are represented as sets of neighbor sets in objects (directed graphs with embedded neighbor sets). As all objects, which may be matched by COLLECT-EXPRESSIONS-1, must be reachable from a procedure via the depicted relations, we simply generate several loops over the relevant neighbor sets. In our case the rule test condition is very simple: whenever we have enumerated a suitable permutation of nodes (a *redex*), we draw an edge between the current `Proc`-node and the current `Expr`-node. If we look at the resulting algorithm superficially, it has cost $O(|\texttt{Proc}||\texttt{Blocks}||\texttt{Stmt}||\texttt{Expr}|)$. However, in most intermediate representations the mentioned relations form a tree-like structure which partitions the objects. Thus we get a linear algorithm in the number of expressions.

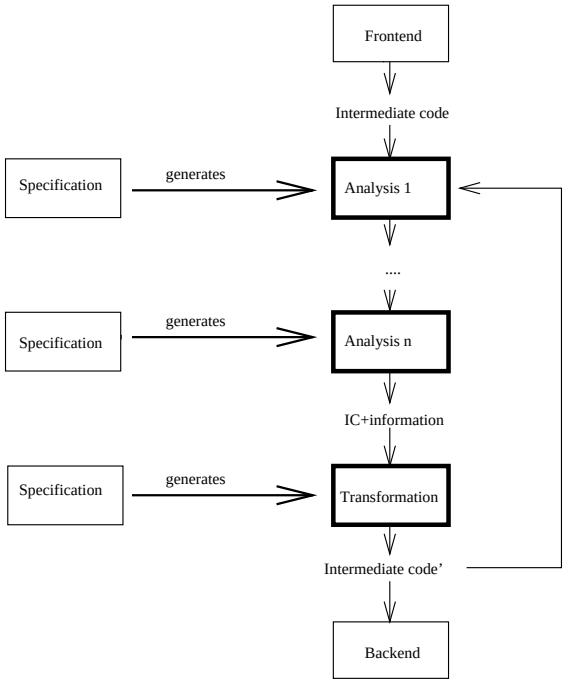

Figure 3: We regard the optimization process as a sequence of graph rewrite system applications. Thick boxes are phases which are generated from specifications.

1.2 Overview of the method

When we specify program analysis and transformation with graph rewriting we attain a uniform view of the intermediate language and the analysis information: everything is represented as graphs, and all actions are done by graph rewriting. Moreover, we can split and coalesce specifications easily, which enables modular composition of program optimizations. However, when we increase the number of rules, we may easily specify non-terminating or non-deterministic systems. Thus it is better to divide the optimization process into a sequence of graph rewrite system applications which are separate and thus smaller (Figure 3). Each of these graph rewrite systems generates a procedure in the implementation language of the compiler, and this procedure is called from the main compiler routine appropriately. The optimization process starts with a graph given by the front-end, which is the abstract syntax tree or the intermediate code sequence. First program analysis is performed. Typically this is divided into several graph rewrite systems, among which are one or several data-flow analyses. Then the transforming graph rewrite systems can be applied. After some iterations of analyses and transformations the modified code is handed over to the back-end.

The advantages of this scheme are the following. First, a suitable split into smaller graph rewrite systems isolates non-confluence effects and/or resolves non-termination. Second, for smaller systems we can often generate linear or quadratic algorithms because smaller rewrite systems often have a lower *order* than bigger ones (section 5). Third, this division breaks the optimization task down into component problems which can be solved independently. If we do not find a graph rewrite system for a certain task, or some developed system turns out to be too slow, we can always substitute it by a hand-written algorithm. In any event, this is the case for the front-end and the back-end; they are implemented as usual.

In order to make this scheme work, all specified and hand-written algorithms have to refer to a uniform data model of the intermediate representation. Therefore the first step of the method consists

of developing a *data model* or *graph schema* for the intermediate representation [Sch91] [SWZ95]. We have to model the following:

1. the types of graph nodes. Among these are intermediate code instructions such as expressions, statements, loops, and procedures. Also previously analyzed information can be encoded in nodes of graphs, e.g. variable definitions, or expression equivalence classes.

2. the types of graph edges (relations). We have to model all predicates we wish to infer about the program, i.e. the relations between the objects of the intermediate representation. Some examples are relations such as classical flow dependencies, equivalence relations, calling relations, or control flow relations. Also the information the front-end produces, e.g. expression tree pointering or statement lists, must be modeled.

Before we demonstrate the power of our specification method with several examples we define some theory which ensures that the given graph rewrite systems behave properly.

2 Termination orderings for GRS

This section defines our notion of graph rewrite system, some new termination orderings, and *stratification* of graph rewriting. The resulting graph rewrite system classes can be used not only for program analysis but also for transformation. First we define some new classes of terminating graph rewrite systems, called *edge-additive*, *edge-subtractive*, and *subtractive* graph rewrite systems. A subclass of edge-additive systems are *edge addition rewrite systems* [Aß95b]. Then we present a *stratification* heuristic for graph rewrite systems, which gives a unique semantics to many non-deterministic systems.

2.1 Graph rewriting

2.1.1 Graphs

Throughout this paper we consider rewritings of *directed, relational*, and *labeled* graphs.

Definition 1 *Let Σ be a finite set of constants, the* labels. *Σ is the disjunct union of two sets $\Sigma = \Sigma_N \cup \Sigma_E$, the* node *and* edge labels. *Then a* (relational) *Σ-graph $G = (N, E, nlab)$ consists of the following.*

1. *N is a finite set of* nodes.

2. *Nodes are labeled with labels from Σ_N by a function $nlab : N \to \Sigma_N$. We call a node with label l l-node. The set $N_l = \{n \in N | nlab(n) = l\} \subseteq N$ of all l-nodes forms a* node domain.

3. *The* edges *$E \subseteq (N \times N \times \Sigma) = \{E_l \subseteq N \times N\}_{l \in \Sigma_E}$, is a family of binary* relations *over $N \times N$, where each edge $e \in E$ is labeled by a $l \in \Sigma_E$. We call an edge with label l l-edge.*

Implicitly associated with G are several functions. $elab : E \to \Sigma_E$ projects an edge to its label. The functions $source, target : E \to N$ provide source and target nodes for edges. We say that $e \in E$ is *incident* to $source(e)$ and $target(e)$ while $source(e)$ and $target(e)$ are *adjacent*. All adjacent nodes of a node are called its *neighbor set*. The function *in-degree*: $N \to I\!N$ of a node is the number of incoming incident edges $in\text{-}degree(n) = |\{e \in E | target(e) = n\}|$. Similarly define the function *out-degree* to be the number of outgoing incident edges $out\text{-}degree(n) = |\{e \in E | source(e) = n\}|$. A node with $in\text{-}degree(n) = 0$ is called *root*. If necessary, we index the components of a Σ-graph $G = (N_G, E_G, nlab_G)$, extend the functions $nlab$ and $elab$ to sets of items, and use *in-degree* and *out-degree* restricted to specific edge labels.

Contrary to other approaches in the literature we do not deal with arbitrary multi-graphs. Because $E \subseteq (N \times N \times \Sigma_E)$, between two nodes several edges are allowed, however, their labels must be distinct. For the purposes of optimization it is not necessary to allow several edges of the same label between two nodes: edges represent predicates on their source and target node objects, and a predicate either holds or does not hold. However, it is very important to have different relations because we want to represent different kinds of information. On the other hand, although attributes for nodes may be introduced, this is left out for the sake of simplicity. If designed carefully (only integer values, only matching in rules), node attributes do not change the results obtained in this paper, because they can be seen as extension of the node label set Σ_N.

Several operations on Σ-graphs are standard extensions of set operations over nodes and edges. A Σ-graph $G = G_1 \cup G_2$ is called *union* of G_1 and G_2, if $G = (N_{G_1} \cup N_{G_2}, E_{G_1} \cup E_{G_2}, nlab_{G_1} \cup nlab_{G_2})$. Similarly define the Σ-graph-intersection $\cap$ and the Σ-graph-inclusion $\subset$. The *difference* $G = G_1 - G_2$ of G_1 and G_2 is $G = (N = N_{G_1} - N_{G_2}, E|_N, nlab_{G_1}|_N)$. The set of all Σ-graphs over Σ is called $\mathcal{L}^\Sigma$. The set of all Σ-graphs over a fixed set of nodes N, $\mathcal{L}^\Sigma_N := (\mathcal{L}^\Sigma|_N, \cup, \subset)$, forms a finite lattice.

A Σ-graph-morphism $g : G \to H$ consists of two functions $g_N : N_G \to N_H$ and $g_E : E_G \to E_H$ which fulfil two conditions:

label preservation $\forall n \in N_G : nlab_H(g_N(n)) = nlab_G(n)$, $\forall e \in E_G : elab_H(g_E(e)) = elab_G(e)$

incidence preservation $\forall e \in E_G : source_{E_H}(g_E(e)) = g_N(source_{E_G}(e))$, $target_{E_H}(g_E(e)) = g_N(target_{E_G}(e))$

If both g_N and g_E are injective, g is called *injective*. If both g_N and g_E are surjective, g is called *surjective*. If g is injective and surjective it is an *isomorphism*. g_N/g_E are often extended to sets of nodes and edges. If clear from context, g_N/g_E are also called g. We often write gG instead of $g(G)$ to save brackets.

2.1.2 Graph rewrite rules

In order to define graph rewrite rules and to allow some more expressiveness for their left-hand sides, we first have to extend the definition of a Σ-*graph*.

Definition 2 *A $\Sigma^\neg$-graph $J = (N, E, nlab, negated)$ is a Σ-graph, augmented by a function $negated : E \to \mathbb{B}$ which tells whether an edge e of the graph is negated ($negated(e) = true$). J has a related Σ-graph J^{pos} with all negated edges stripped ($J^{pos} = (N, E - \{e \in E | negated(e) = true\}, nlab)$. The set of all $\Sigma^\neg$-graphs is called $\mathcal{L}^{\Sigma^\neg}$.*

A *graph rewrite rule* $r = (L \supseteq K \subseteq R)$ consists of a *left-hand side* $L \in \mathcal{L}^{\Sigma^\neg}$, a *rule invariant* $K \in \mathcal{L}^\Sigma$ *(interface graph)* , and a *right-hand side* $R \in \mathcal{L}^\Sigma$ (Figure 4). Negated edges are only allowed in left-hand sides; they are used to test the absence of subgraphs. In the given figures of graph rewrite rules we annotate a node by its label, and if necessary by an identification number, in order to identify the same nodes on left- and right-hand sides. Each rule also has a name (here Rs-1). We also use the following terms:

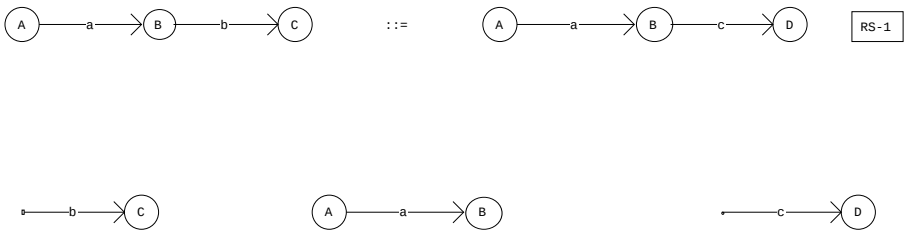

Figure 4: Above: Left- and right-hand side. Down left: rule deletion items. Down middle: rule invariant K. Down right: rule addition items

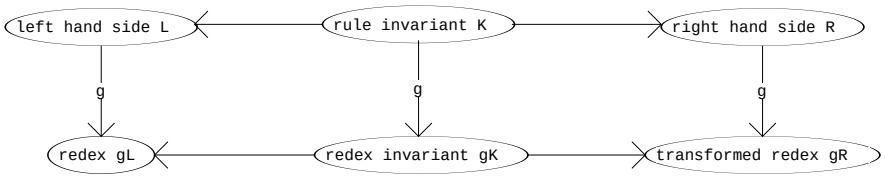

Figure 5: Relating of a rule and a redex with the occurrence g

$N^{test}(r) := N_L$	(rule test nodes)
$E^{test}(r) := E_L$	(rule test edges)
$\Sigma_N^{test}(r) := nlab_L(N_L)$	(rule test node labels)
$\Sigma_E^{test}(r) := elab_L(E_L)$	(rule test edge labels)
$\Sigma^{test}(r) := \Sigma_N^{test} \cup \Sigma_E^{test}$	(rule test labels)
$N^{inv}(r) := N_K$	(rule invariant nodes)
$E^{inv}(r) := E_K$	(rule invariant edges)
$\Sigma_N^{inv}(r) := nlab_K(N_K)$	(rule invariant node labels)
$\Sigma_E^{inv}(r) := elab_K(E_K)$	(rule invariant edge labels)
$\Sigma^{inv}(r) := \Sigma_N^{inv} \cup \Sigma_E^{inv}$	(rule invariant labels)
$N^{add}(r) := N_R - N_K$	(rule addition nodes)
$E^{add}(r) := E_R - E_K$	(rule addition edges)
$\Sigma_N^{add}(r) := nlab_R(N_R - N_K)$	(rule addition node labels)
$\Sigma_E^{add}(r) := elab_R(E_R - E_K)$	(rule addition edge labels)
$\Sigma^{add}(r) := \Sigma_N^{add}(r) \cup \Sigma_E^{add}(r)$	(rule addition labels)
$N^{del}(r) := N_L - N_K$	(rule deletion nodes)
$E^{del}(r) := E_L - E_K$	(rule deletion edges)
$\Sigma_N^{del}(r) := nlab_L(N_L - N_K)$	(rule deletion node labels)
$\Sigma_E^{del}(r) := elab_L(E_L - E_K)$	(rule deletion edge labels)
$\Sigma^{del}(r) := \Sigma_N^{del}(r) \cup \Sigma_E^{del}(r)$	(rule deletion labels)

2.1.3 Rule application

A rule $r = (L \supseteq K \subseteq R)$ is applicable to a Σ-graph G, the *host graph*, if the following conditions hold:

(t1) matching (subgraph test) Find an injective Σ-graph-morphism $g : L^{pos} \to G$, called *occurrence*. $g(L^{pos})$ is called *redex*, its nodes *redex nodes*, its edges *redex edges*. Because L^{pos} is used, negated edges from L are neglected, and we also abbreviate it to gL. gL need not be an *induced*

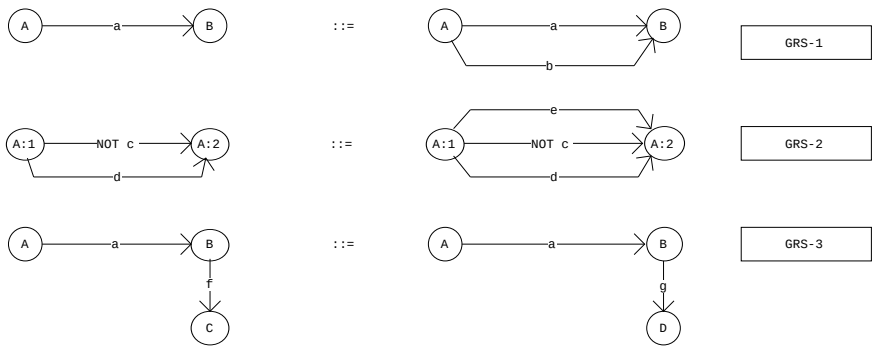

Figure 6: GRS: a simple graph rewrite system

subgraph of G, i.e. if $g_N(N_L)$ is the set of redex nodes from gL, not all linking edges of nodes in $g_N(N_L)$ need to be in gL. The graph $G - gL$ is called *rest graph*. Edges from $G - gL$ into gL may exist (*hidden edges, dangling edges*).

(t2) negated edge test $(e = (n_1, n_2, l) \in E_L, negated(e) = true) \Longrightarrow \neg\exists(g_N(n_1), g_N(n_2), l) \in E_G$, i.e if an l-edge in E_L is negated, between the corresponding nodes in $g_N(N_L)$ no l-edge may exist.

(t3) relation completeness test $\exists e = (n_1, n_2, l) \in E^{add}(r) : (g_N(n_1), g_N(n_2), l) \notin E_G$ i.e. not all edges which the application of r would add are already in G.

If r is applicable, r derives from G the Σ-graph H by performing the following steps in order (*direct derivation* $G \rightarrow H$):

(a1) redex edge deletion All redex edges matched by rule deletion edges are deleted: $E_G := E_G - g_E(E^{del}(r))$

(a2) hidden edge deletion All redex edges are deleted which are incident to nodes matched by rule deletion nodes: $E_G := E_G - \{(n_1, n_2, l) \in E_G | n_1 \vee n_2 \in g_N(N^{del}(r))\}$

(a3) node deletion All redex nodes matched by rule deletion nodes are deleted: $N_G := N_G - g_N(N^{del}(r))$

(a4) node addition A set of new nodes corresponding to $N^{add}(r)$ is added to G: $N_G := N_G \cup \{n | m \in N^{add}(r), nlab_G(n) = nlab_R(m)\}$

(a5) edge addition A set of new edges corresponding to $E^{add}(r)$ is added to G, if they are not already in G: $H := (N_G, E_G \cup \{(g_N(n_1), g_N(n_2), l) | (n_1, n_2, l) \in E^{add}\}, nlab'_G)$ where $nlab_G$ has been appropriately extended. E_G and E_H are sets, i.e. after the application of r there will still be only one l-edge between $g_N(n_1)$ and $g_N(n_2)$, so that H is a well-formed Σ-graph.

2.1.4 Graph rewrite systems

A *(relational) graph rewrite system* $\mathcal{G} = (S, Z)$ consists of a set of graph rewrite rules S and a Σ-graph Z (the *axiom*). The set of all graph rewrite systems is called GRS.

Figure 6 contains an example of a graph rewrite system. Rule GRS-1 adds a `b`-edge between two nodes with labels `A` and `B` which are linked by an `a`-edge (a5). GRS-1 is not applied if there is already

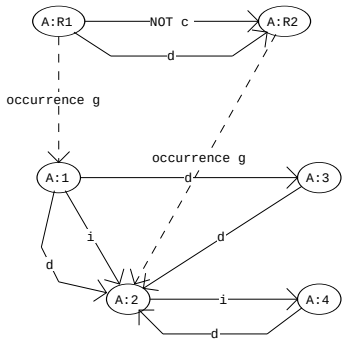

Figure 7: Matching of GRS-2 to an example host graph. Above the rule, down the graph. Rule nodes are marked by R1/R2. Negated rule edges test the absence of edges in the host graph

an `b`-edge (t3). GRS-2 adds an `e`-edge between two `A`-nodes (a5) which are not already linked by an `c`-edge (t3). Because we use an injective occurrence g in rule application condition (t1), $g(\texttt{A:1})$ and $g(\texttt{A:2})$ must be different. GRS-3 removes a `C`-node (a3) together with an incoming `f`-edge (a1) and allocates a new `D`-node (a4) with incoming `g`-edge (a5). Figure 7 shows an occurrence of GRS-2 in a possible host graph. The rule matches a non-induced subgraph containing the redex nodes `A:1` and `A:2`.

In this paper we use a specific graph rewrite approach. None of the approaches reported in the literature seems to meet the execution efficiency criteria which are required for program optimization. This concerns embedding of rules, gluing conditions, and hidden edges. We restrict the rule application conditions to those that can be tested in constant time when a set of nodes of the host graph is under investigation. Thus we use *invariant* rule embedding [BFG95], i.e. we do not allow hidden edges to be redirected, split or coalesced. Redexes always refer to a fixed number of nodes and edges. More liberal embedding mechanisms would form additional rule application conditions. Although this resembles the algebraic approach [EKL90], we do not use its *gluing condition* because it prohibits hidden edges to nodes which are deleted. To test whether hidden edges exist is rather expensive on directed graphs because we have to look up sources of incoming edges at every rule application. Hence rule applications need not be pushouts in the category of graphs and a rule does not always have an inverse. However, this is not important because we are only interested in rewriting and not in parsing.

Some of the approaches reported in the literature are not appropriate because they use induced subgraph tests instead of matching. If — for the application of a rule — all linking relations between nodes were to be enumerated, specifications would be rather difficult to read and write. Realistic optimizer specifications will work on an enormous amount of relations over the object types of the intermediate representation. We estimate that a full-fledged optimizer will use about 30 - 100 object types and around 100 - 500 relations. Also induced subgraph tests prevent specifications from being extended easily; if new relations are added to host graphs, formerly matching rules suddenly need not match anymore [BFG95]. With matching, we can abstract from all non-relevant relations. However, if nodes are deleted, all incident edges of the node must also be deleted. Section 5 clarifies how this can be done with linear execution cost.

2.1.5 Derivations

Let be $\mathcal{G} = (S, Z) \in \mathsf{GRS}$. A *derivation* in $\mathcal{G}$ is a sequence of Σ-graphs $G_0 \rightarrow_{r_1} G_1 \rightarrow_{r_2} \ldots \rightarrow_{r_n} G_n$, linked by direct derivations under rules $r_i \in S$. We say that G_n is *derivable* from G_0 ($G_0 \stackrel{*}{\rightarrow} G_n$). If there is no $r \in S$ which can be applied to G_n, G_n is called a *normal form* of $\mathcal{G}$. A derivation is

terminating if n is finite, otherwise *non-terminating*. If all derivations in $\mathcal{G}$ are terminating, $\mathcal{G}$ is also called *terminating*.

Two alternative direct derivations of rules $r_1 = (L_1 \supseteq K_1 \subseteq R_1)$ and $r_2 = (L_2 \supseteq K_2 \subseteq R_2)$ from S are called *independent*, if they can be interchanged: $G_2 \leftarrow_{r_1} G_1 \rightarrow_{r_2} G_3 \Longrightarrow \exists H : G_2 \rightarrow_{r_2} H \leftarrow_{r_1} G_3$. Otherwise they are called an *overlap*. The following definitions hold for $\mathcal{G}$ and its direct derivation relation $\rightarrow$. $\mathcal{G}$ is called *strong confluent* if $\forall G_1, G_2, G_3 \in \Sigma\text{-}graph : G_2 \leftarrow G_1 \rightarrow G_3 \Longrightarrow \exists H : G_2 \rightarrow_\lambda H \leftarrow_\lambda G_3$ where λ is the reflexive closure of $\rightarrow$. This is at least the case if all pairs of direct derivations are independent. $\mathcal{G}$ is called *confluent*, if $\forall G_1, G_2, G_3 \in \mathcal{L}^\Sigma : G_2 \stackrel{*}{\leftarrow} G_1 \stackrel{*}{\rightarrow} G_3 \Longrightarrow \exists H : G_2 \stackrel{*}{\rightarrow} H \stackrel{*}{\leftarrow} G_3$. $\mathcal{G}$ is called *convergent* if it is terminating and confluent [DJ90]. Such systems deliver a unique normal form [New42]. Graph rewrite systems with several normal forms are called *non-deterministic*.

2.2 Termination by edge addition

A set of graph rewrite rules S terminates on a finite axiom Z if every derivation step adds some edges to a particular relation. At last when this relation is complete the derivation process must stop because we only allow one edge with a certain label between two nodes.

Definition 3 (Edge-additive rule) *Let be* $\mathcal{G} = (S, Z) \in \mathsf{GRS}$. *Define for* $r = (L \supseteq K \subseteq R) \in S$ *the set of* edge-addition-termination labels

$$\Theta_E^+(r) := \{l \in \Sigma_E^{add}(r) | \forall e = (n_1, n_2, l) \in E^{add}(r) : nlab_R(n_1), nlab_R(n_2) \notin \Sigma_N^{add}(r)\}$$

the set of termination-subgraph node labels

$$\Theta_N(r) := \{l \in \Sigma_N | \forall e = (n_1, n_2, m) \in E^{add}(r), m \in \Theta_E^+(r) :$$

$$nlab_R(n_1) = l \vee nlab_R(n_2) = l\}$$

the termination-relations *in* Z

$$E_T(r) := \{e \in E_Z | elab_Z(e) \in \Theta_E^+(r)\}$$

the set of termination-subgraph nodes *in* Z

$$N_T(r) := \{n \in N_Z | nlab_Z(n) \in \Theta_N(r)\}$$

If $\Theta_E^+(r) \neq \emptyset$, *i.e. there are rule addition edges which are only incident to rule invariant nodes,* r *is called* edge-additive.

An edge-additive rule r *with* $N^{add}(r) = N^{del}(r) = E^{del}(r) = \emptyset$ *is called* edge addition rule *because it only adds edges.*

An edge-additive rule adds at least one rule addition edge which is only incident to rule invariant nodes. The label of this edge (these edges) denote a (set of) relations of the host graph (the termination-relations) which are being completed during the applications of the rule. This provides the basis for *edge-additive* graph rewrite systems:

Definition 4 (Edge-additive graph rewrite systems) $\mathcal{G} = (S, Z) \in \mathsf{GRS}$ *is called* edge-additive *if* $\forall r_1 = (L_1 \supseteq K_1 \subseteq R_1), r_2 = (L_2 \supseteq K_2 \subseteq R_2) \in S$:

$$\forall (n_1, n_2, l) \in E^{add}(r_1), l \in \Theta_E^+(r_1) : nlab_{R_1}(n_1), nlab_{R_1}(n_2) \notin \Sigma_N^{add}(r_2)$$

i.e. all rules are edge-additive and no rule adds nodes to the termination-subgraph nodes of another rule. Then define the set of edge-addition-termination labels *of* $\mathcal{G}$

$$\Theta_E^+(\mathcal{G}) := \bigcup_{r \in S} \Theta_E^+(r)$$

and the termination-subgraph *of $\mathcal{G}$ to be the subgraph of Z*

$$T := (N_T := \bigcup_{r \in S} N_T(r), E_T := \bigcup_{r \in S} E_T(r), nlab_Z|_{N_T})$$

The set of all edge-additive graph rewrite systems is called AGRS.

A graph rewrite system with edge addition rules is called edge addition rewrite system (EARS). *The set of all edge addition rewrite systems is called* EARS.

In edge-additive graph rewrite systems we can identify a subgraph of the axiom, the *termination-subgraph*. Its nodes carry labels from $\bigcup_{r \in S} \Theta_N(r)$, its edges from $\Theta_E^+(\mathcal{G})$. Rules may manipulate arbitrary nodes and edges, except that each of them only *adds edges* to the termination-subgraph. Thus the termination-subgraph is being completed and the system terminates. EARS are very specific graph rewrite systems because they only add edges to graphs [Aß95b]. We clearly have EARS $\subset$ AGRS.

Theorem 1 (Termination of AGRS) *Let be $\mathcal{G} = (S, Z) \in$ AGRS. $\mathcal{G}$ terminates on finite Z.*

If $T = (N_T, E_T, nlab_T) \subset Z$ is the termination-subgraph of $\mathcal{G}$ and $\Theta_E^+(\mathcal{G})$ is the set of edge-addition-termination labels, a derivation in $\mathcal{G}$ has at most $|\Theta_E^+(\mathcal{G})| \cdot |N_T|^2$ steps.

Proof: (1) $\mathcal{G} \in$ EARS. Starting from Z, a derivation $d = G \stackrel{*}{\rightarrow} H$ of $\mathcal{G}$ forms an ascending chain in $(\mathcal{L}^{\Sigma}_{N_Z}, \cup, \subset)$ because the rules of $\mathcal{G}$ only add edges. Due to rule addition action (a5) edges are added only *once*. Thus every direct derivation leads to another Σ-graph in $(\mathcal{L}^{\Sigma}_{N_Z}, \cup, \subset)$. Because $(\mathcal{L}^{\Sigma}_{N_Z}, \cup, \subset)$ is finite, the chain of the derivation is noetherian. Thus $\mathcal{G}$ terminates on Z.

(2) $\mathcal{G} \in$ AGRS$-$EARS. Because $\mathcal{G}$ adds and deletes items, the inputs of direct derivations need not be subgraphs of the results, and the derivable graphs do not form a lattice. However, every graph derivable from Z contains a subgraph which is restricted to the termination-subgraph nodes $N_T \subset N_Z$. Consider this set of subgraphs $(\mathcal{L}^{\Sigma}_{N_T}, \cup, \subset)$, and an arbitrary direct derivation $G \rightarrow_r H$. G has a subgraph $G' \in \mathcal{L}^{\Sigma}_{N_T}$. Because r must add an edge to a termination-relation of G, H has a subgraph $H' \in \mathcal{L}^{\Sigma}_{N_T} \subset H$ which is the result of applying r to G'. $G' \subset H'$ holds in $\mathcal{L}^{\Sigma}_{N_T}$. Therefore each derivation $d = G_0 \rightarrow G_1 \rightarrow \ldots \rightarrow G_n$ in $\mathcal{G}$ has associated an ascending chain of subgraphs of the G_i in $\mathcal{L}^{\Sigma}_{N_T}, G'_0 \subset G'_1 \subset \ldots \subset G'_n$. This sequence is always finite because $\mathcal{L}^{\Sigma}_{N_T}$ is finite. Thus d terminates, and also $\mathcal{G}$.

In both cases the number of edges in a complete graph over $N_T \subset N_Z$ with $|\Theta_E^+(\mathcal{G})|$ relations is limited by $|\Theta_E^+(\mathcal{G})| \cdot |N_T|^2$. Because every direct derivation adds an edge, its length can be at most $|\Theta_E^+(\mathcal{G})| \cdot |N_T|^2$.

■

As an example consider the graph rewrite system in Figure 1. Because we have N^{add}(COLLECT-EXPRESSIONS-1) $= \emptyset = N^{del}$(COLLECT-EXPRESSIONS-1) $= E^{del}$(COLLECT-EXPRESSIONS-1), COLLECT-EXPRESSIONS is an EARS. We have Θ_E^+(COLLECT-EXPRESSIONS-1) $= \{$`AllExprs`$\}$, Θ_N(COLLECT-EXPRESSIONS-1) $= \{$`Proc`, `Expr`$\}$. COLLECT-EXPRESSIONS terminates on a finite intermediate representation.

Figure 8 contains two other examples. Although TGRS adds nodes, a derivation terminates when the relation `terminator` is complete. We have Θ_E^+(TGRS-1) $= \{$`terminator`$\}$, Θ_N(TGRS-1) $= \{$`A`$\}$. Figure 9 depicts how a derivation in TGRS behaves: each rule application adds an edge to the termination-subgraph, while its nodes are left untouched. In contradistinction NTGRS additionally creates `A`-nodes. Thus there is no termination-relation and a derivation may be non-terminating.

Because the given termination criterion relies on the features of the rules alone it can be decided in an optimizer generator. EARS which do not use negation in their left-hand sides, are not only terminating, but also strong confluent, and thus convergent [Aß95b]. They can be used to specify intraprocedural distributive data-flow analysis [Aß95b]. Because EARS only add edges, they operate on the complete lattice of relations over the nodes of the axiom. Each direct derivation enlarges a

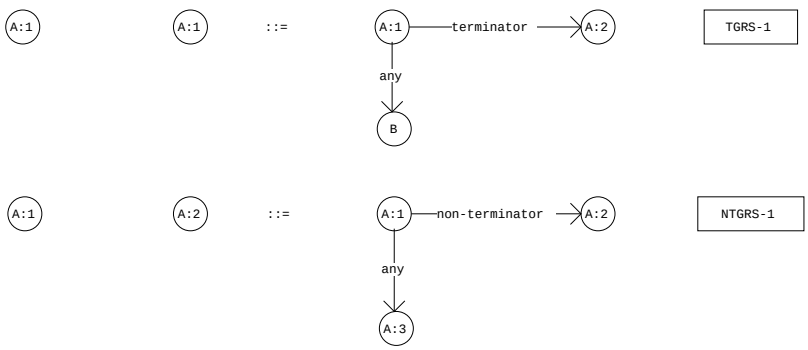

Figure 8: An edge-additive graph rewrite system with the edge-addition-termination label *terminator* and a non-terminating graph rewrite system

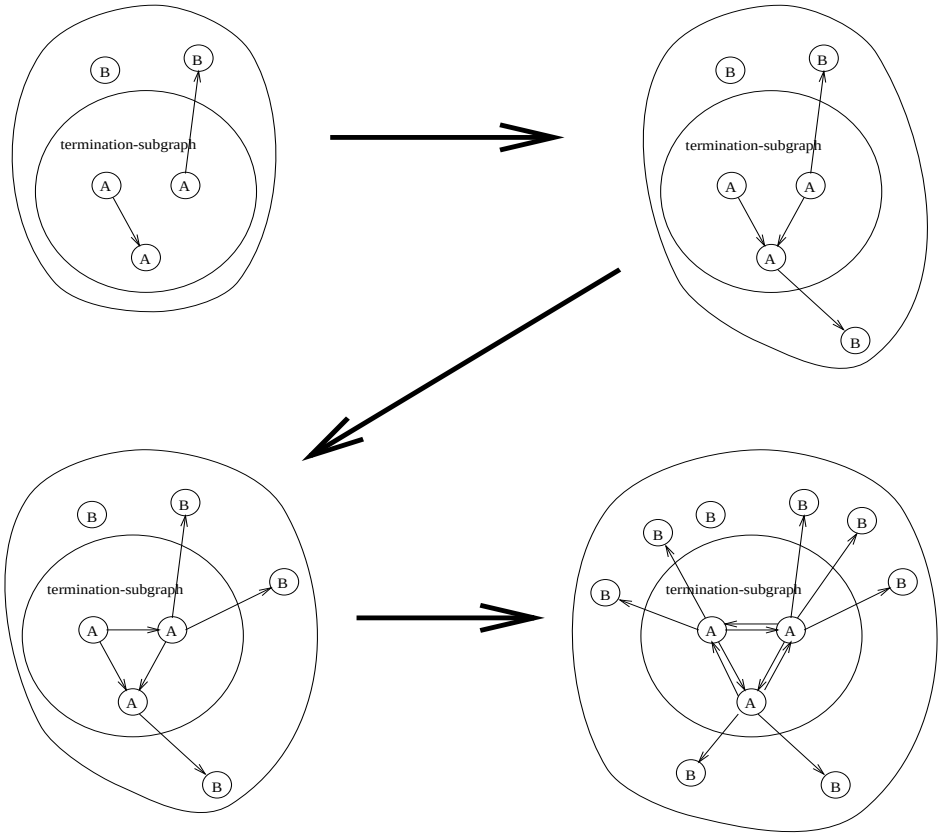

Figure 9: A derivation in the edge-additive graph rewrite system TGRS

relation and a derivation forms an ascending chain in the lattice. Because rule applications can be interchanged arbitrarily, each rule models a distributive function, and the EARS models a distributive data-flow framework [Aß95b]. Thus, finding a normal form of an EARS over an axiom is similar to finding a fixpoint in a distributive data-flow framework: we apply all rules to the host graph until a fixpoint, the normal form, is reached.

2.3 Termination by subtraction

We also can use edge or node deletion to achieve termination.

Definition 5 (Subtractive rule) *Let be* $\mathcal{G} = (S, Z) \in \mathsf{GRS}$. *Define for* $r = (L \supseteq K \subseteq R) \in S$ *the set of* edge-subtraction-termination labels

$$\Theta_E^-(r) := \{l \in \Sigma_E^{del} | \exists (n_1, n_2, l) \in E^{del}(r),$$

$$nlab_L(n_1), nlab_L(n_2) \notin \Sigma_N^{add}(r), l \notin \Sigma_E^{add}(r)\}$$

the set of termination-subgraph node labels

$$\Theta_N(r) := \{l \in \Sigma_N | \forall e = (n_1, n_2, m) \in E^{del}(r), m \in \Theta_E^-(r) :$$

$$nlab_L(n_1) = l \vee nlab_L(n_2) = l\}$$

the set of node-subtraction-termination labels

$$\Theta_N^-(r) := \Sigma_N^{del}(r) - \Sigma_N^{add}(r)$$

the termination-relations *in* Z

$$E_T(r) := \{e \in E_Z | elab_Z(e) \in \Theta_E^-(r)\}$$

the set of termination-subgraph nodes *in* Z

$$N_T(r) := \{n \in N_Z | nlab_Z(n) \in \Theta_N(r) \cup \Theta_N^-(r)\}$$

If $\Theta_E^-(r) \neq \emptyset$, r *is called* edge-subtractive. *If* $\Theta_N^-(r) \neq \emptyset$, r *is called* node-subtractive.

Similar to edge-additive graph rewrite systems our aim is to define a *termination-subgraph* from which items are subtracted until the system stops. Its nodes and edges carry labels from $\bigcup_{r \in S} (\Theta_N(r) \cup \Theta_E^-(r))$ (edge-subtractive systems) or from $\bigcup_{r \in S} (\Theta_N^-(r) \cup \Theta_N(r) \cup \Theta_E^-(r))$ (subtractive systems):

Definition 6 (Edge- and subtractive graph rewrite systems) *Define for* $\mathcal{G} = (S, Z) \in \mathsf{GRS}$ *the set of* edge-subtraction-termination labels *of* $\mathcal{G}$

$$\Theta_E^-(\mathcal{G}) := \bigcup_{r \in S} \Theta_E^-(r)$$

and the set of subtraction-termination labels *of* $\mathcal{G}$

$$\Theta^-(\mathcal{G}) := \bigcup_{r \in S} \Theta_E^-(r) \cup \Theta_N^-(r)$$

$\mathcal{G} = (S, Z) \in \mathsf{GRS}$ *is* edge-subtractive, *if* $\forall r_1 = (L_1 \supseteq K_1 \subseteq R_1), r_2 = (L_2 \supseteq K_2 \subseteq R_2) \in S$:

$$\exists (n_1, n_2, l) \in E^{del}(r_1), nlab_{L_1}(n_1), nlab_{L_1}(n_2) \notin \Sigma_N^{add}(r_2), l \notin \Sigma_E^{add}(r_2)\}$$

i.e. all rules are edge-subtractive *and every rule has an edge-subtraction-termination label whose incident node labels are not added by any other rule.*

$\mathcal{G} = (S, Z) \in \mathsf{GRS}$ *is* subtractive*, if all rules are edge-subtractive or node-subtractive* $\forall r_1 = (L_1 \supseteq K_1 \subseteq R_1), r_2 = (L_2 \supseteq K_2 \subseteq R_2) \in S$:

$$(\exists (n_1, n_2, l) \in E^{del}(r_1), nlab_{L_1}(n_1), nlab_{L_2}(n_2) \notin \Sigma_N^{add}(r_2), l \notin \Sigma_E^{add}(r_2))$$
$$\vee (\exists l \in \Theta_N^-(r_1) : l \notin \Sigma_N^{add}(r_2))$$

i.e. every rule has a subtraction-termination label which is not added by any other rule.

Then define the termination-subgraph *of* $\mathcal{G}$ *to be the subgraph of* Z

$$T := (N_T := \bigcup_{r \in S} N_T(r), E_T := \bigcup_{r \in S} E_T(r), nlab_Z|_{N_T})$$

The set of all edge-subtractive *graph rewrite systems is called* ESGRS*. The set of all* subtractive *graph rewrite systems is called* SGRS*. We have* ESGRS $\subset$ SGRS*. The union of edge-additive and subtractive graph rewrite systems is called the class of* exhaustive graph rewrite systems XGRS := AGRS $\cup$ SGRS.

In a subtractive graph rewrite system each rule may manipulate arbitrary nodes and edges, except that it also deletes *items* from the termination-subgraph. At latest when this subgraph is empty the system terminates. In an edge-subtractive system every rule deletes at least one *edge* with a label from $\Theta_E^-(\mathcal{G})$ from the termination-subgraph. The set $\Theta_N^-(\mathcal{G})$ may be empty. Hence these systems form the counterpart of edge-additive systems.

Theorem 2 (Termination of SGRS) *Let be* $\mathcal{G} = (S, Z) \in \mathsf{SGRS}$*.* $\mathcal{G}$ *terminates on finite* Z*.*

If $T = (N_T, E_T, nlab_T) \subset Z$ *is the termination-subgraph of* $\mathcal{G}$ *and* Θ^- *is the set of subtraction-termination labels, a derivation in* $\mathcal{G}$ *has at most* $|\Theta^-| \cdot |N_T|^3$ *steps.*

Proof: The number of nodes in $T \subset Z$ is $|N_T|$, and the number of edges with $|\Theta_E^-(\mathcal{G})|$ edge-subtraction-termination labels is at most $|\Theta_E^-(\mathcal{G})| \cdot |N_T|^2$. Thus the length of any derivation in $\mathcal{G}$ may be at most $|\Theta^-| \cdot |N_T|^3$. ∎

As example consider Figure 8. If rule TGRS-1 is reversed, Figure 9 displays an example derivation in reverse direction: each rewrite step is edge-subtractive.

2.3.1 Usefulness

Exhaustive graph rewrite systems rely on the fact that all implicit redexes in the host graph are reduced step by step. The rewrite process is always tied to a termination-subgraph which is either completed or consumed. Edge-additive systems create new redexes, but there are at most as many redexes as the number of edges in the termination-subgraph. Subtractive rewrite systems are even more restricted: they do not create redexes but only destroy them. Therefore exhaustive graph rewrite systems can only be used to specify algorithms which perform a finite number of actions depending on the size of the host graph. In fact this is the case for many code optimizations, especially for code motion and replacement algorithms: after calculating the places of code transformations, they perform the transformation for each of the places once. Thus they reduce a number of redexes which is proportional to the size of the host graph, and the transformation stops after a finite number of actions.

[Plu95] presents a termination criterion which is also based on subtraction (called *consumptive* graph rewrite systems). This criterion is more general than ours because it characterizes the exact conditions for a graph rewrite system to be terminating. However, it is based on features of derivations and not on features of rules alone. This is indispensable for use in an optimizer generator because the generated algorithms must terminate on any input graph. Plump's criterion also uses multi-graphs (multiple edges with the same labels between two nodes). Hence he cannot define a criterion similar to edge-additive termination.

3 Stratification of graph rewrite systems

If a terminating graph rewrite system is not confluent, it delivers a correct, but arbitrarily selected result. Often it would be useful to have a technique to automatically select one of these. To this end we transfer the concept of *stratification* from DATALOG$^\neg$ (DATALOG with negation) to graph rewrite systems [CGT89b]. Stratification orders the rules into a list of rule sets (the *strata*). When the rules are executed along this order they yield one unique result. In a lot of cases this turns out to yield a rather natural semantics of the rewrite system.

Stratification takes dependencies of the rules into account. While in DATALOG$^\neg$ these are only due to negation, in graph rewrite systems they are additionally caused by arbitrary manipulation of items. Here the major idea is to defer the application of rules that delete objects or use negation. First we enlarge the axiom as much as possible, then we apply negation rules, and finally we execute rules that delete objects. Using this heuristic we are able to apply more rules that construct or test subgraphs. We conclude that subgraphs are absent only if they really cannot be derived, preferring longer derivations over shorter ones.

3.1 Rule dependencies in graph rewrite systems

A rule of a graph rewrite system may have several effects on a node domain or a relation: *test (t)*, test absence *(negation,n)*, *add (a)*, and *delete (d)*. A dependency between two rules consists of a pair of these effects, e.g. *add-add/aa* or *add-delete/ad*. These pairs form relations between the rules and span up the *rule dependency graph (RDG)* of a graph rewrite system.

Definition 7 (Rule dependency graph (RDG)) *Let be* $\mathcal{G} = (S, Z) \in \mathsf{GRS}$, $r \in S, l \in \Sigma$. *Let* $EFF = \{t, a, d, k\}$ *be a set of relations between rules and labels defined as follows:*

$$t(r,l) :\Longleftrightarrow l \in \Sigma_N^{test}(r) \cup \{l | e \in E^{test}(r), l = elab(e), negated(e) = false\}$$

$$n(r,l) :\Longleftrightarrow e \in E^{test}(r), l = elab(e), negated(e) = true$$

$$a(r,l) :\Longleftrightarrow l \in \Sigma^{add}(r)$$

$$d(r,l) :\Longleftrightarrow l \in \Sigma^{del}(r)$$

The following defines a set of rule dependency relations for a pair of effects $x, y \in EFF$ *and two rules* $r_1, r_2 \in S$*:*

$$\delta_{xy}(r_1, r_2) :\Longleftrightarrow \exists l \in \Sigma : x(r_1, l), y(r_2, l).$$

The relations δ_{tt}, δ_{tn}, δ_{nt} *und* δ_{nn} *are called* harmless. *The relations* δ_{aa}, δ_{at}, δ_{ta} *are called* generating. *The relations* δ_{td}, δ_{an}, δ_{ad}, δ_{nd}, *as well as their inverses* δ_{dt}, δ_{na}, δ_{da}, *and* δ_{dn} *are called* stratifiable. *The relation* δ_{dd} *is called* non-stratifiable.

With $\Sigma_N := \emptyset$ *and* $\Sigma_E := \{t, n, a, d\}^2$ *the* rule dependency graph (RDG) *of* $\mathcal{G}$ *is the* Σ*-graph*

$$RDG(\mathcal{G}) := (S, \delta_{aa} \cup \delta_{at} \cup \delta_{td} \cup \delta_{an} \cup \delta_{ad} \cup \delta_{nd} \cup \delta_{dd}, \emptyset)$$

The first component of a dependency specifies the effect of the first rule on a node domain or relation, the second component that of the second rule. E.g. if $\delta_{td}(r_1, r_2)$ holds for some rules r_1, r_2, then the r_1 tests a node or edge label which r_2 deletes. A stratification would apply r_1 in an earlier stratum than r_2 because r_2 prevents r_1 from being applied. Stratification relies on the fact that only one rule involved in an overlap deletes the redex of the other rule. Then rule applications may be ordered such that concerning one rule either no or as many redexes as possible are destroyed.

Table 1 shows how the heuristic handles the different combinations of rule effects, and which of them are integrated in the RDG. Harmless dependencies (δ_{tt}, δ_{tn}, δ_{nt} und δ_{nn}) between rules can be neglected for the stratification because they do not prevent that corresponding direct derivations can

dependency	(r_1, r_2)	class	remark
test-test	δ_{tt}	harmless	not in RDG
test-negation	δ_{tn}	harmless	not in RDG
test-add	δ_{ta}	generating	δ_{at} in RDG
test-delete	δ_{td}	stratifiable	r_1 first
negation-test	δ_{nt}	harmless	not in RDG
negation-negation	δ_{nn}	harmless	not in RDG
negation-add	δ_{na}	stratifiable	r_2 first, δ_{an} in RDG
negation-delete	δ_{nd}	stratifiable	r_1 first
add-test	δ_{at}	generating	
add-negation	δ_{an}	stratifiable	r_1 first
add-add	δ_{aa}	generating	
add-delete	δ_{ad}	stratifiable	r_1 first
delete-test	δ_{dt}	stratifiable	r_2 first, δ_{td} in RDG
delete-negation	δ_{dn}	stratifiable	r_2 first, δ_{nd} in RDG
delete-add	δ_{da}	stratifiable	r_2 first, δ_{ad} in RDG
delete-delete	δ_{dd}	non-stratifiable	

Table 1: Dependencies between rules

be interchanged. Also generating dependencies (δ_{aa}, δ_{at}) do not lead to overlaps. However, they have to be considered in the stratification because they generate redexes for other rules. δ_{ta} is considered in its inverse form δ_{at}.

Stratifiable dependencies indicate that their rules may form overlaps, in which the application of one rule destroys the redex of the other, but not vice versa. The heuristic is to first execute those rules which add subgraphs, followed by those which test on non-existence of subgraphs, and finally those which delete subgraphs. Thus we consider for the RDG only the relations δ_{td}, δ_{an}, δ_{ad}, and δ_{nd} (and thus also their inverses δ_{dt}, δ_{na}, δ_{da} und δ_{dn}). This strategy retains redexes for rules which test the existence of subgraphs. Rules that test non-existence of subgraphs are applied less. As many subgraphs as possible are derived from the axiom, and the non-existence of a subgraph is concluded only if it is really not derivable from the axiom. Thus the stratification process yields a rather natural semantics of the rewrite system.

The dependency δ_{dd} is not stratifiable because its rules may produce overlaps in which the application of each rule destroys the redex of the other (mutual exclusion). This means that no reasonable order for a stratification can be found.

3.2 Stratification of graph rewrite systems

This section gives a fundamental theorem about normal forms in stratified graph rewrite systems. Although a stratified graph rewrite system may not be convergent, its rules can be ordered such that a single normal form results.

Definition 8 (Stratification of a GRS) *Let be $\mathcal{G} = (S, Z) \in$ GRS. A stratification of $\mathcal{G}$ is a partition of S into a list of sets of rules (*strata*) $[S_1, \ldots, S_n]$, so that the following conditions hold for the relations of RDG($\mathcal{G}$):*
Let be $r_1, r_2 \in S$. If $r_1 \in S_i \wedge r_2 \in S_j$, then

1. $\neg\delta_{dd}(r_1, r_2)$

2. $\delta_{at}(r_1, r_2) \vee \delta_{aa}(r_1, r_2) \Longrightarrow i \leq j$

3. $\delta_{td}(r_1, r_2) \vee \delta_{an}(r_1, r_2) \vee \delta_{ad}(r_1, r_2) \vee \delta_{nd}(r_1, r_2) \Longrightarrow i < j$

If there is a stratification for S, $\mathcal{G}$ is called stratifiable. *The set of all stratifiable graph rewrite systems is called* STRGRS.

Condition 1 excludes that a stratifiable graph rewrite system contains non-stratifiable rules. Condition 2 means that if there is a generating dependency between a rule r_1 and a rule r_2, then r_2 must be evaluated in the same or in a later stratum than r_1. Condition 3 means that if there is a stratifiable dependency between r_1 and r_2, then r_2 must be evaluated in a later stratum than r_1. Equivalent to this condition is, that edges in the loops of the RDG must not be stratifiable dependencies. If so, rules that delete parts of the graph or test the non-existence of parts, depend recursively on each other. Then our stratification heuristic cannot find an execution order.

Theorem 3 (Stratified convergence) *Let be $\mathcal{G} = (S, Z) \in$ STRGRS, and $S' = [S_1, .., S_n]$ a stratification of S. Let every $S_i \in S'$ terminate. Suppose every rule of S_i does not form overlaps with itself on Z. Then S_i yields a unique normal form.*

Suppose every stratum $S_i \in S'$ has a unique normal form. Then $\mathcal{G}$ is called stratified-convergent. *If the strata are computed in stratification order, this process selects a single normal form for $\mathcal{G}$, the* stratified normal form.

Proof: All strata of $\mathcal{G}$ are assumed to terminate. Within a stratum only harmless and generating rule dependencies are allowed. Rules with such dependencies never form overlaps, i.e. their direct derivations are always independent. Additionally, if each rule does not form overlaps with itself (this is at least the case if its redexes do not overlap), all pairs of direct derivations from rules in the stratum may be interchanged directly. Thus the stratum is strong confluent and with [New42] yields a unique normal form.

If all strata of $\mathcal{G}$ yield a unique normal form, and if the strata are computed in stratification order, also the whole derivation process delivers a single normal form. ∎

The above heuristic is a generalization of the stratification of DATALOG$^\neg$ to graph rewrite systems. The rule dependency graph of a DATALOG$^\neg$-program results as a special case of our RDG and only contains relations with *add-* and *negation*-effect on edges [CGT89a].

Whereas the stratification process deals well with overlaps between different rules, the absence of overlaps for single rules can only be decided if a host graph is given. If a rule deletes items and can be applied competitively on two overlapping subgraphs, these form an overlap and the corresponding stratum does not yield a unique normal form. Thus a stratification on the rule set may be possible, but whether a stratified normal form exists can only be checked by investigating the direct derivations on the host graph. However, we expect that this does not disturb specification of program optimizations. An experienced user is able to assert whether there are overlaps concerning one single rule because he knows about the shape of the intermediate representation. Even if this is not possible, stratification reduces the number of solutions enormously.

4 How to use graph rewrite systems for optimization

This section demonstrates how graph rewriting can be used to specify important parts of the *lazy code motion* optimization. We consider local analysis, global analysis and transformation. We employ all the developed theory: termination is essential and stratification already occurs with simpler examples of data-flow analysis.

Lazy code motion moves intermediate expressions within a procedure. Its aim is to remove *partial redundancies*, i.e. repeated computations of the same values. This is achieved by moving computations

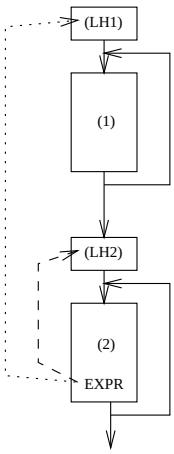

Figure 10: Moving an invariant expression out of a loop. Busy code motion with dotted edge, lazy code motion with dashed edge

higher up in the code and re-using their results later in the program. The original algorithm (also called *busy code motion* [MR79]) pays especially in array-intensive computations because address expressions often can be moved out of loops.

However, busy code motion introduces longer life-times for expression values, which may deteroriate register allocation. Therefore *lazy code motion* was developed [KRS94]. It tries to reduce register occupation (register lifetimes) as long as the number of computations is minimal. This is achieved by computing two kinds of information. First the same information as in busy code motion is computed: for each expression the earliest place in the code is determined for which the most re-uses are possible and the most computations are saved (*earliest information*). Then lazy code motion tries to re-move the expression computation downwards in the code, but only as far as no additional computations have to be introduced (*latest information*). This means that for each expression the latest places in the code are determined, which still are *computationally optimal*, but which yield shortest register life-time of the computed value. Thus lazy code motion moves expressions up in the code as much as possible (to save computations), and then moves them back as far as possible without duplicating them (to reduce register life-times).

Figure 10 contains two loops (1) and (2) with two loop entry headers (LH1) and (LH2). Suppose `EXPR` is a redundant expression in loop 2 and not invalidated in loop 1. While busy code motion would move `EXPR` out of loop body (2) up into LH1 lazy code motion would move it into LH2. This results clearly in a shorter register lifetime and a better register allocation.

Our example specification regards only one procedure. Interprocedural analyses are often more complicated than intraprocedural ones. Instead of applying chaotic iteration to evaluate the rules special evaluation strategies must be used. Thus modeling interprocedural analyses with graph rewriting needs special specification mechanisms. On the other hand, all Ears-specifications model distributive data-flow frameworks over finite power-sets. For these frameworks there exist several simple interprocedural propagation methods, which could be applied [SP81] [KS92] [RHS95].

4.1 Designing the data model

Before we develop graph rewrite systems for program optimization, we have to describe how the graphs of the intermediate representation look like. A data model of an optimizer has to specify all entities of the intermediate language as well as all analysis information which is used for the transformations.

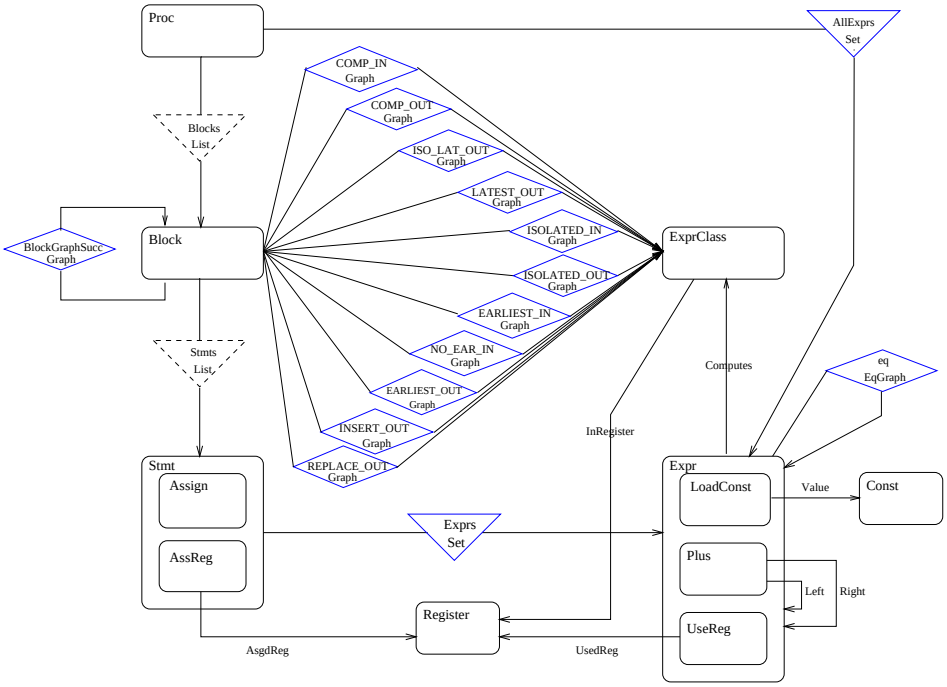

Figure 11: Data model as higraph.

Our running example relies on the data model in Figure 11. The diagram is a higraph, i.e. an extended entity-relationship-model [Har88]. Boxes denote object types, while box inclusion denotes inheritance. The direction of the binary relations is indicated by arrows to their target domains. n:m-relations are depicted by diamond boxes, 1:n-relations by triangles, and 1:1-relations by simple arrows. We use object types, which appear frequently in intermediate representations, such as blocks, statements and expressions. Several of them are generated by the front-end (`Proc`, `Block`, `Assign`, `LoadConst`, `Plus`, `Const`) 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 front-end (`Blocks`, `Stmts`, `Exprs`, `Left`, `Right`). All others are the results of optimizer components, i.e. computed by graph rewrite systems. Some relations are only 1:1 relations, such as `Left` and `Right`, others are 1:n-relations, such as `Blocks`.

Some relations which the front-end generates have to be ordered (expressed by dashed boxes), because the intermediate code obeys control flow constraints. In our definition of graphs we have used only unordered relations. However, an ordered relation can be regarded as an unordered relation overlaid by a second one providing the order. Thus we have to match both of them in a left-hand side. The alternative would be to introduce graphs with ordered relations. Whereas this is no problem for matching, the construction of ordered relations would need a special mechanism. Thus we assume that ordered relations implicitly create a linearizing second relation, which must be matched additionally. This is the style of PROGRES [Zür95].

For an implementation this data model has to be transformed into a concrete data description language, from which a uniform object handling package can be derived. More specifically, each abstract relation must be annotated by a concrete representation class. To be appropriate, a data specification language should provide an abstraction for relations. In the compiler framework CoSy this is solved by the data description language fSDL [WKD94]. fSDL provides two kinds of relation representations (template classes): bidirectional relations (such as `Graph` or `EqGraph`), and directed

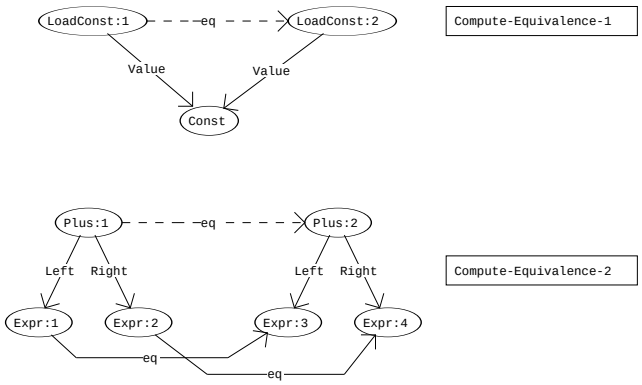

Figure 12: COMPUTE-EQUIVALENCE: Syntactic equivalence of expressions

relations (such as `Set` or `List`). Starting from the fSDL specification a uniform data access interface is generated, which is used by all hand-written and generated phases of a compiler.

4.2 Expression equivalence (value numbering)

The first step of lazy code motion is to assemble all intermediate expressions of a statement list of a procedure. COLLECT-EXPRESSIONS from section 1.1 can be used for this purpose. Because in our data model the relation `AllExprs` is denoted to be a set, the result of executing COLLECT-EXPRESSIONS constructs a set of the procedure's expressions. The result is unique because COLLECT-EXPRESSIONS is an EARS without negation.

The second step of lazy code motion maps all syntactically equal expressions to each other. Figure 12 contains the core of a specification. In intermediate languages we distinguish two kinds of expressions: those without operand expressions (leaf expressions), and those with operands (non-leaf expressions). Among the former there are operators such as `LoadConst`, among the latter there are such as `Plus`. We express the equivalence of two expressions by the relation `eq`. Two expressions are equivalent if they use the same objects and their operands are equivalent. For `LoadConst` expressions the used constant object must be equal (COMPUTE-EQUIVALENCE-1). For `Plus` expressions their operands must be equivalent (COMPUTE-EQUIVALENCE-2). For a complete specification of syntactical tree equivalence of expressions we have to add similar rules for all subtypes of expressions.

COMPUTE-EQUIVALENCE is an edge addition rewrite system and does not use negation. Thus COMPUTE-EQUIVALENCE is convergent. A stratification groups all rules into one stratum.

The information which expressions are equivalent can be used to reduce the number of objects which have to be considered in global data-flow analysis. Because expression classes represent expressions the data-flow analysis can be performed on the classes alone, taking some more modification information into account (define/use/kill information). Thus the next step in the preparation of the data-flow analysis summarizes all strongly connected components of the `eq`-relation into a set of objects, the expression classes (`ExprClass`). This can be specified with an exhaustive graph rewrite system that allocates the objects, one for each partition, and then transitively links all reachable expressions. Then data-flow analysis can be performed.

4.3 Global data-flow analysis

Because EARS may be used to model global distributive data-flow analysis they can be used to specify lazy code motion analysis [KRS94] [Aß96]. This analysis relates the basic blocks of the control flow graph to the expression classes of the program, representing the data-flow information in form of

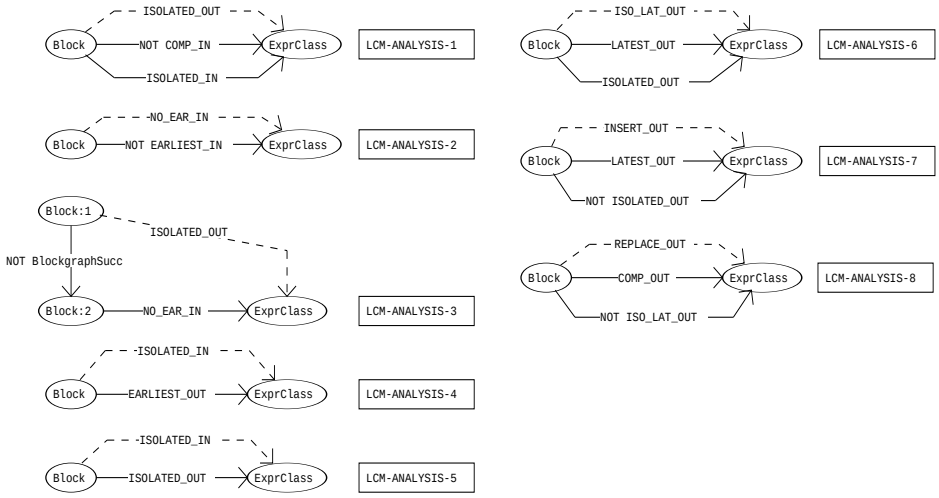

Figure 13: LCM-ANALYSIS: Data-flow analysis for lazy code motion: isolated, insert-out, replace-out.

the edges of several graphs. Here we present only the last two parts of the analysis: the recursive equation system for isolation analysis and the non-recursive equation system which determines the places of transformations at the end of blocks (Figure 13). The remainder of the analysis as well as the initialization can be specified quite similarly.

Our example computes the predicates `INSERT/REPLACE_OUT` which denote those block exits where expressions are to be inserted or replaced. It uses the following predicates which are the results of the first equation systems in [KRS94]. `EARLIEST_IN/OUT` denotes block entries/exits which are the earliest points in the program where an expression is computationally optimal. `LATEST_OUT` denotes those block exits where expressions have to be inserted latest to achieve shortest register lifetime and computational optimality. Code motion can be performed using these predicates (*almost lazy code motion* [KRS94]). However, often this is not sufficient. Also expressions are inserted which have only one use. Such expressions should not be transformed because replacing the single use does not save computations. These expressions are called *isolated* and are marked by the isolation analysis. We represent this by the relations `ISOLATED_IN/OUT`.

Isolation analysis is done by rules LCM-ANALYSIS-1–6. An expression is isolated at the exit of a block if it is either not computed and isolated at the beginning (LCM-ANALYSIS-1) or earliest at the entry of all successor blocks. To specify this we would need all quantified variables. Although this can be integrated the approach of this paper does not provide this. We can also rewrite the rules, expressing that there should be no successor block where an expression is not earliest (LCM-ANALYSIS-2–3). Then, an expression is isolated at the entry of a block if it is earliest (LCM-ANALYSIS-4) or isolated at the exit of the block (LCM-ANALYSIS-5). Thus isolation at the entry and exit of a block is expressed by recursive rules, defined over the successors of the block. This makes a fixpoint evaluation necessary.

Finally, rules LCM-ANALYSIS-6–8 compute the necessary predicates for a transformation. An expression must be inserted at the end of a block, if it is latest, but not isolated there (LCM-ANALYSIS-7). An expression is to be replaced at the end of a block if it is computed there (LCM-ANALYSIS-8) and not isolated and latest (LCM-ANALYSIS-6).

The given graph rewrite rules relate almost one-to-one to the equations in [KRS94]. Thus specifying global data-flow analysis with graph rewrite systems can be done in a very short time.

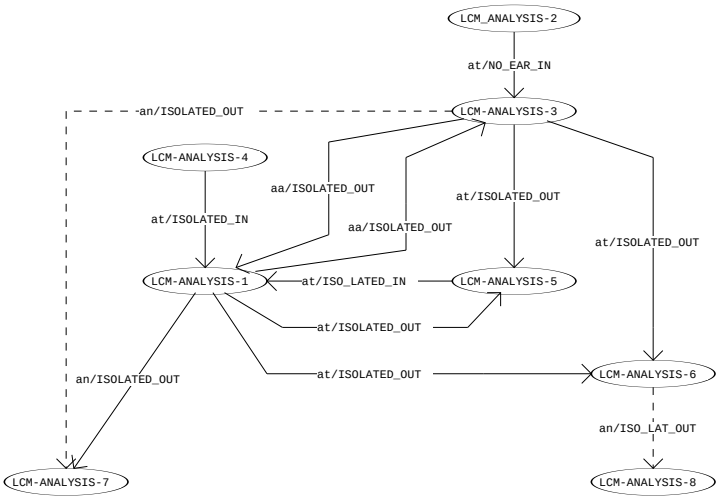

Figure 14: The rule dependency graph of LCM-ANALYSIS. Edges are marked by effect-pairs and by the labels which invoke them. Dashed edges denote stratifiable dependendies, i.e. transitions between the two strata in a possible stratification

4.3.1 The role of stratification in data-flow analysis

For LCM-ANALYSIS — and the complete set of equations of lazy code motion — we have to use negation. This easily destroys the confluence of an EARS [Aß95c], and we have to apply stratification. The rule dependency graph of our example is given in Figure 14. A stratification consists of two strata of which the first is cyclic. One possible solution is [{LCM-ANALYSIS-1, ..., LCM-ANALYSIS-6}, {LCM-ANALYSIS-7, LCM-ANALYSIS-8}].

For the complete data-flow specification several stratifications exist. One of the possible solutions consists of 4 cyclic and 3 non-cyclic strata, which corresponds exactly to the rule groups (equation systems) from [KRS94]. Because no rule deletes anything, there are no overlaps concerning single rules. According to theorem 3, executing the rules in the order of the stratification yields one normal form. This is the data-flow information for lazy-code motion. If all rules of lazy code motion analysis were included in one system, it would not be deterministic. In terms of data-flow analysis this means that the combined equation system would not be distributive, but only monotone. It is interesting that current data-flow analysis theory does not discuss this topic, instead the separation into equations systems is always given ad hoc. Thus stratification also provides a systematic method for structuring complex data-flow analyses.

4.4 Transformation

We conclude our examples by specifying some of the necessary transformation rules for lazy code motion (Figure 15). We regard only the necessary relations for transformation at the end of blocks (`INSERT_OUT`, `REPLACE_OUT`). These relations indicate the blocks where expressions should be inserted or replaced. The first rule inserts an expression at the end of a block. It creates new statements which compute the value of the expression and store it into a register. Edge `INSERT_OUT` is deleted in order to prevent that LCM-TRAFO-1 is applied again. LCM-TRAFO-2 replaces a partially redundant expression at the end of a block whose expression class is marked by an edge `REPLACE_OUT`. The rule replaces the redundant expression to a `UseReg` expression which re-uses the already computed value of the class from the corresponding register. To simplify presentation we assume that only one expression is to be

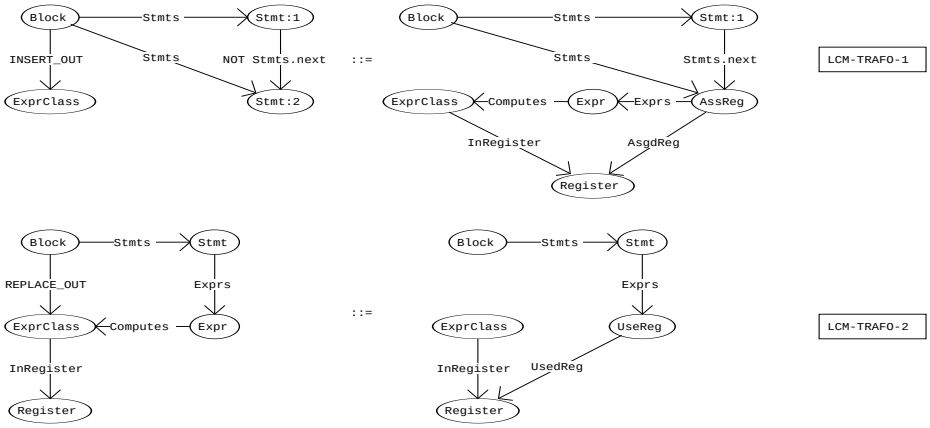

Figure 15: LCM-TRAFO: Insert and replace of an expression at the exit of a basic block

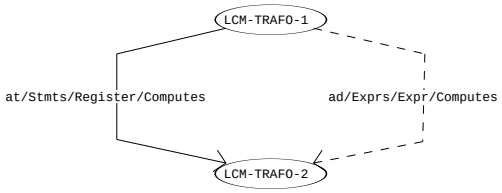

Figure 16: The rule dependency graph of LCM-TRAFO. Edges are marked by effect-pairs and by some of the labels which invoke them. Dashed edge denotes stratifiable dependencies

replaced per block. For a complete lazy code motion transformation similar rules for the beginning of the blocks must be written.

LCM-TRAFO is edge-subtractive. Although its rules add items they do not produce any redex: LCM-TRAFO-1 subtracts the edge `INSERT_OUT` and LCM-TRAFO-2 the edge `REPLACE_OUT`. Thus LCM-TRAFO terminates.

LCM-TRAFO need not be strong confluent, because LCM-TRAFO-1 inserts and LCM-TRAFO-2 deletes expressions. However, due to the rule dependency graph in Figure 16 we can stratify it into [{LCM-TRAFO-1}, {LCM-TRAFO-2}]. Execution along these strata first inserts as many expressions as possible, and then replaces expressions. Each of the rules should not form overlaps with itself, because each expression class is handled separately in the data-flow analysis and all transformations can be carried out in parallel for each expression. Thus according to theorem 3 LCM-TRAFO is stratified-convergent.

The following table summarizes the features of our graph rewrite system examples.

example	termination	normal forms	#strata
COLLECT-EXPRESSIONS	edge-additive	convergent	1
COMPUTE-EQUIVALENCE	edge-additive	convergent	1
LCM-ANALYSIS	edge-additive	stratified-convergent	2
LCM-TRAFO	edge-subtractive	stratified-convergent	2

4.5 Other specifiable optimizations

Other examples for optimizations which can be specified by graph rewrite systems are found easily because the structure of lazy code motion is prototypical for a number of classical optimizations. We expect that at least the following areas of optimization can be handled:

- other code motion algorithms
- dead code elimination
- strength reduction
- induction variable elimination
- simple alias analysis expressing alias equivalence relations
- control flow analysis
- copy propagation
- folding constant propagation

Optimizations, which rely on syntactical transformations of the code and do not need global analysis, can be specified directly with graph rewrite systems:

- branch optimization
- constant folding
- constant propagation with partial evaluation [Bin94]
- idiom recognition [PP94].
- loop transformation algorithms such as loop unrolling

There exist specifications of optimization algorithms in the literature which use graph rewrite systems (e.g. [Bin94]). It is probable that more subclasses of graph rewrite systems will be found which are able to cope with more optimizations.

5 Meta-code-generation by the order algorithm

In this section we present a generic algorithm to execute an exhaustive graph rewrite system, the *order evaluation*. In [Aß95b] a variant of this algorithm was presented for Ears. Here it is extended to handle (stratified) exhaustive graph rewrite systems. The idea of the algorithm is similar to a nested-loop join in relational algebra: edge relations are joined to find all redexes in a graph. However, in contrast to databases, order evaluation assumes a node-based, directed representation of the graph. This is required for an optimizer because front-end and back-end rely on it and the optimizer has to share data with them. In addition, the cost of the algorithm does not refer to the amount of edges in a relation but only to the maximal out-degree of a node under a certain relation. Thus it works best on sparse and width-limited graphs which are often encountered in program analysis. Finally, section 5.4 gives a short overview on other evaluation methods.

5.1 Overview

Order evaluation is based on the notion that we will find all redexes of a rule if we consider all host graph nodes, which can be homomorphic to root nodes of left-hand sides. Then the remainder of the redexes is found by traversing neighbor sets, starting from the redex root nodes. In order to accomplish this, the *order* of a set of rules has to be calculated, which is the maximal number of root nodes in left-hand sides. If a left-hand side does not have a root, we choose an arbitrary node as artificial root, normalizing the order to at least 1.

Definition 9 (Order of a rule set) *Let S be a set of graph rewrite rules, $r = (L \supseteq K \subseteq R) \in S$. Define the* signature *$sig(L)$ to the multiset of the root nodes labels in a left-hand side L. Define the order $k(S)$ to be the maximum of the cardinalities of sig over all left-hand sides, normalized to 1:*

$$k(S) := \max\,(\max_{(L \supseteq K \subseteq R) \in S} |sig(L)|, 1)$$

Let be $\mathcal{G} = (S, Z) \in$ STRGRS, and $[S_1, .., S_n]$ a stratification of S. Then the order of $\mathcal{G}$ is

$$k(\mathcal{G}) := \max_{1 \leq i \leq n} k(S_i)$$

We also call $\mathcal{G}$ a STRGRS(k).

Based on this definition [Aß95b] developed a generic evaluation algorithm for EARS. EARS-*order-evaluation* works on directed graph representations and has complexity $O(n^{k+2}e^{lp})$, where l the length of the longest path of an edge disjoint path cover over all left-hand sides, p the maximal number of paths in such a cover, n the maximal number of nodes in a node domain, and e the maximal out-degree of a node concerning an arbitrary relation. For concrete rule sets k, l, and p are constant, and the generated algorithms are polynomial. In program optimization many relations are one-valued, have a fixed cardinality, are sparse, or are partitioned over some object domains. Hence linear or quadratic algorithms often result.

In this paper we extend this algorithm to XGRS-ORDER in Figure 17. In contrast to EARS we not only have to add edges, but also add nodes and delete items. XGRS-ORDER is always generated for a set of rules, either for one stratum of a stratification or for all rules of a graph rewrite system. For a stratification the generated algorithms have to be executed in the stratification order. XGRS-ORDER is a generic algorithm because it depends on several parameters of the rules: some loops marked by **forall-const** and can be unrolled for a concrete algorithm.

The algorithm consists of the following parts. First all nodes of node domains which can be deleted are marked as valid. Then all permutations of possible redex root nodes are enumerated. We overlap all rule tests which have a similar set of root nodes. To this end the rule set of a stratum S has to be partitioned into a set of equivalence classes V. A rule $r_1 = (L_1 \supseteq K_1 \subseteq R_1)$ is equivalent to a rule $r_2 = (L_2 \supseteq K_2 \subseteq R_2)$, if $sig(L_1) \subseteq sig(L_2)$ or $sig(L_1) \supset sig(L_2)$. This expresses whether the tests for the rules can be done together in loop (3). In the algorithm, k_v is the maximal number of elements in a signature in an equivalence class $v \in V$. For every class v k_v is always smaller or equal to order k. Normally we have to embed the rule tests into a fixpoint loop (loop (1)). After the normal form is reached, the host graph relations must be cleaned, i.e. edges to deleted nodes have to be deleted.

Then, for one single permutation of root nodes and for one rule, RULETEST finds all reachable redexes, traversing neighbor sets. To this end an *edge disjoint path cover* of the left-hand side is statically computed. This is a set of paths which cover the left-hand side such that the paths intersect each other only at their end points. RULETEST computes the cross product over all paths to enumerate all reachable nodes. Thus RULETEST performs a nested loop join over the path problems of the paths (loop (5)). For each permutation of nodes the rule application conditions (t1) – (t3) are tested, and if successful, the rule actions (a1) – (a5), except (a2) are executed. Deletion of hidden edge needs

Input: Generic: rule set S with order k, rule classes V, path covers $P(L)$ for all $(L \supseteq K \subseteq R) \in S$.
Non-generic: host graph Z with termination-subgraph T.
Output: Non-generic: added relations

```
/* Marking nodes as non-deleted */
forall n ∈ N_T(Z) do
    n.deleted ← FALSE;
end
/* fixpoint loop: enumeration of redexes */
change ← TRUE;
while change = TRUE do (1)
    change ← FALSE;
    forall-const v ∈ V do (2)
        forall (x_1, ..., x_{k_v}) ∈ (N_1 × ... × N_{k_v}) do (3)
            /* tests of rules with overlapping signature */
            forall-const r ∈ v do (4)
                /* Test rule r with roots x_1, ..., x_{k_v} */
                change ← change or RULETEST(r, x_1, ..., x_{k_v});
end
/* Cleanup relations of graph */
forall (y_i, y_j, a) ∈ E_T(Z) do
    if y_j.deleted then
        y_i.a -= y_j;
end
/* Rule test for all redexes that can be reached from root nodes */
procedure RULETEST(r = (L ⊇ K ⊆ R), x_1, ..., x_{k_v}) return BOOLEAN
begin
    change ← FALSE;
    /* Cross product of all paths P_1 to P_p ∈ P(L) by traversal of */
    /* the neighbor sets, starting from root nodes x_1, ..., x_{r_{k_v}} */
    forall ((y_{1_1}, ..., y_{1_l}), ..., (y_{p_1}, ..., y_{p_l})) ∈ (P_1 × ... × P_p)(x_1, ..., x_{k_v}) do (5)
        /* First test join conditions... */
        ...
        /* (t1) Test homomorphism */
        forall-const (n_1, n_2, a) ∈ E_L, n_1 ∈ N_i, n_2 ∈ N_j do
            if y_j.deleted or y_i.deleted then (t1-a)
                continue (5); /* Node has been deleted already */
            if y_j ∉ y_i.a then (t1-b)
                continue (5); /* Redex edge is missing */
        /* (t2) Test all negated rule edges */
        forall-const e = (n_1, n_2, a) ∈ E_L, negated(e) =TRUE, n_1 ∈ N_i, n_2 ∈ N_j do
            if y_j ∈ y_i.a then
                continue (5); /* No redex here */
        /* (t3) Test whether all added edges are already there */
        AllEdgesThere ← TRUE;
        forall-const (n_1, n_2, a) ∈ E^{add}(r), n_1 ∈ N_i, n_2 ∈ N_j do
            if y_j ∉ y_i.a then
                AllEdgesThere ←FALSE;
        if AllEdgesThere = TRUE then
            continue (5); /* Terminating relation is complete */
        /* ----------- All tests passed: redex found. ----------- */
        /* (a1) Delete redex edges. */
        forall-const (n_1, n_2, a) ∈ E^{del}(r), n_1 ∈ N_i, n_2 ∈ N_j do
            y_i.a -= y_j;
        /* (a3) Delete nodes by invalidation. */
        forall-const n_1 ∈ N^{del}(r), n_1 ∈ N_i do
            y_i.deleted ← TRUE;
        /* (a4) Add nodes. */
        forall-const n_1 ∈ N^{add}(r), nlab_R(n_1) = i do
            N_i += new(i);
        /* (a5) Add all edges of the rule */
        forall-const (n_1, n_2, a) ∈ E^{add}(r), n_1 ∈ N_i, n_2 ∈ N_j do
            y_i.a -= y_j;
        change ← TRUE;
    end
    return change;
end
```

Figure 17: Generic algorithm XGRS-ORDER

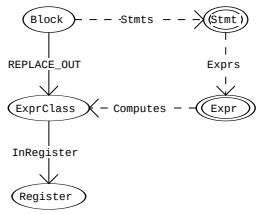

Figure 18: A edge-disjoint path cover for LCM-TRAFO-2 with two paths. Path 1 is drawn with normal edges, path 2 with dotted edges

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

Output: Modified intermediate code: expressions replaced by `UseReg` expressions

```
/* Enumerate path 1 */
forall block ∈ Block do (1)
    forall exprclass ∈ block.REPLACE_OUT do (2)
        register ← exprclass.InRegister; /* 1:1-relation */
        /* Join with path 2 */
        forall stmt ∈ block.Stmts do (3)
            forall expr ∈ stmt.Exprs do (4)
                if expr.deleted = TRUE then /* test on deleted nodes */
                    continue ;
                exprclass2 ← expr.Computes; /* 1:1-relation */
                /* Join condition */
                if not exprclass = exprclass2 then
                    continue ;
                /* Redex found. Do the transformation */
                expr.deleted ← TRUE;
                block.REPLACE_OUT −= exprclass;
                expr.Computes ← NIL;
                stmt.Exprs −= expr;
                usereg ← new(UseReg);
                stmt.Exprs += usereg;
                usereg.UsedReg ← register;
```

Figure 19: Order evaluation for LCM-TRAFO-2

a special handling: if rule nodes have to be deleted, the corresponding redex nodes are invalidated (a3), so that they are not considered anymore (t1-a).

For example consider Figure 18 which contains an edge disjoint path cover of LCM-TRAFO-2. This rule results in the code in Figure 19. The outer loops (1) and (2) enumerate the path problem of path 1, the inner loops (3) and (4) that of path 2. The paths are joined with a nested-loop-join under the join condition $exprclass = exprclass_2$ in order to find the intersection of their enumerated objects. Because LCM-TRAFO-1 is equivalent to LCM-TRAFO-2 concerning its signature, the test for LCM-TRAFO-1 can be overlapped with the loop (1) of algorithm in Figure 19. This illustrates the advantage of the algorithm: traversal of node domains of redex roots are overlaid.

5.2 The cost of XGRS-ORDER

The following theorem deals with the cost of an algorithm generated from XGRS-ORDER.

Theorem 4 (Evaluation of XGRS with XGRS-ORDER) *Let be $\mathcal{G} = (S, G) \in$ XGRS with order k. Let l the length of the longest path of an edge disjoint path cover over all left-hand sides, p the maximal number of paths in a path cover, n the maximal number of nodes in a node domain with an arbitrary label, e the maximal out-degree of a node concerning an arbitrary edge label. The addition and deletion of an edge shall be performed in constant time.*

If $\mathcal{G} \in$ AGRS has one stratum, it can it can be evaluated with a algorithm generated from XGRS-ORDER in $O(n^{k+2}e^{lp})$.

If $\mathcal{G} \in$ SGRS has one stratum, or if $\mathcal{G} \in$ AGRS has one non-cyclic stratum and one rule equivalence class v, it can be evaluated with a algorithm generated from XGRS-ORDER in $O(n^k e^{lp})$.

If $\mathcal{G} \in$ STRGRS it can be evaluated with the maximal cost of one of its strata.

Proof: First assume that $\mathcal{G}$ has one stratum.

(1) Termination. If a fixpoint loop of XGRS-ORDER does not change the host graph, the algorithm stops. (1a) $\mathcal{G} \in$ AGRS. Then each loop adds at least one edge to the termination-relation of Z in action (a5). When this is complete, the relation completeness condition (t3) is always false, and the algorithm terminates.
(1b) $\mathcal{G} \in$ SGRS. Then either action (a1) or (a2) will delete items from the termination-subgraph of $\mathcal{G}$. When it is empty, (t1) will not be true anymore and the algorithm will terminate.

(2) Correctness. XGRS-ORDER enumerates all permutations of root nodes of possible redexes. For one single permutation of root nodes, RULETEST will find all redexes: starting from the root nodes, it traverses all neighbor sets, using the path problems which are induced by the edge disjoint path cover. Thus one fixpoint loop of XGRS-ORDER will find all present redexes in the host graph. New redexes are handled in the next round. If $\mathcal{G}$ is convergent, XGRS-ORDER will compute the unique normal form, otherwise it will select arbitrary solution, depending how the code of the concrete algorithm is generated.

(3) Cost of one fixpoint iteration.
(3a) Redex root node permutations. The enumeration of all redex root node permutations costs $O(n^k)$, as in the case of EARS order evaluation [Aß95b]. This is due to the fact that loop (3) enumerates the cross product of k node domains, and k is chosen maximally for all rule equivalence classes $v \in V \subset S$. Loop (2) and (4) handle all rules of $\mathcal{G}$, but only once, and can be unrolled in a concrete algorithm.
(3b) Rule test. The cost for enumerating all redex nodes to a given set of root nodes, i.e. a nested loop join over the path problems of the path cover, is $O(e^{lp})$ for p paths with maximal length l. The actions which have to be performed for rule tests of a terminating graph rewrite system all have constant effort with regard to a given set of redex nodes. The actions for the test of a single redex, namely rule test actions (t1) – (t3), the addition of new items and the deletion of redex items ((a1), (a3) – (a5)) only depend on the current set of redex nodes. Thus all relevant loops can be unrolled by an optimizer generator and yield constant cost in a generated algorithm.

The main difficulty is the deletion of hidden edges linking the redex and the rest graph (a2). This is not a problem if the host graph is implemented bidirectional. Otherwise, the sources of incoming hidden edges must be looked up in the graph, which would mean additional cost $O(n)$ for each incoming hidden edge. XGRS-ORDER reduces this to a constant factor per redex test: it marks a node invalid which must be deleted (a3) and tests during the rule test whether an invalid node must be neglected (t1-b). This requires additional constant effort in redex testing, but avoids the search for sources of incoming hidden edges. The cost of RULETEST is $O(e^{lp})$.

(3c) Cost of cleaning up. At the end of the rewrite process the graph must be cleaned up, i.e. all invalid edges are finally deleted. To this end all node domains and relations of deletable edges have to be traversed once, i.e. a factor $O(ne)$ is added which does not increase the total cost. Deleted nodes may either be garbage collected or deallocated with an additional linear loop over all corresponding node domains.

(4) Cost of fixpoint. If $\mathcal{G} \in$ AGRS, each round of a fixpoint rule may add one edge to the termination-subgraph. Thus the additional fixpoint cost $O(n^2)$ must be added.

For a subtractive graph rewrite system consisting of one stratum we do not need to perform a fixpoint application of the rules because only existing redexes are reduced, and no new redexes are produced.

For a edge-additive graph rewrite system which is non-recursive, consists of one stratum, and all rules fall into one equivalence class v, we are also able to reach the fixpoint in one iteration. To this end we have to order the rule tests in the loops (2) and (4) along the dependencies of the RDG. This is always possible if the RDG is tree-like (one non-cyclic stratum) and there is only one rule equivalence class (then all rules can be overlapped). Then we will find all redexes in one iteration.

If the graph rewrite system is stratifiable, its cost is the sum of the cost of all strata.

■

We are able to use the order algorithm for graph rewrite systems because we use an invariant embedding: we modify items only in the context of the tested and added nodes, and take special care of the deletion of hidden edges. Allowing general embedding [Nag79] complicates the rule test and leads to context-sensitive search during rule application. This enlarges the cost for the evaluation of a graph rewrite system significantly.

5.3 Order evaluation for our examples

COLLECT-EXPRESSIONS has order 1. It is a non-recursive EARS. Because it contains one stratum, adds only one edge, and has only one path in its left-hand side, the XGRS-ORDER algorithm collapses to a simple nested loop over blocks, statements and procedures (Figure 1). According to the evaluation theorem $O(|\texttt{Proc}|e^3) = O(|\texttt{Proc}||\texttt{Block}||\texttt{Stmt}||\texttt{Expr}|)$ would result. However, because expressions are partitioned by procedures, blocks, and statements, we have $O(|\texttt{Expr}|)$.

COMPUTE-EQUIVALENCE has order 2, because both rules left-hand sides contain two roots. It is recursive. A theorem in [Aß95b] shows that because we deal with expression trees the fixpoint computation can be avoided. Because `Left` and `Right` are 1:1-relations, $l = 1$ and $p = 4$ hold. Thus for COMPUTE-EQUIVALENCE the cost $O(|\texttt{Expr}|^2e^4)$ results. However, e is here much smaller than $|\texttt{Expr}|$ because the relation `eq` partitions `Expr`. If we neglect their cardinality, XGRS-ORDER results in a quadratic algorithm in the number of expressions.

LCM-ANALYSIS has order 1, because `Block` is the only root node domain of all rules. Because it is recursive, we have to apply fixpoint computation. Because $l = 2, p = 2$, we have complexity $O(|\{\texttt{Block},\texttt{ExprClass}\}|^3e^4)$. We can use a bit-vector representation for the relations, which results in the standard round-robin iteration for data-flow analysis. Because bit-vector 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].

LCM-TRAFO is non-recursive, because it does not create new redexes for itself. According to the cost formula it has complexity $O(|\texttt{Block}|^1e^9)$, because $k = 1, l = 3, p = 3$. Then `Computes`, `Stmts.next`, and `InRegister` are 1:1-relations. Also expressions are partitioned over the blocks, i.e. we need to loop only once over all expressions of a procedure. Thus the generated algorithm has cost $O(|\texttt{Expr}||\texttt{ExprClass}|)$.

The order of a graph rewrite system relies on the directions of the relations in the data model. Choosing different directions may result in a different order. If we represent all relations with bidirectional graphs, the order is always the maximal number of non-connected components in a left-hand side, because bidirectional graphs do not have roots.

All exhaustive graph rewrite systems can be solved by order evaluation. For convergent systems it delivers the unique normal form. For non-deterministic systems it delivers a correct, but arbitrarily selected result. Thus we are able not only to specify analysis and transformation uniformly, but also execute the specifications with a uniform execution mechanism.

5.4 Evaluation with other methods

Because a convergent rewrite system yields unique normal forms, an evaluation algorithm is allowed to interchange rule applications arbitrarily. This is the case for EARS, strata in stratified graph rewrite

systems, but also for DATALOG [CGT89b]. In fact, because termination is guaranteed, and all direct derivations can be interchanged, the evaluation methods for the latter can be used for the former. Thus the stratification theory of this paper lays the foundation that all DATALOG methods can be used for EARS and stratifiable graph rewrite systems.

Numerous DATALOG evaluation algorithms have been developed. The fundamental methods are bottom-up (forward-chaining, exhaustive) and top-down evaluation (backward-chaining). The former computes all solutions at once, the later works incrementally and computes only one solution. In graph rewriting the top-down evaluation corresponds to testing whether a single graph edge can be derived. Bottom-up evaluation corresponds to the construction of all derivable subgraphs and edges, i.e. to reduce to a normal form. Because we have used this strategy throughout the paper the order algorithm is a forward-chaining algorithm. However, depending on the context of a specification each evaluation method has its advantages in efficiency.

Order evaluation works *naive*, i.e. reconsiders old redexes in a new round of the fixpoint loop again. This can be easily improved by applying *semi-naive evaluation*, which is known from DATALOG. This method considers only new parts of the database in a new round, by memorizing which parts of the database are new. The memoization can be done in different ways, e.g. by worklists of redexes or marking of nodes.

Very prominent in DATALOG are also tabling methods (*OLDT resolution*, *query-subquery evaluation*), because they reuse partial solutions to cut down the search space [CGT89b]. Also rule transforming schemes such as *magic set transformation* can be applied to the rules. This has the effect, that tuples are pre-selected from the database before joins are performed. In our case it corresponds to narrowing down the node domains of the host graph. Finally, DATALOG databases with special forms (acyclic relations) are also efficiently executable (*counting method*).

The main prerequisites for all these techniques are termination and interchangeability of rule applications. Thus we should be able to use all of them for stratifiable graph rewrite systems. Order evaluation works best if the host graphs are sparse; otherwise a DATALOG-evaluation method may be more apt and more efficient. However, this reveals another strength of our specification method: an optimizer generator can be instructed to use another evaluation algorithm without changing the specification at all.

6 Practical experience

The last section is dedicated to practical experiences with the specification method. In order to show the validity of the approach, the optimizer generator OPTIMIX has been implemented [Aß95a] [Aß95c]. OPTIMIX takes graph rewrite system specifications in textual form and generates an order evaluation procedure in plain C, which can be embedded in compilers. The tool has been used to generate several optimizer components, especially some of a Modula-2 lazy code motion optimizer, using several of the presented example specifications. The generated optimizer parts work on the the intermediate language CCMIR (*common COMPARE medium intermediate representation*) [VvSL93] for C, Fortran-90, Modula-2, and its parallel dialect Modula-P [VH92].

The first important point is the size of the specifications. Table 6 shows the approximate number of rules for several examples. Because a specification of an analysis has to handle a lot of cases it typically needs more rules than that of a transformation. Also, the considered analyses are quite complex: e.g. lazy code motion data-flow analysis contains 7 equation systems.

Also the relation of generated and hand-written part in the optimizers is interesting. Because OPTIMIX still has a lot of restrictions, several parts of the optimizer components are hand-written. We estimate that with an industrial-strength implementation of the generator 90% of an optimizer could be generated. Due to the fact that specifications are very compact the development time of an optimizer should be greatly reduced. Equation systems such as LCM-ANALYSIS can be typed in in a few hours. Also the generated optimizer algorithms can be validated quite quickly, supposed that the

component	approx. number of rules
expression equivalence classes	30
data-flow analysis reaching definitions	20
data-flow analysis live copies	20
data-flow analysis lazy code motion	40
transformation copy propagation	5
transformation lazy code motion	5

Table 2: Number of rules in some example specifications

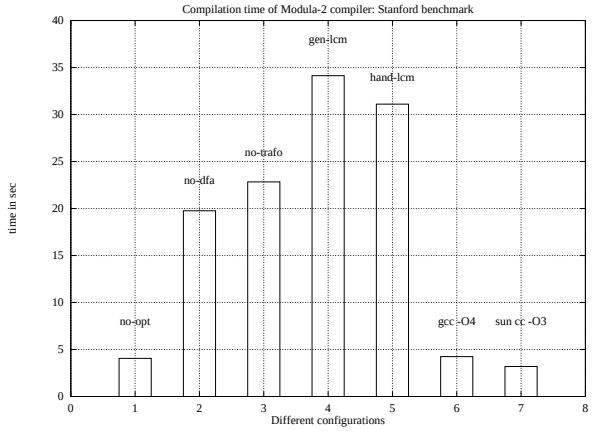

Figure 20: Compilation speed on the Stanford benchmark

generator generates correct code. Thus we estimate that about 50% savings in development time are possible.

6.1 Optimization speed

In order to test the compilation speed of the generated components, we compared the runtime of the Modula-2 compiler with and without optimization. For this test we compiled the Stanford benchmark from Henessy and Nye, resulting in Figure 20. Row (1) depicts the runtime without optimizer, only front-end and back-end. Row (2) adds the time of the analysis except global data-flow analysis and transformation. This adds factor 5 to the runtime of the compiler. Row (3) additionally contains the global data-flow analysis (factor 6). Row (4) contains the time with all generated components. The compiler slows down about factor 9. For the transformation phase there is an alternative implementation by hand, which results in row (5). This phase is not much faster than the generated transformation phase, most time of the optimizer is spent in the generated analysis components.

If we compare this to gcc -O4 (6) and sun-cc -O3 (7), running on an equivalent C program, they are much faster while performing more optimizations. Nevertheless, for an optimizer with generated parts the results are quite encouraging. The lazy code motion optimizer performs a rather computation-intensive optimization: it computes syntactical expression equivalence and 4 data-flow equation fixpoints.

6.2 The speed of expression equivalence

Apparently the optimizer spents a lot of time in program analysis. This is due to the expression equivalence algorithm. Most of the procedures in the benchmark translate quite fast, but there are

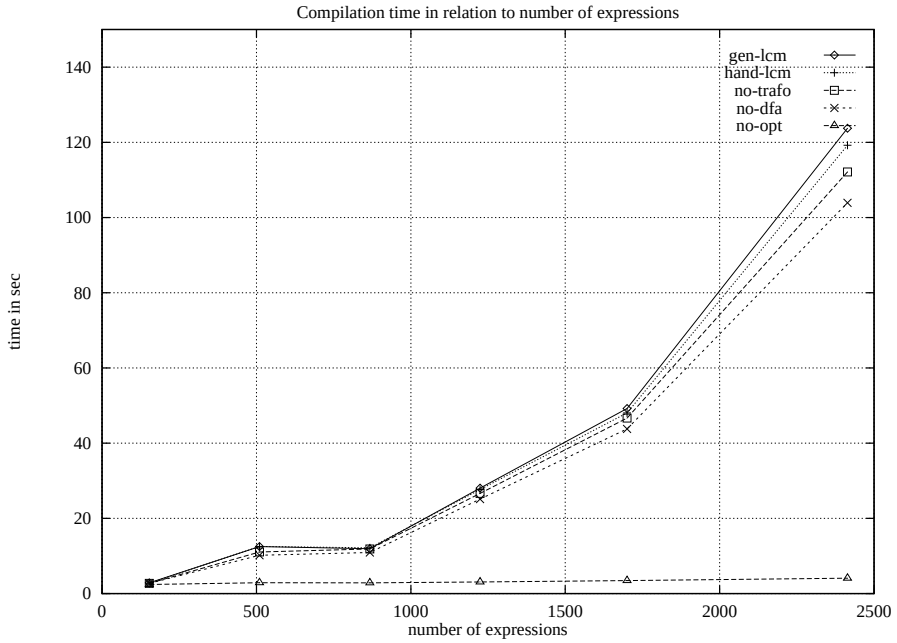

Figure 21: Compile time with different number of expressions per procedure

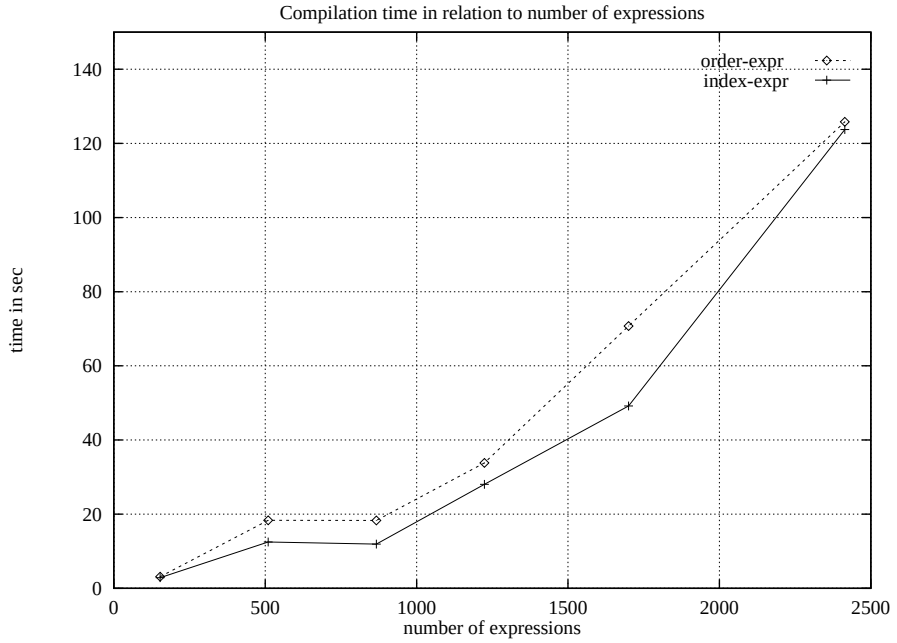

Figure 22: Compile time with different generated expression equivalence algorithms

two which contain up to 1400 expressions and these slow down the expression equivalence algorithm enormously. Compared to that the data-flow analysis, also for large procedures, is reasonably fast. This is assured by diagram 21 where several compilations of procedures with a different number of expressions are shown. Curve `gen-lcm` depicts the compilation time with all generated engines, `hand-lcm` that with hand-written transformation phase, and `no-lcm` everything except transformation. If additionally the global data-flow analysis is skipped, we get curve `no-dfa`. We can see that global data-flow analysis in all cases is quite fast.

Apparently the order algorithm for COMPUTE-EQUIVALENCE is at least quadratic because it has to compare expressions pair-wise (due to the two root nodes in COMPUTE-EQUIVALENCE-1/2). [Aß95b] shows that an index structure can be used to speed it up. It realizes a mapping between the used constants and the using nodes. Hence it simulates virtual edges between the two root nodes of COMPUTE-EQUIVALENCE-1. Then COMPUTE-EQUIVALENCE reduces to order 1. Both algorithms are compared in Figure 22, curve `order-expr` and curve `index-expr`. The algorithm with index optimization uses a hash table index. It is faster for medium-sized procedures. Also the choice of the hash function plays an important role. We experimented with different functions and hash table sizes. Using a inappropriate hash function may slow the whole compiler down about factor 2. Thus it is very important in practice to use index structures and to tune their performance.

Apart from that there are other meta-optimizations for COMPUTE-EQUIVALENCE. Currently OPTIMIX does not exploit the fact that the equivalence for expressions can be computed bottum-up in one pass because the operand relations `Left` and `Right` are tree-shaped [Aß95b]. Because the system is recursive, the generator has to use a fixpoint evaluation algorithm. The fixpoint is reached if no rule can be applied anymore. In the currently generated algorithm this needs a complete round of rule tests on all expressions. Avoiding this we would gain at least another factor 2. Also the semantic knowledge could be used that the constructed relation `eq` is an equivalence relation. Then the standard value numbering algorithm with cost $O(|\texttt{Expr}||\texttt{ExprClass}|)$ can be generated which walks over all expressions linearly and hashes them to their equivalence class [CCL+96]. Normally the number of expression classes is small compared to the number of expressions, and an almost linear behavior can be achieved. However, the generator cannot infer this from the specification. It should be asserted by the optimizer writer. Thus, with an industrial-strength tool the generated optimizer parts could very well reach the velocity of hand-written ones.

6.3 The influence of graph representations

We also measured the influence of the graph representations on the optimization time. To this end a data-flow analysis component was run with a list- and a bit-vector-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 the generator adapts the code generation (of course hand-written parts have to be adapted, too). The union and intersection operation on data-flow sets have different cost: on bit-vectors they are linear and on lists they are quadratic. On the Stanford benchmark the bit-vector implementation beats the the list-based implementation by factor 6. One could also try a factored representation for the graphs [CCF94]. This should speedup data-flow analysis even more.

Thus the last step in writing an optimizer with graph rewriting consists of selecting the right data representation for the host graphs. Only the data model has to be changed; the generator can adjust its code generation automatically. This reveals a major strength of our method: the specification is independent of the concrete representation of the graphs.

7 Related work

Termination of graph rewrite systems was first tackled in [Plu95]. However, the method of *forward closures* relies on the features of a derivation, and cannot be proved regarding the rules alone. Thus it does not seem not to be appropriate in our case. [Plu93b] [Plu93a] treat confluence of graph rewrite systems, but again based on criteria of the derivations. Also, confluence alone is not sufficient for program optimization: many problems are inherently non-confluent. Stratification goes beyond that: it can handle overlaps of different rules automatically and yields normal forms which are quite natural.

Stratification was invented for DATALOG$^\neg$ to handle negation. In the meantime a lot of other methods have been developed [Ull94]. It is an interesting question which of them can be carried over to graph rewrite systems. The idea that DATALOG can been used to describe data-flow analysis has

also been discovered by [Rep94]. However, because our work constrains DATALOG to binary predicates we arrive at a special subclass of graph rewrite systems, EARS. These can be extended very easily to specify general transformations, which is not the case for DATALOG.

The *algorithmic* approach to graph rewrite systems is the most similar to ours. However, there seems to be few work on the efficient automatic execution of terminating or convergent systems. The language PROGRES [Sch90] [Zür95] is its most advanced representative. However, PROGRES is designed for an interactive user environment and not for batch processing. Currently it does not allow for fixpoint computations, it is tied to an underlying database, and does not provide mechanisms for rule overlapped execution, which is indispensable for program optimization. Nevertheless it provides an excellent user interface and program environment.

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. Our method produces much faster algorithms: order evaluation of COMPUTE-EQUIVALENCE performs at least $2400^2/140 = 41000$ redex tests/second (Figure 22)[1] Thus OPTIMIX should also provide one of the fastest existing tools to execute graph rewrite systems.

Several other tools are known which can generate program analyses. Sharlit [TH92] and PAG [AM95] generate efficient global data-flow analyses from lattice-based specifications. With both tools users have to supply C code for the lattice elements and flow functions. Thus exchange of implementations is not so easy. Also the tools do not aid in the preparation of the global analysis. SPARE [Ven89] follows the same approach, but uses a closed specification language. It is additionally tied to the Synthesizer Generator. [Ste91] shows how data-flow analysis algorithms can be generated from modal logic specifications. Although the powerful modal operators allow very short specifications, an application in a real-life compiler is not yet 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 familiar concept of graphs.

Only few approaches are known which integrate analyses and transformations. SPECIFY [Koc92] additionally provides a proof language but was never implemented completely. GENESIS [WS90] provided one of the starting points of this work. It allows 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 the intermediate language is fixed and the code generation scheme is quite ad hoc. Because our method is founded on the theory of graph rewrite systems and DATALOG, it provides a more solid basis for generation optimizers.

Although *term rewriting* can be used for program optimization purposes, there are only few works on it. [FRT95] applies it to *program slicing*, an analysis technique which filters out code parts which are dependent on single statements. However, general criteria for termination and/or normal forms are not given. *Term graph rewriting* [SPvE93] is a new technique to model and implement term rewriting by graph rewriting on shared subterms. Most of this work is only applicable to directed acyclic graphs, so that e.g. data-flow analysis cannot be described. Only [AA93] also allows cycles. This work defines a criterion for strong confluence, and investigates semantic features of a graph rewrite system in order to prove the equivalence of two graphs. Thus this work is somewhat orthogonal to ours.

8 Conclusion

The main contributions of the research described in this paper are the following. First several new rule-based termination criteria for graph rewrite systems have been developed which define *exhaustive graph rewrite systems*. In order to handle terminating, but non-deterministic systems, *stratification* for graph rewrite systems has been defined. Using examples from lazy code motion we demonstrated that

[1] This is a rough estimate because the numbers even comprise the runtime of the front-end and back-end. A quadratic algorithm is assumed for the number of tests. In fact even more tests are done. To compare with arbitrary graph transformation some cost has to be added, because COMPUTE-EQUIVALENCE only adds edges.

stratified graph rewrite systems are apt to specify classical program optimizations. Next, a generic evaluation algorithm for all exhaustive graph rewrite systems has been presented, which often leads to linear and quadratic concrete algorithms. We have shown that our method works in practice by giving some numbers of a concrete implementation of lazy code motion. Thus this paper proposes a new method for the uniform specification and evaluation of program analysis and transformation.

Using graph rewrite systems, for the first time a uniform view on local, global analysis, and transformation results. This facilitates the modular composition of optimizers, allowing coalescing and separation at the specification level. Moreover, the stratification heuristic structures complex specifications automatically, e.g. complex data-flow analyses. Because the methodology covers a broad range of analysis and transformation problems, is independent of the source and intermediate languages, and leads to efficiently executing optimizers, it should greatly facilitate the development of optimizers.

This work associates program optimization, DATALOG, and graph rewrite systems. Program analysis can be seen as a process to query the intermediate representation, or as a process to infer explicit knowledge implicit in the intermediate representation. On the other hand, exhaustive graph rewrite systems extend DATALOG in a straight-forward way to handle transformations. Moreover, they can be evaluated by DATALOG-algorithms. This provides a new approach for their efficient execution.

Future work will consider non-deterministic graph rewrite systems. It is interesting to investigate how a rule-based cost function can be used to select better or best derivations. As in the case of weighted term rewrite systems for codegenerator-generators [Emm92] such a method would lead to generated optimizers which automatically improve the quality of their manipulated code. Finally, stratified rewrite systems provide a powerful rule-based programming paradigm in which the control flow is automatically computed as a result of the stratification. In contrast to logic programming, stratified graph rewrite systems allow general data transformations. Hence they could extend standard programming languages quite usefully.

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*, Lecture Notes in Computer Science, Heidelberg, November 1995. Springer. to appear.

[Aß95c] Uwe Aßmann. Optimix Language Report. Technical Report 31, Universität Karlsruhe, 1995.

[Aß96] Uwe Aßmann. How To Uniformly Specify Program Analysis and Transformation. In P. A. Fritzson, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, Heidelberg, 1996. Springer.

[AA93] Zena Ariola and Arvind. Graph rewriting systems for efficient compilation. In M. R. Sleep, M. J. Plasmeijer, and M. C. van Eekelen, editors, *Term Graph Rewriting—Theory and Practice*, chapter 6, pages 77–90. John Wiley and Sons Ltd, New York, 1993.

[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 (CC)*, volume 786 of *Lecture Notes in Computer Science*, pages 278–293, Heidelberg, April 1994. Springer.

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

[BFG95] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. In J. Cuny, editor, *5th Workshop on Graph Grammars and Their Application To Computer Science*, Lecture Notes in Computer Science, Heidelberg, 1995. Springer. to appear.

[Bin94] David Binkley. Interprocedural Constant Propagation using Dependence Graphs and a Data-Flow Model. In P.A. Fritzson, editor, *Compiler Construction (CC)*, volume 786 of *Lecture Notes in Computer Science*, pages 473–388, Heidelberg, April 1994. Springer.

[CCF94] J.-D. Choi, R. Cytron, and J Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Software Engineering*, 20(2):105–114, feb 1994.

[CCL+96] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In Tibor Gyimothy, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267, Heidelberg, April 1996. Springer.

[CGT89a] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, Heidelberg, 1989.

[CGT89b] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge And Data Engineering*, 1(1):146–166, March 1989.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Science Publishers, 1990.

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

[EKL90] H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *4th International Workshop On Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 24–37, Heidelberg, March 1990. Springer.

[Emm92] H. Emmelmann. Code selection by regularly controled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, Workshops in Computing. Springer, 1992.

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric Program Slicing. In *ACM Symposium on Principles of Programming Languages*, volume 22, pages 379–392. ACM, January 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, Amsterdam, 1977.

[Koc92] Gerd Kock. *Spezifikation und Verifikation von Optimierungsalgorithmen*, volume 201 of *GMD Berichte*. Oldenbourg, München, 1992.

[KRS94] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In U. Kastens and P. Pfahler, editors, *Compiler Construction (CC)*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140, Heidelberg, October 1992. Springer.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[Nag79] M. Nagl. *Graph-Grammatiken, Theorie, Implementierung, Anwendungen.* Vieweg, 1979.

[New42] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[Plu93a] Detlef Plump. *Evaluation of Functional Expresions by Hypergraph Rewriting.* PhD thesis, Universität Bremen, 1993.

[Plu93b] Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M. R. Sleep, M. J. Plasmeijer, and M. C. van Eekelen, editors, *Term Graph Rewriting—Theory and Practice*, chapter 15, pages 201–214. John Wiley and Sons Ltd, New York, 1993.

[Plu95] Detlef Plump. On Termination of Graph Rewriting. In *Graph Theoretic concepts in Computer Science*, Lecture Notes in Computer Science, Heidelberg, 1995. Springer.

[PP94] S. S. Pinter and R. Y. Pinter. Program Optimization and Parallelization Using Idioms. *ACM Transactions on Programming Languages and Systems*, 16(3):305–327, May 1994.

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

[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] Andreas 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, Heidelberg, 1990. Springer.

[Sch91] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen.* PhD thesis, Universität Aachen, Deutscher Universitäts-Verlag, 1991.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis.* Prentice Hall, 1981.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. van Eekelen. *Term Graph Rewriting—Theory and Practice.* John Wiley and Sons Ltd, New York, 1993.

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

[SWZ95] Andreas Schürr, Andreas J. Winter, and Albert Zürndorf. Graph Grammar Engineering with PROGRES. In *European Software Engineering Conference ESEC 5*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234. Springer, September 1995.

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

[Ull94] Jeffrey D. Ullman. Assigning an appropriate meaning to database logic with negation. In H. Yamada, Y. Kambayashi, and S. Ohta, editors, *Computers as Our Better Partners*, pages 216–225. World Scientific Press, March 1994.

[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*, pages 2–12, 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, Computer Society Press, April 1992.

[VvSL93] J. Vollmer, H. van Someren, and M. Libourel. CCMIR Definition. Technical report, COMPARE Consortium, 1993. contact `info@ace`nl.

[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. A. Fritzson, editor, *Proceedings of the Poster Session of Compiler Construction*, number LiTH-IDA-R-94-11 in PELAB Research Reports. 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.

[Zür95] Albert Zürndorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, Universität Aachen (RWTH), jul 1995.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l' Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399