A Tutorial for OPTIMIX

Uwe Aßmann
Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation
Postfach 69 80, Am Zirkel2, 76128 Karlsruhe, Germany
assmann@ipd.info.uni-karlsruhe.de

November 3, 1998

Abstract

OPTIMIX is a tool for generating algorithms which construct and transform directed relational graphs. In particular, it facilitates many tasks in program compilation and optimization. OPTIMIX's input language allows to specify graph queries which localize analysis information as well as graph rewrite systems which describe transformations. The generator type-checks the rewrite systems with a graph data model and tests whether they fulfil a termination criterion. This paper explains the advantages of the OPTIMIX specification language for compiler writers and demonstrates that OPTIMIX can be applied to three major problem classes of program rewriting: *graph reachability problems*, *context-sensitive pattern match problems*, and *mark/transform problems*.

Compilation tasks and program optimizations should be specified abstractly and easily. This is possible with graph rewrite systems [Aßm96b]. To facilitate the construction of program transformers and optimizers, tools are desired which generate algorithms from graph rewrite specifications. OPTIMIX is such a tool.

The central idea of OPTIMIX is to regard all of the information in a compiler (the *intermediate representation*) as a set of relational graphs. Program objects, abstract syntax tree nodes, and intermediate code instructions are represented as nodes; predicates of these program entities are represented as relations. Then, program analysis and program transformation consist of rewriting graphs. 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 with the intermediate representation given by the front-end, and ending with an intermediate graph which is handed over to the back-end for code generation.

Hence, OPTIMIX's specification language is based on relational graph rewriting. In particular, OPTIMIX supports *edge addition rewrite systems (EARS)* and *exhaustive graph rewrite systems (XGRS)* [Aßm95] [Aßm96a] [Aßm96b]. EARS only add

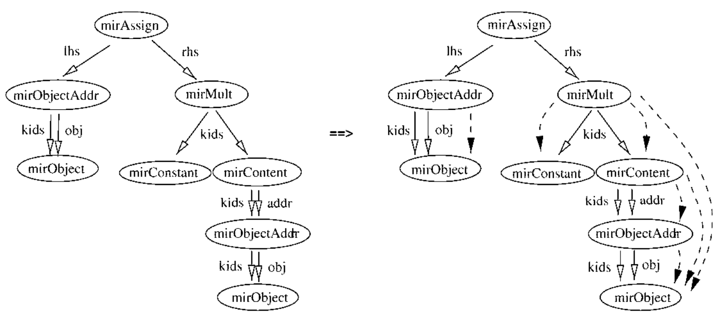

Figure 1: Left: Intermediate code of an assignment with its left- and right-hand side expression trees. Right: after applying `CollectDescendantsOfExpressions`. New edges are dashed.

edges to graphs. They can be used to construct graphs and to materialize implicit relations to explicit ones. They are equivalent to a subset of DATALOG and always yield unique results being *congruent* [Aßm94]. XGRS allow graph manipulations and may result in multiple normal forms (rewrite results). However, since each XGRS rule fulfils an *edge-termination criterion*, the termination of a rule system can be checked [Aßm96a].

In the following, the advantages of OPTIMIX's specification language are explained, with special regard to compiler writers. It is demonstrated that OPTIMIX can be applied to the following problem classes of program transformation and optimization: *graph reachability problems* (next section), *context-sensitive pattern match problems*, and *mark/transform problems* (section 2). Practical experiences with the tool conclude the paper. The examples use a subset of the intermediate language CCMIR [AAvS94].[1]

1 What users can do with OPTIMIX

The first class of problems OPTIMIX is useful for are *graph reachability problems*. Often these can be expressed as queries on a graph database which investigate the structure of a graph, infer new structural knowledge, and materialize the knowledge by augmenting graphs. An example is the problem of transitive closure.

Assume that the intermediate language provides a statement which assigns an object from an expression of operators (see Figure 1). Suppose the user wants to attach an expression with the set of all descendant expressions. Probably, he would have a query in mind such as "For each expression, find all reachable sub-expressions". In essence, this means to calculate the transitive closure

[1]See appendix A. CCMIR is an intermediate language for C, Fortran-90, and Modula-2. All its types are prefixed by the prefix `mir` (medium intermediate representation). Some types are renamed for readability.

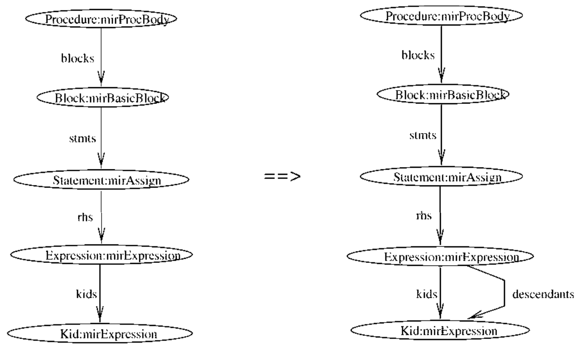

Figure 2: Rule 1 of `CollectDescendantsOfExpressions` depicted graphically

on the expression relation. The following EARS specifies how a relation `kids` over expressions can be used to construct a relation `descendants`.

```
GRS CollectDescendantsOfExpressions(Procedure:mirProcedure)
{  RULES
   blocks(Procedure,Block), stmts(Block,Statement),
    Statement ~ mirAssign,              // pattern match on mirAssign
    rhs(Statement,Expression), kids(Expression,Kid)
   ==> descendants(Expression,Kid);
   blocks(Procedure,Block), stmts(Block,Statement),
    Statement ~ mirAssign, rhs(Statement,Expression),
    descendants(Expression,Descendant), kids(Descendant,GrandKid)
   ==> descendants(Expression,GrandKid), descendants(Descendant,GrandKid);
}
```

The preconditions of the first rule, found before the delimiter ==>, denote the following. There must be a procedure body `Procedure` which has an associated basic block `Block` under relation `blocks`[2]. This block has a statement `Statement` under relation `stmts`. It is pattern matched to be of type `mirAssign` and has a right-hand side expression `Expression`. `Expression` has a child `Kid` in relation `kids`. If all these conditions are true, `Kid` should also be a descendant of `Expression`, i.e. relation `descendants` should hold between `Expression` and `Kid`. The second rule describes the transitive case of the closure problem: if there is a descendant `Descendant` of expression `Expression` which has another child `GrandKid` then `GrandKid` should also be a descendant of `Expression` and of `Descendant`.

`CollectDescendantsOfExpressions` is an EARS because it only adds edges of type `descendants`. According to [Aßm94] it is congruent, i.e. it terminates and yields a unique result. The rule system can also be described graphically (rule 1 is depicted in Figure 2). In the figure, edge labels correspond to predicate literals, the first part of the node labels corresponds to variable names, and the second part of the node label specifies the node type. In this paper only the OPTIMIX format, a textual representation, will be used. Consequently, the terms *variable*

[2]A basic block is a list of statements without interleaved jumps or conditional jumps.

and *predicate* are used for nodes and relations in rules; the terms *node*, *edge*, and *relation* for nodes, edges, and relations of the manipulated host graph.

Predicates need not be written in Datalog style but can be specified in a set-based form. For instance, `CollectDescendantsOfExpressions` can be rewritten as follows, the semantics should be obvious:

```
grs CollectDescendantsOfExpressions(Procedure:mirProcedure)
{  rules
   if  Statment  in Procedure.blocks.stmts
      and Statement matches mirAssign
      and Kid       in Statement.rhs.kids
   then Kid      in Expression.descendants;

   if  Statment   in Procedure.blocks.stmts
      and Statement  matches mirAssign
      and Expression in Statement.rhs
      and Descendant in Expression.descendants
      and GrandKid   in Descendant.kids
   then GrandKid in Expression.descendants
      and  GrandKid in Descendant.descendants;
}
```

The generated code To generate algorithms from the specifications, OPTIMIX applies DATALOG techniques. The main method is an adapted nested-loop join algorithm, the *order algorithm* [Aßm94]. It is efficient on sparse graphs, because the cost of the algorithm is defined in terms of the maximal out-degree of a node. For the example system OPTIMIX generates a C routine with the following layout:

```
void CollectDescendantsOfExpressions(mirProcedure Procedure)
{  int _change = TRUE;
   while (_change) {  _change = FALSE;
      /* code for rule 1: */
      conslist_LOOP(Procedure->blocks, Block) {
         conslist_LOOP(Block->stmts, Statement) {
            if (!GRAPHNODE_HAS_TYPE((GRAPHNODE_TYPENAME)Statement,mirAssign)) continue;
            Expression = Statement->rhs;
            consset_LOOP(Expression->kids, Kid) {
               _change |= consset_insert(Expression->descendants,Kid);
            } consset_ENDLOOP;
         } conslist_ENDLOOP;
      } conslist_ENDLOOP;
      /* similar code for rule 2: */
      conslist_LOOP(Procedure->blocks, Block) {
         conslist_LOOP(Block->stmts, Statement) {
            if (!GRAPHNODE_HAS_TYPE((GRAPHNODE_TYPENAME)Statement,mirAssign)) continue;
            Expression = Statement->rhs;
            consset_LOOP(Expression->descendants, Descendant) {
               consset_LOOP(Descendant->kids, GrandKid) {
               _change |= consset_insert(Expression->descendants,GrandKid);
               _change |= consset_insert(Descendant->descendants,GrandKid);
            } consset_ENDLOOP;
         } conslist_ENDLOOP;
      } conslist_ENDLOOP;
   }
}
```

It is also assumed that all relations in the intermediate representation are represented by neighbor sets in node types. The generated code instantiates the variable `Expression` to all expressions reachable from the parameter `Procedure` and applies the two rules to them. The code navigates on the intermediate representation to find an appropriate set of nodes which is linked in those relations which correspond to the predicates in rules. Hence all relevant neighbor sets (`blocks`, `stmts`, `rhs`, `kids`) are traversed with nested loops or dereferencing statements. Loops are described by C loop macros which contain the name of the set data type by which a neighbor set is represented (e.g. `conslist`). In the innermost loop a redex is found, i.e. all instantiated nodes belong to a valid redex. Then the transformation can be applied, which is to add an edge to the relation `Descendant` between the currently instantiated node for `Expression` and `Kid`. Because the rules are recursive and one round of the loops does not yield the desired result, OPTIMIX embeds the rule applications in a fixpoint evaluation loop. When this loop stops, each expression contains all reachable subexpressions in its set `descendants`.

1.1 How to develop a specification

The example proposes the following steps how a user should proceed to use OPTIMIX to generate optimizer parts:

1. First the user should (informally) define graph queries which filter out the analysis information he wants to know.
2. Together with that a data model of the graphs must be defined. A user has to specify graph node types (objects), relations, and which C modules are used to represent the relations.
3. Then the informal queries should be formulated as graph rewrite system procedures to construct and transform the graphs that were defined in the data model. This may extend the data model.

   Usually an application can be structured into a set of graph rewrite system procedures. OPTIMIX transform to C procedures, which the user has to call from his compiler appropriately.
4. Lastly the specification may be rearranged to tune the generated algorithms. Common parts of rules may be factored out, the representations of the graphs may be exchanged, and certain annotations may be inserted in the specification.

2 An introduction to Optimix specifications

Before another example from the domain of context-sensitive pattern match problems is presented, the basic outline of an OPTIMIX specification is explained.

OPTIMIX-specifications are grouped in *modules* (*data model* and *graph rewrite modules*) which can be combined and reused flexibly. The data model of the graphs is defined by a simple extension of the AST data definition language from the compiler construction toolbox *Cocktail* [Gro89] so that existing AST data specifications can be reused and extended. Also, users can connect other tool environments easily by implementing an abstract interface module for graphs, nodes, and edges. OPTIMIX also supports a *CoSy-mode* in which code is generated for the CoSy compiler construction environment [AAvS94].

```
OptimixModule ::= ASTDataModule | GraphRewriteModule .
GraphRewriteModule ::= 'MODULE' ModuleName (UseClause|GraphRewriteProcedure)+ 'END'.
GraphRewriteProcedure ::= 'GRS' ProcName '(' ParameterDecl // ',' ')' ProcBody
.
ProcBody ::= RuleGroup | '{' RuleGroup+ '}'.
ParameterDecl ::= Parameter ':' NodeTypeName
   | Parameter ':' Functor '(' NodeTypeName [',' NodeTypeName] ')'.
UseClause ::= 'USE' '"' FileName '"'.
```

A *graph rewrite module* consists of use clauses and graph rewrite procedures. A *use clause* specifies a module to be imported. A *graph rewrite procedure* contains one or several groups of graph rewrite rules. For each rewrite procedure, OPTIMIX generates a C routine with the same name. A list of parameters can be specified to pass graphs, sets of nodes, or other variables to the generated C routine.

OPTIMIX coalesces the data modules into a single data model and collects the graph rewrite system procedures into one list. Then it type-checks the graph rewrite rules with the data model and checks the termination of the systems. Finally, OPTIMIX generates C code.

2.1 Rule groups in rewrite procedures

```
RuleGroup ::= '{' [RangeDeclarations] 'RULES' Rule+ '}'.
RangeDeclarations ::= 'RANGE' (Variable '<=' Parameter ';' | 'REUSE' Variable ';')+
.
```

A rewrite procedure contains a sequence of *rule groups*. Such a sequence can be used to master non-terminating or non-confluent rule sets. Often, such a set can be split into several groups. If each of them is terminating and/or confluent also the sequence of groups terminates and/or yields a unique result. Hence rule groups are executed in textual order. For a single rule group, the rules are applied interminglediy; the code is generated according to the *order algorithm* scheme, a variant of a nested-loop-join [Aßm94]. Rule groups may also be nested: in preconditions or transformations of rule tests it is allowed to open additional rule groups so that common parts of rules are factored out.

Range declarations for rule groups To achieve efficient code, it is useful to apply a rule group instead to the entire intermediate representation to a specific set of nodes. Since OPTIMIX's order algorithm scheme is a graph search and navigation algorithm it has to be told at which nodes the navigation should start. Such a declaration is called a *range declaration*. To find out which range declarations are necessary, rule left hand sides are analyzed. A left-hand side can regarded as a graph containing some roots (nodes with indegree 0) which match redex root nodes. Since the order algorithm starts to lookup redexes at potential redex root nodes, each rule variable that is a root node of the left-hand side's graph needs a range declaration, telling from which node domain the redex root nodes can be instantiated. In example 2 the only root node of the left hand side is `Statement`, and for this variable a range declaration is required.

Root variables of left-hand sides may be instantiated from:

- A single node parameter of the graph rewrite procedure. This is assumed as default. In the example from section 1 all redexes start with the parameter `Statement`, so this parameter is automatically predefined as range for variable `Statement`.
- A set parameter of the procedure or a node set of a parameter graph. In the first case the nodes are instantiated from the parameter set, in the second case from the node set of the parameter graph.
- A variable in an outer rule group of a rule group nesting, a reuse.

Termination check for rule groups OPTIMIX can check termination for certain graph rewrite rule groups, which economizes paper proofs. Rule groups that only add edges are EARS and terminate automatically [Aßm94]. Other groups are checked with the *edge-accumulative termination criterion* of XGRS [Aßm96a]. For those OPTIMIX finds the set of *edge-accumulative-termination labels* which causes termination. In the case of our example this would be the edge label set {`descendants`}.

OPTIMIX may be used to specify more general graph transformation problems than XGRS, but it cannot check whether the specified rule groups terminate. Also, a problem with an unknown number of redexes may have a larger number of normal forms, i.e. may provide a larger degree of indeterminism. While the generated algorithms may yield correct results, these may be unexpected. Thus, for such specifications it is up to the user to ensure termination and to cope with indeterminism. Nevertheless the generated algorithm may yield reasonable results.

2.2 Type-checking rules

```
Rule ::= RuleTest '==>' RuleTransformation '.'.
RuleTest ::= (Predicate | PatternMatch | TargetCode)* .
Predicate ::= ['NOT'] PredicateName '(' Variable ',' Variable ')'.
PatternMatch ::= Variable '~' Pattern | Variable ('==' | '!=') Variable .
Pattern ::= NodeTypeName ['{' (FieldName '=>' [Variable ':='] Pattern) // ',' '}']
.
TargetCode ::= '{*' any_C_char* '*}'.
```

To understand the semantics of rules with regard to the data model, their type-checking has to be explained. Each rule consists of a rule test part (specifying rule preconditions) and a rule transformation part (specifying rule actions). Preconditions and actions are specified as predicates, combined in conjunction, as in DATALOG. However, each predicate must be binary because it must correspond to a graph defined in the data model. Looking up the predicates as graphs in the data model, OPTIMIX infers types for rule variables, and finds out the features of the used relations, e.g. whether relations are one- or multi-valued.

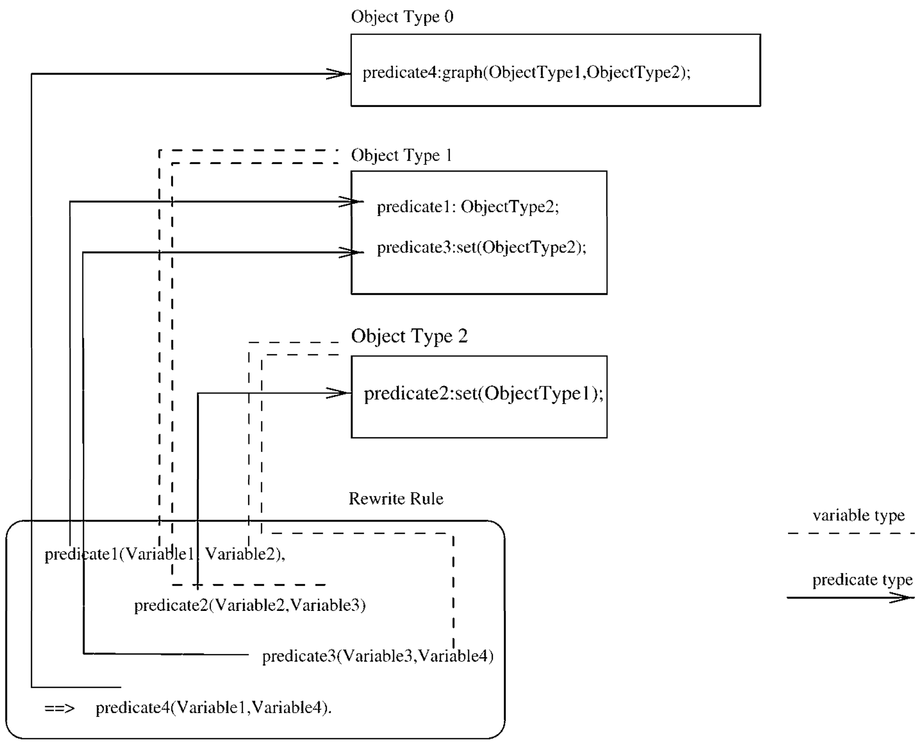
Object Type 0
predicate4:graph(ObjectType1,ObjectType2);
Object Type 1
predicate1: ObjectType2;
predicate3:set(ObjectType2);
Object Type 2
predicate2:set(ObjectType1);
Rewrite Rule
predicate1(Variable1, Variable2),
predicate2(Variable2,Variable3)
predicate3(Variable3,Variable4)
==> predicate4(Variable1,Variable4).
variable type
predicate type

Figure 3: Type-checking a part of a rewrite rule against the data model. Predicates refer to field names (dotted arrows), variables to node types. Inclusion of node types denotes simple inheritance (see app. A).

To define relations in the data model, parametrized set, list, or graph data types must be used (*functors*). Functors are instantiated by one or two node types to specify a concrete graph or set. OPTIMIX's functor library provides cons-cell-style lists and sets, hash sets, bitvectors, unipartite, and bipartite graphs. Graphs may be *explicit* or *implicit*. Explicit graphs provide an explicit root node for the graph, an explicit set of nodes, and their relations. Implicit graphs are represented only by the relation, which is embedded as *neighbor sets* in the nodes (adjacency sets/lists). In section 1 all graphs were implicit, since they were encoded as neighbor sets into the graph nodes.

To check a rewrite procedure against the data model, all predicates are looked up as a field in a node type (Figure 3). The type of the field may be the following:

node type (pointer type) (`rhs` in Figure 3) Then the relation is one-valued. The node type which contains the field determines the type of the left variable of the predicate. The type of the field is the type of the right variable of the predicate.

set/list functor type (`kids` and `descendants` in Figure 3) Then the relation is multi-valued and represented as implicit graph, embedded as neighbor sets. The node type which contains the field determines the type of the left variable of the predicate. The parameter of the functor determines the type of the right variable of the predicate. In case of a list functor the neighbor set is ordered.

graph functor type (`equiv_exprs` in Figure 3) Then the relation is an explicit graph on two node types. The first functor parameter determines the type of the left variable, the second the type of the right variable. The root node of the graph is accessible via an node of the node type which contains the field.

These steps deliver a set of typing constraints which are solved to infer the actual typing of predicates and variables.

Rules may contain more complex tests. Predicates may be *negated*. Then the generated code tests that no edge exists between two instantiated nodes. Also *pattern matches* on variables are allowed. They match variables on node types, select field values of nodes into variables, or match fields against constant values. Finally, it is possible to specify arbitrary C *target code* in place of a predicate. Target codes allow arbitrary custom checks and are copied unchanged to the generated file.

OPTIMIX specifications provide *graph representation transparency*. It is transparent by which functor a predicate is represented since the functor can be changed in the data model, and OPTIMIX's code generation is adapted automatically. OPTIMIX uses the type information to generate correct navigation and manipulation code, generating calls to macros and functions from its functor library (loop macros on neighbor sets, operations to add edges, delete edges, delete nodes, add nodes, and test existence of edges). Functor parametrization does not lead to code explosion because OPTIMIX expands functors only internally. All generated function calls invoke the same library module, passing objects as void-pointers. This is type-safe because specifications are type-checked by OPTIMIX anyway.

Furthermore, OPTIMIX allows users to define their own functors. For such functors, a user has to write a C module which supports the same functions as a standard functor. OPTIMIX provides a *node-macro interface* that manipulates nodes in the generated code. Suppose, a user wants to use a tool T instead of AST. Then he has to redefine the node-macro interface to map to T's node manipulation functions. Since OPTIMIX only generates calls to the macro interface, the generated code allocates and manipulates T-nodes automatically.

Other elements of rule conditions In rule tests not only binary predicates may be combined in conjunction, but several other actions or tests may be performed.

In place of predicates external functions may be called. These functions must be declared; they must take a node and return a node or a set of nodes. Then OPTIMIX generates calls to these functions, and uses their results for further navigation. For further details see [Aßm98]. Also external functions delivering a boolean may be called. These are called during the navigation.

Problem class: context-sensitive pattern match problems OPTIMIX-rules allow to specify context-sensitive pattern match problems. Often subgraphs of the intermediate representation have to be compared to subgraphs that are not directly linked, i.e. are remote. Context-sensitive matching is possible if a rule's left-hand side consists of disconnected predicate conjunctions (i.e. the graph of the left-hand side consists of disconnected subgraphs). Such a rule can match subgraphs in remote parts of the intermediate representation. In this way, context-dependent knowledge can be accessed very easily and can be made explicit.

An important subclass are *structure-based equivalence class problems* ("*Are two subgraphs in the intermediate representation equivalent?*"). A typical example calculates structurally equivalent expressions (*value numbering*) [Aßm96b]. Consider the following rule from a specification of value numbering. The rule relates all integer constant nodes which carry the same value in the field `intvalue`. These expressions need not be linked directly; the range of the expressions, i.e. the parameter set, is searched exhaustively for integer constant nodes whose values are equal.

```
GRS StructuralEquivalence(expressions:consset(mirExpression))
{  RANGE Expr1 <= expressions; Expr2 <= expressions; RULES
   Expr1 ~ mirIntConst{intvalue => Val1}, // pattern match on type mirIntConst
    Expr2 ~ mirIntConst{intvalue => Val2},
    Val1 == Val2,                          // are values equal?
    hasType(Expr1,Type1),hasType(Expr2,Type2),
    Type1 == Type2,                        // are pointers equal?
   ==> equiv_exprs(Expr1, Expr2),
      {* printf("heureka, two equivalent expressions found!\n"); *} ;
}
```

Line 2 shows the range declaration which instantiates the variables `Expr1` and `Expr2` to nodes of the parameter set `expressions`. Line 3 and 4 show two pattern matches. Line 8 in the rule action links the two expressions under a new relation `equiv_exprs` for further processing. Line 9 shows a target code with a print statement which is inserted into the output file after the code for line 8.

Problem class: filtering by pattern matching OPTIMIX allows to specify pattern matching in queries in order to filter out certain nodes or paths during graph navigation. The following example collects all reachable nodes in the expression trees of an assignment into two relations `uses` and `assigns`. The first rule represents the query *"Find all objects in the right-hand side of assigns and connect them to the assign under relation uses"*, the second the query *"Find the object in a left hand side and connect it to the assign under the relation assign"*. In our case, there is only one node on a left-hand side, i.e. `assign` is a one-valued relation.

```
GRS AttachObjectsToAssigns(Procedure:mirProcedure)
{   RULES
    blocks(Procedure,Block), stmts(Block,Assign),
      Assign ~ mirAssign{ rhs => Expression },
      descendants(Expression,Descendant),
      Descendant ~ mirObjectAddr { obj => Object, deleted => FALSE }
    ==>
      uses(Assign,Object)
    ;
    blocks(Procedure,Block), stmts(Block,Assign),
      Assign ~ mirAssign{ lhs => Expression },
      descendants(Expression,Descendant),
      Descendant ~ mirObjectAddr { obj => Object, deleted => FALSE }
    ==>
      assigns(Assign,Object)
      ;
}
```

In graph reachability problems, testing attributes of nodes (*pattern matching*) may be used to filter out certain subgraphs for further processing. Many preparatory work for program analysis consists in filtering out interesting parts of the intermediate representation and transforming them into a shape which can be used for further analysis.

Problem class: data-flow analysis Interestingly, [RHS95] has shown that classical data-flow problems can be expressed as graph reachability problems. Then both the flow graph and the values of the semi-lattice of the data-flow problem need to be encoded in graphs. Given some basic reachability rules, these problems ask: *"Which data-flow nodes can be reached from which statements in the flow graph?"*. In data-flow analysis theory, the reachability rules are described by linear dependency equations and model a set of functions on a (powerset) lattice. In graph rewriting, the rules are described by graph rewrite rules on a graph-lattice, and the reachability problem reduces to solving the rewrite rules on the initial graph [Aßm94]. The following system calculates a simple bitparallel data-flow analysis for reaching definitions.

```
GRS ReachingDefinitions(ReachdefIN, ReachdefOUT, ReachdefTRANSP:
                           graph(mirBasicBlock,mirStatement))
```

```
{
      RANGE b <= ReachdefIN;
      RULES
      ReachdefGEN(b,s)                        ==> ReachdefOUT(b,s);
      ReachdefIN(b,s),    ReachdefTRANSP(b,s) ==> ReachdefOUT(b,s);
      blocks.pred(b,b1), ReachdefOUT(b1,s)    ==> ReachdefIN(b,s);
}
```

If-expressions and path expressions give to the same set of rules a nice look:

```
grs ReachingDefinitions(ReachdefIN, ReachdefOUT, ReachdefTRANSP:
                          graph(mirBasicBlock,mirStatement))
{     range b <= ReachdefIN;
      rules
      if s in b.ReachdefGEN            then s in b.ReachdefOUT;
      if s in b.ReachdefIN
      && s in b.ReachdefTRANSP         then s in b.ReachdefOUT;
      if s in b.blocks.pred.ReachdefOUT  then s in b.ReachdefIN;
}
```

2.3 Transformation actions in rules

```
RuleTransformation ::= [NodesToBeDeleted] [NodesToBeAdded] Predicate+ '.'.
NodesToBeDeleted ::= 'DELETE' Variable // ',' ('MARK'|'FREE'|'REDEXREMOVE')+ ','.
NodesToBeAdded ::= 'NEW' (Variable // ',' ':' NodeTypeName ';') * ','.
```

This section demonstrates OPTIMIX's capabilities for program transformation. Transformational parts of rules may contain – besides edge additions – edge deletions, node additions, and node deletions. If a predicate appears *negated* in a rule transformation, between the two nodes that instantiate the rule's variables a potentially existing edge is deleted. Nodes which are added have to be declared together with their types. OPTIMIX generates appropriate calls to allocation macros.

Nodes are deleted in different modes, which can also be combined. The *free-mode* deallocates the node by calling a deallocation macro. The *mark-mode* marks the nodes by setting a default node field `deleted`. Then they can be recognized by other rules as being invalid. Marking is necessary when a node belongs to several graphs, not only those that were tested in the rule. Then subsequent navigations can remove all incident edges from those graphs before the node is deallocated. The *redex-remove-mode* removes the node from its redex context automatically, i.e. from all containing graphs which are mentioned in the rule test.

The *delayed-remove mode* is important when nodes together with incident edges have to be deleted. In a first pass this mode deletes the attached edges of deleted nodes. Then in a second pass nodes are removed from containing graphs. When these rules test and delete edges of the same graphs, the deletion of edges may destroy redexes which otherwise would have been valid. which are candidates for deletion should be deleted delayed should be used for further navigation. Thus this mode generates a second XGRS which is executed after the original one. This XGRS walks the graphs of the rule test a second time, tests on deleted (marked) nodes and then performs removal of incident edges.

In rule transformation parts also negated predicates may be specified. They specify edge deletions.

Non-ground facts Rules without preconditions correspond to non-ground facts in DATALOG-databases, e.g. in Coral [RSS92]. Non-ground facts are like facts which refer to a certain domain of nodes. Since they will always match these rules may be used to initialize graphs. Negated non-ground facts may be used to empty graphs. As example consider an initialization of the `copies_to_be_replaced-relation` with a non-ground fact:

```
GRS InitializeGraph(copies_to_be_replaced:graph(mirStatement))
{  RANGE Statement1 <= copies_to_be_replaced;
         Statement2 <= copies_to_be_replaced.target;
   RULES
   copies_to_be_replaced(Statement1,Statement2); // initialization to complete graph
}
```

Problem class: mark/transform-problems *Mark/transform-problems* mark a finite set of subgraphs in an intermediate representation where transformations have to be applied. A subsequent phase performs the transformations, reducing all redexes in the graph. XGRS model this kind of transformation because their termination criterion requires that during the reduction only a finite number of redexes appear in the graph [Aßm96a]. Example problems are *code motion*, *common subexpression elimination (structure sharing)*, or *copy propagation*.

```
GRS CopyPropagation(copies_to_be_replaced:graph(mirStatement))
{  RANGE CopyAssign <= copies_to_be_replaced; RULES
   copies_to_be_replaced(CopyAssign, UsingAssign),
    CopyAssign ~ mirAssign{rhs=>mirObjectAddr{obj=>Original},
                           lhs=>mirContent{addr=>mirObjectAddr{obj=>CopyObj}}}
    rhs(UsingAssign,RhsExpr), descendants(RhsExpr,UsingExpression),
    UsingExpression ~ mirObjectAddr{obj=>ReplaceObj}
    ReplaceObj == CopyObj
   ==> DELETE CopyAssign, CopyObj MARK REDEXREMOVE, // mark and cut out
      obj(UsingExpression,Original); // Replace all CopyObj objects by Original
}
```

In copy propagation, i.e. elimination of useless copy assignments, first all copy statements are assembled, then reaching-definitions and live-copy dataflow analysis calculate which copies of definitions reach a certain statement. Here it is assumed that this information has been *marked* in the relation `copies_to_be_replaced`. Then the following system can be used to remove all useless copy assignments and to replace the useless copied program object by the original program object. The system finds all expressions which use the copy object of the copy assignment, instantiating variable `UsingExpression`. It also replaces the use of the copy by the use of the original by redefining the one-valued-relation `obj` between the variables `UsingExpression` and `Original`.

Since the delete statement applies *mark-mode*, both `CopyAssign` and `CopyObj` are not deallocated, but marked as deleted. This might be necessary in order to remove dangling edges later on. Since also *redex-remove-mode* is specified, the generated code removes both nodes automatically from all graphs that were mentioned in the rule test, e.g. `copies_to_be_replaced`.

The following system could be executed subsequently and removes the a useless copy assignment which had been marked from the graphs `dep`, `isLiveCopy`.

```
GRS RemoveEdgesFromGraphs(copies_to_be_replaced:graph(mirStatement),
                          dep:graph(mirStatement),
                          isLiveCopy:graph(mirStatement))
{
   RANGE CopyAssign <= copies_to_be_replaced;
   RULES
   copies_to_be_replaced(CopyAssign, UsingAssign),
    CopyAssign ~ mirAssign{ deleted => TRUE }
   ==>
    NOT dep(CopyAssign, UsingAssign),
    NOT isLiveCopy(CopyAssign, UsingAssign)
   ;
}
```

2.3.1 Fixpoint checking for rule groups

For rule groups it must be checked when the fixpoint is reached. This is done by comparing old and new values of certain parts of the graphs. OPTIMIX can generate different kinds of fixpoint detection:

direct fixpoint check This method checks whether calls to edge-addition functions change the graph. Because it works without memoization it is the fastest method, but can only be used if the functions for edge addition return a flag, when something changed. This is the case for all functors from the OPTIMIX-graph-library.

central neighbor set comparison This method compares the neighbor sets certain nodes before and after a fixpoint loop iteration. Thus it has to memorize the old values of neighbor sets which may be expensive. However, it can be used with arbitrary functors.

2.4 Some techniques to accelerate the generated code

Although the order algorithm for XGRS works well for sparse graphs in many cases, it may be that the code is too slow for an appliation, in particular when the preconditions of a rule are very complex. Then there are some other methods to speed up the code.

Rule annotations for code generation In case the standard order algorithm is too slow users can annotate rules with rule options so that OPTIMIX uses other code generation strategies. Annotating a rule with the **element-test-join** option replaces some inner loops by testing membership of nodes in node sets. Because some set representations allow membership tests in constant time, often the cost of a join can be reduced. Annotating a rule with the option **pre-select-filtering** inserts pre-select-operations in the generated code which cut down the search space for joins. This corresponds to the principle of selections-before-joins in databases. Since each of these options may not pay off in all cases, users have to write them explicitly.

Factoring out common parts of rules In certain cases algorithms generated from a rule group can be speeded up by *rule factoring*. Normally code is generated for each graph rewrite rule separately. Hence many parts of the intermediate representation are traversed repeatedly, which could be economized. To this end, OPTIMIX allows to factor out common parts of rules by nesting rule groups. In place of rules or predicates complete rule groups can be specified. The code for the nested rule group is generated after the code for the textually proceeding rule or predicate. For nested rule groups, variables from outer rule groups can be reused in range declarations. For instance, the example from section 1 can be modified to:

```
GRS CollectDescendantsOfExpressions(Procedure:mirProcedure)
{  RULES
   blocks(Procedure,Block), stmts(Block,Statement),
    Statement ~ mirAssign, rhs(Statement,Expression),
    { RANGE REUSE Expression; RULES
      kids(Expression,Kid) ==> descendants(Expression,Kid) .
      descendants(Expression,Descendant), kids(Descendant,GrandKid)
      ==> descendants(Expression,GrandKid), descendants(Descendant,GrandKid) ;
    }
    ==> /* empty transformation */;
}
```

Then the relations `blocks`, `stmts`, and `rhs` are traversed only once for the evaluation of both rules, and the generated algorithm should be almost twice as fast:

```
void CollectDescendantsOfExpressions(mirProcedure Procedure)
{  int _change = TRUE;
   while (_change) {  _change = FALSE;
      conslist_LOOP(Procedure->blocks, Block) {
         conslist_LOOP(Block->stmts, Statement) {
            if (!GRAPHNODE_HAS_TYPE((GRAPHNODE_TYPENAME)Statement,mirAssign)) continue;
            Expression = Statement->rhs;
            /* code for rule 1: */
            consset_LOOP(Expression->kids, Kid) {
               _change |= consset_insert(Expression->descendants,Kid);
            } consset_ENDLOOP;
            /* code for rule 2: */
            consset_LOOP(Expression->descendants, Descendant) {
               consset_LOOP(Descendant->kids, GrandKid) {
               _change |= consset_insert(Expression->descendants,GrandKid);
               _change |= consset_insert(Descendant->descendants,GrandKid);
               } consset_ENDLOOP;
            } conslist_ENDLOOP;
         } conslist_ENDLOOP;
      } conslist_ENDLOOP;
   }
}
```

Factoring with choice rule groups Choice-rule-groups allow to specify alternative rule conditions, in particular for pattern matching alternatives. Such a rule group consists of a number of rules that are tried in textual order until the first redex is found; rules are not evaluated until a fixpoint is found. If choice-rule-groups are nested into other rule groups, they allow to specify graph queries with complex preconditions. The following system shows a nested choice-rule-group which tests several pattern matching alternatives. Common rule preconditions are factored out.

```
{  blocks(Procedure,Block), stmts(Block,Assign), Assign ~ mirAssign,
      rhs(Assign,RhsExpression), descendants(RhsExpression,Expression),
      {| RANGE REUSE Expression; RULES
         Expression ~ mirMult   ==> {* printf("mult op"); *};
         Expression ~ mirObjectAddr { obj=>NULL }
            ==> {* printf("no object for address"); *}.
         Expression ~ mirObject ==> {* printf("object"); *};
      |} ==>  /* empty transformation */ ;
   }
```

3 Experiences

OPTIMIX has been used for several areas in program optimization, such as data-flow analysis, lazy code motion, and copy propagation. Additionally, it can be used for arbitary program rewriting tasks, also semantical analysis or code generation. Typically, a program analysis specification requires only around 30-40 rules, indicating that program rewriting and optimization is facilitated with OPTIMIX. The following table shows the approximate number of rules for several analysis and transformation examples. Some analyses are rather complex: e.g. lazy code motion data-flow analysis contains 7 data-flow equation systems. Typically, a program analysis specification requires more rules than a transformation since it has to handle a lot of complicated cases.

component	problem class	appr. #rules
data-flow analysis live copies	reachability	20
data-flow analysis lazy code motion	reachability	40
structural expression equivalence	cont.-sens. matching	30
termination check in OPTIMIX	cont.-sens. matching	7
transformation copy propagation	mark/transform	5
kernel of lazy code motion transform.	mark/transform	5

For a lot of standard problems, the generated code is satisfactorily fast. In particular, reachability problems, such as data-flow analyses, are almost as fast as hand-written ones; numbers have been published already [Aßm96b]. With context-sensitive graph problems and mark/transform problems, the performance of the code depends on how complex rule preconditions are and how large the intermediate representation is. In complex cases, the code generation can be tuned by rule options. Also, OPTIMIX integrates several automatic optimizations to accelerate the generated code. Also many other DATALOG evaluation strategies could be used (top-down evaluation, semi-naive evaluation, query-subquery [CGT89]). Better fixpoint checking strategies should be applied, e.g. checking fixpoints only at the critical points of a weak topological ordering [Bou93].

OPTIMIX generates code that optionally emits debug output. Also, all functors provide routines to print a graph to a file to be displayed with VCG [San95]. This facilitates debugging of optimizations enormously. Figure 4 on page 20 contains a relation which represents live copy information between statements, and which is automatically generated by OPTIMIX's functor library.

OPTIMIX eases the development of an optimization. Since the specifications are abstract they can be better read and understood. The core of the specifications, the graph rewrite systems, forms a declarative language. Preconditions and transformations are specified in widely-known DATALOG-oriented style. Since OPTIMIX infers types for predicates and variables, specifications tend to contain only the essential information. This pays off especially in program optimization, since optimizers tend to be complex, hard to write, and hard to understand. It is possible to start an optimizer by simply stating informal queries. These queries can subsequently be refined and extended, and the generator supports the development process by its type-checking capabilities. In essence, with OPTIMIX optimizer development can be viewed as a kind of graph database development which makes it much more comprehensible for the average programmer.

OPTIMIX is not the only tool to implement graph rewrite problems. However, PROGRES [SWZ95] as well as its variant UBS-systems [Dör95] require control-flow to be stated together with the rules to guarantee termination. Other systems are oriented towards interactive graph rewriting [LB93]. Also, none of these seems to have been applied to program rewriting.

Also in artificial intelligence rule-based systems have been built. Among the newer works are [CM95] [For94] [Mir] and [Pac95], which embed production rules into C++ and Smalltalk. These systems use the RETE/LEAPS pattern match algorithms to find rule matches in their workspace. However, it is unclear whether these systems can be used for graph rewriting because their data model is not graph-based. The specifications cannot be checked for termination and the code generation does not use standard database techniques. It is unclear how RETE/LEAPS compares to join algorithms in practice, although theoretical works exist [Alb89].

4 Conclusion

With OPTIMIX optimizer development can be viewed as a kind of graph database development. It is possible to start an optimizer by simply stating informal queries. These queries can subsequently be refined and extended, and the generator supports the development process by its type-checking capabilities.

Hence OPTIMIX eases the development of program rewriting and optimization. It allows the specification of graph reachability problems, context-sensitive pattern match problems, and mark-transform problems. Since the language core, the graph rewrite rules, consists of a rule-based declarative language, rewrite specifications are concise and easy to read. Rules can be arranged in rule groups to master difficult specifications. The rewrite systems can be checked for termination. On the other hand, specifications are open and flexible, since C code can be embedded. Rules can be nested, and complex pattern matching is possible. OPTIMIX allows specification of arbitrary data models, i.e. intermediate representations, and can be adapted to new tool environments. It allows the user to speed up the generated code by rule factoring and rule options. Hence we believe that OPTIMIX provides a valuable practical tool which saves a lot of coding work in program transformation. A free version of the tool can be fetched from its home page [Aßm98].

The work on OPTIMIX opens a wide field for further investigations. On the theoretical side more general graph rewrite systems should be investigated whether more general termination and confluence criteria can be found. Termination criteria in graph rewrite systems are a new field [Plu95] and one could try to take over the work in term rewrite systems [DJ90]. Another important work is to annotate rules with a cost function. This could be used to select derivations in non-confluent systems, just as in code-generator generators [NKWA96] [NK96]. Also, the automatic choice of graph representations could be tackled so that users could be sure that the generator selects both the best possible graph representation and an appropriate solution algorithm. This would be an enormous improvement on Prolog, because in Prolog the layout of graph structures is fixed.

References

[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.

[Alb89] Luc Albert. Average case complexity analysis of RETE pattern-match algorithm and average size of join in databases. In *Foundations of Software Technology and Theoretical Computer Science*, number 405 in Lecture Notes in Computer Science, pages 223–241, Heidelberg, December 1989. Springer.

[Aßm94] Uwe Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *5th Int. Workshop on Graph Grammars and Their Application To Computer Science, Williamsburg*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335, Heidelberg, November 1994. Springer.

[Aßm95] Uwe Aßmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen*. PhD thesis, Universität Karlsruhe, Oldenbourg-Verlag, München, July 1995. GMD-Bericht 262.

[Aßm96a] Uwe Aßmann. Graph Rewrite Systems For Program Optimization. Technical Report RR-2955, INRIA Rocquencourt, 1996.

[Aßm96b] 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*, pages 121–135, Heidelberg, 1996. Springer.

[Aßm98] Uwe Aßmann. OPTIMIX Language Manual (for OPTIMIX 2.5). Technical Report 15, Universität Karlsruhe, 1998. Available at http://i44www.info.uni-karlsruhe.de/~assmann/optimix.html.

[Bou93] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.

[CGT89] 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.

[CM95] Stephen Correl and Daniel P. Miranker. Integrating database concurrency control into the venus rule language. Technical Report UTEXAS.CS//CS-TR-95-16, The University of Texas at Austin, Department of Computer Sciences, April 1995.

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

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

[For94] Charles L. Forgy. RAL/C and RAL/C++, Rule-Based Extensions to C and C++. July 1994.

[Gro89] Josef Grosch. Ast - A Generator for Abstract Syntax Trees. Technical report, Gesellschaft fuer Mathematik und Datenverarbeitung, Forschungstelle Karlsruhe, August 1989. Language manual.

[LB93] Michael Löwe and Martin Beyer. AGG — an implementation of algebraic graph rewriting. In *Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 451–456. Springer, 1993.

[Mir] Daniel P. Miranker. Encapsulating rules. The University of Texas at Austin, Department of Computer Sciences.

[NK96] Albert Nymeyer and Joost-Pieter Katoen. Code Generation based on formal BURS theory and heuristic search. Technical report inf 95-42, University of Twente, 1996.

[NKWA96] Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code Generation = A* + BURS. volume 1060 of *Lecture Notes in Computer Science*, pages 160–176, Heidelberg, April 1996. Springer.

[Pac95] Francois Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming (JOOP)*, pages 19–24, July 1995.

[Plu95] Detlef Plump. On Termination of Graph Rewriting. In *Graph Theoretic concepts in Computer Science*, Lecture Notes in Computer Science, Heidelberg, 1995. 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, San Francisco, California, January 1995. ACM.

[RSS92] R. Ramakrishnan, D Srivastava, and S. Sudarshan. CORAL - Control, Relations and Logic. In *Proceedings of the 18th VLDB Conference*, 1992.

[San95] Georg Sander. Graph Layout through the VCG Tool. In *Graph Drawing, DIMACS International Workshop (GD)*, number 894 in Lecture Notes in Computer Science, pages 194–205. Springer, 1995.

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

Appendix A

The data definition syntax of OPTIMIX is similar to the syntax of the data structure generator AST [Gro89]. Node types (classes) are composed from a collection of typed fields. Inner classes in angle brackets <> inherit from outer classes (simple inheritance). Fields may consist of set and graph attributes in brackets ().

```
MODULE AssigmentData TREE IntermediateRep RULES
mirProcedure =                          // the procedure type
   (blocks:list(mirBasicBlock))         // a list of basic blocks
   (equiv_exprs:graph(mirExpression,mirExpression))        // a graph of expressions
   (copies_to_be_replaced:graph(mirStatement,mirStatement)) // a graph of statements
   (ReachdefIN:graph(mirBasicBlock,mirStatement))          // reaching definitions block entries
   (ReachdefOUT:graph(mirBasicBlock,mirStatement))         // reaching definitions block exits
   (ReachdefTRANSP:graph(mirBasicBlock,mirStatement))      // transparents
.
```