History

- 1.0 2000, IBM.
- 1.1 2002, ACC
- 1.2 Feb 16, 2004. ACC.
Program Optimization with Graph Rewrite Systems

Prof. Uwe Aßmann
RISE
Linköpings Universitet
Problem and Goal

- We need optimizers
  - new processor types
  - memory hierarchy
- Optimizers are big beasts
  - Current techniques are hard to understand
- We need a uniform specification methodology
  - covering many parts
  - short specifications
  - flexibility
  - effective code improvements
  - efficient optimizer components
Overview

- Examples
- Termination Criteria
- Stratification
- Evaluation
- Practice
There clearly remains more work to be done in the following areas:

- discovery of other properties of transformations that appear to have relevance to code optimization,
- development of simple tests of these properties, and
- the use of these properties to construct efficient and effective optimization algorithms that apply the transformations involved.

Aho, Sethi, Ullmann in Code Optimization and Finite Church-Rosser Systems, 1972
Idea: Optimization With Graph Rewriting

- Represent everything as graphs
  - program code
  - the analysis information
  - “everything is a graph”

- Use graph rewrite systems (GRS) to
  - construct the graphs
  - transform the graphs

- Relational graphs with
  - node and edge types
  - node attributes
  - one-edge condition
Terms

- **redux**: application place in the manipulated graph
- **manipulated graph (host graph)**: graph which is rewritten
- **start graph (axiom)**: input graph to rewriting
- **normal form**: manipulated graphs without further redex
- **unique normal form**: unique result of a rewrite system, applied to one start graph
- **terminating GRS**: rewrite system that stops after finite number of rewrites
- **confluent GRS**: two derivations always can be joined together
- **convergent GRS**: rewrite system that always yields unique results
The Development Method

- **Specification of the data model:** specification of the graph data with an ER or UML model:
  - start graph (axiom): program code as objects and basic relationships. This data is provided as result of the frontend analysis information (the inferred predicates over the program objects) as objects or relationships.

- **Program analysis:**
  - querying graphs, enlarging graphs
  - materializing implicit knowledge to explicit knowledge.

- **Program transformation:**
  - querying graphs, transforming graphs.
A Program Model

Proc

Block

Stmt

Assign

AssReg

ExprClass

AllExprs

INSERT_IN

INSERT_OUT

LATEST_IN

Expr

Exprs

InRegister

Exprs

Exprs

Exprs

Left

UsedReg

AsgdReg

Register

Plus

IntConst

UseReg
Examples
Collection of Expressions

AllExprs(Proc,Expr) :- Blocks(Proc,Block), Stmts(Block,Stmt), Expr(Stmt,Expr).

if Blocks(Proc,Block), Stmts(Block,Stmt), Expr(Stmt,Expr) then AllExprs(Proc,Expr);
Value Numbering
(Expression Equivalence)

eq(\text{IntConst}_1, \text{IntConst}_2) :-
\text{IntConst}_1 \sim \text{IntConst}(\text{Value}),
\text{IntConst}_2 \sim \text{IntConst}(\text{Value}).

eq(\text{Plus}_1, \text{Plus}_2) :-
\text{Plus}_1 \sim \text{Plus}(\text{Type}),
\text{Plus}_2 \sim \text{Plus}(\text{Type}),
\text{Left}(\text{Plus}_1, \text{Expr}_1),
\text{Right}(\text{Plus}_1, \text{Expr}_2),
\text{Left}(\text{Plus}_2, \text{Expr}_3),
\text{Right}(\text{Plus}_2, \text{Expr}_4).
\text{eq}(\text{Expr}_1, \text{Expr}_3),
\text{eq}(\text{Expr}_2, \text{Expr}_4).
Classical Reaching Definition Analysis

- On instructions or on blocks

reach-end(B,E) :- gen(B,E).
reach-end(B,E) :-
    reach-begin(B,E), not killed(B,E).
reach-begin(B,E) :-
    pred(B,P), reach-end(P,E).
Data-flow Analysis for Lazy Code Motion (LCM)
Conclusions from LCM Analysis

- The equation systems in LCM analysis correspond to the GRS
- Idea:
  - Look at every set in the equation as neighbor set in the graph
  - A set equation becomes a rewrite rule
  - While the equation systems are specified with sets, the rules are specified in terms of singular objects (like in logic)
- LCM analysis is a nice example for a program analysis which can be specified easily with Datalog
Example: Lazy Code Motion Without Overlaps
Lazy Code Motion as Datalog

if Stmts.last(Block,Stmt),
    INSERT_OUT
    (Block,ExprClass)
then
    new Register:Register;
    new Expr:Expr;
    new AssReg:AssReg;
    InRegister(ExprClass,Register),
    AsgdReg(AssReg,Register),
    Exprs(AssReg,Expr)
;

if Stmts(Block,Stmt),
    Exprs(Stmt,Expr),
    REPLACE_OUT
    (Block,ExprClass),
    InRegister(ExprClass,Register),
    Computes(Expr,ExprClass)
then
    new UseReg:UseReg;
    delete Expr;
    Exprs(Stmt,UseReg),
    UsedReg(UseReg,Register)
;
Problems with GRS

- With graph rewriting there are some hard problems:
  - Termination
    - termination graph
  - Non-convergence
    - unique normal forms by rule stratification
I) Termination Criteria

- Question: How do I terminate a rewrite process?
Additive Termination

- Identify a termination (sub-)graph
- Conditions in the additive case:
  - nodes of termination (sub-)graph are not added (remain unchanged)
  - its edges are only added
- If the termination graph is complete, the system terminates
- This leads to the definition of AGRS (edge-Accumulative Graph Rewrite Systems).
- AGRS use the completion of the termination graph as their termination ordering
EARS – A Simple Form of AGRS

- A subclass of edge-accumulative graph rewrite systems are EARS (Edge Addition Rewrite Systems).
  - They can be used for the construction of graphs (analysis information).
- **terminating**: noetherian on finite lattice of subgraphs
- **strongly confluent**: direct derivations can always be interchanged.
- **congruent**: unique normal form
Data-flow Analysis with EARS

- Every distributive data flow problem on finite-height powerset lattices can be represented by an EARS:
  - defined/used-data-flow analysis
  - partial redundancies
  - local analysis and preprocessing:
- EARS work for other problems which can be expressed with DATALOG-queries:
  - equivalence classes on objects
  - alias analysis
  - program flow analysis
Example: Subexpressions

- "Find all subexpressions which are reachable from a statement"

Exprs(Stmt,Expr) :- Child(Stmt,Expr).
Exprs(Stmt,Expr) :- Child(Stmt,Expr2), Descendant(Expr2,Expr).

Descendant(Expr1,Expr2) :- Child(Expr1,Expr2).
Descendant(Expr1,Expr2) :- Descendant(Expr1,Expr3), Child(Expr3,Expr2).
Assign

Expr

Plus

Const
1

Var
X

Descendants
Assign

Expr

Plus

Const

1

Var

X
Edge-accumulative Rules and AGRS

- A GRS is called *edge-accumulative* (an AGRS) if
  - all rules are edge-accumulative and
  - no rule adds nodes to the termination-subgraph nodes of another rule.
- Edge-accumulative rules are defined on label sets of nodes and edges in rules
- Criterion statically decidable
Example: Allocation of Register Objects

"Allocate a register object for every subexpression of a statement which has a result and link the expression to the statement"

if Exprs(Stmt, Expr), HasResult(Expr)
then
    ObjectExprs(Stmt, Expr),
    RegisterObject := new Register;
    UsedReg(Expr, RegisterObject)
;
Features: terminating
Assign

Expr

Plus

Const

1

Var

X
Subtractive Termination

- Conditions in the subtractive case:
  - the nodes of the termination subgraph are not added (remain unchanged)
  - its edges are only deleted
- If the termination subgraph is empty, the system terminates
- Results in:
  - edge-subtractive GRS (ESGRS)
  - subtractive GRS (SGRS)
- AGRS, SGRS make up XGRS (eXhaustive Graph Rewrite Systems).
Peephole Optimization as Subtractive XGRS
Constant Folding as Subtractive XGRS
Assign

Expr

Const

1

Plus

Var

X
The Termination Subgraph of the Examples

Collection of subexpressions:
\[ T = (\{\text{Stmt,Expr}\}, \{\text{Exprs, Descendant}\} ) \]

Allocation of register objects:
\[ T = (\{\text{Proc,Expr}\}, \{\text{ObjectExprs}\} ) \]
The Nature of XGRS

- All redex parts in the termination-subgraph of the host graph are reduced step by step.
- The termination-subgraph is either completed or consumed.
  - Edge-accumulative systems may create new redex parts in the termination-subgraph, but
    - there will be at most as many of them as the number of edges in the termination-subgraph.
  - Subtractive systems do not create sub-redexes in the termination-subgraph but destroy them.
- XGRS can only be used to specify algorithms which
  - perform a \textit{finite} number of actions depending on the size of the host graph.
II) Stratification (the Problem of Determinism)

- EARS are congruent, i.e.
  - terminating and
  - confluent.
- The always deliver unique results (unique normal forms)
- They specify *functions*
- They are *deterministic*
Problem: Multiple Normal Forms of Graph Rewrite Systems

- If we allow negation, deletion and addition of items we get the following problem:
  - unique normal forms not guaranteed
  - derivable subgraphs get lost because of redex conflicts
  - A GRS can be *indeterministic*
  - Then it does no longer specify a function, but a *relation*, i.e., a set of solutions
  - Sets of solutions are hard to deal with (indeterministic)
- Reason: rule dependencies.
  - Rules perform the following actions competitively:
    - test (use), test absence (negation), add (define), delete (kill)
- However, we are interested in everything that can be derived.
Example: Leaf Expressions

"Find all leaf expressions of a statement"

Exprs(Stmt,Expr) :- Child(Stmt,Expr).
Exprs(Stmt,Expr) :- Child(Stmt,Expr2), Descendant(Expr2,Expr).
Descendant(Expr1,Expr2) :- Child(Expr1,Expr2).
Descendant(Expr1,Expr2) :- Descendant(Expr1,Expr3), Child(Expr3,Expr2).
/* NEW */
LeafExprs(Stmt,Expr) :- Exprs(Stmt,Expr), NOT Descendant(Expr,Expr2).

Features:
- terminating
- recursive
- no unique normal form (negation)
Stratification

- **Heuristic:** execute rules in the following order:
  - rules which produce subgraphs
  - rules which test on absence of subgraphs (negation)
  - rules which delete subgraphs

- **Investigate the graph of rule dependencies (RDG):**
  - use-use, use-neg, use-def, use-kill
  - def-use, def-neg, def-def, def-kill
  - .....
Stratification

- The RDG must be stratifiable, i.e. dividable in levels (rule groups, strata).
- Then the class of STRGRS (STRatifiable GRS) results.
- The strata are then executed in their order:
  - rules that use must be executed first
  - then rules that use negation (test absence)
  - then rules that kill
  - kill-kill dependencies are forbidden
- *many more subgraphs* („true statements“) *are derived*
- Weak stratification neglects self-dependencies of rules.
Unique Normal Forms with Stratified Convergence

- Let $G$ be a XGRS and $S' = [S_1, \ldots, S_n]$ a stratification of $G$.
- If every stratum $S_i$ in $S'$ has a unique normal form, the strata are computed in stratification order,
- the whole process selects a single normal form, the **stratified normal form**.
- Then $G$ is called **stratified-convergent**.
Unique Normal Form of All Stratifications

- There are many stratifications possible.

- Let be G be a XGRS and S' and S" two arbitrary stratifications of S.
- Then S' and S" yield the same normal form.

- A stratified system has a „deterministic“ semantics, i.e. is a function.
Proof Idea
Example: Replacement

"Replace all expressions of a statement which are a Plus(1) to an Incr"

...  
/* NEW */
if Exprs(Stmt,Expr), Expr == Plus(Var,Const(1))
then
  delete Expr;
  Expr2 := new Incr(Var),
  Exprs(Stmt,Expr2).

Features:
  - no unique normal form (edge deletion, node deletion, node addition)
Assign → Expr → Const → 1 → Var → X → Plus → Assign → Var → X → Expr
Resulting: 3 Strata for Example

\[
\begin{align*}
\{ \text{Exprs(Stmt,Expr) :- Child(Stmt,Expr).} \\
\text{Exprs(Stmt,Expr) :- Child(Stmt,Expr2), Descendant(Expr2,Expr).} \\
\text{Descendant(Expr1,Expr2) :- Child(Expr1,Expr2).} \\
\text{Descendant(Expr1,Expr2) :- Descendant(Expr1,Expr3), Child(Expr3,Expr2).} \\
\} \\
\{ \\
\text{LeafExprs(Stmt,Expr) :- Exprs(Stmt,Expr), NOT Descendant(Expr,Expr2).} \\
\} \\
\{ \\
\text{if Exprs(Stmt,Expr), Expr == Plus(Var,Const(1)) then} \\
\text{delete Expr;} \\
\text{Expr2 := new Incr(Var),} \\
\text{Exprs(Stmt,Expr2).} \\
\}
\]
Stratification for Lazy Code Motion
Excerpt from LCM Analysis with Overlaps

- `NOT earliest_out`
- `social_out`
- `comp_in`
- `latest_in`
- `NOT social_in`
RDG and Strata LCM Isolation Analysis

fixpoint of isolatedness analysis

add-negation (social_in)

add-negation (iso_lat_in)
Stratification for LCM Analysis

- The cyclic equations (fixpoint calculations) in LCM analysis conform to the strata a RDG analyzer would find.
- It is well known that fixpoints have to be computed before the later equation systems can be solved.
  - In this case the stratification achieves correctness of the analysis, i.e. the stratified normal form is the MFP.
- If the system is written down in Datalog, the strata comprise the same rules.
- Actually, LCM analysis is a nice example for a program analysis which can be specified easily with Datalog.
Example: Lazy Code Motion

- INSERT
  - ExprClass: 1
  - Stmt: 1
  - Stmt: 2
- REPLACE
  - ExprClass: 1
  - Expr: 1
  - Register: 1
- AssignRegister: 1
- UseRegister: 1
- R1
- R2
Not stratifiable, but weakly stratifiable since it does not generate overlaps with itself.

Yields a unique normal form.
Evaluation
Efficient Evaluation Algorithms

- **Order algorithm**
  - variant of nested loop join
  - works effectively on very sparse directed graphs
  - sometimes fixpoint evaluations can be avoided
  - index structures
  - bitvector unions can be used

- **DATALOG optimization techniques**
  - bottom-up/top-down evaluation
  - semi-naive evaluation
  - index structures
  - magic set transformation
  - transitive closure optimizations
Practical Features

- Short specifications expression
  - equivalence classes  30 rules
  - DFA reaching definitions 20-40
  - copy propagation 5
  - lazy code motion  5

- Flexibility: intermediate language CCMIR for C, Modula-2, Fortran

- Velocity:
  - Optimix generates the Order algorithm for a GRS.
  - Compiler with generated components is slower, but ..
  - important algorithms run as fast as hand-written algorithms (DFA).
Related Work

- Analysis Generators
  - Sharlit (Tijang)
  - PAG (Alt, Martin)
  - MetaFrame with modal logic (Knoop, Steffen)
  - Slicing-Tools (Reps, Field/Tip, Kamkar)

- Graph rewrite systems
  - Clean
  - Progres

- Uniform Approaches
  - GENESIS (Whitfield)
  - SPECIFY (Kock)
  - TRS (Field, ELAN, ASF-SDF)
Results
Results for Program Rewriting

- **Termination theory:**
  - If a termination graph can be identified, a graph rewrite systems terminates.
    - EARS are the most special case

- **Determinism theory**
  - EARS are strong confluent, direct derivations can always be interchanged. EARS have unique normal forms.
  - Stratification yields for many graph rewrite systems unique and natural normal forms because larger subgraphs are derived.

- Graph rewriting, DATALOG and data-flow analysis have a common core: EARS
The Common Core

Datalog

GRS

Program Analysis

EARS
Relation DFA/DATALOG/GRS

- Data-flow analysis, DATALOG and graph rewrite systems have a common kernel: EARS
  - As DATALOG, graph rewrite systems can be used to query the graph.
- Contrary to DATALOG graph rewrite systems materialize their results instantly.
- Graph rewriting is restricted to binary predicates and always yields all solutions.
- Graph rewriting can do transformation, i.e. is much more powerful than DATALOG.
- Graph rewriting enables a uniform view of the entire optimization process
Results for Program Optimization

- Program optimization:
  - Spezification of program optimizations is possible with graph rewrite systems. Short specifications, fewer effort.
  - First uniform methodology for analysis and transformation
  - Practically usable optimizer components can be generated.
Covered Analyses

- Every optimization where a mapping of the abstract domains to graphs can be found.
  - Powerset Lattices
  - Lattices of finite height or breadth
- Flow functions
  - Subset constraint systems
  - Equational systems
- Analysis
  - monotone and distributive data-flow analysis
  - control flow analysis
  - SSA
Covered Optimizations

- Local transformations
  - copy propagation, constant propagation
  - loop optimizations (unrolling etc.)
  - branch optimization, strength reduction
  - idiom recognition
  - dead code elimination

- Global transformations
  - lazy and busy code motion (loop invariant code motion)
  - message optimization
Limitations

- Currently there is no methodology on how to specify general abstract interpretations with graph rewrite systems.
- In interprocedural analysis, instead of chaotic iteration special evaluation strategies must be used [Reps95] [Knoop92].
- Currently these have to be modeled in the rewrite specifications explicitly.
- Several optimizations can be specified with GRS which are not exhaustive (peephole optimization, constant propagation with partial evaluation).
- As general rule embedding is not allowed, a rule only matches a fixed number of nodes.
  - Thus those transformations, which refer to an arbitrary set of nodes, cannot be specified.
Practice: How to Build an Optimizer

- Specify the optimizer in steps
  - Preprocessing steps (XGRS and EARS)
    - that convert the abstract syntax tree to an abstract syntax graph with definition-use relations
    - that diminish the domains of the analyses (e.g., equivalence classing)
    - that build summary information for procedures
    - that build indices for faster (constant) access
  - Analyses: specify with EARS
    - reaching-definition information, value flow information
    - SSA
  - Transformation (XGRS)
Other Application Areas

- Aspect Weaving
- Reengineering
- Modelling (OCL)

OPTIMIX 2.5 is out
optimix.sourceforge.net

- Works with CoSy, Cocktail, plain C, or Java
Outlook

- Take over the techniques of graph rewriting to DATALOG with update
- Take over some data-flow evaluation techniques to DATALOG
- Investigate non-stratifiable graph rewrite systems for optimization
- Find out about the connection to abstract interpretation