33. Model Transformation and Program Optimization with Graph Rewrite Systems

Prof. Dr. Uwe Aßmann

Softwaretechnologie

Technische Universität Dresden

Version 11-0.4, 29.12.11

1) Basic Setting
2) Examples
3) More on the Graph-Logic Isomorphism
4) Implementation in Tools

Obligatory Literature

- Uwe Aßmann. Graph rewrite systems for program optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 22(4):583-637, June 2000.
  - http://portal.acm.org/citation.cfm?id=363914
- Tom Mens. On the Use of Graph Transformations for Model Refactorings. GTTSE 2005, Springer, LNCS 4143
  - http://www.springerlink.com/content/5742246115107431/

Other References

- Uwe Aßmann. OPTIMIX, A Tool for Rewriting and Optimizing Programs. In Graph Grammar Handbook, Vol. II. Chapman-Hall, 1999.
- K. Lano. Catalogue of Model Transformations
  - http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf

33.1 Using GRS for Analysis and Transformation of Models and Code

Problem and Goal

- We need analyzers, transformers, and optimizers
  - For models: For model refactoring, adaptation and specialization, weaving and composition
  - For code: Portability to new processor types and memory hierarchies
  - For optimization (time, memory, energy consumption)
- However, transformers and optimizers are big beasts
  - Current implementation techniques are hard to understand and to a large extent unsystematic
- We need a uniform specification methodolody
  - covering many phases of optimizations
  - short specifications
  - effective code improvements
  - efficient optimizer components
- Idea: Use graph-logic isomorphism

An Old Citation

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

Model Transformation and Optimization with Graph Rewriting

- Represent everything as directed graphs
  - Program code (control flow, statements, procedures, classes)
  - Model elements (states, transitions, ...)
  - Analysis information (abstract domains, flow info ...)
- Directed graphs with node and edge types, node attributes
  - one-edge condition (no multi-graphs)
- Use edge addition rewrite systems (EARS) to
  - Query the graphs
  - Analyze the graphs
- Use graph rewrite systems (GRS) to
  - Construct and augment the graphs
  - Transform the graphs
- Preferably, the GRS should terminate (XGRS, exhaustive GRS)
- Use the graph-logic isomorphism to encode
  - Facts in graphs
  - Logic queries in graph rewrite systems

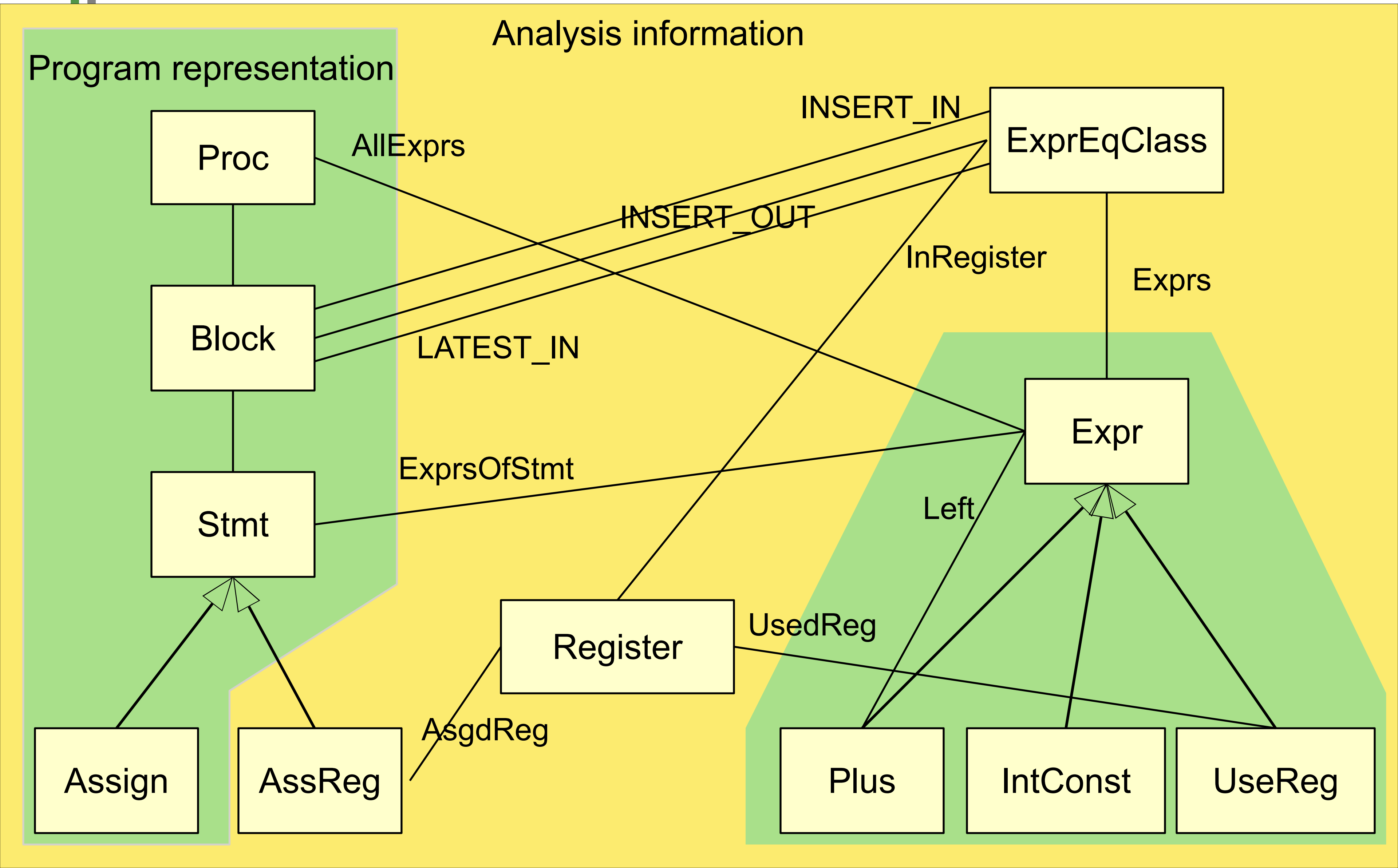
Terminology for Automated Graph Rewriting

- **Graph rewrite rule:** rule (left, right hand side) to match left-hand side in the graph and to transform it to the right-hand side
- **Graph rewrite system:** set of graph rewrite rules
- **Start graph (axiom):** input graph to rewriting
- **Graph rewrite problem:** a graph rewrite system applied to a start graph
- **Manipulated graph (host graph):** graph which is rewritten in graph rewrite problem
- **Redex**: (reducible expression) application place of a rule in the manipulated graph
- **Derivation:** a sequence of rewrite steps on the manipulated graph, starting from the start graph and ending in the normal form
- **Normal form:** result graph of rewriting; 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 commuted, resp. joined together to one result
- **Convergent GRS:** rewrite system that always yields unique results (terminating and confluent)

Specification Process

1) Specification of the data model (graph schema)
   - Specification of the graph schema with a graph-like DDL (ERD, MOF, GXL, UML or similar):
     - **Schema of the program representation**: program code as objects and basic relationships. This data, i.e., the start graph, is provided as result of the parser
     - **Schema of analysis information** (the infered predicates over the program objects) as objects or relationships
2) Program analysis (preparing the abstract interpretation)
   - Querying graphs, enlarging graphs
   - Materializing implicit knowledge to explicit knowledge
3) Abstract Interpretation (program analysis as interpretation)
   - Specifying the transfer functions of an abstract interpretation of the program with graph rewrite rules on the analysis information
4) Program transformation (optimization)
   - Transforming the program representation

A Simple Program (Code) Model (Schema) in UML

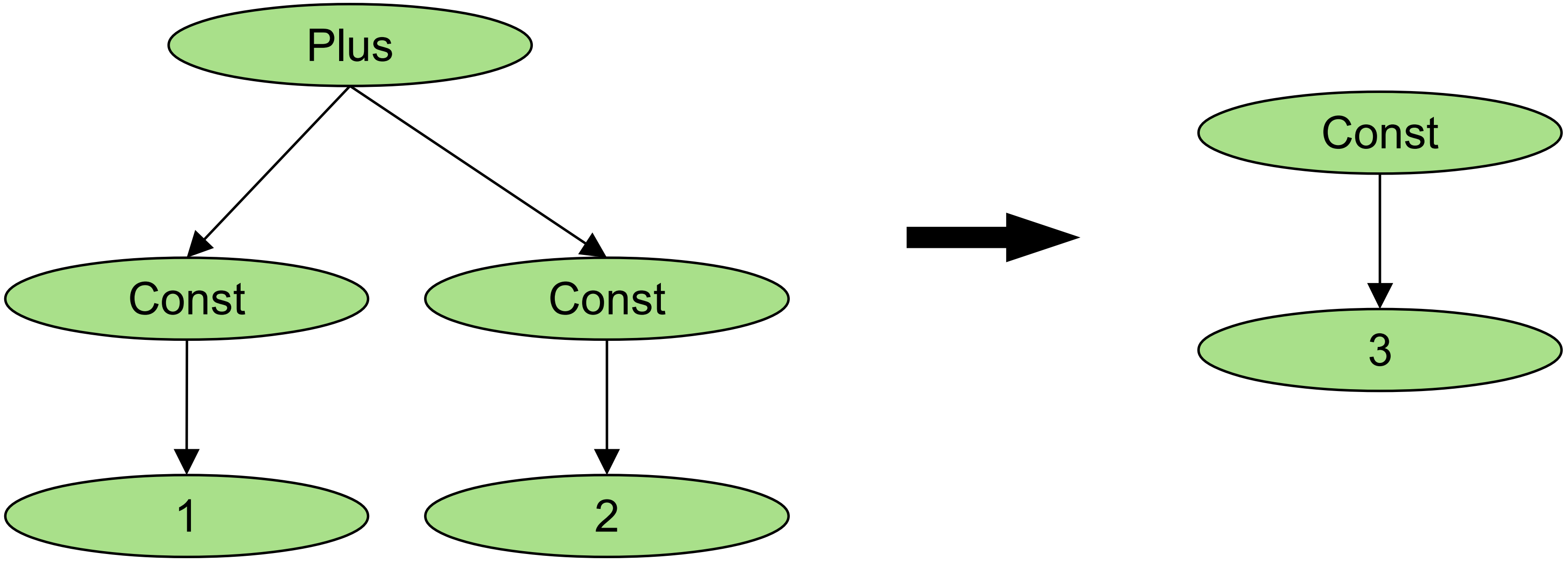


33.2 Examples

33.2.1. Local Rewritings

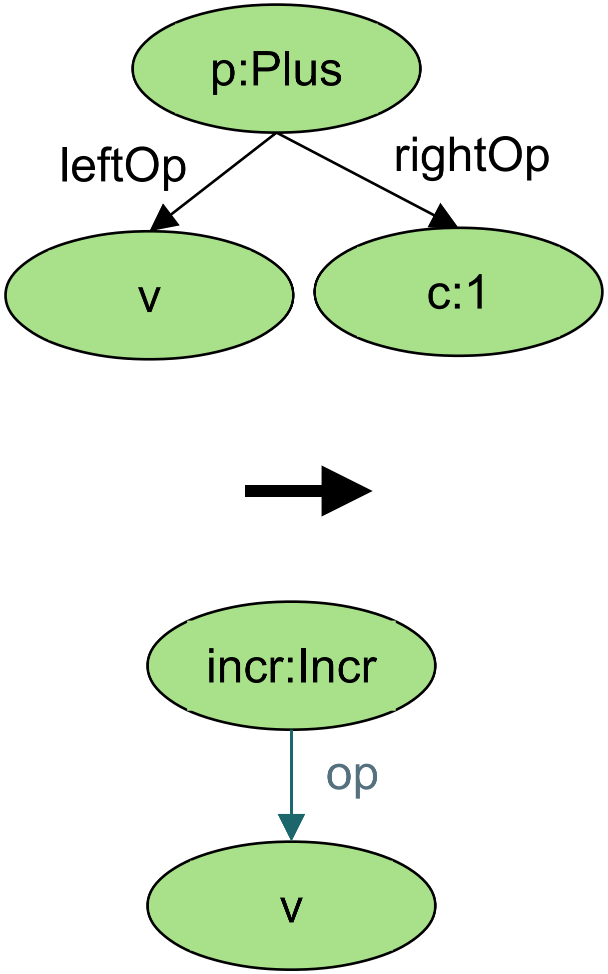
Constant Folding

- A **local rewriting (context-free rewriting)** matches a weakly connected left-hand side graph with a redex.
  - Matching of one redex can be done in constant time
- Subtractive because redexes are destroyed



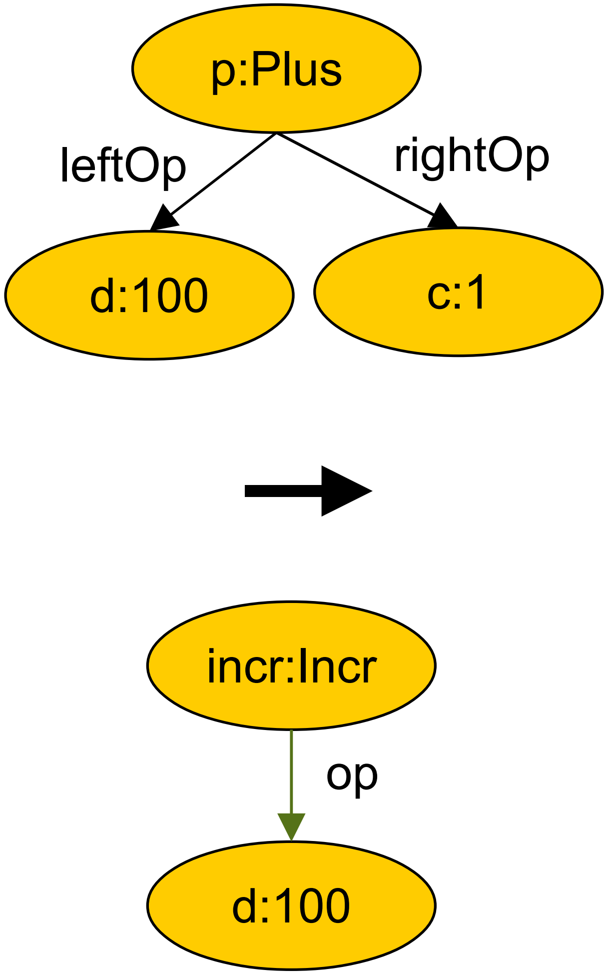
Context-Free Local Rewritings: Operator Strength reduction

```
// if-then rules:
if  leftOp(p:Plus,v),
    rightOp(p,c:1),
then
   Delete p,
   Delete c,
   Add incr:Incr,
   op(incr,v);
```



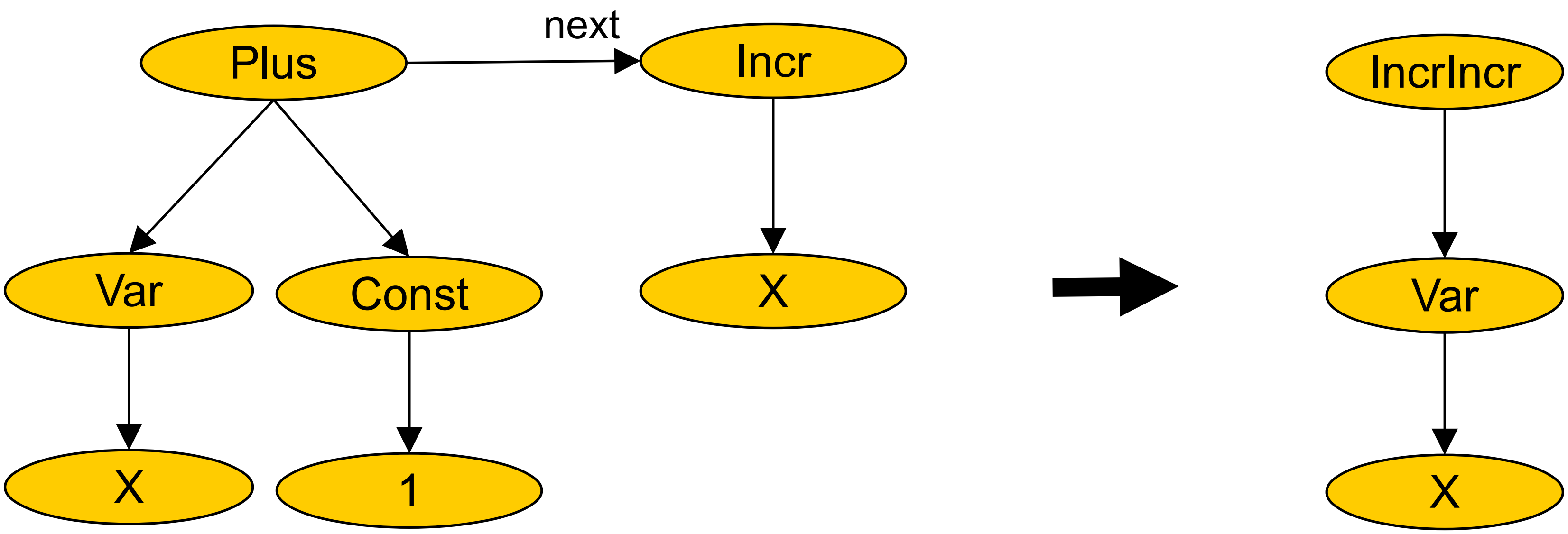
Context-Free Local Rewritings: Constant Folding

```
// if-then rules (logic):
if  leftOp(p:Plus,d:100),
    rightOp(p,c:1),
then
   Delete p,
   Delete c,
   d.value=100,
   op(incr,v);
```



Peephole Optimization

- Peephole optimization is done on statement lists or trees
- Subtractive problem, because redexes are destroyed



33.2.2. Path Abbreviations in Graph Analysis

- With edge addition rewrite systems

Path Abbreviations

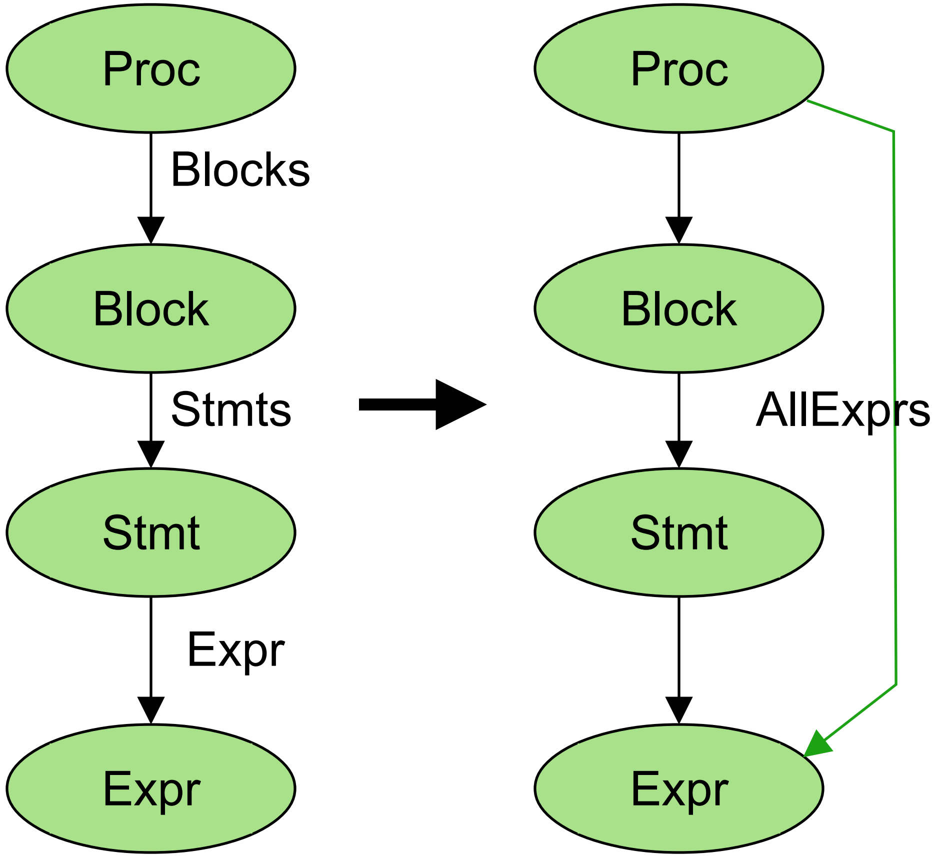
Collection of Expressions for a procedure: edge addition

-- Datalog notation:

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

-- if-then rules:

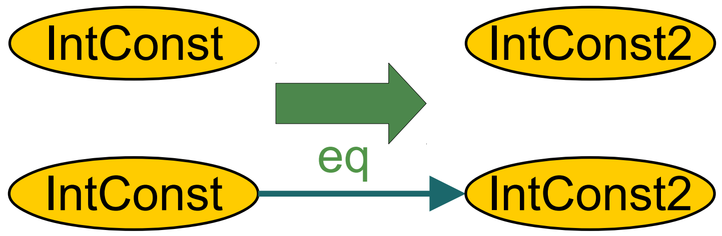
if Blocks(Proc,Block),
    Stmts(Block,Stmt),
    Expr(Stmt,Expr)
then
   AllExprs(Proc,Expr);

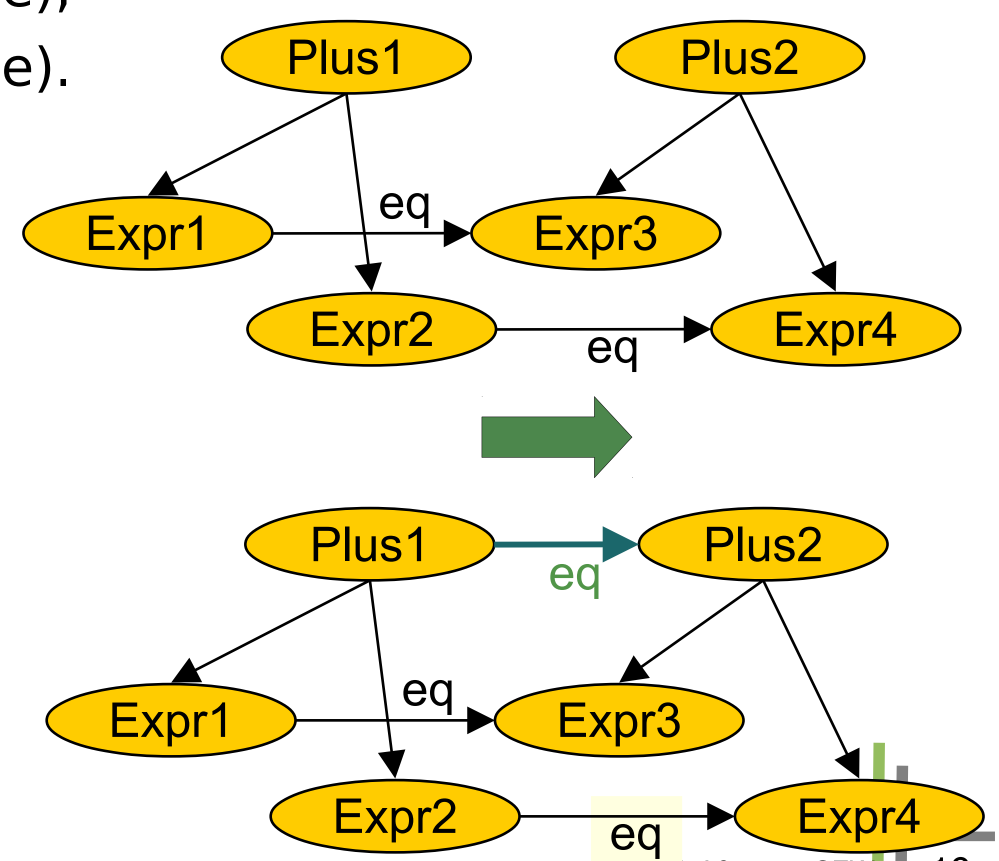


Value Numbering (Expression Equivalence)

Computing equivalent expressions

```
baserule:
eq(IntConst1,IntConst2) :-
   IntConst1 ~ IntConst(Value),
   IntConst2 ~ IntConst(Value).
recursive_rule:
eq(Plus1,Plus2) :-
   Plus1 ~ Plus(Type),
   Plus2 ~ Plus(Type),
   Left(Plus1,Expr1),
   Right(Plus1,Expr2),
   Left(Plus2,Expr3),
   Right(Plus2,Expr4).
   eq(Expr1,Expr3),
   eq(Expr2,Expr4).
```





33.2.3. Program Analysis with Abstract Interpretations

- with edge additions

Abstract Interpretations: Data-flow Analysis

- **Data-flow analysis** is an abstract interpretation computing the flow of data through the program, from variable assignments to variable uses
  - It results in the **value-flow graph (data-flow graph)**
- Examples:
- **Reaching Definitions Analysis:** Which Definitions (Assigments) of a variable can reach which statement?
- **Live Variable Analysis:** At which statement is a variable live, i.e., will further be used
- **Busy Expression Analysis:** Which expression will be used on all outgoing paths?
  - Central part: 1 recursive system

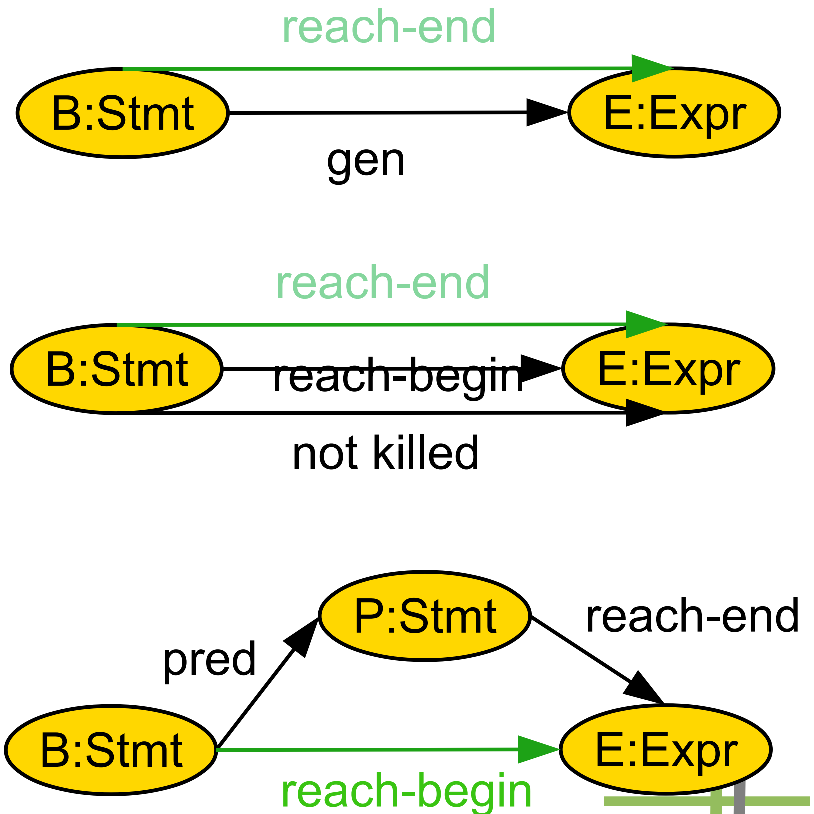
Reaching Definition Analysis

- Graph rewrite rules implement an abstract interpreter
- On instructions or on blocks of instructions
- Recursive system (via edge reach-begin)

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).



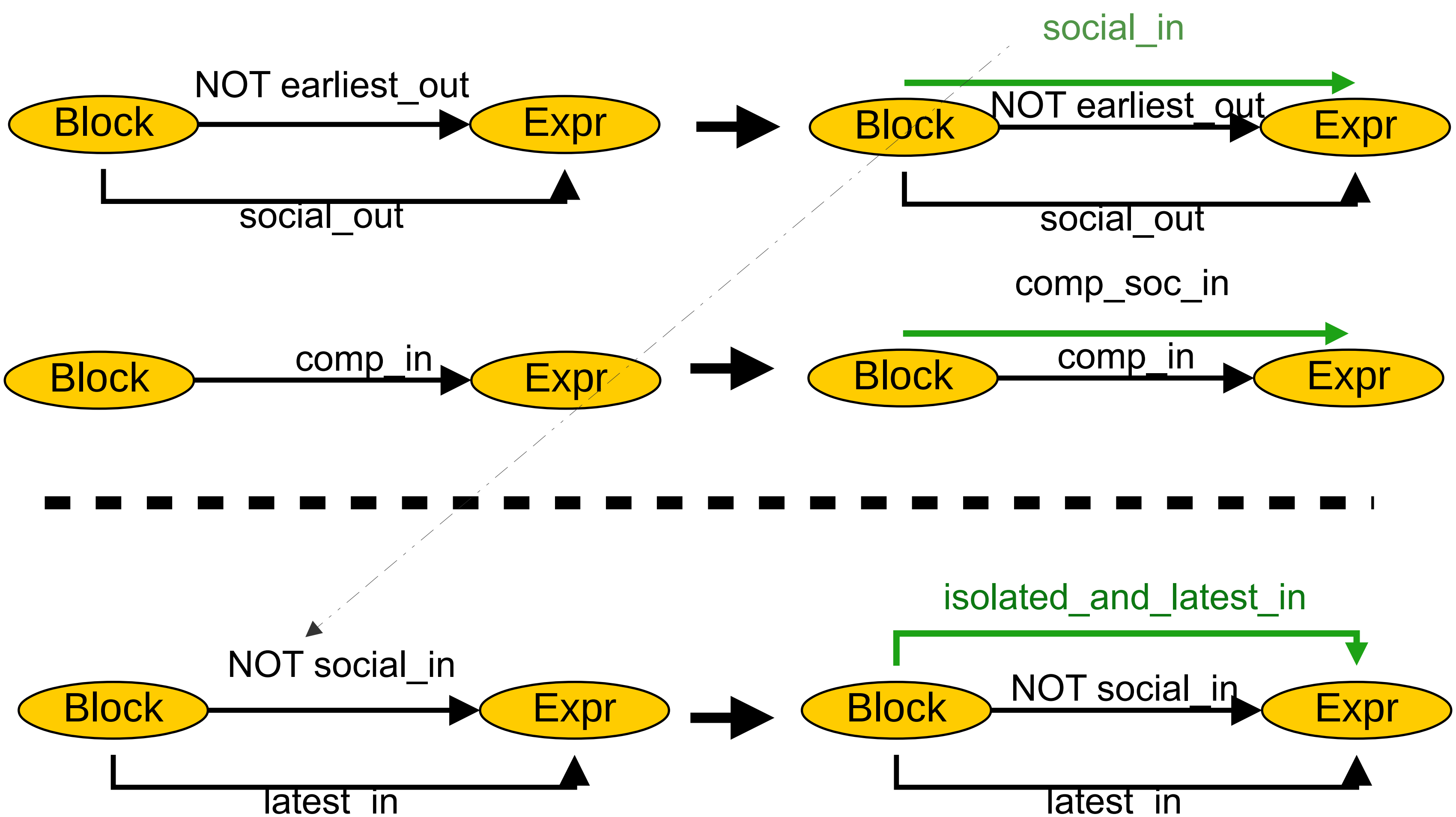
Code Motion Analysis

- Code motion is a complex transformation:
  - Moving loop-invariant expressions out of loops upward

- Busy Code Motion (BCM) moves expressions as upward (early) as possible
- Lazy Code Motion (LCM)
  - Moving expressions out of loops to the front of the loop, upward, but carefully:
  - Moving expressions to an optimal place so that register lifetimes are not too long (optimally early)
  - Shorter register lifetimes
- Code motion needs complex data-flow analysis:
  - Lazy Code Motion Analysis (LCM analysis) computes this optimal early place of an expression [Knoop/Steffen]
  - Analyze an optimally early place for the placement of an expression
  - About 6 equation systems similar to reaching-definitions
  - Every equation system is an EARS

Excerpt from LCM Analysis with Overlaps

- Compute an optimally early block for an expression (out of a loop)

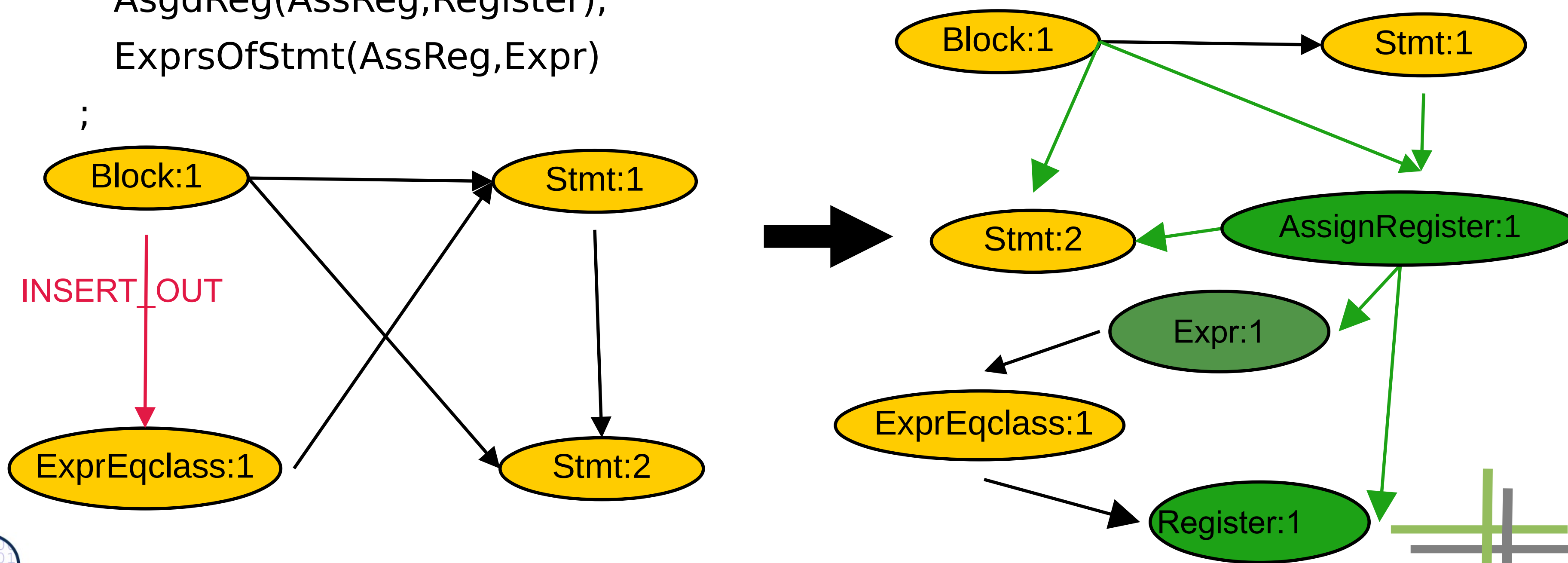


33.2.4. Complex Local Rewritings

Example: Lazy Code Motion Transformation

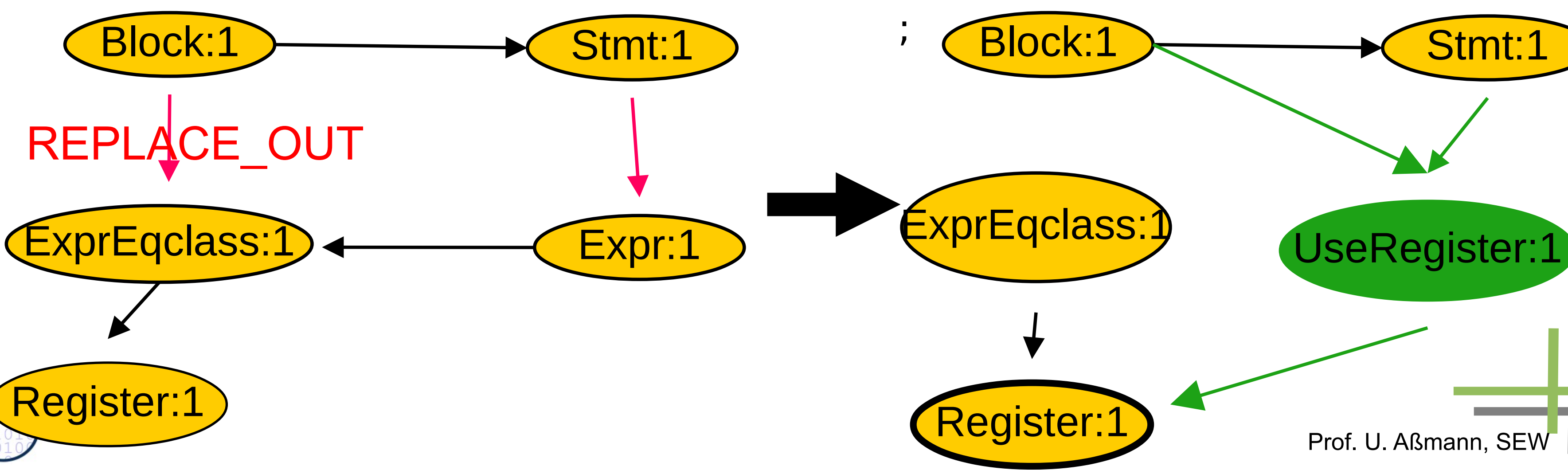
```
if Stmts.last(Block,Stmt),
   INSERT_OUT(Block,ExprEqclass)
then
   new Register:Register;
   new Expr:Expr;
   new AssReg:AssReg;
   InRegister(ExprEqclass,Register),
   AsgdReg(AssReg,Register),
   ExprsOfStmt(AssReg,Expr)
;
```

- Insert expressions at an optimally early place



Lazy Code Motion Transformation

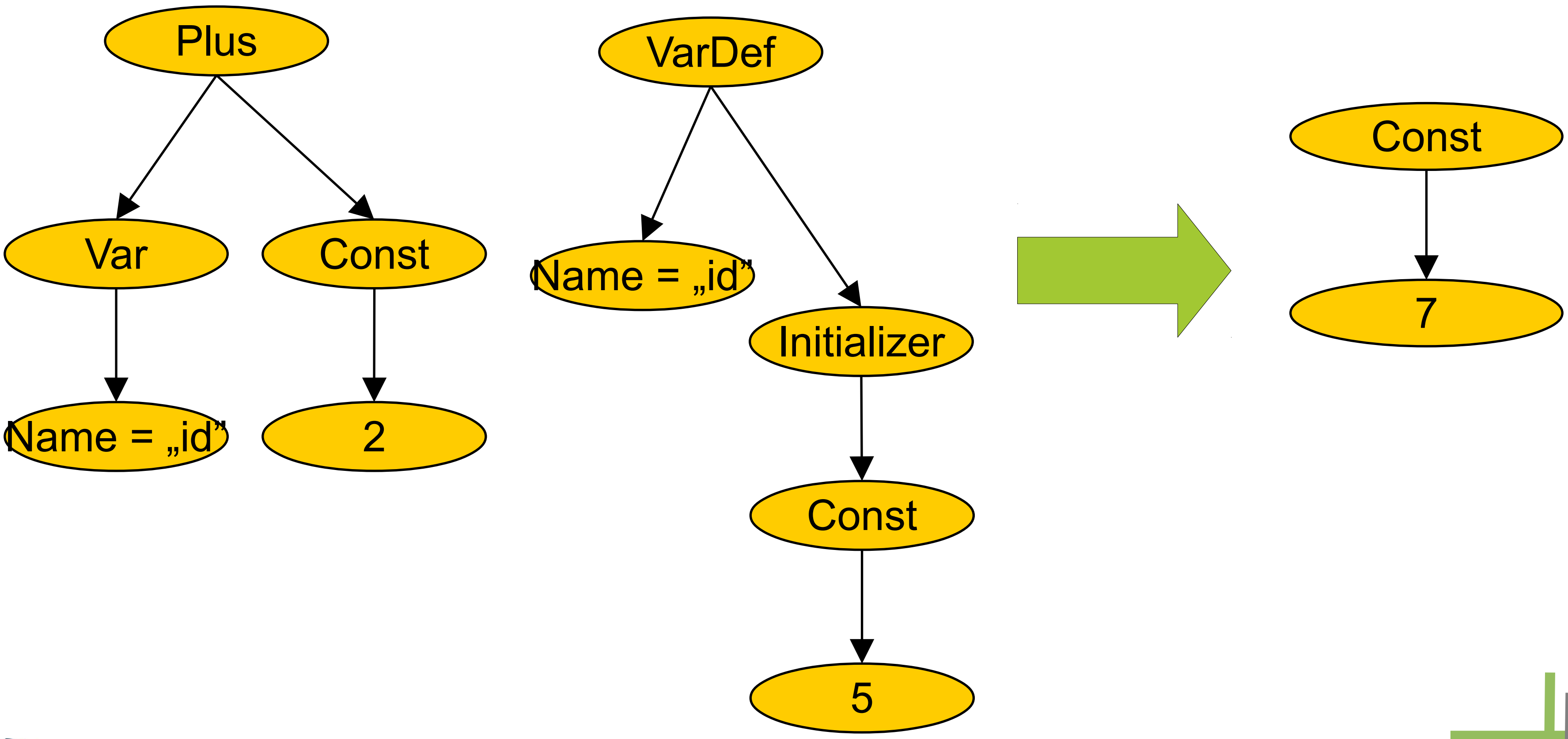
```
if  Stmts(Block,Stmt),
    ExprsOfStmt(Stmt,Expr),
    REPLACE_OUT(Block,ExprEqclass),
    InRegister(ExprEqclass,Register),
    Computes(Expr,ExprEqclass)
then
    new UseReg:UseReg;
    delete Expr;
    ExprsOfStmt(Stmt,UseReg),
    UsedReg(UseReg,Register)
;
```



33.2.5. Context-Sensitive Rewritings

Extended Constant Folding as Subtractive GRS

- A **context-sensitive rewriting** matches a non-connected left-hand side graph with a redex.
  - Matching of one redex can be done in quadratic time, because non-connected nodes have to be pairwise compared



33.3 More on the Logic-Graph Isomorphism

Covered Optimizations

- Analysis: Every analysis where a mapping of the abstract domains to graphs can be found.
  - Abstract interpretations
  - monotone and distributive data-flow analysis
  - control flow analysis
  - SSA construction
  - Interprocedural IDFS framework (Reps)
- Local transformations of the program representation
  - 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

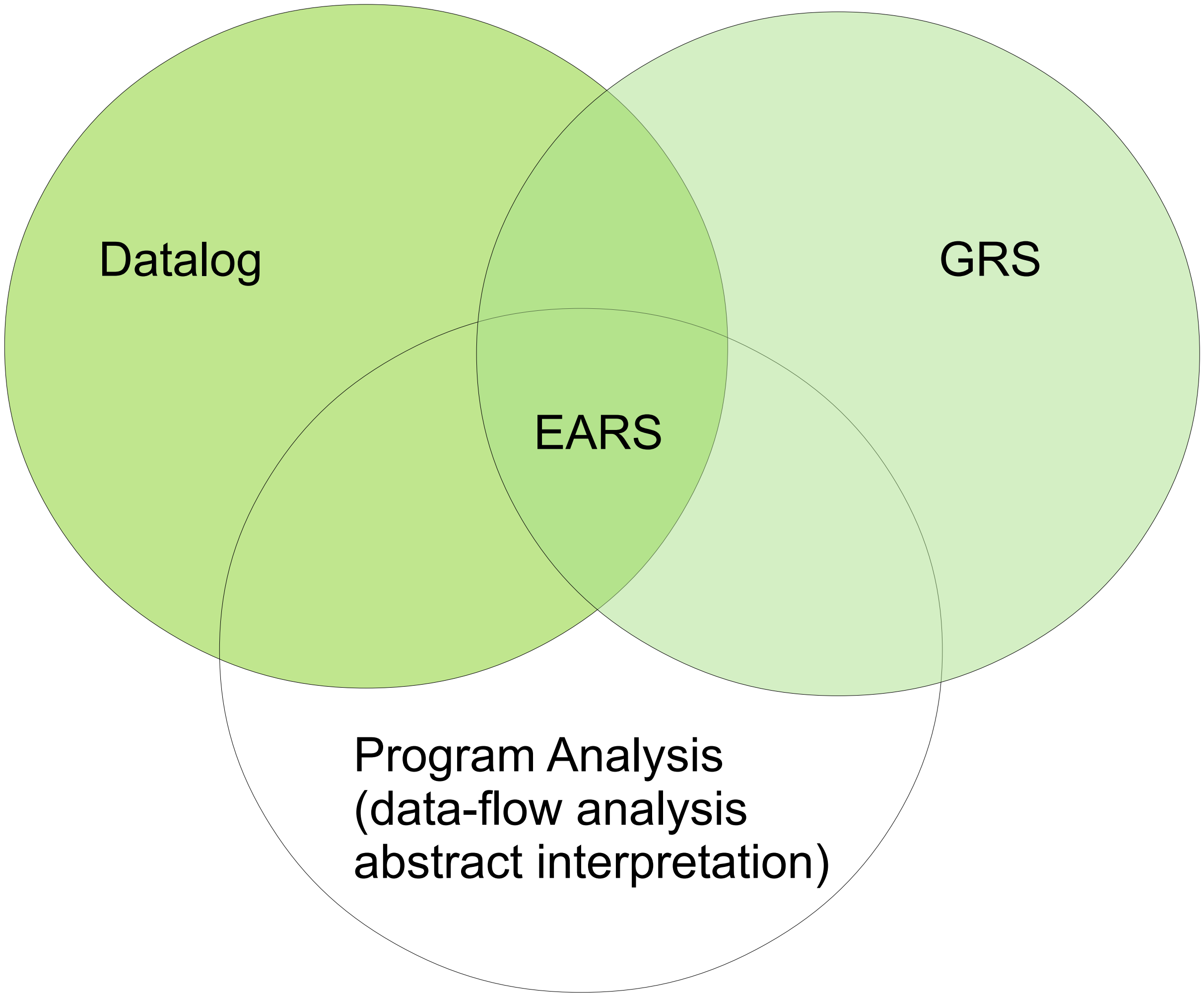
Results

- Theory:
  - If a termination graph can be identified, a graph rewrite systems terminates.
  - Graph rewriting, DATALOG and data-flow analysis have a common core: EARS
- Program optimization:
  - Spezification of program optimizations is possible with graph rewrite systems. Short specifications, fewer effort.
  - Practically usable optimizer components can be generated.
- Uniform Specification of Analysis and Transformation
  - If the program analysis (including abstract interpretation) is specified with GRS
  - It can be unified with program transformation

Limitations

- Currently there is no methodology on how to specify general abstract interpretations, beyond classical data-flow analysis, 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.

The Common Core of Logic, Rewriting and Program Analysis



Relation DFA/DATALOG/GRS

- Abstract interpretation (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

33.4 Implementation in Tools

Process: How to Build an Optimizer or Model Transformer

- Specify the optimizer in steps:
  - Preprocessing steps with 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 abstract interpretations with EARS
    - reaching-definition information, value flow information
    - SSA
  - Transformation: apply XGRS and stratifiable XGRS

Efficient Evaluation Algorithms from Logic Programming

- „Order algorithm“ scheme [Aßmann00]
  - Variant of nested loop join
  - Easy to generate into code of a programming language
  - Works effectively on very sparse directed graphs
  - Sometimes fixpoint evaluations can be avoided
  - Use of index structures possible
  - Linear bitvector union operations can be used
- DATALOG optimization techniques can be employed
  - Bottom-up evaluation is normal, as in Datalog
  - Top-down evaluation as in Prolog possible, with resolution
  - 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
- Velocity:
  - Tool 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)
- Flexibility:
  - intermediate language CCMIR for C (CoSy), Modula-2, Fortran (Aßmann)
  - Model transformations (Alexander Christoph)
  - Aspect weaving (Aßmann, Heidenreich, many others)
  - Refactorings (Aßmann, Mens)
- OPTIMIX 2.5 on optimix.sourceforge.net
  - Works with CoSy, Cocktail, or plain C
  - A prototype code generator for Java exists

Tools for Model-Driven Software Development

- In MDSD and MDA, horizontal and vertical model transformations should be specified with graph rewrite systems
- Example tools:
  - **Fujaba**
  - **MOFLON**
  - VIATRA2 on EMF http://eclipse.org/gmt/VIATRA2/

Related Work

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